This is a draft (date: 2020-Jan-25). It will have errors. It is also subject to change, possibly extensive change. I apologize for the errors and I’d be glad for reports; visit joshua.smcvt.edu/computation. In any event, be warned that this version is provided simply, as I say, as a draft.

---

**Greek letters with pronounciation**

<table>
<thead>
<tr>
<th>character</th>
<th>name</th>
<th>character</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>alpha</td>
<td>ν</td>
<td>nu</td>
</tr>
<tr>
<td>β</td>
<td>beta</td>
<td>ξ, Ξ</td>
<td>xi</td>
</tr>
<tr>
<td>γ, Γ</td>
<td>gamma</td>
<td>ο</td>
<td>omicron</td>
</tr>
<tr>
<td>δ, Δ</td>
<td>delta</td>
<td>π, Π</td>
<td>pi</td>
</tr>
<tr>
<td>ε</td>
<td>epsilon</td>
<td>ρ</td>
<td>rho</td>
</tr>
<tr>
<td>ζ</td>
<td>zeta</td>
<td>σ, Σ</td>
<td>sigma</td>
</tr>
<tr>
<td>η</td>
<td>eta</td>
<td>τ</td>
<td>tau</td>
</tr>
<tr>
<td>θ, Θ</td>
<td>theta</td>
<td>υ, Υ</td>
<td>upsilon</td>
</tr>
<tr>
<td>ϵ</td>
<td>iota</td>
<td>ϕ, Φ</td>
<td>phi</td>
</tr>
<tr>
<td>κ</td>
<td>kappa</td>
<td>χ</td>
<td>chi</td>
</tr>
<tr>
<td>λ, Λ</td>
<td>lambda</td>
<td>ψ, Ψ</td>
<td>psi</td>
</tr>
<tr>
<td>μ</td>
<td>mu</td>
<td>ω, Ω</td>
<td>omega</td>
</tr>
</tbody>
</table>

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

---

Preface

The Theory of Computation is a wonderful thing. It is beautiful. It has deep connections with other areas of computer science and mathematics, as well as with the wider intellectual world. And, looking forward into this century, clearly a theme will be the power 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 easy reach are as-yet-unknown areas.

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

For students  We will study the idea of computation, so that we will be less interested in semicolons and cache hits than in what things can be done with a mechanism at all. This has been intensively studied for nearly a hundred years; we will not see all that is known but we will see enough that you will end with an overview, with some central insights.

Unifying it all is the computational perspective, that there are organizing principles in computation and that seeing how those principles hold across different models gives a deep understanding of both the problems and their solutions.

We do not stint on mathematics. But you will find the story told in a liberal way, in a way that, in addition to technical detail, also aims for a breadth of knowledge. We will be eager to make connections with other fields and with other modes of thinking. We learn things more completely when they fit in a larger picture, as the first quote below expresses.

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

The subject is big, and a challenge. It is also a great deal of fun. Enjoy!

For instructors  This text covers the standard topics: the definition of computability, unsolvability, languages, automata, and complexity. The audience is undergraduate majors in computer science, mathematics, and nearby areas.

The prerequisites are from the standard discrete mathematics course: propositional logic and truth tables, proof methods including induction, graphs, basic number theory, sets, functions, and relations. For some of these topics, such as graphs or big-\(O\), there are optional review sections. I also expect programming experience.
A text does students a disservice if its presentation is not precise. The details matter. But students can also fail to understand the subject because they have not had a chance to reflect on and flesh out the underlying ideas. The presentation here aims to give a fuller understanding of the results both by stressing motivation and naturalness and, where practical, by setting the results in a context.

An example of this difference comes at the start in our choice of computing model, the Turing machine. This is the traditional model, leads naturally to Finite State machines, and is standard in complexity. Its downside is that it is awkward for extensive computation. However, here we don’t do extensive computations. We immediately pass to a discussion of Church’s Thesis, thereby relying on the intuition that students have from their programming experience. Besides giving some motivation and meaning to the formalities, this allows us to sketch algorithms, which is both clearer and faster.

A second example is nondeterminism. We introduce it in the context of Finite State machines and pair that introduction with a discussion giving students a chance to reflect on this key idea. Another example comes with the inclusion in the complexity material of a section of the kind of problems that drive the work today. Still another is the discussion of the current state of P versus NP. All of these examples, and many more, taken together encourage students to reflect on the subject’s larger ideas.

Schedule  This is my semester's schedule. Chapter I defines models of computation, Chapter II covers unsolvability, Chapter III does languages, Chapter IV does automata, and Chapter V is computational complexity.

<table>
<thead>
<tr>
<th>Week</th>
<th>Sections</th>
<th>Reading</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I.1, I.3, I.4</td>
<td>I.2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>II.1, II.2</td>
<td>II.A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>II.3, II.4</td>
<td>II.B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>II.5, II.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>II.8, III.1</td>
<td>II.C</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>III.2</td>
<td>III.A</td>
<td>EXAM</td>
</tr>
<tr>
<td>7</td>
<td>III.4, IV.1</td>
<td>III.3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IV.2, IV.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>IV.3, IV.4</td>
<td>IV.A</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>IV.5, IV.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>V.1</td>
<td>IV.1.4</td>
<td>EXAM</td>
</tr>
<tr>
<td>12</td>
<td>V.2, V.3</td>
<td>V.A</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>V.4, V.5</td>
<td>V.5.4</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>V.5</td>
<td>V.B</td>
<td>REVIEW</td>
</tr>
</tbody>
</table>

An instructor could omit the readings, except for the first, but I give them as homework and quiz on them. I also assign programming homework and am developing a set of Jupyter notebooks for the material.
Speaking in code  As the quote below by A Downey expresses, for communicating effective procedures, a modern programming language is hard to beat as a blend of precision and concision. We use Scheme as a single, uniform, practical language for this. One reason is that it is very easy to learn. Another is that it suits the topics. But the main reason is that, as E Dijkstra says below, it is a delight.

Exploration, Enrichment, Connections  This book encourages readers to be active. It is written in lively language and contains many illustrations, from pictures of some of the people who have developed the topics through to automata diagrams and graphs. The Theory of Computation is a stimulating subject and it deserves a presentation to match.

The presentation also encourages connection-making through the topics that end each chapter, which are both fascinating and fun. These are suitable as single day lectures, or to assign for group work, or for extra credit, or just for looking at.

One way to stimulate readers to explore is with links. There are many notes in the back that fill out, and perhaps add a spark to, the core presentation. Where practical, the references are clickable. For example, the pictures of the people who discovered the subject are a link to their Wikipedia page. I hope readers enjoy these and I know that because it is linked-to, they are very much more likely to be seen than is the same content in a library.

License  This book is Free. You can use it without cost. You can also redistribute it—an instructor can make copies and distribute it through their bookstore or as a file on their school’s intranet. You can also get the \LaTeX{} source and modify it to suit your class; see http://joshua.smcvt.edu/computation.

One reason that it is Free is that it is written in \LaTeX{}, which is Free, as is our Scheme implementation, as is Asymptote that drew the illustrations, as is Emacs and all of GNU software, and the entire Linux platform on which this book was developed. And besides those, all of the research that this text presents was made freely available by scholars. I believe that the synthesis here adds value—I hope so, indeed—but the masters have left a well-marked trail and following it seems only right.

Acknowledgements  I owe a great debt to my wife, whose patience with this project has gone beyond all reasonable limits. My students have also been patient and have made the book better in many ways.

And, I must acknowledge my teachers. There are many but first among them is M Lerman. Thank you, Manny.

They also include H Abelson, G J Sussmann, and J Sussmann, who had the courage with Structure and Interpretation of Computer Programs to show students just how mind-blowing it all is. When I see a programming text where the 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

Teach concepts, not tricks.
– Gian-Carlo Rota, TEN LESSONS I WISH I HAD LEARNED BEFORE I STARTED TEACHING DIFFERENTIAL EQUATIONS

While 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.
– Elaine 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

Lisp has jokingly been called “the most intelligent way to misuse a computer.” I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.
– E Dijkstra, CACM, 15:10

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
joshua.smcvt.edu/computing
Draft: version 0.92, 2019-Dec-27.
## Contents

### I Mechanical Computation

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? .................................. 17
   5 Using Church’s Thesis .................................. 18
3 Recursion .................................................... 21
   1 Primitive recursion ..................................... 21
4 General recursion ........................................... 30
   1 Ackermann functions ................................... 30
   2 μ recursion ............................................ 33
A Turing machine simulator .................................... 36
B Hardware ..................................................... 40
C Game of Life ................................................. 45
D Ackermann’s function is not primitive recursive ....... 49
E LOOP programs ............................................... 52

### II Background

1Infinity ......................................................... 60
   1 Cardinality .............................................. 60
2 Cantor’s correspondence .................................... 67
3 Diagonalization .............................................. 75
   1Diagonalization ........................................... 75
4 Universality .................................................. 81
   1 Universal Turing machine ............................... 81
   2 Uniformity ............................................. 84
   3 Parametrization ......................................... 85
5 Unsolvability ................................................ 90
   1 The Halting problem .................................... 90
   2 Discussion ............................................. 91
   3 Significance ............................................ 93
   4 General unsolvability .................................. 94
   5 Rice’s Theorem ......................................... 97
6 Computably enumerable sets ............................... 101
7 Oracles ....................................................... 105
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 Alan Turing that the first step toward defining the set of computable functions is to reflect on the details of what mechanisms can do. The context of Turing's thinking was the Entscheidungsproblem, proposed in 1928 by D Hilbert and W Ackermann, which asks for an algorithm that decides, after taking as input a mathematical statement, whether that statement is true or false. So he considered the kind of symbol-manipulating computation familiar in mathematics, 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 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 simple case as a prototype, Turing posited conditions for the computing agent.

First, it (or he, or she) has a memory facility, like 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. So there is no question about the precision
of operations as there might be, say, when 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 $q_0, q_1, \ldots$

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

The final question Turing faced is: how smart is the computing agent? For instance, can it multiply? We don’t need to include a special facility for multiplication because we can in principle multiply via repeated addition. We don’t even need addition because we can iterate the successor operation, the add-one operation. In this way he pared the computing agent down until it was 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 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,

---

† It is true that a physical computer such as your 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.

‡ 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 existing device.
Section 1. Turing machines

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 and 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 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 be in only 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 what it is now reading on the tape, that it next needs to carry a 1. The agent’s next step is to transition to a new state, one whose intuitive meaning is that carries take place there.

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

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 then 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 character set \( \Sigma = \{ B, 1 \} \) has six instructions.

\[
P_{\text{pred}} = \{ q_0 BL q_1, q_0 1 R q_0, q_1 BL q_2, q_1 1 B q_1, q_2 B R q_3, q_2 1 L q_2 \}
\]

To trace its execution, below we’ve represented this machine in an initial configuration. This shows a stretch of the tape, including all its non-blank contents, along with the machine’s state and the position of its read-write head.

![Diagram of Turing machine](image)

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

†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 drawings we hold the tape steady and move the head because then comparing graphics step by step is easier.
picture shows it reading \( 1 \) so instruction \( q_0 1 R q_0 \) applies. Thus the first step is
that the machine moves its tape head right and stays in state \( q_0 \). Below, the first
line shows this and later lines show the machine’s configuration after later steps.
Roughly, the computation slides to the right, blanks out the final \( 1 \), and slides back
to the start.

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1</td>
</tr>
<tr>
<td></td>
<td>( q_0 )</td>
</tr>
<tr>
<td>2</td>
<td>1 1 1</td>
</tr>
<tr>
<td></td>
<td>( q_0 )</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1</td>
</tr>
<tr>
<td></td>
<td>( q_0 )</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1</td>
</tr>
<tr>
<td></td>
<td>( q_1 )</td>
</tr>
<tr>
<td>5</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>( q_1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>( q_2 )</td>
</tr>
<tr>
<td>7</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>( q_2 )</td>
</tr>
<tr>
<td>8</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>( q_2 )</td>
</tr>
<tr>
<td>9</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>( q_3 )</td>
</tr>
</tbody>
</table>

Next, 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 \( 1 \)'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 ten-instruction machine adds two natural numbers.

\[
P_{\text{add}} = \{ q_0 B B q_1, q_0 1 R q_0, q_1 B 1 q_1, q_1 1 1 q_2, q_2 1 L q_2, q_2 B B q_3, q_3 B R q_3, q_3 1 B q_4, q_4 B R q_5, q_4 1 1 q_5 \}
\]

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 to the start of the string. Here are the steps.
Instead of giving a machine’s instructions as a list we can use a table or a diagram. Here is the transition table for $P_{\text{pred}}$ and its transition graph.

$$
\begin{array}{c|cc}
\Delta & B & 1 \\
\hline
q_0 & Lq_1 & Rq_0 \\
q_1 & Lq_2 & Bq_1 \\
q_2 & Rq_3 & Lq_2 \\
q_3 & - & - \\
\end{array}
$$

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

$$
\begin{array}{c|cc}
\Delta & B & 1 \\
\hline
q_0 & Bq_1 & Rq_0 \\
q_1 & 1q_1 & 1q_2 \\
q_2 & Lq_2 & Bq_3 \\
q_3 & Rq_3 & Bq_4 \\
q_4 & Rq_5 & 1q_5 \\
q_5 & - & - \\
\end{array}
$$

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 $P_{\text{inf loop}} = \{ q_0 Bq_0, q_0 11q_0 \}$ never halts, regardless of the input.
The exercises ask for examples of Turing machines that halt on some inputs and not on others.

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

1.4 Definition A Turing machine $\mathcal{P}$ is a finite set of four-tuple instructions $q_p T_p T_n q_n$. 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 a Turing machine by $\mathcal{P}$ because the thing from our everyday experience that a Turing machine is most like is a computer program.

Of course, the point of a machine is what it does. 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 how these machines act.

We saw in tracing through Example 1.1 and Example 1.2 that a machine acts by transitioning from one configuration to the next. A configuration of a Turing machine is a four-tuple $C = \langle q, s, \tau_L, \tau_R \rangle$, 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 $\epsilon$. 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 $C(t)$ 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 $C(0)$ is the machine’s configuration before we press Start.

Suppose that at step $t$ a machine $\mathcal{P}$ is in configuration $C(t) = \langle q, s, \tau_L, \tau_R \rangle$. 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 are three possibilities. (1) If $T_n$ is a symbol in the tape alphabet set $\Sigma$ then

---

† 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 an instrument might have in distinguishing two values near its limit of resolution.
Section 1. Turing machines

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 $T_n = \lambda$ then the machine moves the tape head to the left. That is, the next configuration is $C(t + 1) = \langle q_n, \hat{s}, \hat{t}_L, \hat{t}_R \rangle$, where $\hat{s}$ is the rightmost character of the string $\tau_L$ (if $\tau_L = \varepsilon$ then $\hat{s}$ is the blank character), where $\hat{t}_L$ is $\tau_L$ with its rightmost character omitted (if $\tau_L = \varepsilon$ then $\hat{t}_L = \varepsilon$ also), and where $\hat{t}_R$ 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 $C(i) \vdash C(i + 1)$. A computation is a sequence $C(0) \vdash C(1) \vdash C(2) \vdash \cdots$. We abbreviate such a sequence with $\vdash^*$. 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.

$$\langle q_0, 1, \varepsilon, 11 \rangle \vdash \langle q_0, 1, 1, 1 \rangle \vdash \langle q_0, 1, 11, \varepsilon \rangle \vdash \langle q_0, B, 111, \varepsilon \rangle \vdash \langle q_1, 1, 11, \varepsilon \rangle$$

$$\vdash \langle q_1, B, 11, \varepsilon \rangle \vdash \langle q_2, 1, 1, \varepsilon \rangle \vdash \langle q_2, 1, \varepsilon, 1 \rangle \vdash \langle q_2, B, \varepsilon, 11 \rangle \vdash \langle q_3, 1, \varepsilon, 1 \rangle$$

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 this chapter’s opening we declared that our interest is not so much in the machines as it is in the things that they compute. We close with a definition of the set of functions that are mechanically computable.

A function is an association of inputs with outputs. The simplest candidate for a definition of the function computed by a machine is the association of the string on the tape when the machine starts with the string on the tape when it halts, if indeed it does halt. (For the following definition, note that where $X$ is a set of characters, we use $X^*$ to denote the set of finite-length strings of those characters).

1.6 Definition Let $P$ be a Turing machine with tape alphabet $\Sigma$, and let $\Sigma_0$ be $\Sigma - \{B\}$. The function $\phi_P : \Sigma_0^* \to \Sigma_0^*$ computed by $P$ is this: to find $\phi(\sigma)$ for $\sigma \in \Sigma_0^*$, place that string on an otherwise blank tape, point the read/write head to its first symbol, its left-most symbol, and start the machine. If $P$ halts, and the non-blank characters are consecutive with the first such character under the head, and that string is $\tau \in \Sigma_0^*$, then we take $\phi_P(\sigma) = \tau$.

This illustrates the computation of a function where $\phi(111) = 11111$. (If there is

---

\(^\dagger\) Read the turnstile symbol $\vdash$ aloud as “yields.” We could, where $I$ is the applicable instruction, write $\vdash^I$, but we will never need that construct. \(^\ddagger\) Read this aloud as “yields eventually.” \(^\spadesuit\) For more on functions see the Appendix on page 318. \(^\clubsuit\) Mathematicians began the subject by studying the effective computation of mathematical functions, principally functions from number theory. They often worked with the simplest tape alphabet, $\Sigma = \{B, 1\}$. This approach has proven fruitful so researchers still often discuss the subject in these terms.
only one machine under discussion then we omit the subscript and just write $\phi$.)

That definition has two fine points, both needed to make the input-output association well-defined. One is that just specifying that the machine starts with $\sigma$ on the tape is not enough since the initial position of the head can change the output, so the definition also specifies the head’s initial position. And, the definition omits blanks from $\sigma$ and $\tau$ since the machine would not be able to distinguish blanks at the end of those strings from blanks that are part of the unbounded tape.

The definition says “If $P$ halts . . . ” What if it doesn’t?

1.7 Definition If for a Turing machine the value of a computation is not defined on some input $\sigma \in \Sigma_0$ then we say that the function computed by the machine diverges, written $\phi(\sigma)\uparrow$. Where the machine does have an associated output value, we say that its function converges, written $\phi(\sigma)\downarrow$. If $\phi$ is defined for each input in $\Sigma_0$ then it is a total function. If it diverges for at least one member of $\Sigma_0$ then it is a partial function.

There is a difference between a 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 write as $x \mapsto \text{pred}(x)$. Another example of the difference is that machines halt or fail to halt, while functions converge or diverge.

1.8 Remark Aren’t there more functions that are computable; what about functions with multiple inputs or outputs? For instance, suppose that we want to show that the function that takes in two natural numbers $a$ and $b$ and returns $a^b$ is mechanically computable. The trick here is to consider the input string of 1’s to be an encoding of multiple inputs. For instance, we could set an exponentiation routine up so that it takes in as a string of $x$-many 1’s, performs a prime factorization as $x = 2^a3^b \cdot k$ for some $k \in \mathbb{N}$ to recover the two numbers, and then returns $a^b$. In this way we can get a two-input function from a single input string.

OK then, what about computing with non-numbers? For instance, we may want to find the shortest path through a graph. In an extension of the prior paragraph, to compute with a graph we find a way to represent it with a string. Programs that work with graphs may first decode the input string, then compute the answer, and finish by encoding the answer as a string.

These codings may seem awkward, and they are. (Of course, a programming language does conversions from decimal to binary and back again that are somewhat similar.) But a Turing machine simply manipulates characters and we externally provide an interpretation.

When we describe the function computed by a machine we will often gloss

\footnote{Some authors don’t require that the first character in the output is under the head. But this way is neater.}
over the part about interpreting the strings. We might say, “this shows $\phi(3) = 5$” rather than, “this shows $\phi$ taking a string representing 3 in unary to a string representing 5.” That may seem minor but early researchers, working before there were computing machines available, needed convincing 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 worked through those details. But for us it is different. Our everyday experience with the machines around us is that they are able to use their alphabet, binary, to get a reasonable representation of anything that our intuition says is computable. So in this book to show that the functions we expect are computable are indeed computable we will not fix an encoding and extensively work through the details of producing those functions. We omit this just to get sooner to interesting material. The next section says more.

1.9 **Definition** A *computable function*, or *recursive function†*, is a total or partial function that is computed by some Turing machine. A *computable set*, or *recursive set*, is one whose characteristic function is computable. A Turing machine is a *decider* for 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 given a characterization of 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 has led to a precise definition of which functions are mechanically computable. In the next subsection we will discuss this characterization, including the evidence that leads to its widespread acceptance.

effective function

1.1 **Exercises**

For each exercise, unless it says otherwise assume the alphabet is $\Sigma = \{ B, 1 \}$. Also assume that any machine you will write starts with its head under the leftmost non-blank character, if there are any non-blank characters, and arrange for it to end in the same way.

✓ 1.10 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.

✓ 1.11 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.

†The term ‘recursive’ used to be used universally but now is old-fashioned.
1.12 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.

1.13 Trace the execution of \( P_{\text{inf loop}} \) for ten steps, from a blank tape. Show the sequence of tapes.

1.14 Trace the execution on each input of this Turing machine with alphabet \( \Sigma = \{B, 0, 1\} \) for ten steps, or fewer if it halts.

\[
\{ q_0BBq_4, q_00Rq_0, q_01Rq_1, q_1BBq_4, q_10Rq_2, q_11Rq_0, q_2BBq_4, q_20Rq_0, q_21Rq_3 \}
\]

(A) 11 (B) 1011 (C) 110 (D) 1101 (E) \( \epsilon \)

1.15 Give the transition table for the machine in the prior exercise.

1.16 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.

1.17 Produce Turing machines to perform these Boolean operations, using \( \Sigma = \{B, 0, 1\} \).

(a) Take the ‘not’ of a bit \( b \in \Sigma_0 = \Sigma - \{B\} \); that is, convert an input of \( b = 0 \) into 1, and an input of \( b = 1 \) into 0. (b) Take as input two characters drawn from \( \Sigma_0 \) and give as output the single character that is their logical ‘and’. (c) Do the same for ‘or’.

1.18 Give a Turing machine that takes as input a bit string, using the alphabet \( \{B, 0, 1\} \), and adds \( \text{01} \) at the back.

1.19 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.

1.20 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).

1.21 Produce a doubler, a Turing machine that computes \( f(x) = 2x \).

(a) Assume that the input and output is in unary. \textit{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 \( \Sigma = \{B, 0, 1\} \) and the input is represented in binary.

1.22 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).

1.23 Write a machine \( P \) with tape alphabet \( \Sigma \) that, in addition to blank \( B \) and stroke 1, also contains the comma ‘,’ character. Where \( \Sigma_0 = \Sigma - \{B\} \), 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, 
\( \phi_P(1111, 111, 1) = 11111111 \).

1.24 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 \( I = \hat{q} \hat{s}T_nq_n \) such that if a machine is in configuration \( \hat{C} \) then instruction \( I \) applies and \( \hat{C} \vdash C \)?

1.25 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 \( b_3b_2b_1b_0 \) you end with \( b_2b_1b_0b_3 \).

(C) Take as input two 4-bit strings and perform a bitwise AND.

✓ 1.26 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.

1.27 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 and one as a rejecting state, and the language decided by the machine is the set of strings that it accepts. 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.

1.28 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.

1.29 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.

1.30 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.

1.31 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?

1.32 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.

In the 1670’s Gottfried Wilhelm 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 symbols and could thereby determine the truth of scientific statements. To settle any dispute, Leibnitz wrote, scholars could just say, “Calculemus!”† 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,§ 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 and number-theoretic functions such as finding the remainder on division. They could not find any such function that the \( \lambda \)-calculus could not do. Church suggested to Gödel, the most prominent expert in the area, that the set of effective functions, the set of functions that are intuitively mechanically computable, which is not precisely given, is the same as the set of functions that are \( \lambda \)-computable, which is. But Gödel was unconvinced.

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

---

† Latin for “Let us calculate!”  
§ After producing his machine model, Turing became a PhD student of Church at Princeton.
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.†

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 researchers agree that it correctly explicates ‘computable on a discrete and deterministic mechanism’ and consent to work within that formalization, they are then free to proceed with reasoning mathematically about these systems, without fear of vagueness or ambiguity. 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.”

Nonetheless, despite not being the conclusion of a deductive system, Church’s Thesis is very widely accepted. We will outline the arguments 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.

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 a 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: all of the models of computation that were proposed, by all of the researchers then and since, yield the same set of computable functions. For instance, Turing showed that the set of functions computable with his machine model is equal to the set of functions computable with Church’s $\lambda$-calculus.

§ Some authors call this the Church-Turing Thesis. Here we figure that because Turing has the machine, we can give Church the thesis.
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 well suppose that it is the right answer. At the least,
convergence says that there is something natural and compelling about this set of
functions.

An argument not available to Turing, Church, Gödel, and others in the 1930’s,
since it depends on work done since, 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 done
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 just by taking a single symbol beyond the blank that covers the all
but finitely-many cells on the starting tape, that is, with $\Sigma = \{1, B\}$. Likewise,
restricting to write-only machines that cannot change marks 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
the unboundedly-many states machines given in Definition 1.4.

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 $q_5 \uparrow Rq_6$ and $q_5 \uparrow Lq_4$ in the same machine,
because they both begin with $q_5 \uparrow$. 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 posses 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.

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 studying an automobile antilock braking system then the Turing machine model accounts 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 but which passes statistical tests. Instead, well-established physics tells us these bits are truly random. Its relevance here is that Church’s Thesis only claims that Turing machines model the discrete and deterministic computations that we can do after we are given input bits from such a device.

**An empirical question?** Church’s Thesis posits that Turing machines can do any computation that is discrete and deterministic. That 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 these an ultimate in physically-possible machines? Did Turing, on that day, lying on that grassy river bank, intuit everything that experiments with reality would ever find to be possible?

For a taste of the conversation, we can prove that there is a case where the wave equation° has initial conditions that are computable (for the initial real numbers \( x_i \) there is a program that inputs \( i \in \mathbb{N} \) and outputs the \( i \)-th decimal place of \( x \)), but the unique solution is not computable. So does the wave tank modelled by this

---

Brain scientists also find Turing machines to be not the most suitable model. Note, though, that saying that an interrupt-driven brain model is a better fit is not the same as saying that the brain operations could not, in principle, be done using a Turing machine as the substrate. ° A partial differential equation that describes the propagation of waves.
equation compute something that Turing machines cannot? Stated for rhetorical effect: do the planets in their orbits compute a solution to the Three-Body Problem?

In this case we can object that an experimental apparatus is subject to 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. As yet no analysis of a wider notion of physically-possible mechanical computation in the non-discrete case has the support that Turing’s analysis has garnered in its more narrow domain. In particular, no one has yet produced a generally accepted example of a non-discrete mechanism that computes a function that no Turing machine computes.

We will not pursue this any 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, λ 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 for which we can prove that that no Turing machine can compute it then, with the thesis in mind, we will take 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 insight, after a short while you may well find that doing more machines does not give any more illumination. Worse, focusing too much on Turing machine details (or on the low-level details of any computing model) can obscure larger points. So if we can be clear and rigorous without actually having to handle a mass of detail then we will be delighted.

Church’s Thesis helps with this. Often when we want to show that something is computable by a Turing machine, we will first argue that it is intuitively computable and then cite Church’s Thesis to assert that it is therefore Turing machine computable. With that, our argument can proceed, “Let \( P \) be that machine . . .” without us ever having exhibited 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 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 a program, or sketch a program, doing that thing. For these 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 Each of these is frequently voiced on the interwebs as a counterargument to Church’s Thesis. Explain why each is bogus, said by clueless noobs. Plonk!
(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.

✓ 2.5 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.

2.6 List two benefits from adopting Church’s Thesis.

✓ 2.7 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.”

2.8 The computers we use every day are binary. Use Church’s Thesis to argue that if they were trinary, where instead of bits with two values they used trits with three, then they would compute exactly the same set of functions.

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

✓ 2.10 Suppose that \( f : \mathbb{N} \to \mathbb{N} \) is a total computable function. Use Church's Thesis to argue that this function is computable.

\[
h(n) = \begin{cases} 
0 & \text{if } n \text{ is in the range of } f \\
\uparrow & \text{otherwise}
\end{cases}
\]

2.11 Let \( f, g : \mathbb{N} \to \mathbb{N} \) be computable functions that may be either total or partial functions. Use Church’s Thesis to argue that this function is computable.

\[
h(n) = \begin{cases} 
1 & \text{if both } f(n) \downarrow \text{ and } g(n) \downarrow \\
\uparrow & \text{otherwise}
\end{cases}
\]

✓ 2.12 If you allow processes to take infinitely many steps then you can have all kinds of fun. Suppose that you have infinitely many dollars. Feeling flush you go to a bar. The Devil is there. 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, \ldots At each step he buys your lowest-numbered bill and pays you with higher-numbered bills. Thus, on the first transaction he 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 remaining exercises involve multitape Turing machines. A good way to define a \( k \)-tape machine is to start with Definition 1.4’s single tape transition function \( \Delta : Q \times \Sigma \to (\Sigma \cup \{L,R\}) \times Q \) and extend it to \( \Delta : Q \times \Sigma^k \to (\Sigma \cup \{L,R\})^k \times Q \). Thus, a typical four-tuple for a \( k = 2 \)-tape machine with alphabet \( \Sigma = \{0,1,B\} \) is \( q_4\langle1,B\rangle\langle0,L\rangle q_3 \). 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 \( q_3 \).

2.13 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 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.

2.14 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 3. Recursion

In the 1930's researchers other than Turing also saw the need to make precise the notion of mechanical computability. 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.†

This approach has a classical mathematics flavor. It lists initial functions that are intuitively mechanically computable, along with intuitively computable ways to combine existing functions, to make new functions from old. An example is that one effective initial function is successor \( S : \mathbb{N} \rightarrow \mathbb{N} \) described by \( S(x) = x + 1 \), and an effective combiner is function composition. Then the composition \( S \circ S \), the plus-two operation, is also intuitively mechanically computable.

We now introduce another combiner that is intuitively mechanically computable.

**Primitive recursion** Grade school students learn addition and multiplication as mildly complicated algorithms (“carry the one”). H Grassman produced a more elegant definition. Here is the formula for addition, plus : \( \mathbb{N}^2 \rightarrow \mathbb{N} \), which takes as given the successor map, \( S(n) = n + 1 \).

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

This is definition by recursion, since ‘plus’ recurs in its definition.‡

A common reaction on first seeing recursion is to wonder whether it is logically problematic — isn’t defining something in terms of itself a fallacy? Consider the expansion below. In it, plus(3, 2) is not defined in terms of itself, it is defined in terms of plus(3, 1). In turn, plus(3, 1)

† It also has the advantage of not needing the codings discussed in Remark 1.8 since it works directly with the functions. ‡ That is, recursion is discrete feedback.
is defined in terms of plus(3, 0), whose definition is clearly not a problem.

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

The key is to define the function on later values using only earlier ones.

One elegant thing about Grassmann’s approach is that it extends naturally to other operations. Multiplication has the same form.

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

3.1 Example The expansion of product(2, 3) reduces to a sum of three 2’s.

\[
\text{product}(2, 3) = \text{plus}(\text{product}(2, 2), 2) \\
= \text{plus}(\text{plus}(\text{product}(2, 1), 2), 2) \\
= \text{plus}(\text{plus}(\text{plus}(\text{product}(2, 0), 2), 2), 2) \\
= \text{plus}(\text{plus}(\text{plus}(0, 2), 2), 2)
\]

Exponentiation works the same way.

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

We are interested in Grassmann’s definition because it is effective; it translates immediately into a program. Here is code based on the definition of plus.† Starting with a successor operation,

\[
(\text{define (successor x)} \\
(\text{+ x 1})
\]

this code exactly fits the definition of plus.

\[
(\text{define (plus x y)} \\
(\text{let ((z (- y 1))}) \\
(\text{(if (= y 0) x)} \\
(\text{(successor (plus x z)))))
\]

(The \text{(let \ldots)} creates the local variable \text{z}.) The same is true for product and power.

\[
(\text{define (product x y)} \\
(\text{let ((z (- y 1))}) \\
(\text{(if (= y 0) 0)} \\
(\text{(plus (product x z) x)))))
\]

†Obviously Racket comes with an addition operator, as in \text{(+ 3 2)}, with a multiplication operator, as in \text{(* 3 2)}, and with exponentiation, \text{(expt 3 2)}.
Section 3. Recursion

3.2 Definition  A function $f$ is defined by the schema\(^\dagger\) of primitive recursion from the functions $g$ and $h$ if it has this form.

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

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 $x_0, \ldots, x_{k-1}$ as $\bar{x}$.

3.3 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 $h(w, x_0, z) = \text{product}(w, x_0)$.

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

3.4 Example  The predecessor function is like an inverse to successor. However, with our restriction to the natural numbers we can’t give a predecessor of zero, so instead consider $\text{pred}: \mathbb{N} \to \mathbb{N}$ described 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 $h(a, b) = b$.

3.5 Example  As with predecessor, we can’t do subtraction because we don’t have negative numbers so consider proper subtraction, denoted $x - y$, described by: if $x \geq y$ then $x - y$ equals $x - y$ and otherwise if $x - y$ equals 0. This definition of that function fits the primitive recursion scheme.

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

In the terms of Definition 3.2, $g(x_0) = x_0$ 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 $f$.

\(^\dagger\) A schema is an underlying organizational pattern or structure.
The computer code above make clear that primitive recursion fits into the plan of specifying combiners that preserve the property of effectiveness: if \( g \) and \( h \) are effective then so is \( f \).

### 3.6 Definition

The set of **primitive recursive functions** consists of those that can be derived from the initial operations of the zero function \( Z(\bar{x}) = Z(x_0, \ldots, x_{n-1}) = 0 \), the successor function \( S(\bar{x}) = x + 1 \), and the projection\(^\dagger\) functions \( I_i(\bar{x}) = x_i \), 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 \( f \circ g(\bar{x}) = f(g(\bar{x})) \). It also covers the case of simultaneous substitution, where from \( f(x_0, \ldots, x_n) \) and \( h_0(y_1, \ldots, y_{m_0}), \ldots, h_n(y_1, \ldots, y_{m_n}) \), we get \( f(h_0(y_{0,0}, \ldots, y_{0,m_0}), \ldots, h_n(y_{n,0}, \ldots, y_{n,m_n})) \), which is a function with \((m_0 + 1) + \cdots + (m_n + 1)\)-many inputs.

Besides addition and proper subtraction, we commonly use many other primitive recursive functions such as finding remainders and testing for less-than. See the exercises for these. The list is so extensive that a person could wonder whether every mechanically computed function is primitive recursive. The next section shows that the answer is no, that there are intuitively mechanically computable functions that are not primitive recursive.

### I.3 Exercises

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

3.8 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?

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

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

(a) Find \( f(0) \) and \( f(1) \). (b) Try to find \( f(2) \).

3.10 Consider this function.

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

(a) Find \( F(0), \ldots, F(10) \).

\(^\dagger\)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 use the letter \( I \).
(b) Show that $F$ is primitive recursive by describing it in the form given in Definition 3.2, 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.

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

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

✓ 3.13 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 $t(0), \ldots t(10)$.

(b) Show that $t$ is primitive recursive by describing it in the form given in Definition 3.2, 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.

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

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

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

(b) Verify that $t$ is primitive recursive by putting it in the form given in Definition 3.2, 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.

3.15 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 $d(0), \ldots d(5)$.

(b) Verify that $d$ is primitive recursive by putting it in the form given in Definition 3.2, 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 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 Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, 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 Bramahns alike will crumble into dust, and with a thunderclap the world will vanish. It gives the recurrence below because to move a pile of discs you first move to one side all but the bottom, which takes \( H(n-1) \) steps, then move that bottom one, which takes one step, then re-move the other disks into place on top of it, taking another \( H(n-1) \) steps.

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

(a) Compute the values for \( n = 1, \ldots, 10 \).
(b) Verify that \( H \) is primitive recursive by putting it in the form given in Definition 3.2, 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 Define the factorial function \( \text{fact}(y) = y \cdot (y-1) \cdots 1 \) by primitive recursion, using product and a constant function.

✓ 3.18 Recall that the greatest common divisor of two positive integers is the largest integer that divides them both. We can compute the greatest common divisor using Euclid’s recursion

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

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

(a) \( \gcd(28, 12) \)  
(b) \( \gcd(104, 20) \)  
(c) \( \gcd(309, 25) \)

✓ 3.19 Many familiar mathematical operations are primitive recursive. Show that these functions and predicates are primitive recursive. (A predicate is a truth-valued function and we take an output of 1 to mean ‘true’ while 0 is ‘false’.)

For each you may use functions already shown to be primitive recursive in the subsection body, or in a prior item.

(a) Constant function: \( C_k(\vec{x}) = C_k(x_0, \ldots, x_{n-1}) = k \) for any \( k \in \mathbb{N} \).
(b) Maximum and minimum of a set having two numbers: \( \max(\{x, y\}) \) and \( \min(\{x, y\}) \).
(c) Absolute difference function: \( \text{absdiff}(x, y) = |x-y| \), the absolute value of \( x - y \).
(D) Sign predicate: \( \text{sign}(y) \), which gives 1 if \( y \) is greater than zero and 0 otherwise.

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

(F) Less-than predicate: \( \text{lessthan}(x, y) = 1 \) if \( x \) is less than \( y \), and 0 otherwise. (The greater-than predicate is similar.)

3.20 Show that each of these is a primitive recursive function. You can use functions from this section already shown to be primitive recursive, or functions from the prior exercises.

(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.

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

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

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

3.21 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.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rem}(a, 3) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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.

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

(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 3.19 and Exercise 3.20. (Compared with Definition 3.2, 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.

3.22 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.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>( \text{div}(a, 3) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 3.19 and Exercise 3.20. (Compared with Definition 3.2, here the two arguments are switched, which is only a difference of appearance.)

(d) Show that \( \text{div}(a, b) \) is primitive recursive.

3.23 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, \( S_g(y) = \sum_{0 \leq i < y} g(i) = g(0) + g(1) + \cdots + g(y - 1) \) (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 \( p(\bar{x}, i) \) be a predicate. Then the minimization operator \( M(\bar{x}, i) \), typically written \( \mu^m i[p(\bar{x}, i)] \), returns the smallest \( i \leq m \) such that \( p(\bar{x}, i) = 0 \), or else returns \( m \). Hint: Consider the bounded sum of the bounded products of the predicates.

3.24 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 \( p(\bar{x}, i) \) is a predicate. Then \( U(\bar{x}, m) \), typically written \( \forall i \leq m p(\bar{x}, i) \), has value 1 if \( p(\bar{x}, 0) = 1, \ldots, p(\bar{x}, m) = 1 \) and value 0 otherwise. (The point of writing the functional expression \( U(\bar{x}, m) \) is to emphasize the required uniformity. Stating one formula for the \( m = 1 \) case, \( p(\bar{x}, 0) \cdot p(\bar{x}, 1) \), and another for the \( m = 2 \) case, \( p(\bar{x}, 0) \cdot p(\bar{x}, 1) \cdot p(\bar{x}, 2) \), etc., is the best we can do. We can get a single derivation, that follows the rules in Definition 3.6, and that works for all \( m \).)

(b) Bounded existential quantification: let \( m \in \mathbb{N} \) and let \( p(\bar{x}, i) \) be a predicate. Then \( A(\bar{x}, m) \), typically written \( \exists i \leq m p(\bar{x}, i) \), has value 1 if \( p(\bar{x}, 0) = 0, \ldots, p(\bar{x}, m) = 0 \) 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.

3.25 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.

3.26 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 3.19 and Exercise 3.23.)

(B) Show that \( F \) is primitive recursive. You may use the results from earlier, including Exercise 3.19, Exercise 3.20, Exercise 3.23, and Exercise 3.24.

3.27 Let \( C(x, y) = 0 + 1 + 2 + \cdots + (x + y) + y \).

(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 3.19, Exercise 3.19, Exercise 3.24, and Exercise 3.24.

3.28 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 3.19 and Exercise 3.23.

✓ 3.29 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 $x \in \{0, \ldots, 101\}$? 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 3.19.

3.30 Show that every primitive recursive function is total.

3.31 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 3.2 then they are equal, so the same inputs they will yield the same outputs.

Primitive recursion |)

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’.†

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.

\[
\begin{align*}
x + y &= S(S(\cdots S(x))) \\
x \cdot y &= x + x + \cdots + x \\
x^y &= x \cdot x \cdots \cdot x
\end{align*}
\]

This is a compelling pattern.

†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 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, $\mathcal{H}_0 = S$.

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

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

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

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 $\mathcal{H} : \mathbb{N}^3 \to \mathbb{N}$.

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

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

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

We prove the level 1 statement $\mathcal{H}_1(x, y) = x + y$ by induction on $y$. For the $y = 0$ base step, the definition is that $\mathcal{H}(1, x, 0) = x$, which equals $x + 0 = x + y$. For the inductive step, assume that the statement holds for $y = 0, \ldots, y = k$ 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, \(H_3(4, 4)\) is much greater than the number of elementary particles in the universe.

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

\[
\begin{align*}
\text{(define } (H n x y) & \text{)} \\
\text{(cond} & \text{)} \\
[ (= n 0) (+ y 1)] & \\
[ (= n 1) (= y 0)] & x \\
[ (= n 2) (= y 0)] & 0 \\
[ (> n 2) (= y 0)] & 1 \\
[ \text{else } (H (- n 1) x (H n x (- y 1)))]] & \\
\end{align*}
\]

However, hyperoperations’s recursion line

\[
H(n, x, y) = H(n - 1, x, H(n, x, y - 1))
\]

does not fit the form of primitive recursion.

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

The problem is not that the arguments are in a different order; that is cosmetic. The reason \(H\) does not work as \(h\) is that the definition of primitive recursive function, Definition 3.2, 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\(^\dagger\) proved that there isn’t, that \(H\) is not primitive recursive. The proof is a detour for us so it is in an Extra Section but in summary: \(H\) 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 \(H(n, x, y) > f(n, x, y)\). This proof is about uniformity. At every level, the function \(H_n\) is primitive recursive but no primitive recursive function encompasses all levels at once — there is no single, uniform, primitive recursive way to compute them all.

4.4 **Theorem** The hyperoperation \(H\) 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

\(^\dagger\)We have seen Ackermann already, as one of the people who stated the Entscheidungsproblem. Functions having the same recursion as \(H\) are Ackermann function.
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 that in the Church’s Thesis discussion we called coverage. To cover all mechanically computable functions under a recursive rubric we need to expand from primitive recursive functions to a larger set.

\[\mu\text{ recursion}\] The right direction is hinted at in Exercise 3.23 and Exercise 3.24. Primitive recursion does bounded operations. We can prove that an operation is primitive recursive if and only if it is bounded by showing 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 unbounded operations.

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 unbounded minimization or \(\mu\)-recursion, written \(f(\bar{x}) = \mu y [g(\bar{x}, y) = 0]\), if \(f(\bar{x})\) is the the least number \(y\) such that \(g(\bar{x}, y) = 0\).

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

<table>
<thead>
<tr>
<th>(y)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p(y))</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
<td>61</td>
<td>71</td>
<td>83</td>
<td>97</td>
<td>113</td>
<td>131</td>
</tr>
</tbody>
</table>

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

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

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

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 (add1 c))]))
  (prime-helper n 2))
```

\[^\dagger\] Recall that \(\bar{x}\) abbreviates \(x_0, \ldots x_{n-1}\).
and a way to compute the output of $y \mapsto x_2y^2 + x_1y + x_0$.

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

Now, this is $g$.

```
(define (f-sub-g x0 x1 x2)
 (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 $\mathcal{Z}(\bar{x}) = 0$, the **successor** function $\mathcal{S}(x) = x + 1$, and the **projection** functions $\mathcal{I}_i(x_0, \ldots, x_i \ldots x_{k-1}) = x_i$ 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.

### Exercises

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

1. **Exercise** 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 $\mathcal{H}_4(2, 4)$.
2. **Exercise** 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 $H_4(3, 3)$ seconds?

4.12 What is the ratio $H_3(3, 3)/H_2(2, 2)$?

4.13 Finish the proof of Lemma 4.2 by verifying that $H_2(x, y) = x \cdot y$ and $H_3(x, y) = x^y$.

4.14 This variant of $H$ is often labelled “the” Ackermann function.

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

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

4.15 Prove that the computation of $H(n, x, y)$ 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 $g(x, y) = x + y$ and let $f(x) = \mu y \left[ g(x, y) = 100 \right]$. For each, find the value or say that it is not defined. (A) $f(0)$ (b) $f(1)$ (c) $f(50)$ (d) $f(100)$ (e) $f(101)$ Give an expression for $f$ that does not include $\mu$-recursion.

4.18 Let $g(x, y) = [(x + 1)/(y + 1) - 1]$ 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.

4.19 Consider the Turing machine $P = \{ q_0B1q_1, q_01Rq_0, q_1BRq_2, q_11Lq_1 \}$. Define $g(x, y) = 0$ if the machine $P$, 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 $f(x) = \mu y \left[ g(x, y) = 0 \right]$ for $x < 6$.

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

4.21 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 $f(x) = \mu y \left[ g(x, y) = 0 \right]$. Find: (A) $f(0)$ (b) $f(1)$ (c) $f(2)$ (d) $f(x)$. 


4.22 Define

\[ h(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n + 1 & \text{else} \end{cases} \]

and 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 \( C(n) = \mu k \left[ H(n, k) = 1 \right] \).

(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. It has three design goals. The main one is to track closely the description of the action of a Turing machine on section 1. 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.

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

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

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 this machine’s file description.

The data structure for a Turing machine is the simplest one, a list of instructions. For the instructions, the program converts each of the above six lines into a list with four members, a number, two characters, and a number. Thus, a Turing machine is stored as a list of lists. The above machine is this (the line break is there only to make it fit in the margins).

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

After some convenience constants
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 Δ 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)]) ;; X is arbitrary placeholder char
    (if (not inst)
      (list #\X HALT-STATE) ;; X is arbitrary placeholder char
      (list (third inst) (fourth inst)))))

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

Turing machine 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
(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)]
         [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)]]
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)])
    (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)])
    (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.

;; execute Run a turing machine step-by-step until it halts
(define (execute tm initial-config)
  (define (execute-helper config s)
    (if (= (get-current-state config) HALT-STATE)
        (fprintf (current-output-port) "step ~s: HALT\n" s)
        (begin
          (fprintf (current-output-port) "step ~s: ~a\n" s
            (configuration->string config))
          (execute-helper (step config tm) (add1 s))))
  (execute-helper initial-config 0))
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.

```scheme
;;; 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))]]
    "\[state-string state-number right-tape left-tape current\]
    "))
```

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.

```scheme
(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)
  (let* ([instruction (string-split (string-trim s))]
         [current-state (current-state-string->number (first instruction))]
         [current-symbol (current-symbol-string->char (second instruction))]
         [action (action-symbol-string->char (third instruction))]
         [next-state (next-state-string->number (fourth instruction))])
    (list current-state
      current-symbol
      action
      next-state)))
```

And, there is a bit more code for getting the file name from the command line, etc., that does not bear at all 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: 11*1*B
step 5: q1: 11*B*B
step 6: q2: 1*1*BB
step 7: q2: *B*11BB
step 8: q2: *B*11BB
step 9: q3: B*1*1BB
step 10: HALT
```

The output is crude but good enough for small scale experiments.

**I.A Exercises**
A.1 Run the simulator on the predecessor machine $\mathcal{P}_{\text{pred}}$ starting with five 1’s. Also run it on an empty tape.

A.2 Run the simulator on Example 1.2’s $\mathcal{P}_{\text{add}}$ to add 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 $\Sigma = \{\emptyset, 0, 1, B\}$, 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.

1. 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.
2. 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.
3. 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.

1. Modify it so that, given $k \in \mathbb{N}$, if the Turing machine hasn’t halted after $k$ steps then the simulator stops.
2. 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 that starts by considering general physical computing devices and ends at transition tables. What about the converse; given a finite transition table, is there a physical implementation with that behavior?

Put another way, in programming languages there are built-in 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. A proposition is a statement
that has a Boolean value, either \( T \) or \( F \). For instance, ‘7 is odd’ and ‘8 is prime’ 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 often combine propositions. We might cojoin two by saying, ‘5 is prime and 7 is prime’, or we might negate with ‘it is not the case that 8 is prime’.

These truth tables define the behavior of the logical operators not (sometimes called negation), and (or conjunction), and or (or disjunction). The first is a unary, or one input, operator while the others are binary operators. The tables write \( F \) as 0 and \( T \) as 1, as is the convention in electrical engineering. In an electronic computer, these would stand for different voltage levels. For both tables, inputs are on the left while outputs are on the right.

\[
\begin{array}{c|c}
\text{not } P & P \\hline
0 & 1 \\
1 & 0 \\
\end{array}
\quad
\begin{array}{c|c|c|c}
P & Q & P \land Q & P \lor Q \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

Thus, where ‘7 is odd’ is \( P \), and ‘8 is prime’ is \( Q \), get the value of the conjunction ‘7 is odd and 8 is prime’ from the third line of the right-hand table: 0. The value of the disjunction ‘7 is odd or 8 is prime’ is 1.

Truth tables help us work out the behavior of complex propositional logic statements, by building them up from their components. This shows the input/output behavior of the statement \((P \lor Q) \land \neg(P \lor (R \land Q))\).

\[
\begin{array}{c|c|c|c|c|c|c}
P & Q & R & P \lor Q & R \land Q & P \lor (R \land Q) & \neg(P \lor (R \land Q)) & \text{statement} \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

There are operators other than ‘not’, ‘and’, and ‘or’ but an advantage of the set of these three is that with them we can reverse the activity of the prior paragraph: we can go from a table to a statement with that table. That is, we can go from a specified input-output behavior to a statement with that behavior.

Below are two examples. To make a statement with the behavior shown in the table on the left, focus on the row with output 1. It is the row where \( P \) is false and \( Q \) is false. Therefore the statement \( \neg P \land \neg Q \) makes this row take on value 1 and every other row take on value 0.
Next consider the table on the right and again focus on the rows with 1’s. Target the second row with \( \neg P \land \neg Q \land R \). For the third row use \( \neg P \land Q \land \neg R \) and for the fifth row use \( P \land \neg Q \land \neg R \). To finish, put these parts together with \( \lor \)'s to get the overall statement.

\[
(\neg P \land \neg Q \land R) \lor (\neg P \land Q \land \neg R) \lor (P \land \neg Q \land \neg R)
\]  

(\text{\textasteriskcentered})

Thus, we can produce statements with any desired behavior. Statements of this form, clauses connected by \( \lor \)'s, where inside each clause the statement is built from \( \land \)'s, are in **disjunctive normal form**. (Also commonly used is **conjunctive normal form**, where statements consist of clauses connected by \( \land \)'s and each clause uses only \( \lor \)'s as binary connectives.)

Now we translate those statements 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 statement to systematically design logic circuits was made by Claude 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

\[
\text{[A schematic of a circuit that implements statement (\textasteriskcentered}), given below, shows three input signals on the three wires at left. For instance, to implement the first clause, the top \text{\textasteriskcentered} gate is fed the not P, the not Q, and the R signals. The second and third clauses are implemented in the other two \text{\textasteriskcentered} gates. Then the output of the \text{\textasteriskcentered} gates goes through the \text{\textasteriskcentered} gate.} \]
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 energy 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 we shall see provides 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 through the D-S line. (Because it is sometimes open and sometimes closed it is depicted as a switch, although internally it has no moving parts.) This transistor is manufactured such that an input voltage $V_{in}$ of 5 volts will trigger this event.

To verify that this circuit inverts the signal, assume first that $V_{in} = 0$. Then there is 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 truth table for each of these propositions. (a) \((P \land Q) \land R\) (b) \(P \land (Q \land R)\) (c) \(P \land (Q \lor R)\) (d) \((P \land Q) \lor (P \land R)\)

B.2 Make a truth table for these. (a) \(\neg(P \lor Q)\) (b) \(\neg P \land \neg Q\) (c) \(\neg(P \land Q)\) (d) \(\neg P \lor \neg Q\)

B.3 (a) Make a three-input table for the behavior: the output is 1 if a majority of the inputs are 1’s. (b) Draw the circuit.

B.4 For the table below, construct a disjunctive normal form propositional logic statement and use that to make a circuit.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

B.5 For the tables below, construct a disjunctive normal form propositional logic statement and use that to make a circuit. (a) the table on the left, (b) the one on the right.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 \(T\) if \(P \neq Q\), and is \(F\) otherwise. Make a truth table, from it construct a disjunctive normal form propositional logic statement, and use that to make a circuit.

B.7 Construct a circuit with the behavior specified in the tables below: (a) the table on the left, (b) the one on the right.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
B.8 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 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 iron oxide, rust. 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.)

His thinking, along with a suggestion from S Ulam, who was studying crystal growth, led him to use a cell-based approach. So von Neumann laid out some computational devices in a grid of interconnections, making a cellular automaton.

Interest in cellular automata greatly increased when J Conway invented one called Life. It was featured in an October 1970 magazine column of Scientific American. The rules of the game are simple enough that a person could immediately take out a pencil and start experimenting. Lots of people did. When personal computers came out, Life became one of the earliest computer crazes, since it is easy for a beginner to program.

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 neighbours will again be live at the next generation but any other live cell dies, (ii) a dead cell with exactly three live neighbours 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 inpenetrable complexity.

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

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

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

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

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

![Generation 0](image1)
![Generation 1](image2)

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

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

![Generation 0](image3)
![Generation 1](image4)

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

![Generation 0](image5)
![Generation 1](image6)

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.
Chapter I. Mechanical Computation

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 D. Ackermann’s function is not primitive recursive

We have see that the hyperoperation, whose definition is repeated below, 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 quoted a result that \( \mathcal{H} \), while intuitively mechanically computable, is not primitive recursive. The details of the proof are awkward. For technical convenience we will instead show that a closely related function, which is also intuitively mechanically computable, 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.

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

Any function based on the recursion in the bottom line is called an Ackermann function.\(^\dagger\) We will prove that \( \mathcal{A} \) is not primitive recursive.

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

<table>
<thead>
<tr>
<th>( k )</th>
<th>( y = 0 )</th>
<th>( y = 1 )</th>
<th>( y = 2 )</th>
<th>( y = 3 )</th>
<th>( y = 4 )</th>
<th>( y = 5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>125</td>
<td>253</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>65</td>
<td>533</td>
<td>\ldots</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

\(^\dagger\)There are many different Ackermann functions in the literature. A common one is the function of one variable \( \mathcal{A}(k, k) \).
D.1 Lemma (a) \( A(k, y) > y \)
(b) \( A(k, y + 1) > A(k, y) \), and in general if \( \hat{y} > y \) then \( A(k, \hat{y}) > A(k, y) \)
(c) \( A(k + 1, y) \geq A(k, y + 1) \)
(d) \( A(k, y) > k \)
(e) \( A(k + 1, y) > A(k, y) \) and in general if \( \hat{k} > k \) then \( A(\hat{k}, y) > A(k, y) \)
(f) \( A(k + 2, y) > A(k, 2y) \)

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 \( k \).

\[ \forall k \forall y \left[ A(k, y) > y \right] \tag{*} \]

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

We must verify this statement,

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

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

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

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

D.2 Definition A function \( s \) is level \( k \), where \( k \in \mathbb{N} \), if \( A(k, \max(\vec{x})) > s(\vec{x}) \) 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 \( g_0, \ldots, g_{m-1}, h \) is level \( k \), and if the function \( f \) is obtained by composition as \( f(\vec{x}) = h(g_0(\vec{x}), \ldots, g_{m-1}(\vec{x})) \), then \( f \) is level \( k + 2 \).

Proof Apply Lemma D.1’s item c, and then the definition of \( A \).

\[ A(k + 2, \max(\vec{x})) \geq A(k + 1, \max(\vec{x}) + 1) = A(k, A(k + 1, \max(\vec{x}))) \tag{*} \]

Focusing on the second argument of the right-hand expression, use Lemma D.1.e and the assumption that each function \( g_0, \ldots, g_{m-1} \) is level \( k \) to get that for each index \( i \in \{1, \ldots, m - 1\} \) we have \( A(k + 1, \max(\vec{x})) > A(k, \max(\vec{x})) > g_i(\vec{x}) \). Thus \( A(k + 1, \max(\vec{x})) > \max(\{g_1(\vec{x}), \ldots, g_{m-1}(\vec{x})\}) \).
Lemma D.4 says that $\mathcal{A}$ is monotone in the second argument, so returning to equation (*)& returning to equation (**) and swapping out $\mathcal{A}(k + 1, \max(\bar{x}))$ gives the first inequality here

\[ \mathcal{A}(k + 2, \max(\bar{x})) > \mathcal{A}(k, \max(\{ g_1(\bar{x}), \ldots, g_{m-1}(\bar{x}) \})) \]
\[ > h(g_0(\bar{x}), \ldots, g_{m-1}(\bar{x})) = f(\bar{x}) \]

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

D.4Lemma 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

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

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

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

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

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

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

\[ \mathcal{A}(k + 1, \max(\bar{x}) + z) > \max(\{ f(\bar{x}, z), x_0, \ldots, x_{n-1}, z \}) \]

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

\[ \mathcal{A}(k + 1, \max(\bar{x}) + z + 1) = \mathcal{A}(k, \mathcal{A}(k + 1, \max(\bar{x}) + z)) \]
\[ > \mathcal{A}(k, \max(\{ f(\bar{x}, z), x_0, \ldots, x_{n-1}, z \})) \]
\[ > h(f(\bar{x}, z), \bar{x}, z) = f(\bar{x}, z + 1) \]

That finishes the inductive verification of statement (*).

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

\[ \mathcal{A}(k + 3, \max(\{ x_0, \ldots, y \})) > \mathcal{A}(k + 1, 2 \cdot \max(\{ x_0, \ldots, y \})) \]
\[ \geq \mathcal{A}(k + 1, \max(\bar{x}) + y) \]

(the latter holds because $2 \cdot \max(\bar{x}, y) \geq \max(\bar{x}) + y$ and because of Lemma D.1.b). In turn, by the first part of this proof, that is greater than $f(\bar{x}, y)$. □
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 3.6 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 \( A(0, x) = x + 1 > 0 \). The successor function \( S(x) = x + 1 \) is level 1 since \( A(1, x) > A(0, x) = x + 1 \) by Lemma D.1.e. Each projection function \( I_i(x_0, \ldots, x_i, \ldots, x_n-1) = x_i \) is level 0 since \( A(0, \max(x)) = \max(x) + 1 \) is greater than \( \max(x) \), which is greater than or equal to \( x_i \).

D.6 **Corollary** The function \( A \) is not primitive recursive.

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

I.D **Exercises**

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

D.8 Show that for any \( k, y \) the evaluation of \( A(k, y) \) 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) \( A(0, y) = y + 1 \) (B) \( A(1, y) = 2 + (y + 3) - 3 \) (C) \( A(2, y) = 2 \cdot (y+3) - 3 \) (D) \( A(3, y) = 2y+3-3 \) (E) \( A(4, y) = 2 \uparrow\uparrow (n+3) - 3 \)

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 \( A \) are primitive recursive. In fact, all levels are. But how does that work: all the parts of \( A \) are primitive recursive but as a whole it is not?

D.12 \( A(k + 1, x) = A(k, A(k, \ldots A(k, 1) \ldots)) \) where there are \( x + 1 \)-many \( A \)’s.

D.13 Prove that \( A(k, y) = H(k, 2, n + 3) - 3 \). Conclude that \( H \) 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, and that is easy.
The most familiar looping constructs are \texttt{for} and \texttt{while}. The difference is that a \texttt{while} loop can go an unbounded number of times, but in a \texttt{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 \(r_0, r_1, \ldots\), which hold natural numbers.

A \texttt{LOOP program} is a sequence of instructions, 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) \texttt{loop } \ldots \texttt{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 \texttt{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).

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

Very important: in \texttt{loop } \ldots \texttt{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 \(r_0\) will be twice what it was at the start.

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

We want to interpret \texttt{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 \( f(x, y) = x - y \).

\[
\text{loop } r1 \\
r0 = 0 \\
\text{loop } r0 \\
r1 = r0 \\
r0 = r0 + 1 \\
\text{end} \\
\text{end}
\]

That is, if we load \( x \) into \( r0 \) and \( y \) into \( r1 \), and run the above routine, then the output \( x - y \) 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 \( Z(x) = 0 \) is computed by the LOOP program whose single line is \( r0 = 0 \). The successor function \( S(x) = x + 1 \) is computed by the one-line \( r0 = r0 + 1 \). Projection \( I_i(x_0, ..., x_i, ..., x_{n-1}) = x_i \) is computed by \( r0 = r_i \).

The composition of two functions is easy. Suppose that \( \phi(x_0, ..., x_n) \) and \( f(y_0, ..., y_m) \) are computed by LOOP programs \( P_\phi \) 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 \( P_\phi \) 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

\[
f(x_0, ..., x_n), \quad h_0(y_{0,0}, ..., y_{0,m_0}), \quad ... \quad \text{and} \quad h_n(y_{n,0}, ..., y_{n,m_n})
\]

and produces \( f(h_0(y_{0,0}, ..., y_{0,m_0}), ..., h_n(y_{n,0}, ..., y_{n,m_n})) \). The issue is that were we to load the sequence of inputs \( y_{0,0}, ... \) into \( r0, ... \) 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 \( P_f, P_{h_0}, \ldots, P_{h_n} \) 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 \( (m_0 + 1) + \cdots (m_n + 1) \)). 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.

\[
f(x_0, ..., x_{k-1}, y) = \begin{cases} 
g(x_0, ..., x_{k-1}) & \text{if } y = 0 \\
h(f(x_0, ..., x_{k-1}, z), x_0, ..., x_{k-1}, z) & \text{if } y = S(z) \
\end{cases}
\]
Suppose that we have LOOP programs $P_d$ 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 $P_d$. 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 $f(x_0, ... x_{k-1}, t + 1) = h(f(x_0, ... x_{k-1}, t), x_0, ... x_{k-1}, t)$ 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.

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 $r_0, r_1, \ldots$, that hold natural numbers. We keep track of them in a list of pairs.

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.
Chapter I. Mechanical Computation

; 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 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-regs data)
  (intr-body progr)
  (get-reg-value (make-reg-name 0)))

; init-regs Initialize the registers r0, r1, r2, .. to the values in data
(define (init-regs data)
  (define (init-regs-helper i data)
    (if (null? data)
      '()
      (begin
       (set-reg-value! (make-reg-name i) (car data))
       (init-regs-helper (+ i 1) (cdr 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 r0). Similarly, $r_0 = r_0 + 1$ becomes (incr r0) and $r_0 = r_1$ becomes (copy r0 r1). Finally, loop r0 becomes (loop r0 (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 Scheme 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 Scheme 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 $r_0 = r_0 + 2$.

E.5  Modify the interpreter to allow statements like $r_0 = 1$.

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.

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & \cdots \\
0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & 81 & 100 & 121 & \cdots \\
\end{array}
\]

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. So there really is a lot in the background. Robert Williams and the Hubble Deep Field Team (STScI) and NASA. (Also see the Deep Field movie.)
As with Galileo’s Paradox, the puzzle is that we might think of the set of points on the circumference of a larger circle as being a bigger set. But the right idea is that the two sets have 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.

Recall that a correspondence is a function that is both one-to-one and onto. A function \( f : D \to C \) is one-to-one if \( f(x_0) = f(x_1) \) implies that \( x_0 = x_1 \) for \( x_0, x_1 \in D \). It is onto if for any \( y \in C \) there is an \( x \in D \) such that \( y = f(x) \). 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.

**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, and if such a function is not one-to-one then its domain has more elements than its range. Therefore 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** ??.
1.6 **Lemma**  The relation between two sets $S_0$ and $S_1$ of ‘there is a correspondence $f: S_0 \rightarrow S_1$’ is an equivalence relation.

**Proof**  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. 

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

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, Galelio’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 $\mathbb{N}$ by the definition of $S$, and so $y = f(n)$.

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, $\mathbb{N}^+ = \{1, 2, \ldots \}$ has the same cardinality as $\mathbb{N}$. A correspondence is $f: \mathbb{N} \rightarrow \mathbb{N}^+$ given by $n \mapsto n + 1$.

Comparing the sizes of sets, even infinite 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 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, \ldots, n\} for some \(n \in \mathbb{N}\). Otherwise the set is infinite.

In the Theory of Computation, the most important infinite set is \(\mathbb{N}\).

1.12 **Definition** A set with the same cardinality as the natural numbers is countably infinite. If that set is either finite or countably infinite then it is countable. A function whose domain is the natural numbers enumerates, or is an enumeration of, its range.

We say 'enumeration' with the idea that \(f: \mathbb{N} \to S\) lists the range set: first \(f(0)\), then \(f(1)\), etc. The listing may have repeats, so perhaps for some \(n_0 \neq n_1\) we have \(f(n_0) = f(n_1)\). As always, we are most interested when the function is computable.

1.13 **Example** The set of multiples of three, \(3\mathbb{N} = \{3k \mid k \in \mathbb{N}\}\), is countable. The natural map \(g: \mathbb{N} \to 3\mathbb{N}\) is \(g(n) = 3n\). Of course, this function is effective.

1.14 **Example** The set \(\mathbb{N} - \{2, 5\} = \{0, 1, 3, 4, 6, 7, \ldots\}\) is countable. The function below, both defined and illustrated with a table, closes up the gaps.

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

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 \(p: \mathbb{N} \to P\) 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 \(\Sigma = \{a, \ldots, z\}\). 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.

<table>
<thead>
<tr>
<th>(n \in \mathbb{N})</th>
<th>(0)</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(n) \in \Sigma^*)</td>
<td>(\varepsilon)</td>
<td>a</td>
<td>b</td>
<td>...</td>
<td>z</td>
<td>aa</td>
<td>ab</td>
<td>...</td>
</tr>
</tbody>
</table>

(The first entry is the empty string, \(\varepsilon = \"\"\).) This correspondence is also effective.

1.17 **Example** The set of integers \(\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}\) is countable. The natural correspondence, alternating between positive and negative numbers, is also effective.

<table>
<thead>
<tr>
<th>(n \in \mathbb{N})</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(n) \in \mathbb{Z})</td>
<td>0</td>
<td>+1</td>
<td>-1</td>
<td>+2</td>
<td>-2</td>
<td>+3</td>
<td>-3</td>
<td>...</td>
</tr>
</tbody>
</table>
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 Galelio’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 \((0..\infty)\).

![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).)](image)

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

## II.1 Exercises

### ✓ 1.19 Verify Example 1.13, that the function \( g: \mathbb{N} \to \{ 3k \mid k \in \mathbb{N} \} \) 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 \( f, g: \mathbb{Z} \to \mathbb{Z} \) be \( f(x) = 2x \) and \( g(x) = 2x - 1 \). Give a proof or a counterexample for each. (A) If \( f \) one-to-one? Is it onto? (B) If \( g \) one-to-one? Onto? (c) Are \( f \) and \( g \) inverse to each other?

### ✓ 1.22 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) \( f: \mathbb{N} \to \mathbb{N} \) given by \( f(n) = n + 1 \)  (B) \( f: \mathbb{Z} \to \mathbb{Z} \) given by \( f(n) = n + 1 \)  (c) \( f: \mathbb{N} \to \mathbb{N} \) given
by \( f(n) = 2n \) (d) \( f : \mathbb{Z} \to \mathbb{Z} \) given by \( f(n) = 2n \) (e) \( f : \mathbb{Z} \to \mathbb{N} \) given by \( f(n) = |n| \)

1.23 Decide if each is a correspondence (you must also verify): (A) \( f : \mathbb{Q} \to \mathbb{Q} \) given by \( f(n) = n + 3 \) (B) \( f : \mathbb{Z} \to \mathbb{Q} \) given by \( f(n) = n + 3 \) (C) \( f : \mathbb{Q} \to \mathbb{N} \) given by \( f(a/b) = |a \cdot b| \). Hint: this is a trick question.

1.24 Decide if each set finite or infinite and justify your answer. (A) \( \{1, 2, 3\} \) (B) \( \{0, 1, 4, 9, 16, \ldots\} \) (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 \( \{1, 2, 3, 4, \ldots\} \) and \( \{7, 10, 13, 16, \ldots\} \).

✓ 1.29 Consider the set of characters \( C = \{\emptyset, 1, \ldots, 9\} \) and the set of integers \( A = \{48, 49, \ldots, 57\} \).

(A) Produce a correspondence \( f : C \to A \).

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

✓ 1.30 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 \( f : \mathbb{R} \to (0..\infty) \) 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 \((3..5)\) and \((-1..10)\) 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) \(\{0, 1, ..., 99\}\), \(\{m \in \mathbb{N} \mid m^2 < 10000\}\) (c) \(\{0, 1, 3, 6, 10, 15, ...\}\), \(\mathbb{N}\)

✓ 1.38 Produce a correspondence between each pair of open intervals of reals. (a) \((0..1), (0..2)\) (b) \((0..1), (a..b)\) for real numbers \(a < b\) (c) \((0..\infty), (a..\infty)\) for the real number \(a\) (d) This shows a correspondence \(x \mapsto f(x)\) between a finite interval of reals and an infinite one, \(f : (0..1) \rightarrow (0..\infty)\).

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

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

1.40 Let \(\mathcal{B}\) be the set of characters from which bit strings are made, \(\mathcal{B} = \{0, 1\}\). (a) Let \(B\) be the set of finite bit strings where the initial bit is 1. Show that \(B\) is countable.

(b) Let \(\mathcal{B}^*\) be the set of finite bit strings, without the restriction on the initial bit. Show that it also is countable. \textit{Hint:} use the prior item.

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

1.42 Example 1.9 restates Aristotle’s Paradox as: the intervals \(I_0 = [0..2\pi r_0)\) and \(I_1 = [0..2\pi r_1)\) have the same cardinality, for \(r_0, r_1 \in \mathbb{R}^+\). (a) Verify it by checking that \(g : I_0 \rightarrow I_1\) given by \(g(x) = x \cdot (r_1/r_0)\) is a correspondence.

(b) Show that where \(a < b\), the cardinality of \([0..1)\) equals that of \([a..b)\).

(c) Generalize by showing that where \(a < b\) and \(c < d\), the real intervals \([a..b)\) and \([c..d)\) have the same cardinality.

1.43 Suppose that \(D \subseteq \mathbb{R}\). A function \(f : D \rightarrow \mathbb{R}\) is \textit{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 common notion that “the whole is greater than the part,” because
they name sets that are equinumerous with a proper subset. Each of these pairs has a set and a proper subset. Show that the sets in each pair have 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 if we take away a finite number of elements from the set \( \mathbb{N} \) then we do not change the cardinality. Prove it: prove that if \( S \) is a finite subset of \( \mathbb{N} \) then \( \mathbb{N} - S \) is countable.

1.46 We show that the inverse of a correspondence is also a correspondence.

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

(B) Let \( f : D \to C \) be a correspondence. Show that the inverse function exists. That is, show that associating each \( y \in C \) with the \( x \in D \) such that \( f(x) = y \) gives a well-defined function \( f^{-1} : C \to D \).

(C) Show that the inverse 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. \textit{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. \textit{Hint:} again use induction on the size of the domain.

(C) If such a function 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 they correspond, that is, if and only if there is a correspondence from one to the other.

\textbf{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.

\textbf{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. So informally, \( S \) is twice the natural numbers and, as in Galileo’s Paradox, 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.
Chapter II. Background

2.2 Animation: Enumerating \( \{0, 1\} \times \mathbb{N} \).

This illustrates the correspondence as a table.

<table>
<thead>
<tr>
<th>( n \in \mathbb{N} )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle i, j \rangle \in \mathbb{N} \times \mathbb{N} )</td>
<td>( \langle 0, 0 \rangle )</td>
<td>( \langle 1, 0 \rangle )</td>
<td>( \langle 0, 1 \rangle )</td>
<td>( \langle 1, 1 \rangle )</td>
<td>( \langle 0, 2 \rangle )</td>
<td>( \langle 1, 2 \rangle )</td>
<td>...</td>
</tr>
</tbody>
</table>

The map from the top row to the bottom row 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 \( \{0, 1, 2\} \times \mathbb{N} \), 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 countable.

\[ \text{Proof} \quad \text{Exercise 2.33; use the above example as a model.} \]

2.4 Example The natural next set has infinitely many copies: \( \mathbb{N} \times \mathbb{N} \).

<table>
<thead>
<tr>
<th>( \langle 0, 0 \rangle )</th>
<th>( \langle 1, 0 \rangle )</th>
<th>( \langle 0, 1 \rangle )</th>
<th>( \langle 1, 1 \rangle )</th>
<th>( \langle 0, 2 \rangle )</th>
<th>( \langle 1, 2 \rangle )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle 0, 3 \rangle )</td>
<td>( \langle 1, 3 \rangle )</td>
<td>( \langle 2, 3 \rangle )</td>
<td>( \langle 3, 3 \rangle )</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle 0, 2 \rangle )</td>
<td>( \langle 1, 2 \rangle )</td>
<td>( \langle 2, 2 \rangle )</td>
<td>( \langle 3, 2 \rangle )</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle 0, 1 \rangle )</td>
<td>( \langle 1, 1 \rangle )</td>
<td>( \langle 2, 1 \rangle )</td>
<td>( \langle 3, 1 \rangle )</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle 0, 0 \rangle )</td>
<td>( \langle 1, 0 \rangle )</td>
<td>( \langle 2, 0 \rangle )</td>
<td>( \langle 3, 0 \rangle )</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 correspondence as a table.

\[
\begin{array}{c|cccccccc}
\text{Number} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\
\text{Pair} & \langle 0, 0 \rangle & \langle 0, 1 \rangle & \langle 1, 0 \rangle & \langle 0, 2 \rangle & \langle 1, 1 \rangle & \langle 2, 0 \rangle & \langle 0, 3 \rangle & \ldots \\
\end{array}
\]

That this procedure gives a correspondence is perfectly evident. But because it is amusing we will develop the formula for going from the bottom line to the top. Animation 2.5 numbers the diagonals.

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 \( 1 + 2 + \cdots + d \). 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 \( \text{cantor} : \mathbb{N}^2 \to \mathbb{N} \) or unpairing function, or diagonal enumeration\(^\dagger\) is \( \text{cantor}(x, y) = x + [(x + y)(x + y + 1)/2] \). Its inverse is the pairing function, \( \text{pair} : \mathbb{N} \to \mathbb{N}^2 \).

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 \( \mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N} \), \( \mathbb{N}^4 \), \ldots are all countable.

*Proof* The function \( \text{cantor} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) 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 \( \text{cantor}_3 : \mathbb{N}^3 \to \mathbb{N} \) by \( \text{cantor}_3(x, y, z) = \text{cantor}(\text{cantor}(x, y), z) \). Exercise 2.27 shows that this function is a correspondence. The full induction step details are routine.

\(^\dagger\)Some authors use diamond brackets, writing \( \langle x, y \rangle \) where we write \( \text{cantor}(x, y) \).
2.9 **Corollary** The cross product of finitely many countable sets is countable.

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

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 nonnegatives 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, $f(0), f(1), \ldots, f(i-1)$. It loops, using the pairing function $\text{cantor}^{-1}$ to generate pairs: $\text{cantor}^{-1}(0), \text{cantor}^{-1}(1), \text{cantor}^{-1}(2), \ldots$. 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, $S = \{s_0, \ldots, s_{n-1}\}$, 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 $f(k) \not\in \{\hat{f}(0), \ldots, \hat{f}(n-1)\}$. 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.

This section starts off by noting that it is natural to see how countability interacts with set operations.
Section 2. Cantor’s correspondence

2.12 Corollary (1) Any subset of a countable set is countable. (2) The intersection of two countable sets is countable. The intersection of countably many countable sets is countable. (3) The union of two 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 \( f : \mathbb{N} \to S \). If \( \hat{S} \) is empty then it is countable, and if not then fix some \( \hat{s} \in \hat{S} \) so that this map \( \hat{f} : \mathbb{N} \to \hat{S} \) is onto.

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

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

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 \( f_0 : \mathbb{N} \to S_0 \) and \( f_1 : \mathbb{N} \to S_1 \) are onto. Lemma 2.3 gives a correspondence taking \( \mathbb{N} \) to \( \{0, 1\} \times \mathbb{N} \), inputting natural numbers and outputting pairs \((i, j)\) where \( i \) is either 0 or 1. Call that function \( g : \mathbb{N} \to \{0, 1\} \times \mathbb{N} \). Then this is the desired function onto the set \( S_0 \cup S_1 \).

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

This approach extends to any finite number of countable sets.

Finally, we start with countably many countable sets, \( S_i \) for \( i \in \mathbb{N} \), and show that their union \( S_0 \cup S_1 \cup \cdots \) is countable. If all but finitely many are empty then we can fall back to the finite case so 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 \( f_j : N \to S_j \). Then use Cantor’s pairing function: the desired map from \( \mathbb{N} \) onto \( S_0 \cup S_1 \cup \cdots \) is \( \hat{f}(n) = f_j(k) \) where \( \text{pair}(n) = (j, k) \).

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 numbering then their cross product corresponds to \( \mathbb{N} \) via an effective numbering. We finish this section by applying that to Turing machines—we will give a way to effectively number the Turing machines.

Turing machines are sets of instructions. Each instruction is a four-tuple, a member of \( Q \times \Sigma \times (\Sigma \cup \{L, R\}) \times Q \), where \( Q \) is the set of states and \( \Sigma \) is the tape alphabet. So by the above numbering results, we can number the instructions: there is an instruction whose number is 0, one with number 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 (see Exercise 2.23).
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 $\{i_0, i_1, \ldots, i_n\}$, and then output the number associated with that machine as $e = g(\mathcal{P}) = 2^{i_0} + 2^{i_1} + \cdots + 2^{i_n}$.

The inverse association is much the same. Given a natural number $e$, represent it in binary $e = 2^{j_0} + \cdots + 2^{j_k}$, form the set of instructions corresponding to the numbers $j_0, \ldots, j_k$, and that is the output Turing machine $g^{-1}(e)$. (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 define $g^{-1}(e)$ to 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 source-equivalent — we can go effectively from the index to the machine source, the set of 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 Every computable function has infinitely many indices: if $f$ is computable then there are infinitely many distinct $e_i \in \mathbb{N}$ with $f = \phi_{e_0} = \phi_{e_1} = \cdots$.

Proof Suppose that $f = \phi_e$. The machine $\mathcal{P}_e$ has some set of states; let $q_j$ be the highest-numbered state mentioned in that set. For each $k \in \mathbb{N}^+$ consider the Turing machine obtained from $\mathcal{P}_e$ by adding the instruction $q_{j+k} B B q_{j+k}$. This gives a sequence of Turing machines $\mathcal{P}_{e_1}, \mathcal{P}_{e_2}, \ldots$ with distinct indices, all having the same behavior, $\phi_{e_k} = \phi_e = f$. \qed
2.16 Remark Stated in terms of more everyday programming, we can get infinitely different many source codes that have the same compiled behavior, by starting with one source 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 functions from \( \mathbb{N} \) to \( \mathbb{N} \) that they cannot be counted. This will prove that there are functions not computed by any Turing machine.

II.2 Exercises

\( \checkmark \) 2.17 Extend the table of Example 2.1 through \( n = 12 \). Where \( f(n) = \langle x, y \rangle \), give formulas for \( x \) and \( y \).

\( \checkmark \) 2.18 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 \( f_T : \mathbb{N} \to T \) and \( f_F : \mathbb{N} \to F \). Give a table listing the values of \( f_T(0), \ldots f_T(9) \) and give another table listing \( f_F(0), \ldots f_F(9) \).

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

\( \checkmark \) 2.19 Give an enumeration of \( \mathbb{N} \times \{ 0, 1 \} \). Find the pair corresponding to 0, 10, 100, and 1000. Find the number corresponding to \( \langle 2, 1 \rangle, \langle 20, 1 \rangle \), and \( \langle 200, 1 \rangle \).

\( \checkmark \) 2.20 Example 2.1 says that the method for two columns extends to three. Give an enumeration \( f \) of \( \{ 0, 1, 2 \} \times \mathbb{N} \). That is, where \( f(n) = \langle x, y, z \rangle \) give a formula for \( x, y, \) and \( z \) Find the pair corresponding to 0, 10, 100, and 1000. Find the number corresponding to \( \langle 1, 2 \rangle, \langle 1, 20 \rangle \), and \( \langle 1, 200 \rangle \).

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

\( \checkmark \) 2.22 Extend the table of Example 2.4 to cover correspondences up to 20.

\( \checkmark \) 2.23 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: \( \mathbb{N} \to \mathbb{N}^2 \) is also effective by sketching a way to compute it with a program.

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

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

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

(A) Fix a natural number \( n \). Prove that the set of polynomials with \( n + 1 \)-many terms \( P_n[x] = \{ a_n x^n + \cdots + a_0 \mid a_n, \ldots a_0 \in \mathbb{Z} \} \) is countable by producing a suitable onto function.

(B) Finish the argument.

\( \checkmark \) 2.27 The proof of Lemma 2.8 says that the function \( \text{cantor}_3 : \mathbb{N}^3 \to \mathbb{N} \) given by \( \text{cantor}_3(a, b, c) = \text{cantor}(\text{cantor}(a, b), c) \) is a correspondence. Verify that.
2.28 Define \( c_3 : \mathbb{N}^3 \to \mathbb{N} \) by \( \langle x, y, z \rangle \mapsto \text{cantor}(x, \text{cantor}(y, z)) \). (A) Compute \( c_3(0, 0, 0), c_3(1, 2, 3), \) and \( c_3(3, 3, 3) \). (B) Find the triples corresponding to 0, 1, 2, 3, and 4. (C) Give a formula.

2.29 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 number that is a multiple of four.

2.30 Give an alternative proof of the three set case of Corollary 2.12 where \( C_0 \) is the set of numbers that are divisible by 3, \( C_1 \) is the set of numbers that leave a remainder of 1 on division by 3, and \( C_2 \) is the set of numbers that leave a remainder of 2.

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

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

2.33 Give the proof of Lemma 2.3.

2.34 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.

2.35 Besides Cantor’s correspondence from \( \mathbb{N}^2 \) to \( \mathbb{N} \), there are others.

(A) Show that \( g : \mathbb{N}^2 \to \mathbb{N} \) given by \( \langle n, m \rangle \mapsto 2^n(2m+1)−1 \) is a correspondence.

(B) Find the number corresponding to the pairs in \( \{ \langle n, m \rangle \in \mathbb{N}^2 \mid 0 \leq n, m < 4 \} \).

You may want to write the correspondence as a program and use the program.

(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?

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

(A) To count the union of two sets we partitioned the natural numbers into the odds and evens, that is, into the set of numbers whose binary representation does not end in 0 and the set of numbers whose representation does end in 0. 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.) Each is clearly countably infinite so there are correspondences \( g_0 : \mathbb{N} \to C_0 \), \( g_1 : \mathbb{N} \to C_1 \) and \( g_2 : \mathbb{N} \to C_2 \). Produce an onto function \( f : \mathbb{N} \to S_0 \cup S_1 \cup S_2 \).

(B) To show that the countable union of countable sets is countable start with
countably many countable sets, \( S_i \) for \( i \in \mathbb{N} \). If all but finitely many are empty then we can fall back to the finite case so 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. We now introduce a powerful technique to show that no correspondence exists. This technique 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, \( \mathbb{R} \).

**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.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Decimal expansion of ( f(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>42 . 3 1 2 7 7 0 4 ...</td>
</tr>
<tr>
<td>1</td>
<td>2 . 0 1 0 0 0 0 0 ...</td>
</tr>
<tr>
<td>2</td>
<td>1 . 4 1 4 1 5 9 2 ...</td>
</tr>
<tr>
<td>3</td>
<td>-20 . 9 1 9 5 9 1 9 ...</td>
</tr>
<tr>
<td>4</td>
<td>0 . 1 0 1 0 0 1 0 ...</td>
</tr>
<tr>
<td>5</td>
<td>-0 . 6 2 5 5 4 1 8 ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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: take the digits 3, 1, 4, 5, 0, 1 \( ... \). Now, construct the number \( z \) by making its first decimal place something other than 3, making its second decimal place something other than 1, etc. To be specific: 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 \( z = 0.121112 ... \). 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.

(In this argument we have skirted a technicality, that some real numbers have two different decimal representations. For instance, 1.0000 \( ... = 0.999 ... \).)
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.)

This technique is diagonalization.

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 \( \{0, \ldots, 9\} \) given by: \( g(j) = 2 \) if \( j \) is 1, and \( g(j) = 1 \) otherwise.

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

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, says something both 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 Galelio’s Paradox, this is not just that the reals are a superset of the naturals. 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 \( P_0, P_1, \ldots \) but there are just too many real numbers to fit in them.

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 3.30.
3.6 Theorem (Cantor's Theorem) A set's cardinality is strictly less than that of its power set.

Before stating the proof we first illustrate it. The easy half is starting with a set $S$ and producing a function to $\mathcal{P}(S)$ that is one-to-one: just map $s \in S$ to $\{s\}$.

The harder half is showing that no map from $S$ to $\mathcal{P}(S)$ is onto. We illustrate the proof with $S = \{a, b, c\}$ and this function $f : S \to \mathcal{P}(S)$.

$$
\begin{align*}
\text{a} & \mapsto \{b, c\} & \text{b} & \mapsto \{b\} & \text{c} & \mapsto \{a, b, c\}
\end{align*}
$$

In the table below the first row lists the values of the characteristic function $1_{f(a)} : S \to \{0, 1\}$ on the inputs $a, b,$ and $c$. The second row lists the input/output values for $1_{f(b)}$. And, the third row lists $1_{f(c)}$.

<table>
<thead>
<tr>
<th>$s \in S$</th>
<th>$1_{f(s)}(a)$</th>
<th>$1_{f(s)}(b)$</th>
<th>$1_{f(s)}(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We show that $f$ is not onto by producing a subset of $S$ that is not one of the three sets in $(*)$. For that, diagonalize: go down the table's diagonal 011 and flip the bits from 0 to 1 or from 1 to 0, giving 100. That's the characteristic function of $R = \{a\}$. This set is not equal to $f(a)$ because it differs on $a$, it is not $f(b)$ because it differs on $b$, and it is not $f(c)$ because it differs on $c$.

Proof One half is easy: consider the injection map $i : S \to \mathcal{P}(S)$ given by $i(s) = \{s\}$. It is one-to-one so the cardinality of $S$ is less than or equal to the cardinality of $\mathcal{P}(S)$.

For the other half, to show that no map from a set to its power set is onto, fix any $f : S \to \mathcal{P}(S)$ and consider this element of $\mathcal{P}(S)$.

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

We will show 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 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 all natural number functions $f : \mathbb{N} \rightarrow \mathbb{N}$.

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

In conclusion, we have seen that some infinite sets have more members than others. Specifically, the set of real numbers has more members than the set of natural numbers. This is not about that the reals are a superset of the naturals, but instead it is about that there is no correspondence between the two.

Likewise, there are more members of the set of functions from $\mathbb{N}$ to itself, than there are members of the set $\mathbb{N}$. Lemma 2.8 shows that there are as many Turing machines as there are members of $\mathbb{N}$. So, the cardinality of the set of functions $f : \mathbb{N} \rightarrow \mathbb{N}$ is greater than the cardinality of the set of Turing machines. Consequently some function $f : \mathbb{N} \rightarrow \mathbb{N}$ is without a Turing machine $\mathcal{P}_0, \mathcal{P}_1, \ldots$.

This is an epochal result. In the light of Church’s Thesis we understand it to prove that there are functions not computed by any program—there are jobs that no computer can do.

II.3 **Exercises**

3.8 Your friend 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!” Straighten them out.

3.9 Verify Cantor’s Theorem Theorem 3.6 for these finite sets.

**(A)** $\{0, 1, 2\}$

**(B)** $\{0, 1\}$

**(C)** $\{0\}$

**(D)** $\{\}$

3.10 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

3.11 Name a set with a larger cardinality than $\mathbb{R}$.

3.12 Characterize each set as countable or uncountable.

**(A)** $[1..4] \subset \mathbb{R}$

**(B)** $[1..4] \subset \mathbb{N}$

**(C)** $[5..\infty) \subset \mathbb{R}$

**(D)** $[5..\infty) \subset \mathbb{N}$

3.13 One set is countable and one 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 Short answer: if $A$ is a finite set and $B$ is finite then the most we can say about $A \cup B$ is: (A) uncountable (B) countable or uncountable, (C) finite, (D) countable, (E) at most countable, (F) finite, countable, or uncountable.
3.15 Where $A = \{0, 1, 2\}$ list all the functions with domain $A$ and codomain $\mathcal{P}(A)$.

3.16 Suppose that $S$ is countable and $f : S \to T$ is onto. Which of these is possible?
(A) $S$ is finite  (B) $T$ is finite  (C) $S$ is countably infinite  (D) $T$ is countably infinite  (E) $T$ is uncountable.

3.17 Suppose that $S$ is countable and $f : S \to T$ is one-to-one. Which of these is possible?
(A) $S$ is finite  (B) $T$ is finite  (C) $S$ is countably infinite  (D) $T$ is countably infinite  (E) $T$ is uncountable.

✓ 3.18 Short answer: if $A$ is a countable set and $B$ is finite then the most we can say about $A \cup B$ is: (A) uncountable (B) countable or uncountable, (C) finite, (D) countable, (E) at most countable, (F) finite, countable, or uncountable.

✓ 3.19 Short answer: if $A$ is a countable set and $B$ is uncountable then the most we can say about $A \cup B$ is: (A) uncountable (B) countable or uncountable, (C) finite, (D) countable, (E) at most countable, (F) finite, countable, or uncountable.

✓ 3.20 Short answer: if $A$ is a countable set and $B$ is uncountable then the most we can say about $A \cap B$ is: (A) uncountable (B) countable or uncountable, (C) finite, (D) countable, (E) at most countable, (F) finite, countable, or uncountable.

✓ 3.21 Recall that $\mathbb{B} = \{0, 1\}$.

  (A) Show that the set of finite binary sequences, $\langle b_0 b_1 \ldots b_{k-1} \rangle$ where $b_i \in \mathbb{B}$ and $k \in \mathbb{N}$, is countable.

  (B) An infinite binary sequence $f = \langle b_0, b_1, \ldots \rangle$ is a function $f : \mathbb{N} \to \mathbb{B}$. Show that the set of infinite binary sequences is uncountable with diagonalization.

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

3.23 The proof of Theorem 3.1 diagonalizes. Could we instead “verticalize?”

3.24 Show that a product of a countable number of nonempty sets of natural numbers $S_0 \times S_1 \times \cdots$ is uncountable.

3.25 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 rational ones. Recall that the union of two countable sets is countable. Prove that while the set of rational numbers is countable, the set of irrational numbers is uncountable.

✓ 3.26 Example 2.10 shows that the rational numbers are countable. What happens with the argument given in Theorem 3.1 is applied to the rationals? Consider a listing $q_0, q_1, \ldots$. For each $q_i$ fix a decimal expansion $q_i = d_{i,0} d_{i,1} d_{i,2} \ldots$ that does not end in all 9’s. We will show that the diagonal number $d = \sum_{1 \leq n < \infty} d_{n,n} 10^{-n}$ is irrational. (Note that the rationals are not all of the decimal expansions and so the fact that the diagonal entry is not on the list is not a contradiction to the fact that we can enumerate all of the rationals.)
(A) Let \( g \) be the map on the decimal digits 0, 1, \ldots, 9 given by \( g(1) = 2 \), and \( g(j) = 1 \) for \( j \neq 1 \). Define \( y = \sum_{1 \leq n < \infty} g(d_n)10^{-n} \). Show that \( y \) is irrational.

(B) Use the prior item to conclude that \( d \) is irrational. \textit{Hint:} show that it has no repeating pattern in its decimal expansion.

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

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

3.29 Show that there is no set of all sets. \textit{Hint:} use Theorem 3.6.

3.30 Extending the definition of equal cardinality, we say \(|A| \leq |B|\) if there is a one-to-one function from \( A \) to \( B \). The \textit{Schröder–Bernstein theorem} is that if \( |S| \leq |T| \) and \( |T| \leq |S| \) then \( |S| = |T| \). We will walk through the proof, which 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 \( s, f(s), g(f(s)), f(g(f(s))), \ldots \) and to the left, as here.

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

For any \( t \in T \) define the associated chain similarly. For instance, let \( S = \{0, 1, 2\} \) and \( T = \{a, b, c\} \) (where the elements of \( T \) are characters, not variables). Let the correspondences be \( f : S \to T \) and \( g : T \to S \).

\[
\begin{array}{c|c|c}
 s & f(s) & g(t) \\
 0 & b & a \\
 1 & c & b \\
 2 & a & c \\
\end{array}
\]

Starting at \( 0 \in S \) gives a single chain that is cyclic, \( \ldots, 0, b, 1, c, 2, a, 0 \ldots \)

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

(C) Show that we can assume without loss of generality that \( S \) and \( T \) are disjoint.

(D) Assume \( S \) and \( T \) are disjoint and that \( f : S \to T \) and \( g : T \to S \) are one-to-one. Show that every element of either set is in a unique chain, and that each chain is of one of four kinds: (i) those that repeat after some number of terms (ii) those that continue infinitely in both directions (iii) those that
Section 4. Universality

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 many Turing machines: one whose behavior is that the output is double the input, one that takes two inputs and adds them, etc. These are single-purpose devices, where to get different input-output behavior we needed to get a new machine, that is, new hardware. This was what we meant in saying that a first take on Turing machines is that they are more like a modern program than like a modern computer.

The picture below shows programmers of an early electronic computer changing its behavior by changing its circuits, using the patch cords.

![ENIAC, reconfigure by rewiring.](image)

Imagine having a laptop where to change from running a browser to a spreadsheet you must pull one chip and replace it with another. The 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 case 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 paper cards. Getting a different pattern does not require a new loom, it only requires swapping cards. Turing introduced the analog for computing devices.

4.1 **Theorem (Turing, 1936)** There is Turing machine that when given the input $e, x$ will have the same output behavior as does $P_e$ on input $x$.

This asserts that a Turing machine can simulate Turing machines — there is a single Turing machine that can act like any Turing machine at all. This is a **Universal Turing Machine**. We denote such a machine with $U\mathcal{P}$.

A universal machine may seem to present a chicken and egg problem: how can we give a Turing machine as input to a Turing machine? In particular, if the universal machine is $P_e$, the theorem seems to allow the possibility of giving the universal machine to itself — won’t feeding a machine to itself lead to infinite regress?

We don’t feed a machine to itself. Instead, the machine gets symbols as input, specifically it gets a pair of numbers. True, that is computationally equivalent to the source of a machine. But we run Turing machines by putting symbols on the tape as data and pressing Start, and even if those symbols are equivalent to the machine’s source then the universe won’t explode. The machine just runs. You could absolutely write a text editor and use it to edit its own source. Or you could write a compiler and use it to compile itself. Similarly, you could feed a machine its own number. This is not to say that nothing interesting happens as a result, but rather to say that there is no inherent impossibility.

It remains to argue that there are universal machines. These devices change behavior in software — this is like the loom’s punched cards, which alter the behavior of the machine without any hardware patch chords. We first appeal to the experience that we have from everyday computing work.

---

*A technical point: Turing machines have a tape alphabet. So what we put on the tape has to be input that the machine is defined as able to take. But every Turing machine takes at least two symbols, one of which is a blank, so if we represent a number $e$ in binary by taking blank as 0 and any nonblank character as 1 then we are OK; we can feed a machine numbers.*
Our first appeal to experience is an analogy: a universal machine is like an operating system.† Imagine your computer, with its operating system and some program that runs under that environment. Church’s Thesis allows us to picture this program as a Turing machine, $P_e$, which may take input, a string of 0’s and 1’s, that we can interpret as the binary representation of a number $x$. The point of the operating system here is that its job is to run programs and feed them input. That is, from input $e, x$, the operating system responds with the behavior of machine $e$ when fed input input $x$.

Another computer experience that we have everyday that is like a universal machine, where a computing system acts like an arbitrary source code, is an interpreter. Below is a Scheme interpreter. After it’s command line invocation, in line 1 the system gets the source of a program that takes in $i$ and sum the first $i$ numbers. In line 2 we run that source with the input $i = 4$. The interpreter returns the answer of 10.

```
$ csi
CHICKEN
(c) 2008-2013, The Chicken Team
(c) 2000-2007, Felix L. Winkelmann
Version 4.8.0.5 (stability/4.8.0) (rev 5bd53ac)
linux-unix-gnu-x86-64 [ 64bit manyargs dload xtables ]
compiled 2013-10-03 on aeryn.xorinia.dim (Darwin)
;p1> (define sum
    (lambda (i tot)
      (if (= i 0)
          tot
          (sum (- i 1) (+ tot i)))))
p;2> (sum 4 0)
 10
;p;3> ((lambda (i) (if (= i 0) 1 0)) 1)
 0

Even more directly, in line 3 (it is shown highlighted) the interpreter gets as a single expression both the source of a routine, and the input, and it returns the result of applying the source to the input.

The most concrete example of our everyday experience with computing systems that act as universal machines is an `eval` statement.

```
p;p1> (define (utm s)
    (eval s (scheme-report-environment 5)))
p;p2> (define TEST '((lambda (i) (if (= i 0) 1 0)))
p;p3> TEST
    (lambda (i) (if (= i 0) 1 0)))
p;p4> (utm TEST) 5)
    0
;p;5> (utm TEST) 0)
    1
```

†This is a flow chart, which we use to give a high level outline of programs. We use only three types of boxes. Rectangles are for the ordinary flow of control. Round corner rectangles are for the Start and End. Diamond boxes (which will appear in later flow charts) are for decisions, if statements.
(The `(scheme-report-environment 5)` is a technicality that ensures that the routines defined in the specification for Scheme 5 are available to `eval` so it is inside a definition.) The prior examples's routine that tests if its input is 0 has a quote so that Scheme will keep it as source code, and not execute it. Lines 4 and 5 show that source fed to `utm`, which runs `eval` on it, so the first entry in the list is now a routine, which acts on the argument, producing the correct output.

Finally, we have already exhibited what amounts to a Universal Turing machine. In the Extra section titled Turing Machine simulator, on Extra A, we gave code that reads an arbitrary Turing machine from a file. By Church’s Thesis we could write this code as a Turing machine.

**Uniformity** We've argued that there are functions that no program can perform. Consider this job: given a real number \( r \in \mathbb{R} \), write a program to produce its digits. More precisely, this asks for a family of programs, a \( P_r \) for each \( r \), such that when given input \( n \in \mathbb{N} \) the program returns the value of the \( n \)-th decimal place of \( r \) (for \( n = 0 \) the output is the integer to the left of the decimal point).

Because of cardinality there is no such family of programs: there are countably many Turing machines but uncountably many reals. But why can't we write it? One of the enjoyable things about coding is the feeling that we can get the machine to do whatever we want; what's stopping us from producing whatever digits we like?

There are some real number \( r \)'s for which there indeed is such a program. An easy one is \( r = 1/4 \). For a more generic number, say \( r = 0.703 \ldots \), we might momentarily imagine brute-forcing it.

```
read n
if n==0:
    print 0
if n==1:
    print 7
elif n==2:
    print 0
...
```

An experienced programmer knows that this is silly because programs have finite length. We could have `if .. elif ..` branches for a few cases but past some point it must have code that handles all but finitely many input \( n \)'s, uniformly.

Thus, the fact that Turing machines have only finitely-many instructions imposes on what we can do a requirement of uniformity. This can have surprising consequences.

**Example** Consider \( \pi = 3.14159 \ldots \). Imagine being given the problem of writing a program that inputs a number \( n \) and decides whether somewhere in \( \pi \)'s decimal expansion there is a sequence of \( n \) consecutive nines.

---

\(^\dagger\) 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 this example, however, the software engineering issues are not relevant.
Here is an answer: there are two possibilities. Either for all inputs \( n \) such a sequence exists or else there is some \( n_0 \) where the sequence exists for inputs less or equal to \( n_0 \) and no such sequence exists for \( n \geq n_0 \). Therefore one of these two programs solves the problem — we don’t know which one, but one of the two does.

```
read n
print 1
```

```
read n
if n < n0:
  print 1
else:
  print 0
```

It is surprising that the answer, whichever of the two it is, seems not to have much to do with \( \pi \). It is also surprising, and perhaps unsettling, that we have shown that the problem is mechanically solvable without giving any information as to what the solution is. That is, there is a difference between defining this function and giving an algorithm to compute it.

\[
f(n) = \begin{cases} 
1 & \text{if there are } n \text{ consecutive } 9\text{'s in } \pi \\
0 & \text{otherwise}
\end{cases}
\]

There is an alternative. Consider \( P_\pi \), that inputs \( i \in \mathbb{N} \) and outputs \( \pi \)'s \( i \)-th decimal place. With it, we can write a routine that takes in \( n \) and uses \( P_\pi \)'s ability to give the digits to search for \( n \) nines in a row. This approach is constructive in that we are computing the answer not just saying that it exists. It is also uniform in the sense that we could modify the routine to take as inputs both \( n \) and \( P_r \). However, this approach has the disadvantage that if \( n_0 \) is such that for \( n \geq n_0 \) never does \( r \) have \( n \) nines in a row, then this routine will just search forever, never printing 0.

**Parametrization** Universality says that there is a single Turing machine \( U_{\mathcal{P}} \), which takes in two inputs \( e, x \) and returns the same value as we would get by running \( P_e \) on input \( x \) (including not halting, if that machine does not halt).

That is, there is a computable function \( F: \mathbb{N}^2 \rightarrow \mathbb{N} \) such that \( F(e, x) = P_e(x) \) (again, including that if \( P_e(x) \uparrow \) then \( F(e, x) \uparrow \) also). There, the letter \( e \) travels from the function’s argument to an index. We now generalize this observation.

Consider a two-input program \( P_e(x, y) \). Freezing the first argument, that is, locking \( x = a \) for some \( a \in \mathbb{N} \), gives a one-input program \( Q_a(y) = P_e(a, y) \). For example, starting with

```
(define (P a b)
  (+ a b))
```

and freezing the first input at \( a = 7 \) gives a one-input routine.

```
(define (P_7 b)
  (P 7 b))
```

This is partial application.

Obviously the second program is related to the first. The next result is that we can go from the index of the first program and the value that is frozen, over to the index of the second program, via a computable function.
4.3 **Theorem (Parameter theorem, or s-m-n theorem)** For every \( m, n \in \mathbb{N} \) there is a computable total function \( s_{m,n} : \mathbb{N}^{1+m} \rightarrow \mathbb{N} \) with this behavior: for the \( m + n \)-ary function \( \phi_e(x_0, \ldots, x_m, \ldots, x_{m+n-1}) \), freezing the initial \( m \) variables at \( a_0, \ldots, a_{m-1} \in \mathbb{N} \) gives a function equal to \( \phi_{s(e,a_0,\ldots,a_{m-1})}(x_m, \ldots, x_{m+n-1}) \).

(The function \( \phi_e(x_0, \ldots, x_{m-1}, x_m, \ldots, x_{m+n-1}) \) could be partial, that is, it could be that the Turing machine \( P_e \) fails to halt on some inputs \( x_0, \ldots, x_{m-1}, x_m, \ldots, x_{m+n-1} \).)

**Proof** We will produce the function \( s \) to satisfy three requirements: it must be effective, it must input an index \( e \) and an \( m \)-tuple \( x_0, \ldots, x_{m-1} \), and it must output the index of a machine \( \hat{P} \) that, when given the input \( x_m, \ldots, x_{m+n-1} \), will return the value \( \phi_e(x_0, \ldots, x_{m-1}, x_m, \ldots, x_{m+n-1}) \).

The idea is that \( s \) constructs the instructions for \( \hat{P} \). We know how to get from the instruction set to the index, using Cantor’s encoding, so with that we will be done. The flowchart for \( s \) is on the left below. The flowchart for \( \hat{P} \), the Turing machine that \( s \) constructs, is on the right.

Recall that we are being flexible about the convention for Turing machines that take more than one input. To be clear here, we assume for this argument that input is encoded in unary, with any number \( k \in \mathbb{N} \) represented by a string of \( k \)-many \( 1 \)'s, 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.

With that, the machine \( \hat{P} \) does not start by reading its inputs \( x_m, \ldots, x_{m+n-1} \). Instead, it moves left and puts \( x_0, \ldots, x_{m-1} \) on the tape, in unary and separated by blanks, and with a blank between \( x_{m-1} \) and \( x_m \). Thus when this phase is done the machine will have all of the inputs \( x_0, \ldots, x_{m+n-1} \) on the tape, in unary and separated by blanks. Next \( \hat{P} \) moves the read/write head until it is on the leftmost 1 of the \( x_0 \). Finally, using universality, \( \hat{P} \) simulates Turing machine \( P_e \), and lets it run on that input list.
4.4 Example  Consider the two-input Turing machine sketched by this flowchart.

\[
\text{Start} \\
\text{Read } x, y \\
\text{Print } x \cdot y \\
\text{End}
\]

(∗)

Suppose that it has index \(e\).

We can use the Parameter Theorem to freeze the value of \(x\) to 0. On the left below is the flowchart for \(P_{s_1,1}(e,0)\). It is a one-input machine and the function that it computes is \(\phi_{s_1,1}(e,0)(y) = 0\). For example, \(\phi_{s_1,1}(e,0)(5) = 0\).

\[
\text{Start} \\
\text{Read } y \\
\text{Print } 0 \cdot y \\
\text{End}
\]

\[
\text{Start} \\
\text{Read } y \\
\text{Print } 1 \cdot y \\
\text{End}
\]

\[
\text{Start} \\
\text{Read } y \\
\text{Print } 2 \cdot y \\
\text{End}
\]

Similarly the other two are flowcharts for \(P_{s_1,1}(e,1)\) and \(P_{s_1,1}(e,2)\), freezing the value of \(x\) at 1 and 2. The machine sketched in the center computes \(\phi_{s_1,1}(e,1)(y) = y\), so for instance \(\phi_{s_1,1}(e,1)(5) = 5\). On the right the machine computes \(\phi_{s_1,1}(e,2)(y) = 2y\), and an example is \(\phi_{s_1,1}(e,2)(5) = 10\).

In general, this is the flowchart for \(P_{s_1,1}(e,x)\).

\[
\text{Start} \\
\text{Read } y \\
\text{Print } x \cdot y \\
\text{End}
\]

The difference between this and the flowchart in (∗) above is that here the machine does not read in \(x\). Rather, like the three above, \(x\) is hard-coded into the program body. This is a family of functions, the first three of which are in the prior paragraph. This family is parametrized by \(x\) and the indices of these functions are uniformly computable from \(e\) and \(x\), using the function \(s\).

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 example suggests, they can be more annoying than illuminating. We often omit them.

II.4 Exercises

4.5 Your friend asks, “What can a Universal Turing machine do that a regular Turing machine cannot?” Help them out.
4.6 Has a Universal Turing machine, or a machine equivalent to one, ever been built?

4.7 Can a Universal Turing machine simulate another Universal Turing machine, or itself?

4.8 Your class has a jerk who keeps throwing out pronouncements that the prof has to patiently correct. This time it's, “Universal Turing machines make no sense. How can a machine simulate another machine if it has a different alphabet? Obviously it can't. And how could a machine simulate another machine that has more states? It makes no sense at all.” Clue this chucklehead in.

4.9 Is there more than one Universal Turing machine?

4.10 Consider the function $f(x_0, x_1) = 3x_0 + x_0 \cdot x_1$.
   (a) Freeze $x_0$ to have the value 4. What is the resulting one-variable function?
   (b) Freeze $x_0$ at 5. What is the resulting one-variable function?
   (c) Freeze $x_1$ to be 0. What is the resulting function?

4.11 Consider $f(x_0, x_1, x_2) = x_0 + 2x_1 + 3x_2$.
   (a) Freeze $x_0$ to have the value 1. What is the resulting two-variable function?
   (b) What two-variable function results from fixing $x_0$ to be 2?
   (c) Let $a$ be a natural number. What two-variable function results from fixing $x_0$ to be $a$?
   (d) Freeze $x_0$ at 5 and $x_1$ at 3. What is the resulting one-variable function?
   (e) What one-variable function results from fixing $x_0$ to be $a$ and $x_1$ to be $b$, for $a, b \in \mathbb{N}$?

4.12 Consider the Turing machine sketched by this flowchart.

Imagine that it has index $e$.

(A) Draw the flowchart for the Turing machine $P_{s_1,1}(e,1)$. Also describe the function $\phi_{s_1,1}(e,1)$.

(B) What are the values of $\phi_{s_1,1}(e,1)(0)$, $\phi_{s_1,1}(e,1)(1)$, and $\phi_{s_1,1}(e,1)(2)$?

(C) Draw the flowchart for $P_{s_1,1}(e,0)$. Also describe the function $\phi_{s_1,1}(e,0)$.

(D) What are the values of $\phi_{s_1,1}(e,0)(0)$, $\phi_{s_1,1}(e,0)(1)$, and $\phi_{s_1,0}(e,0)(2)$?

4.13 Consider the Turing machine sketched by this flowchart.

Imagine that it has index $e$.

(A) Draw the flowchart for the Turing machine $P_{s_1,1}(e,1)$. Also describe the function $\phi_{s_1,1}(e,1)$.

(B) What are the values of $\phi_{s_1,1}(e,1)(0)$, $\phi_{s_1,1}(e,1)(1)$, and $\phi_{s_1,1}(e,1)(2)$?

(C) Draw the flowchart for $P_{s_1,1}(e,0)$. Also describe the function $\phi_{s_1,1}(e,0)$.

(D) What are the values of $\phi_{s_1,1}(e,0)(0)$, $\phi_{s_1,1}(e,0)(1)$, and $\phi_{s_1,0}(e,0)(2)$?
Imagine that it has index $e$.

(A) Draw the flowchart for the Turing machine $P_{s_{1,2}(e,1)}$. Also describe the function $\phi_{s_{1,2}(e,1)}$.

(B) What are the values of $\phi_{s_{1,2}(e,1)}(0,1)$, $\phi_{s_{1,2}(e,1)}(1,0)$, and $\phi_{s_{1,2}(e,1)}(2,3)$?

(c) Draw the flowchart for $P_{s_{2,1}(e,1,2)}$. Also describe the function $\phi_{s_{2,1}(e,1,2)}$.

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

\[ \checkmark 4.14 \] Suppose that the Turing machine sketched by this flowchart has index $e$.

\[ \checkmark 4.15 \] 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 4.3, the Parameter Theorem, to fix $x_0 = 0$ and $x_1 = 3$. Give a flowchart for $P_{s(e,0,3)}$ and describe $\phi_{s(e,0,3)}$. What is $\phi_{s(e,0,3)}(5)$?

(B) Use the Parameter Theorem to fix $x_0 = 1$. Give a flowchart for $P_{s(e,1,3)}$ and describe $\phi_{s(e,1,3)}$. What is $\phi_{s(e,1,3)}(5)$?

(c) Give a flowchart for $P_{s(e,a,b)}$ and describe $\phi_{s(e,a,b)}$.

4.16 We can write programs that process the source code of programs.

(A) Sketch a program that, when given the source code $S$ of a program, will return the first character in that source (or some nominal value if the source is empty).

(B) Sketch a program that, when given the source code $S$ of a program, will return the final character in that source, or some nominal value.

(c) What happens when you feed the program in the prior part its own source?
4.17 We can write programs that process the source code of programs.

(A) Sketch a program that, when given the source code $S$ of a program, will return the number of times the character $a$ appears in that source.

(B) What happens when you feed your program’s source to itself?

(C) Sketch two programs that perform the same task as each other, that is, their input-output behavior is the same, but the program in the first item gives different results for those two.

Section II.5 Unsolvability

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, our taste is to prefer to construct a specific unsolvable function rather than just to know that there is one. We will now do that.

The Halting problem The natural approach is to go through Cantor’s Theorem and effectivize it, to see how much of the proof we can express as a construction.

Consider an illustrative table adapted from the discussion of Cantor’s Theorem on page 75. This table’s rows are the computable functions and its columns are the inputs. For instance, this table imagines that $\phi_2(3) = 5$.

Diagonalize. That is, consider the program on the right. Its output seems not to be any of the table’s rows, which appears to contradict the supposition that the rows are all of the computable functions.

In the Cantor’s Theorem argument, the contradiction showed that there are more than countably many rows. But here we know that there are countably many Turing machines so we can imagine the rows that are shown. So what’s the resolution of the apparent contradiction? The program’s first, second, fourth and fifth boxes are trivial so the problem must lie with getting through its third box.

The issue is halting — among the computable functions there are ones does not converge on some inputs. In particular the apparent contradiction shows that this
must happen on the diagonal; there must be an \( e \in \mathbb{N} \) so that \( \phi_e(e) \uparrow \). Recognizing this makes the difficulty dissolve because for those input \( e \)'s in the flowchart, the program will never leave the middle box and so will never get to the contradictory output box. In short, in the above table some of the entries must be up arrows, \( \uparrow \).

5.1 **Definition** \( K = \{ e \in \mathbb{N} \mid P_e \) halts on input \( e \), that is, \( \phi_e(e) \downarrow \} \)

5.2 **Problem (Halting problem)** Given \( e \in \mathbb{N} \), determine whether Turing machine \( P_e \) halts on input \( e \), that is, whether \( \phi_e(e) \downarrow \).

Certainly for any \( e \in \mathbb{N} \), either \( \phi_e(e) \downarrow \) or \( \phi_e(e) \uparrow \). Thus there is a function giving 1 in the first case and 0 in the second. The Halting problem is to decide if that function is computable, to find a Turing machine with that behavior.

5.3 **Theorem (Unsolvability of the Halting problem)** The Halting problem is mechanically unsolvable.

*Proof* Assume otherwise, that there is a Turing machine whose behavior is this.

\[
K(e) = \text{halt\_checker}(e) = \begin{cases} 
1 & \text{if } \phi_e(e) \downarrow \\
0 & \text{if } \phi_e(e) \uparrow 
\end{cases}
\]

Then the function below is also mechanically computable. The flowchart illustrates how \( f \) is constructed; it uses \( \text{halt\_checker} \) in its decision box.

\[
f(e) = \begin{cases} 
0 & \text{if } \phi_e(e) \uparrow \\
\uparrow & \text{if } \phi_e(e) \downarrow 
\end{cases}
\]

(In \( f \)'s first case the output value 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 on the right the input \( \hat{e} \). If it diverges then by the first clause in the definition of \( f \) we have \( f(\hat{e}) \downarrow \), which contradicts divergence. If it converges then by \( f \)'s second clause we have \( f(\hat{e}) \uparrow \), also impossible. Since assuming that \( \text{halt\_checker} \) is mechanically computable leads to a contradiction, \( \text{halt\_checker} \) is not mechanically computable.

With Church’s Thesis in mind will say that a task is **unsolvable** if it is mechanically unsolvable, if no Turing machine computes that task. If the task is to compute the answer to ‘yes’ or ‘no’ questions, such as the task of determining membership in a set, then we will also use as a synonym that the problem is **undecidable**.

**Discussion** The fact that the Halting Problem is unsolvable does not mean that we cannot tell if any program halts. This program obviously halts for every input.
Nor does the unsolvability of Halting problem mean we cannot tell if a program does not halt. Consider this one.

```
#;1> (define (f x)
    (+ 1 (f x)))
```

Once we start it, it just keeps going.

```
#;2> (f 0)
^C
```

Instead, the unsolvability of the Halting Problem says that there is no single program that, for all \( e \), correctly computes in a finite time whether \( P_e \) halts on input \( e \).

That has the qualifier ‘finite time’ because we could perfectly well write this program

```
read e
simulate machine e on input e
print 1
```

but if \( P_e \) on input \( e \) fails to halt then we would not find out in a finite time. The ‘single program’ qualifier is there because for any index \( e \), either \( P_e \) halts on \( e \) or else it does not. That is, for any \( e \) one of these two programs gives the right answer.

```
read e
print 0

read e
print 1
```

Of course, solving the Halting problem by guessing is not what we had in mind.
We had in mind a program, an effective procedure, that inputs \( e \) and outputs an answer 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. That impression is wrong.

Imagine a utility for programmers, always\_halt, to patch non-halting programs. After writing a program \( \mathcal{P} \) we run it through the utility, which modifies the source so that for any input where \( \mathcal{P} \) fails to halt, the modified program will halt (and output 0) but the utility does not change any outputs where \( \mathcal{P} \) does halt. That would cause the list of all programs to give rise to a list of total functions like the one on 90, and diagonalization gives a contradiction.

Thus, halting, or rather failure to halt, is central to 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 \( \text{halt\_checker} \) were a computable function 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. These were studied as far back as Euclid and today we understand the form of all even perfect numbers. But we do not know if there are any odd perfect numbers.

With a solution to the Halting Problem we could settle that question. The program shown here searches for an odd perfect number.\(^\dagger\) If it finds one then it halts. If not then it does not halt. So giving \( \text{halt\_checker} \) the index of this program would settle whether there exists an odd perfect number. 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 \( \text{halt\_checker} \).

Before moving on, we note the appearance here of the theme of unbounded search. It does not just appear in looking for perfect numbers; we have seen it

\(^\dagger\) It takes an input \( x \) but ignores it; in this book we have a convention that we like Turing machines to all take an input and all give an output.
before, in defining $\mu$-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 for a stage at which the computation halts.

**General unsolvability** The Halting problem is part of a larger unsolvability phenomenon.

### 5.4 Example

Consider the problem of determining whether a given Turing machine halts on the input 3. That is, given $x$, does $\phi_x(3) \downarrow$?

$$\text{halts\_on\_three\_checker}(x) = \begin{cases} 1 & \text{if } P_x \text{ halts on input } 3 \\ 0 & \text{otherwise} \end{cases}$$

We will show that if $\text{halts\_on\_three\_checker}$ were a computable function then we could solve the Halting problem.

The plan is to create a system where being able to determine whether a program halts on 3 allows us to settle Halting problem questions. Consider the program below on the right. It reads the input $y$ and ignores it, and has a nominal output statement. The action is in the middle box, where it simulates running $P_x$ on input $x$. If that simulation halts then the program as a whole halts. If that simulation does not halt then this program as a whole does not halt. Thus, the program on the right halts on input $y = 3$ if and only if $P_x$ halts on $x$. So with this flowchart we could leverage being able to answer questions about halting on 3 to answer questions about whether $P_x$ halts on $x$.

$$\begin{align*}
\text{Start} & \\
\text{Read } x, y & \\
\text{Run } P_x \text{ on } x & \\
\text{Print 42} & \\
\text{End} & \\
\end{align*}$$

With that motivation we are ready for the argument. For the proof that $\text{halts\_on\_three\_checker}$ is not mechanically computable, assume otherwise. Consider this function.

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

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 $s$-$m$-$n$ 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. Each has a ‘Read $y$’ but no ‘Read $x$’; rather, the value used in the middle box is hard coded into each source.

As in the plan above, for all $x \in \mathbb{N}$ we have that $\phi_x(x) \downarrow$ if and only if $\text{halts\_on\_three\_checker}(s(e,x)) = 1$. We know that the $s$-$m$-$n$ function $s$ is computable so the supposition that $\text{halts\_on\_three\_checker}$ is also computable gives that the composition $\text{halts\_on\_three\_checker} \circ s$ is computable, which in turn gives that the Halting problem is computable, which is false. We conclude that $\text{halts\_on\_three\_checker}$ is not mechanically computable.

**5.5 Example** Show that this function is not mechanically computable: given $x$, determine whether $\mathcal{P}_x$ outputs a 7 for any input.

$$\text{outputs\_seven\_checker}(x) = \begin{cases} 1 & \text{if } \mathcal{P}_x \text{ outputs 7 on some input} \\ 0 & \text{otherwise} \end{cases}$$

Assume otherwise, that it is a computable function, and consider this.

$$\psi(x, y) = \begin{cases} 7 \quad & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

It is computed by the flowchart on the left. Because it is intuitively mechanically computable, Church’s Thesis gives 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 $x$. On the right is a flowchart for an associated machine; again, $x$ is hard-coded so
there is a different machine for each value of $x$. Then, $\phi_x(x) \downarrow$ if and only if \text{outputs\_seven\_checker}(s(e, x)) = 1. If \text{outputs\_seven\_checker} is computable then the composition \text{outputs\_seven\_checker} \circ s is computable, so the Halting problem is computably solvable, which it isn’t. So \text{outputs\_seven\_checker} is not computable.

5.6 Example  Show that this problem is unsolvable: given $x$, determining whether $\phi_x$ doubles its input, that is, $\phi_x(y) = 2y$ for all $y$. We want to show that this function is not mechanically computable.

$$\text{doubler\_checker}(e) = \begin{cases} 
1 & \text{if } P_e \text{ outputs } 2x \text{ for all input } x \\
0 & \text{otherwise}
\end{cases}$$

Assume otherwise. This function

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

is computed by the flowchart on the left below. Because $\psi$ is intuitively mechanically computable, Church’s Thesis says that there a Turing machine that computes it. This Turing machine has some index, $e$, so that $\psi = \phi_e$.

Apply the s-m-n theorem to get a family of functions $\phi_{s(e, x)}$, associated with the flowchart on the right, parametrized by $x$. Then $\phi_x(x) \downarrow$ if and only if \text{outputs\_seven\_checker}(s(e, x)) = 1. So the supposition that \text{doubler\_checker} is computable gives that the Halting problem is computable, which is wrong.

These examples show that one of the roles that the Halting problem plays is that it is a touchstone for demonstrating algorithmic unsolvability. Most proofs that something is unsolvable involve showing that if we could solve the given problem then we could solve the Halting problem, but we cannot.

We close with two comments. The first is to reiterate that when we say that a problem is unsolvable we mean that it is unsolvable by a mechanism, that there is no Turing machine that can compute the solution to the problem. There can be a function that solves it but no computable function that solves it.

The other comment is some intuition. There certainly are problems that start with “Given an index $x$” that we can solve. Just to list one: given $e$ decide if one
instruction in $\mathcal{P}_e$ is $q_0 BL q_1$. But that problem has a different character than the ones above. The problems above are about the behavior of the computed function — each is about $\phi_e$ rather than $\mathcal{P}_e$. This echoes the opening of the first chapter, which says that we are more interested in the input-output behavior of the machines than in their implementation details. The next subsection expands.

**Rice’s Theorem** The intuition from the unsolvability examples is that we cannot, given $x$, mechanically analyze the behavior of $\mathcal{P}_x$. We now make that intuition precise. These two definitions formalize ‘behavior’.

5.7 **Definition** Two computable functions have the same behavior $\phi_e \simeq \hat{\phi}_e$ if they converge on the same inputs $x \in \mathbb{N}$ and, when they do converge, they have the same outputs.

5.8 **Definition** A set $\mathcal{I}$ of natural numbers is an index set when for all indices $e, \hat{e} \in \mathbb{N}$, if $e \in \mathcal{I}$ and $\phi_e \simeq \hat{\phi}_e$ then also $\hat{e} \in \mathcal{I}$.

5.9 **Example** This set of indices $\mathcal{I} = \{ e \in \mathbb{N} \mid \phi_e(x) = 2x \text{ for all } x \}$ is an index set. Suppose that $e \in \mathcal{I}$ and that $\hat{e} \in \mathbb{N}$ is such that $\phi_e \simeq \hat{\phi}_e$. Then the behavior of $\hat{\phi}_e$ is also to double its input: $\hat{\phi}_e(x) = 2x$ for all $x$. Thus $\hat{e} \in \mathcal{I}$ also.

5.10 **Example** 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 $\phi_e \simeq \hat{\phi}_e$ 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 $\phi_e \simeq \hat{\phi}_e$ we know that either $\phi_e(x) = 3x$ for all $x$ or $\phi_e(x) = x^3$ for all $x$, and thus $\hat{e} \in \mathcal{J}$.

5.11 **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.

5.12 **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 5.31.

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 $\phi_{e_0} \neq \phi_{e_1}$ and 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 $\mathcal{P}_e$. Apply the $s$-$m$-$n$ theorem to parametrize $x$, resulting in the uniformly computable family of functions $\hat{\phi}_{s(e,x)}$, whose behavior is charted on the right.
We've constructed the functions on the right so that if $\phi_x(x) \uparrow$ then $\phi_{s(e,x)} \simeq \phi_{e_0}$ and thus $s(e,x) \notin I$, and also so that if $\phi_x(x) \downarrow$ then $\phi_{s(e,x)} \simeq \phi_{e_1}$ and thus $s(e,x) \in I$. Therefore being able to check whether an index $s(e,x)$ is an element of $I$ would allow us to solve the Halting problem.

5.13 Example Show using Rice’s Theorem that this problem is unsolvable: given $e$, decide if $\phi_e(3) \downarrow$.

Consider the set $I = \{ e \in \mathbb{N} \mid \phi_e(3) \downarrow \}$. To apply Rice’s Theorem we must show that this set is not empty, that it is not all of $\mathbb{N}$, and that it is an index set. This set is not empty because we can write a Turing machine that acts as the identity function $\phi(x) = x$, and if $e_0$ is the index of that Turing Machine then $e_0 \in I$. The set is not all of $\mathbb{N}$ because, where $e_1$ is the index of a Turing Machine that never halts, we have that $e_1 \notin I$.

To show that $I$ is an index set, assume that $e \in I$ and let $\hat{e} \in \mathbb{N}$ be such that $\phi_e \simeq \phi_{\hat{e}}$. Then $e \in I$ gives that $\phi_e(3) \downarrow$ and $\phi_e \simeq \phi_{\hat{e}}$ gives that $\phi_{\hat{e}}(3) \downarrow$ also. Hence $\hat{e} \in I$, and it is an index set.

5.14 Example Show that this problem is unsolvable: given $e$, decide if $\phi_e(x) = 7$ for some $x$.

We will show that $I = \{ e \in \mathbb{N} \mid \phi_e(x) = 7 \text{ for some } x \}$ is a nontrivial index set. It is not empty because, where $e_0$ is the index of a Turing Machine that acts as the identity function $\phi_{e_0}(x) = x$, the input $x = 7$ produces the output 7 so $e_0 \in I$. This set is not all of $\mathbb{N}$ because, where $e_1$ is the index of a Turing Machine that never halts, $e_1 \notin I$. So this set is nontrivial.

Showing that $I$ is an index set is straightforward. Assume that $e \in I$ 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, $\phi_{\hat{e}}(x_0) = 7$ also. Consequently, $\hat{e} \in I$.

We close by reflecting on the significance of Rice’s Theorem. Rice’s Theorem speaks to the behavior of the machines. It addresses the properties of the functions associated with the machines. But it does not speak about the properties of the machines that aren’t about the functions; for example, the set of functions computed by C programs whose 100-th character is a ‘k’ would not be an index set.

Thus, Rice’s Theorem is about understanding what the machines do. Remember that we declared, in the first paragraph of the first chapter of this book, that we are more interested in what the machines do than in their internal construction.

At the start of this chapter we saw by a counting argument that unsolvable
Section 5. Unsolvability

problems exist. Later the Halting problem showed us that there are some interesting problems that are unsolvable, some problems that are significant for the kinds of things we want to compute in practice. Here we saw the definition of index set, which is the natural way to encapsulate a behavior of interest, and Rice's Theorem says that for every nontrivial index set, computing the characteristic function is unsolvable. In this sense, Rice’s Theorem says that every interesting problem is unsolvable.

Now, the prior paragraph is an overstatement; we’ve all seen and written real-world programs with behaviors that interested us. Nonetheless, Rice’s Theorem is especially significant for an understanding what can be done mechanically.

II.5 Exercises

✓ 5.15 True or false?
   (a) The problem of determining, given $e$, whether $\phi_e(3)\downarrow$ is unsolvable because no function halts_on_three_checker exists.
   (b) The existence of unsolvable problems indicates weaknesses in the models of computation, and we need stronger models.
   (c) Rice’s Theorem says that every nontrivial problem is unsolvable.

✓ 5.16 Show that each of these is an unsolvable problem by showing that solving it would enable us to solve the Halting problem.
   (a) The problem of deciding if a function is the squaring function $x \mapsto x^2$.
   (b) The problem of determining if a function returns the same value on two consecutive inputs, so $f(x) = f(x + 1)$ for some $x \in \mathbb{N}$.
   (c) The problem of determining if a function is total, that is, converges on every input.

✓ 5.17 Show that each of these is an unsolvable problem by showing that solving it would enable us to solve the Halting problem.
   (a) The problem of determining if a function is partial, that is, fails to converge on some input.
   (b) The problem of deciding if the function ever converges, on any input.

5.18 Show that each of these is an unsolvable problem.
   (a) Given an index $e$, decide if the function $\phi_e$ computed by machine $P_e$ is the function $x \mapsto x + 1$.
   (b) Given an index $e$, decide if the function $\phi_e$ fails to converge on input 5.
   (c) Given an index $e$, decide if the function $\phi_e$ fails to converge on successive inputs $x$ and $x + 1$.

✓ 5.19 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 $P_e$.
   (b) Given $e$, determine whether $P_e$ halts when the input is the empty string.
   (c) Given $e$, determine if $P_e$ halts on input $e$ within one hundred steps.
5.20 Show that each of these is an unsolvable problem by applying Rice’s Theorem.
(a) The problem of deciding if a function is the squaring function \( x \mapsto x^2 \).
(b) The problem of determining if a function returns the same value on two consecutive inputs, so \( f(x) = f(x + 1) \) for some \( x \in \mathbb{N} \).
(c) The problem of determining if a function is total, that is, converges on every input.
(d) The problem of determining if a function is partial, that is, fails to converge on some input.

5.21 For each problem in Exercise 5.16, show it is unsolvable using Rice’s Theorem.

5.22 Is \( \{ e \mid P_e \text{ runs for at least 100 steps on input 5} \} \) an index set?

5.23 Show that each problem is unsolvable, using Rice’s Theorem.
(a) The problem of deciding, given \( e \in \mathbb{N} \), whether \( P_e \) accepts an infinite language.
(b) The problem of deciding, given \( e \in \mathbb{N} \), whether \( P_e \) accepts the string 101.

5.24 Give an example of a computable function, which is total, whose range is not computable.

5.25 Fix integers \( a, b, c \in \mathbb{Z} \). Consider the problem of determining, given \( \langle x, y \rangle \), whether \( ax + by = c \). Is that problem solvable or unsolvable?

5.26 (a) Show the problem of determining, given \( e \), whether \( P_e \) only halts on an empty input tape is unsolvable. (b) What’s wrong with this argument the other way, “Give it an empty input and see if it goes on”?

5.27 Give a trivial index set: fill in the blank \( I = \{ e \mid \text{_____} e \text{_____} \} \) so that the set is empty.

5.28 As described in Example 5.11, produce two Turing machines giving an output of 0 for all inputs, and such that and one machine contains an instruction starting with \( q_{10} \) while the other does not.

5.29 Show that each of these is an index set.
(a) \( \{ e \in \mathbb{N} \mid \text{machine } P_e \text{ halts on at least five inputs} \} \)
(b) \( \{ e \in \mathbb{N} \mid \text{the language accepted by } P_e \text{ is the empty language} \} \)
(c) \( \{ e \in \mathbb{N} \mid \text{the function } \phi_e \text{ is one-to-one} \} \)
(d) \( \{ e \in \mathbb{N} \mid \text{the function } \phi_e \text{ is either total or one-to-one} \} \)

5.30 Consider the relation \( \simeq \) between natural numbers given by \( e \simeq \hat{e} \) if \( \phi_e \simeq \phi_{\hat{e}} \).
(a) Show that it is an equivalence.
(b) Describe the equivalence classes.
(c) Show that each index set is the union of some of the equivalence class.

5.31 Do the \( e_0 \in I \) case in the proof of Rice’s Theorem, Theorem 5.12.

5.32 Show this problem is mechanically unsolvable: give \( e \), is there an input \( x \) so that \( \phi_e(x) \downarrow \)?

5.33 Show that most problems are unsolvable by using a counting argument to show that there are uncountably many functions \( f : \mathbb{N} \to \mathbb{N} \) that are not computed
by any Turing machine, while the number of functions that are computable is countable.

5.34 Consider the class of all index sets. (A) Prove that it is closed under complementation. (B) Prove it is also closed under union. (C) Is it closed under intersection? If so prove that and if not then give a counterexample.

5.35 A set of bit strings is a **decidable language** if its characteristic function is computable. Show that the collection of decidable languages is closed under these operations. (A) Union (B) Intersection (C) Complement

### Section II.6 Computably enumerable sets

Recall from Definition 1.12 that we say a function \( f : \mathbb{N} \to S \) enumerates its range. The intuition is that we are outputting the elements of the set: \( f(0) \), then \( f(1) \), etc.

The natural attack on the Halting problem is to simulate running \( \mathcal{P}_0 \) on input 0 for a few steps, and then simulate \( \mathcal{P}_1 \) on input 1 for a few steps. Then run \( \mathcal{P}_0 \) on 0 for a more few steps, followed by \( \mathcal{P}_1 \) on 1 for a few more, and then \( \mathcal{P}_2 \) on 2 for a few steps. Keep cycling among the \( \mathcal{P}_e \) on \( e \) simulations, running each for a few steps before going on. Over time some of these will halt and so the elements of \( K \) will start to fill in. On computer systems this interleaving is called time-slicing but in theory discussions it is called dovetailing.

Why won't this scheme of gradual enumeration allow us to decide membership in \( K \)? The problem is with \( K \)'s complement. Enumeration will tell us what is in \( K \) — if \( e \in K \) then we will find that out eventually — but if \( e \notin K \) then the computation never halts and we just wait forever.

#### 6.1 Definition
A set of natural numbers is **decidable**, or **computable**, if its characteristic function is computable, if we can mechanically determine both membership and non-membership.

#### 6.2 Definition
A set is **computably enumerable** (or recursively enumerable, or c.e., or r.e.) if it is effectively listable, that is, if it is the set of outputs of some computable function \( W_e = \{ k \mid k = \phi_e(i) \text{ for some } i \in \mathbb{N} \} \).

Picture a stream of numbers generated by a computable function \( \phi_e(0), \phi_e(1), \phi_e(2), \ldots \), gradually filling out the set \( W_e \). Note that a computably enumerable set can be empty, if \( \phi_e \) never converges for any input.

A set \( S \) is computable if and only if there is a Turing machine that decides membership: given an input number \( x \), the machine decides either ‘yes’ or ‘no’ whether \( x \in S \). In contrast, with computably enumerable sets there is a Turing machine that decides ‘yes’ but there is no requirement that the machine also decides ‘no’. Restated, a set is computable if there is a Turing machine that recognizes both members and nonmembers, while a set is computably enumerable if there
is a Turing machine that recognizes members. Computably enumerable sets are sometimes called **semidecidable**.

One reason that the computably enumerable sets are interesting is philosophical; with spurred by Church’s Thesis we think that these are the only sets that we will, in a sense, ever know.

**6.3 Lemma** (a) If a set is computable then it is computably enumerable.
(b) A set is computable if and only if both it and its complement are computably enumerable.

**Proof** For the first item let the set \( S \subseteq \mathbb{N} \) be computable and we will produce an effective enumeration \( f : \mathbb{N} \to \mathbb{N} \). If \( S \) is finite then take \( f(0) = s_0, f(1) = s_1, \ldots f(n-1) = s_{n-1} \), and for any larger input \( m \geq n \), define \( f(m) \uparrow \). 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 \( 1 \in S, \ldots \) This unbounded search is effective because \( S \) is computable and it must halt because \( S \) is infinite. Similarly, define \( f(k) \) to 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. Since the complement of a computable set is also computable, the prior item also gives that \( S^c \) is computably enumerable.

For the other direction suppose that both \( S \) and \( S^c \) are computably enumerable, and that \( S \) is enumerated by \( f_S \) while \( S^c \) is enumerated by \( f_{S^c} \). Given an input \( x \), effectively determine which of the two sets contains \( x \) by dovetailing their enumerations. That is, first run the computation of \( f_S(0) \) for a step, and the computation of \( f_{S^c}(0) \) for a step. Then run the computations of \( f_S(0) \) and \( f_{S^c}(0) \) for a second step, etc. The sets are complements so eventually \( x \) is enumerated into one or the other.

**6.4 Lemma** 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 6.3 would imply that \( K \) is computable but it is not computable.

That result gives one more reason to be interested in computable enumerability sets, in that the 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 \( \{ e \mid \text{there is an } x \text{ so that } \phi_e(x) = 7 \} \).

**6.5 Lemma** A set is computably enumerable if and only if it is the domain of a partial computable function.

**Proof** Assume first that \( S \) is computably enumerable, so it is the range of a computable function \( f \). We will produce an effective \( g : \mathbb{N} \to \mathbb{N} \) that converges if and only if the input is an element of \( S \). Given the input \( x \in \mathbb{N} \), enumerate the outputs \( f(0), f(1), \ldots \), and wait for \( x \) to appear. If it does appear then halt and return some nominal value. If it never appears then this procedure never halts.
The implication in the other direction is more involved. Assume that \( S \) is the domain of the partial computable function \( g \). If \( S \) is empty then it is computably enumerable by definition. So assume that \( S \) is not empty and fix some \( \hat{s} \in S \). We will produce a computable function \( f \) whose range is \( S \). Given input \( n \in \mathbb{N} \), run the computations of \( g(0), g(1), \ldots, g(n) \) for \( n \)-many steps each. Possibly some of these computations halt. Define \( f(n) \) to be the least \( k \) so \( g(k) \) halts within \( n \) steps and such that \( k \notin \{ f(0), f(1), \ldots, f(n-1) \} \). If no such \( k \) exists then define \( f(n) = \hat{s} \).

If \( s \in S \) then eventually \( g(s) \) must converge, in some number \( n_s \) of steps. At that point the number \( s \) is queued for output by \( f \) in the sense that it will be enumerated by \( f \) as, at most, \( f(n_s + s) \).

II.6 Exercises

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

6.7 Give a function that enumerates each set (A) the prime numbers (B) the natural numbers where the digits come in non-increasing order (e.g., 531 or 5331 but not 513).

✓ 6.8 You got a quiz question to define computably enumerable. A friend of yours says they answered with, “A set that can be enumerated by a Turing machine but that is not computable.” Is your friend right?

6.9 One of these two is computable and the other is computably enumerable. Which is which?
   (A) \{ e \mid P_e \text{ halts on input 4 in less than twenty steps} \}
   (B) \{ e \mid P_e \text{ halts on input 4 in more than twenty steps} \}

✓ 6.10 Which of these sets is decidable, which is semidecidable but not decidable, and which is neither? Justify in one sentence.
   (A) The set of indices \( e \) such that \( P_e \) takes more than 100 steps on input 7.
   (B) The set of indices \( e \) such that \( P_e \) takes less than 100 steps on input 7.
   (C) The set of indices \( e \) such that \( P_e \) takes more than 100 steps on every input.
   (D) The set of indices \( e \) such that \( P_e \) takes less than 100 steps on all inputs.

✓ 6.11 Can there be a set such that the problem of determining membership in that set is unsolvable, and also the set is computably enumerable?

6.12 Short answer: for each set state whether it is decidable, semidecidable (but not decidable), or neither. Give a brief justification.
   (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.

✓ 6.13 You read someone online who says, “every countable set \( S \) is computably enumerable
enumerable because if \( f : \mathbb{N} \to S \) then you have the enumeration \( S \) as \( f(0), f(1), \ldots \)” Explain why that argument is wrong.

✓ 6.14 Name three sets that are computably enumerable but not computable.

6.15 Write a program to enumerate each set. (A) the even numbers (B) the perfect squares (C) the primes (D) the set \{5, 7, 11\}.

✓ 6.16 (A) Prove that every finite set is computably enumerable. (B) Write a program that takes in a finite set and returns a routine that enumerates the set.

6.17 (A) Show that the set \( A_5 = \{e \mid \phi_e(5)\downarrow\} \) is computably enumerable.
(B) Generalize the previous item to show that \( A_k = \{e \mid \phi_e(k)\downarrow\} \) is computably enumerable.

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

6.19 Suppose that there is a set \( A \) and a computable function \( f \) such that the set difference \( A - \text{range}(f) \) is finite. Show that \( A \) is computably enumerable.

✓ 6.20 Let the set \( K_0 \) be the set \( \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\} \).
(A) Show that \( K_0 \) 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.

6.21 This section starts by recalling that we say that a function \( f : \mathbb{N} \to S \) enumerates its range \( R \subseteq S \). Suppose that the set \( R \) is infinite and that the function is computable. Show that if the function is not total then we can produce a total \( \hat{f} : \mathbb{N} \to R \) that enumerates \( R \).

6.22 Write a program that will, given two enumerating programs, dovetail them.

6.23 A set is enumerable in nondecreasing order if there is a computable function whose range is the set, and that is nondecreasing: \( n \leq m \) implies that \( f(n) \leq f(m) \). Similarly a set is enumerable in increasing order if \( n < m \) implies \( f(n) < f(m) \).

(A) Prove that a set is computable and nonempty if and only if it is enumerable in nondecreasing order.
(B) Prove that a set is computable and infinite if and only if it is enumerable in increasing order.
(C) Prove that every infinite computably enumerable set has an infinite computable subset.

6.24 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.

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

6.26 Computable enumerability is a property of sets so we can consider its interaction with set operations. (A) Must a subset of a computably enumerable set
be computably enumerable? (b) Must the union of two computably enumerable sets be computably enumerable? (c) The intersection? (d) The complement?

6.27 Show that if $h$ is a partially computable function and $S$ is a computably enumerable set, then the inverse image $h^{-1}(S)$ is computably enumerable.

6.28 A set $S \subseteq \mathbb{N}^k$ is a projection of the set $T \subseteq \mathbb{N}^{k+1}$ if $S$ is the set of $k$-tuples $\langle x_0, \ldots, x_{k-1} \rangle$ such that there is an $i \in \mathbb{N}$ so that $\langle i, x_0, \ldots, x_{k-1} \rangle \in T$. Prove that the projection of a computably enumerable set is also computably enumerable.

**Section II.7 Oracles**

The Halting problem is so hard that it is unsolvable. It is 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 our idea is that one problem is harder than a second if solving the first would also give us a solution to the second.†

Under Church’s Thesis we interpret the unsolvability of the Halting problem to say that no mechanism can answer all questions about membership in $K$. So if we want to answer questions about sets 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 that to the Turing machine box we attach a device, an oracle, that acts as the characteristic function of a set. To see what could be computed if we could solve the Halting problem we can attach a $K$-oracle. It answers questions of the form, “Is $x \in K$; does $\phi_x(x)\downarrow$?” This is a black box, meaning that we don’t open it to see how it works.‡

†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. ‡Opening it would let out the magic smoke.
One way to formally define computation with an oracle \( A \subseteq \mathbb{N} \) extending the definition of Turing machines is: to the action symbols \( \mathsf{L} \) and \( \mathsf{R} \), add one more, \( \mathsf{O} \). When this symbol appears, where the read/write head is pointing to the leftmost 1 of a sequence of \( n \)-many 1’s, if \( n \in A \) then that sequence is replaced by a single 1 followed by \( n \)-many \( \emptyset \)'s, while if \( n \notin A \) then the sequence is replaced by \( n \)-many \( \emptyset \)'s. This is computation relative to the oracle \( A \).

In this section, instead of working with the details of the definition we will use Church’s Thesis. We imagine enhancing our programming language by introducing a Boolean function \( \text{oracle}(x) \).

\[
\text{(if (oracle } x) \\
\text{ ... )}
\]

Each program in the enhanced language is a string and we can list them effectively, for instance in ascending alphabetical order, so each program has an index. The index is source-equivalent, meaning that from an index we can compute the source associated with that index, and from a source we can find the index.

In the setup above, the Turing machine does not change if we change the oracle — if we have an \( A \) oracle box plugged in, and unplug it and replace it with a \( B \) oracle box, then the white box is unchanged. Of course, the values returned by \( \text{oracle}(x) \) may change, resulting in changes to the outcome of running the Turing machine with the oracle, the two-box system. But the Turing machine itself, what we think of as the program in the enhanced language, stays the same.

Thus, to specify the outcome of a relative computation, in addition to specifying which program we are using and which inputs, we must also specify the oracle set. That explains all the parameters in the notations for the oracle Turing machine, \( \mathcal{P}_c^A \) and for the outcome of the function computed relative to an oracle, \( \phi_e^A(x) \).

### Definition

If a function computed from \( A \) is the characteristic function of the set \( B \) then we say that \( B \) is \( A \)-computable, or that \( B \) is Turing reducible to \( A \) or that \( B \) reduces to \( A \), denoted \( B \leq_T A \).†

Think of the set \( B \) as being easier, or at least no harder, than \( A \). For instance, if \( E \) is the set of even numbers then \( E \leq_T K \).

†The terminology “\( B \) reduces to \( A \)” can at first seem reversed. The idea is that we can solve problem \( B \) by using a solution to \( A \). We use this phrase in other areas of Mathematics. For instance, we may say that the problem of finding the area under a curve reduces to the problem of antidifferentiation.
7.2 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, a set is computable if its characteristic function is computable. If the characteristic function is computable without reference to an oracle then it can be computed by an oracle machine, by ignoring the oracle. For the other direction, suppose that a characteristic function can be computed by reference to the empty set or any other computable oracle. Then it can be computed without reference to an oracle by replacing the oracle calls with computations.

The second item is clear. For the third, suppose that \( P^B_e \) computes the characteristic function of \( A \) and that \( P^C_e \) computes the characteristic function of \( B \). Then in the computation of \( A \) from \( B \) we can replace the \( B \)-oracle calls with calls to \( P^C_e \). That computes the characteristic function of \( A \) directly from \( C \).

7.3 Example Recall the problem of determining, given \( e \), whether \( P_e \) halts on input 3. It asks for a program that acts as the characteristic function of the set \( A = \{ e \mid P_e \text{ halts on 3} \} \). We will show that \( K \leq_T A \).

This is a reprise of Example 5.4. We consider the function \( \psi : \mathbb{N}^2 \to \mathbb{N} \) that returns 0 if \( \phi_x(x) \downarrow \) and that diverges otherwise. By Church’s Thesis there is a Turing machine whose input-output behavior is \( \psi \). Let that machine have index \( e \), so that \( \psi = \phi_e \). Apply the \( s-m-n \) theorem to parametrize \( x \), giving \( \phi_{s(e,x)} \). Observe that \( \phi_x(x) \downarrow \) if and only if \( A(s(e,x)) = 1 \). That computes \( K \) from an \( A \) oracle.

The Halting problem is to decide whether \( P_e \) halts on input \( e \). A person may perceive that a more natural problem is to decide whether \( P_e \) halts on input \( x \).

7.4 Definition \( K_0 = \{ \langle e, x \rangle \mid P_e \text{ halts on input } x \} \)

We will argue, though, that the two problems are equivalent. We will prove that the problems of computing \( K \) and of computing \( K_0 \) are inter-solvable, meaning that if you can solve the one then you can solve the other, and so your choice of problem is a matter of convenience and convention.

7.5 Definition Two sets \( A, B \) are Turing equivalent or \( T \)-equivalent, denoted \( A \equiv_T B \), if \( A \leq_T B \) and \( B \leq_T A \). For any set \( A \), the collection of sets Turing equivalent to \( A \) is the Turing degree of \( A \). The Turing degree of a set named \( A \) is denoted using the boldface lower case letter, here \( a \).

Showing that two sets are \( T \)-equivalent shows that two seemingly-different problems are actually versions of the same problem.

7.6 Theorem \( K \equiv_T K_0 \).

Proof For \( K \leq_T K_0 \) suppose that we have access to a \( K_0 \)-oracle. Since it can say whether \( P_e \) halts on \( x \) for any input \( x \), it can clearly say whether \( P_e \) halts on \( e \).

For the \( K_0 \leq_T K \) half, suppose that we have access to a \( K \) oracle. The problem
we must solve is: given a pair \( \langle e, x \rangle \), determine whether \( P_e \) halts on input \( x \).

Consider the program sketched on the left below; obviously it halts for all input triples exactly if \( \langle e, x \rangle \in K_0 \) (it doesn’t matter what it outputs). By Church’s Thesis there is a Turing machine implementing it; let it be machine \( \hat{P}_e \).

```
Start
Read e, x, /y.alt
Simulate P_e on input x
Output 0
End
```

Get the flowchart on the right by applying the s-m-n theorem to parametrize \( e \) and \( x \) (that is, the flowchart represents \( P_{s(\hat{e}, e, x)} \)). That flowchart represents a family of programs with infinitely many members, one for each pair \( \langle e, x \rangle \).

Now suppose that we are given a pair \( \langle e, x \rangle \) and consider the right-side flowchart for that pair. It either halts on all inputs \( y \) or fails to halt on all inputs, depending on whether \( \phi_e(x) \downarrow \). In particular, taking the input to be the number of this machine \( y = s(\hat{e}, e, x) \), we have that \( P_{s(\hat{e}, e, x)} \) halts on input \( s(\hat{e}, e, x) \) if and only if \( \phi_e(x) \downarrow \). So with a \( K \)-oracle, we can solve the \( K_0 \) problem.

**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 6.5 the computably enumerable sets are the columns of \( K_0 \).

\[
\{ y \mid \phi_e(y) \downarrow \} = \{ \langle e, y \rangle \in K_0 \mid y \in \mathbb{N} \}
\]

So \( W_e \leq_T K_0 \equiv_T K \).

**Theorem** The Relativized Halting problem, the problem of determining the characteristic function of \( K^K = \{ x \mid \phi_x(x) \downarrow \} \), is not computable from a \( K \)-oracle. That is, there is no index \( e \in \mathbb{N} \) such that \( \phi^K_e = K^K \).

**Proof** We adapt the proof that the Halting problem is unsolvable. Assume otherwise, that there is a mechanical computation relative to a \( K \)-oracle with this behavior.

\[
K^K(e) = \begin{cases} 
1 & \text{if } \phi^K_e(e) \downarrow \\
0 & \text{if } \phi^K_e(e) \uparrow 
\end{cases}
\]

Then the function below is also computable relative to a \( K \)-oracle. The flowchart illustrates \( f \)'s construction; it uses the above function for the branch.
Since $f$ is an oracle-computable function, it has an index. Let that index be $\hat{e}$, so that $f(x) = \phi_{\hat{e}}^K(x)$ for all $x$.

Now feed $f$ its own index—consider $f(\hat{e}) = \phi_{\hat{e}}^K(\hat{e})$. If it diverges then the first clause in the definition of $f$ gives that $f(\hat{e}) \downarrow$, which is a contradiction. If it converges then $f$'s second clause gives $f(\hat{e}) \uparrow$, which is also impossible. Either way, assuming that $f$ is computable relative to a $K$ oracle leads to a contradiction. We constructed $f$ by assuming only that the characteristic function of $K^K$ is $K$-computable, so that also is not true.

**7.9 Theorem** Any set $S$ is reducible to its relativized Halting problem, $S \leq^T K^S$.

**Proof** The flowchart on the left sketches a function that is intuitively mechanically computable, relative to an oracle $X$. So Church’s Thesis gives that there is an oracle Turing machine that computes it, $P^X_e$ for some index $e$. Apply the $s$-$m$-$n$ theorem to parametrize $x$, giving the uniformly computable family of machines $P^X_{s(e,x)}$ charted on the right.

The computations on the right do not use the input $y$, so that $\phi_{s(e,x)}^X(y) \downarrow$ if and only if $x \in X$. Then taking the oracle to be $S$ and the input to be $s(e,x)$ gives that $x \in A$ if and only if $\phi_{s(e,x)}^A(s(e,x)) \downarrow$, which holds if and only if $s(e,x) \in K^K$.

**7.10 Corollary** $K \leq^T K^K$, but $K^K \not\leq^T K$.

**Proof** This follows from the prior two results.

That answers the question posed at the start of this section. There are problems strictly harder than the Halting problem. One is the relativized Halting problem, that of computing the characteristic function of $K^K$.

**II.7 Exercises**
Recall the definition of decider from page 11.

✓ 7.11 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.

✓ 7.12 Both oracles and deciders take in a number and return, 0 or 1, whether that number is in the set. What's the difference?

✓ 7.13 Prove that $A \leq_T A^c$ for all $A \subseteq \mathbb{N}$.

✓ 7.14 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 $\phi^A(x) = \phi^B(x)$.

✓ 7.15 Prove that if $A \leq_T B$ and $B$ is computable then $A$ is computable.

✓ 7.16 Show that the Halting problem set $K$ reduces to each.
   (A) $\{ x \mid \mathcal{P}_x \text{ outputs a 7 for some input} \}$
   (B) $\{ x \mid \phi_x(y) = 2y \text{ for all input} \}$

7.17 Let $A$ and $B$ be sets. Produce a set $C$ so that $A \leq_T C$ and $B \leq_T C$.

✓ 7.18 Fix an oracle. Prove that the collection of sets computable from that oracle is countable.

7.19 Let $A \subseteq \mathbb{N}$.
   (A) Define when a set is recursively enumerable in $A$.
   (B) Show that $\emptyset$ is recursively enumerable in $A$ for all sets $A$.
   (C) Define $K^A$.
   (D) Show that $K^A$ is recursively enumerable in $A$.
   (E) Show that the relation $B \sim A$ if $B$ is recursively enumerable in $A$ is not transtive.

Section II.8 Fixed point theorem

Recall our first example of diagonalization, the proof that the set of real numbers is not countable, on page 75. We assume that there is an $f : \mathbb{N} \to \mathbb{R}$ and consider the table of inputs and outputs, as illustrated below.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$'s decimal expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>42.312704 ...</td>
</tr>
<tr>
<td>1</td>
<td>2.0100000 ...</td>
</tr>
<tr>
<td>2</td>
<td>1.4141592 ...</td>
</tr>
<tr>
<td>3</td>
<td>-20.9195919 ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Let a decimal representation of the number on row \( n \) be \( d_n = \hat{d}.d_{n,0}d_{n,1}d_{n,2} \ldots \) Go down the diagonal to the right of the decimal point to get the sequence of digits \( \langle d_{0,0}, d_{1,1}, d_{2,2}, \ldots \rangle \). With that sequence, construct a number \( z = 0.z_0z_1z_2 \ldots \) by making its \( n \)-th decimal place \( z_n \) 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 \( z = 0.1211 \ldots \) 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 the diagonalization fails then the transformation has a fixed point.

We will apply this to sequences of computable functions, \( \phi_{i_0}, \phi_{i_1}, \phi_{i_2}, \ldots \) We are interested in effectiveness so we don’t consider just arbitrary sequences of indices but instead take the indices to be computable, \( i_0, i_1, i_2 \ldots = \phi_e(0), \phi_e(1), \phi_e(2) \ldots \) for some \( e \), so we are considering sequence of this form.

\[
\phi_{\phi_e(0)}, \phi_{\phi_e(1)}, \phi_{\phi_e(2)} \ldots
\]

This table’s rows are the effective sequences of effective functions.

<table>
<thead>
<tr>
<th>( e )</th>
<th>( n = 0 )</th>
<th>( n = 1 )</th>
<th>( n = 2 )</th>
<th>( n = 3 )</th>
<th>( \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( \phi_{\phi_e(0)} )</td>
<td>( \phi_{\phi_e(1)} )</td>
<td>( \phi_{\phi_e(2)} )</td>
<td>( \phi_{\phi_e(3)} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( \phi_{\phi_1(0)} )</td>
<td>( \phi_{\phi_1(1)} )</td>
<td>( \phi_{\phi_1(2)} )</td>
<td>( \phi_{\phi_1(3)} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 2 )</td>
<td>( \phi_{\phi_2(0)} )</td>
<td>( \phi_{\phi_2(1)} )</td>
<td>( \phi_{\phi_2(2)} )</td>
<td>( \phi_{\phi_2(3)} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 3 )</td>
<td>( \phi_{\phi_3(0)} )</td>
<td>( \phi_{\phi_3(1)} )</td>
<td>( \phi_{\phi_3(2)} )</td>
<td>( \phi_{\phi_3(3)} )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

That is, each entry \( \phi_{\phi_e(n)} \) is a computable function. As part of row \( e \), it is a part of a sequence of computable functions where the elements of the sequence have indices that are computed, using \( \phi_e \).

The natural transformation is to use some computable function \( f \).

\[
\phi_x \overset{tf}{\longrightarrow} \phi_{f(x)}
\]

The next result shows that under this transformation diagonalization fails, because the transformed diagonal is a computable sequence of computable functions. Thus the transformation \( tf \) has a fixed point.

### 8.1 Theorem (Fixed Point Theorem, Kleene 1938)†

For any total computable function \( f \) there is a number \( k \) such that \( \phi_k = \phi_{f(k)} \).

†This is often known as the Recursion theorem but there is another widely used result of that name. Besides, this name is more descriptive so is the one that we will use in this book.
Proof In the array the diagonal is $\phi_{\psi_0}, \phi_{\psi_1}, \phi_{\psi_2}, \ldots$. The flowcharts below sketch the computation of these functions. On the left is a mechanical computation of $\phi_{\psi_e}(x)$. Church’s Thesis says that some Turing machine computes it; let that machine have index $e_0$. Apply the s-m-n theorem to get the right chart, which describes a family of machines parametrized by $e$ that compute $\phi_{s(e_0, e)}$, the $e$-th function on the diagonal.

The index $e_0$ is fixed so $s(e_0, e)$ is a function of one variable. Let $d(e) = s(e_0, e)$, so the diagonal functions are $\phi_{\psi_e}$. This function $d$ is computable and total.

Under $t_f$ those functions are transformed to $\phi_{f \circ d(e)}$. The composition $f \circ g$ is computable and total, since $f$ is specified as total.

As the flowchart indicates, this is a computable sequence of computable functions. Hence it is one of the table’s rows. Let it be row $v$, so that $\phi_{f \circ d(e)} = \phi_{\psi_v}$ for all $n$. Consider where the diagonal sequence $\phi_{g(e)}$ intersects that row.

$$\phi_{f \circ g(e)}(x) = \begin{cases} \phi_{f \circ \psi_e}(x) & \text{if } \psi_e \downarrow \\ & \text{otherwise} \end{cases}$$

The desired fixed point for $f$ is $k = g(v)$.

The power of this result is that it applies to any total computable function. As a consequence of that, it leads to a number of surprising results.

8.2 Corollary There is an index $e$ so that $W_e = \{ e \}$.

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 just the index of the left-hand program), we have a total computable function $f$ of one variable $f(m) = s(e_0,m)$ such that $W_f(x) = \{ x \}$ for all $x$. The Fixed Point Theorem gives a fixed point $m = e$, so $W_e = W_{f(e)} = \{ e \}$.

Restated, this result says that there is a Turing machine that halts only on one input, its index. Rephrased for rhetorical effect, its name is its behavior.

8.3 Remark Every Turing machine has some index number but why must there be a machine numbered in a way related to its behavior? At first glance it seems to be an accident of the particular numbering scheme. But it isn’t an accident, the corollary says that it must happen as long as the numbering is acceptable. These are deep results, showing very surprising and interesting behaviors exhibited by any sufficiently powerful computation system. We say more in the discussion below and in Extra D.

8.4 Corollary There is an $m \in \mathbb{N}$ such that $\phi_m(x) = m$ for all inputs $x$.

Proof As in the prior corollary, use Church’s Thesis and the $s$-$m$-$n$ theorem to construct this family of computable functions, parametrized by $m$.

$$\phi_{s(e_0,m)}(x) = \begin{cases} 0 & \text{if } x = m \\ \uparrow & \text{otherwise} \end{cases}$$

The function $f$ with action $m \mapsto s(e_0,m)$ is computable and total so it has a fixed point $e$, giving $\phi_e(x) = \phi_{f(e)}(x) = e$ for all $x$. 

The prior result gives a Turing machine that prints its own index. Since the index is source-equivalent, it raises the question of whether there is a program that self-reproduces, that prints its own source. The answer is that there is such a program; see Extra D.

Discussion The Fixed Point Theorem and its proof are often considered mysterious, or at any rate obscure. One aspect of the theorem that bears explication is how it employs the use-mention distinction.

Compare the sentence Atlantis is a mythical city to There are two a’s in “Atlantis”. In the first, ‘Atlantis’ refers to something. We say that the word is used; it has a value, it points to something. In the second sentence ‘Atlantis’ is not used to refer
to something, its value is itself. We say that it is mentioned. This difference is the reason for the second sentence’s quotation marks.†

We see a version of the use-mention distinction in computer programming, with pointers. The C language program below illustrates. The asterisk means that both \( x \) and \( y \) are pointers. The thing about a pointer is that while the compiler associates \( x \) and \( y \) with memory cells, each of which has room for an integer, we are not interested in the contents of these so much as we are interested in the contents of the memory cells that they name. The first diagram imagines that the compiler associates \( x \) with memory address 123 and \( y \) with 124. It also imagines that the contents of memory 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.‡ The key here: it does not put 42 in location 123, rather it puts 42 in the location referred to by the contents of 123. 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.

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,

---

† A version of this distinction comes up in programming books. In the sentence, “The number of players is players” the first ‘players’ refers to people while the second is the computer code, the syntactic entity. Quoting everything would be awkward and ugly so standard practice is to put the computer code in a typewriter font. We have done something like the same thing with the paragraph’s single-quoted “Atlantis” mentions. (This is quoted because it mentions a mention.) ‡ 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. Both allow a programmer to control the indirection that pointers provide.
the proof starts by taking \( g(e) \) to be the name of this procedure.\(^\dagger \)

\[
\phi_{g(e)}(x) = \phi_{s(e_0,e)}(x) = \begin{cases} 
\phi_{\phi_e(e)}(x) & \text{if } \phi_e(e) \downarrow \\
\uparrow & \text{otherwise}
\end{cases}
\]

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 \( P_e \) on input \( e \) and if it halts with output \( w \) then run \( P_w \) on input \( x \).” Shorter: “Produce \( \phi_e(e) \) and then do \( \phi_{\phi_e(e)}(x) \).”

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.8 Exercises**

8.6 Where \( f : \mathbb{N} \to \mathbb{N} \) is computable, show that there is a natural number \( n \) so that \( W_n = W_{f(n)} \).

8.7 Show there is an index \( e \) so that \( W_e = \{0, 1, \ldots, e\} \).

8.8 What conclusion can you draw about acceptable enumerations of Turing machines by applying the Fixed Point Theorem to each of these?

(a) the doubling function \( x \mapsto 2x \)
(b) the adds-five function \( x \mapsto x + 5 \)
(c) the squaring function \( x \mapsto x^2 \)
(d) the function that returns 0 for all inputs except \( x = 5 \), in which case it returns 1.

What if the function is constant, such as \( x \mapsto 42 \)?

✓ 8.9 Show there is an index \( e \) so that \( W_e \) is the set consisting of one element, the \( e \)-th prime number.

\(^\dagger \) Here, ‘name’ is an equivalent of ‘index’ that is meant to be evocative.
8.10 Let \( f : \mathbb{N} \rightarrow \mathbb{N} \) be the function whose effect on Turing machines is that \( P_{f(e)} \) is the same as \( P_e \) except that the final states and non-final states are interchanged. Give a fixed point.

8.11 Prove or disprove the existence of \( m \) with the given property.
- (A) \( W_m = \{ m^2 \} \)
- (B) \( W_m = 10^m \)
- (C) \( W_m = \mathbb{N} - \{ m \} \)
- (D) \( W_m = \{ x \mid \phi_m(x) \text{ diverges} \} \)

8.12 Corollary 8.2 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 \( \{ 2m \} \)?

8.13 Use the \( s\)-\( m\)-\( n \) Theorem to show that there is an effective procedure to go from \( e \) to the machine sketched by the flowchart on the left. Call that machine \( \phi_{P(e)} \).

Now argue that there is an effective procedure to go from \( e \) to the routine sketched on the right. Call that routine \( \phi_{f(e)} \). Let \( \hat{e} \) be a fixed point for \( f \). What is \( \phi_{\hat{e}} \)?

✓ 8.14 Extend the Fixed Point Theorem to show that any total computable \( f : \mathbb{N} \rightarrow \mathbb{N} \) has infinitely many distinct fixed points.

8.15 Prove that if \( f : \mathbb{N}^2 \rightarrow \mathbb{N} \) is a total computable function of two variables then there is an index \( e \) so that \( \phi_e(y) = f(e, y) \) for all \( y \in \mathbb{N} \).

8.16 Corollary 8.2 shows that there is an index \( e \) with \( W_e = \{ e \} \). Is there a set \( A = \{ e_0, e_1 \} \) with \( A = W_{e_0} = W_{e_1} \)?

8.17 Show that there is an increasing sequence of integers \( e_0 < e_1 < \cdots \) such that \( W_{e_i} = \{ e_{i+1} \} \).

8.18 Use the Fixed Point Theorem to prove that \( K \) is not an index set.

Extra II.A  A Mathematical Fable: Hilbert’s Hotel

Once upon a time there was an infinite hotel. The rooms were naturally numbered 0, 1, \ldots  One day, every room was occupied when someone new came. Could the hotel accommodate? The clerk hit on the idea of moving each guest up a room, that is, moving the guest in room \( n \) to room \( n + 1 \). With that, room 0 is empty. So this hotel always has space for a new guest, or a finite number of new guests.
Next a bus rolled in, with infinitely many people \( p_0, p_1, \ldots \). Again the clerk had an idea: 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 \( p_i \) goes in room \( 2i + 1 \) and everyone has a room.

Just then a convoy of buses rolled in, infinitely many of them, each with infinitely many people.

\[
B_0 = \{ p_{0,0}, p_{0,1}, \ldots \}, \quad B_1 = \{ p_{1,0}, p_{1,1}, \ldots \}, \quad \ldots
\]

By now the spirit of the thing was clear: move each current guest to a new room with twice the number, and the new people go into the odd-numbered rooms, in the breadth-first order that we use to count \( \mathbb{N} \times \mathbb{N} \).

After this experience the clerk could be forgiven for thinking that there is always room in the infinite hotel — that with a sufficiently clever method it can fit any set of guests at all. Restated, Hilbert’s story makes natural the guess that all infinite sets have the same cardinality. That guess is wrong. There is a set so large that if its members were people they could not all fit in the hotel. One such set is \( \mathbb{R} \).†

II.A Exercises

A.1 Imagine the hotel is empty. A hundred buses arrive, where bus \( B_i \) contains passengers \( b_{i,0}, b_{i,1}, \ldots \). Give a scheme for putting them in rooms.

A.2 Give a formula assigning each person from the bus convoy in the story to a room.

A.3 The hotel builds a parking lot. Each floor \( F_i \) has infinitely many spaces \( f_{i,0}, f_{i,1}, \ldots \). And, no surprise, there are infinitely many floors \( F_0, F_1, \ldots \). 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 build a new hotel, with a room for each \( r \in \mathbb{R} \). Do they now have a large enough number of rooms to cover every possible set of potential guests?

II.B Cantor in Code

The definitions of cardinality and countability do not require that the functions must be effective. In this section we effectivize, counting sets such as \( \{ 0, 1 \} \times \mathbb{N} \) and \( \mathbb{N} \times \mathbb{N} \) using functions that are mechanically computable. The most straightforward way to show that these functions can be computed is to exhibit code, so here it is.

†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, 0–2. 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.
Scheme’s `let` creates a local variable.

```scheme
(use numbers)

;; triangle-num return 1+2+3+...+n
(define (triangle-num n)
  (/ (* (+ n 1)
       n)
     2))

;; cantor Cantor number of the pair (x,y) of integers
(define (cantor x y)
  (let ((d (+ x y)))
    (+ (triangle-num d)
      x)))

Use this code in the natural way.

```scheme
#include "godelnumbering.scm"
; including godelnumbering.scm ...
(#> (cantor 1 2)
7)
```

We will need both the map and its inverse, which goes from the number to the pair. Here is the routine that inverts `cantor`. The `let*` variant allows us to compute the local variable `t` by using the local variable `d` computed before it, in the prior line.

```scheme
;; xy given the cantor number, return (x y)
(define (xy c)
  (let* ((d (diag-num c))
         (t (triangle-num d))
         (list (- c t)
               (- d (- c t)))))

This is a sample use.

```scheme
#include "godelnumbering.scm"
; including godelnumbering.scm ...
(#> (xy 7)
(1 2))
```

The `xy` routine depends on a `diag-num` to compute the number of the diagonal. For that, 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

\[
d = \frac{-1 + \sqrt{1 - 4 \cdot 1 \cdot (-2t)}}{2 \cdot 1} = \frac{-1 + \sqrt{1 + 8t}}{2}
\]

(of the ‘±’, we keep only the ‘+’ because the other root is negative). Not every pair corresponds to a triangle number so to find the number of the diagonal lying before the pair \((x, y)\) with cantor\((x, y) = c\), take the floor \(d = \lfloor(-1 + \sqrt{1 + 8c})/2\rfloor\).

†The code for `diag-num` has two implementation details of interest. One is that in Scheme the `floor` function returns a floating point number. We want `xy` to be the inverse of `cantor`, which inputs integers, so we want `diag-num` to return an integer. That explains the `inexact->exact` conversion. The second detail is that the code leads to numbers large enough to give floating point overflows. For instance, `(cantor-n 1 2 3 4 5 6 7)` returns `1.05590697087673e+55`. So the code shown has the naive version of `diag-num` commented out and instead uses a library for bignums, integers of unbounded size.
Extra B. Cantor in Code

```
(define (diag-num c)
  (let ((s (exact-integer-sqrt (+ 1 (* 8 c)))))
    (floor-quotient (- s 1) 2)))

Extending to triples is straightforward.

;;; cantor-3 number triples
(define (cantor-3 x0 x1 x2)
  (cantor x0 (cantor x1 x2)))

;; xy-3 Return the triple that gave (cantor-3 x0 x1 x2) => c
(define (xy-3 c)
  (cons (car (xy c))
        (xy (cadr (xy c))))))

Using those routines is also straightforward.

#;2> (cantor-3 1 2 3)
172
#;3> (xy-3 172)
(1 2 3)

Turing machines instructions are four tuples so we are interested in those.

;;; cantor-4 Number quads
(define (cantor-4 x0 x1 x2 x3)
  (cantor x0 (cantor-3 x1 x2 x3)))

;; xy-4 Un-number quads: give (x0 x1 x2 x3) so that (cantor-4 x0 x1 x2 x3) => c
(define (xy-4 c)
  (let ((pr (xy c)))
    (cons (car pr)
          (xy-3 (cadr pr)))))

What the heck, let's extend to tuples of any size. We don't need these but they are fun. The cantor-n routine takes a tuple of any length and outputs the Cantor number of that tuple. Also there is xy-arity that takes two inputs, the length of a tuple and its Cantor number, and produces the tuple.

;; These routines generalize: number any tuple, or find the tuple corresponding
;; to a number.
;; The only ugliness is that the empty tuple is unique, so there is only
;; one tuple of that arity.

;;; cantor-n number any-sized tuple
(define (cantor-n . args)
  (cond ((null? args) 0)
        ((= 1 (length args)) (car args))
        ((= 2 (length args)) (cantor (car args) (cadr args)))
        (else
         (cantor (car args) (apply cantor-n (cdr args))))))

;;; xy-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)
(define (xy-arity arity c)
  (cond ((= 0 arity)
         (if (= 0 c)
             '()
             (begin
              (display "ERROR: xy-arity with arity=0 requires c=0")
              (newline)
              #f)))
         ((= 1 arity) (list c)))
```
The \textit{xy-arity} routine is not uniform in that it covers only one arity at a time. Said another way, \textit{xy-arity} is not the inverse of \textit{cantor-n} in that we have to tell it the tuple’s arity.

To cover tuples of all lengths we define two matched routines, \textit{cantor-omega} and \textit{xy-omega} that communicate using a simple data structure, a pair where the first element is the length of the tuple and the second is the tuple’s cantor number. These two are correspondences between the natural numbers and the set of sequences of natural numbers. They are inverse.

\begin{verbatim}
;; cantor-omega encode the arity in the first component
(define (cantor-omega . tuple)
  (let ((arity (length tuple)))
    (cond ((= arity 0) (cantor 0 0))
          ((= arity 1) (cantor 0 (+ 1 (car tuple))))
          (else
           (let ((newtuple (list (- arity 1)
                               (apply cantor-n tuple))))
             (apply cantor newtuple))))))

;; xy-omega Inverse of cantor-omega
(define (xy-omega c)
  (let* ((pr (xy c))
         (a (car pr))
         (cantor-number (cadr pr)))
    (cond ((and (= a 0) (= cantor-number 0)) '())
          ((= a 0) (list (- cantor-number 1)))
          (else (xy-arity (+ 1 a) cantor-number)))))
\end{verbatim}

This shows their use.

\begin{verbatim}
#;4> (xy-omega 0)
  ()
#;5> (xy-omega 1)
  (0)
#;6> (xy-omega 2)
  (0 0)
#;7> (xy-omega 3)
  (1)
#;8> (xy-omega 4)
  (0 1)
#;9> (xy-omega 5)
  (0 0 0)
#;10> (cantor-omega 1 2 3 4)
  12693900784
#;11> (xy (cantor-omega 1 2 3 4))
  (4 159331)
#;12> (xy-omega (cantor-omega 1 2 3 4))
  (1 2 3 4)
\end{verbatim}

**Numbering Turing machines** We will define a correspondence between natural numbers and Turing machines via Scheme code.

We represent an instruction with four-tuple of natural numbers. In the code below this is a \textit{quad}. A Turing machine is then represented as a list, a \textit{quadlist}.  

To go from a number to a quadlist we first apply \( xy-\omega \). This turns the number into a sequence of numbers, below called a numlist. We convert each of its numbers into a quad to get a quadlist.

\[
\begin{align*}
\text{;; natural->quad Return the quad corresponding to the natural number} \\
\text{;; quad->natural Return the natural matching the quad} \\
\text{(define (natural->quad n)} \quad (xy-4 n) \\
\text{)(define (quad->natural q)} \quad (apply cantor-4 q) \\
\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{;; numlist->quadlist Convert list of naturals to list of corresponding quads} \\
\text{;; quadlist->numlist Convert list of quads to list of corresponding naturals} \\
\text{(define (numlist->quadlist nlist)} \quad (map natural->quad nlist) \\
\text{)(define (quadlist->numlist qlist)} \quad (map quad->natural qlist) \\
\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{;; get-nth-quadlist Get the quadlist with the given cantor number} \\
\text{(define (get-nth-quadlist n)} \quad (numlist->quadlist (xy-omega n)) \\
\text{)}
\end{align*}
\]

This illustrates. The last line associates the number 2558 with the three-element quadlist.

\[
\begin{align*}
\text{;;1> (natural->quad 1)} \\
\text{} \quad (0 0 0 1) \\
\text{;;2> (natural->quad 2)} \\
\text{} \quad (1 0 0 0) \\
\text{;;3> (natural->quad 3)} \\
\text{} \quad (0 1 0 0) \\
\text{;;4> (cantor-omega 3 2 1)} \
\text{2558} \\
\text{;;5> (get-nth-quadlist 2558)} \\
\text{} \quad ((0 1 0 0) (1 0 0 0) (0 0 0 1))
\end{align*}
\]

A Turing machine is a set so to check if a quadlist is a Turing machine we must check that no two quads are the same. In addition, a Turing machine is deterministic so we must check that different quad’s begin with a different first two numbers. The second condition implies the first so we check here only the second.

This routine checks for determinism by sorting the quadlist alphabetically, so that if there are two quad’s beginning with the same pair they will then be adjacent. Checking for adjacent quad’s with the same first two elements only requires walking once down the list.

\[
\begin{align*}
\text{;; quadlist-is-deterministic? Is the list of quads deterministic?} \\
\text{;; qlist list of length 4 lists of numbers} \\
\text{(define (quadlist-is-deterministic? qlist)} \quad (let ((sorted-qlist (sort qlist quad-less?))) \\
\text{)(quadlist-is-deterministic-helper sorted-qlist)))} \\
\text{;; quadlist-is-deterministic-helper look for adjacent quads that differ} \\
\text{;; sq sorted list of quads} \\
\text{(define (quadlist-is-deterministic-helper sq)} \quad (cond \\
\text{((null? sq) #t)} \\
\text{((= 1 (length sq)) #t)} \\
\text{((first-two-equal? (car sq) (cadr sq)) #f)} \\
\text{((else (quadlist-is-deterministic-helper (cdr sq))))}) \\
\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{;; quadlist-is-tm Decide if a quadlist is a Turing machine}
\end{align*}
\]
Chapter II. Background

We count Turing machines by brute force: we get the numlist’s in ascending order — the one whose Cantor number is 0, the one whose number is 1, etc. — and convert each to a quadlist. We test each to see if it is a Turing machine, and if so assign it the next index. This routine picks out the quadlist that is a Turing machine, and that corresponds to a numlist greater than or equal to the input argument.

With that, here is the function that takes in a Turing machine as a list of quads and finds a natural number index for that Turing machine, along with the inverse function, taking the machine to an index for its machine.

Use is straightforward. The last one takes a few seconds.

Earlier we saw a Turing machine simulator. We can translate the machines written here to the earlier format. That is, we can interpret each of the quad’s above \((a_0 \ a_1 \ a_2 \ a_3)\) as an instruction \((q_p, T_p, T_n, q_n)\).
(two (caddr q))
(three (cadddr q)))
(list (nat->inst-zero zero)
      (nat->inst-one one)
      (nat->inst-two two)
      (nat->inst-three three)))

(define (tminstruction->quad i)
  (let ((zero (car i))
        (one (cadr i))
        (two (caddr i))
        (three (cadddr i)))
    (list (inst->nat-zero zero)
          (inst->nat-one one)
          (inst->nat-two two)
          (inst->nat-three three))))

;; quadlist->instructionlist ql convert a quadlist to a list of instructions
(define (quadlist->instructionlist ql)
  (map quad->tminstruction ql))
(define (instructionlist->quadlist tm)
  (map tminstruction->quad tm))

These rely on helper routines. Handling the states is trivial: for instance, \(a_0 = 0\) and \(a_3 = 0\) translate to the state \(q_0\).

(define (nat->inst-zero i)
  i)
(define (inst->nat-zero i)
  i)

(define (nat->inst-three i)
  i)
(define (inst->nat-three i)
  i)

The present tape character \(T_p\) has three possibilities. It can be a blank, which we associate with \(a_1 = 0\). Second, for readability we allow lower case letters \(a–z\), which we associate with \(a_1 = 1\) through \(a_1 = 26\). Finally, for higher-numbered \(a_1\)’s we just punt and write them as natural numbers. For instance, \(a_1 = 27\) is associated with \(T_p = 0\).

(define ASCII-a (char->integer #\a))

(define (nat->inst-one i)
  (cond
    ((= i 0) #\B)
    ((and (> i 0) (<= i 26))
      (integer->char (+ (- i 1) ASCII-a)))
    (else (- i 27))))
(define (inst->nat-one i)
  (cond
    ((equal? i #\B) 0)
    ((char? i) (+ 1 (- (char->integer i) ASCII-a)))
    (else (+ i 27))))

Note Scheme’s notation for characters, for instance \#\a and \#\B represent the characters a and B.

The tape-next description \(T_n\) is much the same, except that it also can be L or R.
(define (nat->inst-two i)
  (cond
    ((= i 0) #\L)
    ((= i 1) #\R)
    ((= i 2) #\B)
    ((and (> i 2) (<= i 28))
      (integer->char (+ (- i 3) ASCII-a)))
    (else (- i 29))))

(define (inst->nat-two i)
  (cond
    ((equal? i #\L) 0)
    ((equal? i #\R) 1)
    ((equal? i #\B) 2)
    ((char? i) (+ 3 (- (char->integer i) ASCII-a)))
    (else (+ i 29))))

The machine here is simple; if started on a blank tape it writes an a and then halts in the next step.

#;1> (instructionlist->quadlist '((0 #\B #\a 0) (0 #\a #\a 1)))
((0 0 3 0) (0 1 3 1))

We could find its index number as here.

#;2> (godel (instructionlist->quadlist '((0 #\B #\a 0) (0 #\a #\a 1))))

The list (machine 0), (machine 1),... contains all the Turing machines.

II.B Exercises

B.1 The code for machine, the routine that inputs a natural number and produces the Turing machine corresponding to that number, is slow. Find how long it takes to produce $P_n$ for the numbers $n = 0, 100, ..., 700$. You can use, e.g., (time (machine 100)). Graph $n$ against the time.

B.2 What does Turing machine number 666 do? Does it halt on input 0? On input 666?

B.3 The set of Turing machines can be numbered in ways other than the one given here. One is to use the same coding of states and tape symbols but instead of leveraging Cantor’s correspondence, it uses the powers of primes to get the final index. For instance, the Turing machine $P = \{q_0B1q_0, q_011q_1\}$ has the two quad’s (0 0 4 0) and (0 3 4 1). We can take the index of $P$ to be the natural number $2^13^15^17^111^113^417^519^2$ (we add 1 to the exponents because if we did not then we could not tell whether the four-tuple (0,0,0,0) is one of the instructions). (A) What are some advantages and disadvantages of the two encodings? (B) Compute the index of the example $P$ under this encoding.

Extra

II.C The Halting problem in Wider Intellectual Culture

The Halting problem and related results are about limits. In the light of Church’s Thesis, they say that there are things that we can never do. To understand their
impact on the intellectual world outside mathematics as well as inside we must place them in their historical setting.

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 commentators to be an extented 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.‡

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.

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

† These statements are in the context of European general intellectual culture, which is the context in which early Theory of Computation results appeared. A broader view is outside our scope.‡ 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.
fully penetrate').† There was of course a range of opinion but the zeitgeist was that we could expect that any question would be settled, and perhaps soon.

But starting in the early 1900’s, that changed. Exhibit A is the picture to the right. That the modern mastery of mechanisms can have terrible effect became apparent to everyone during World War I, 1914–1918. Ten million military men died. Overall, seventeen million people died. With universal conscription, probably the men in this picture did not want to be here. They were killed by a man 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 then it didn’t matter how brave you were, or how strong you were, or how right was your cause—you died. All that stuff they had told you about your people and honor and god, that was bullshit. Suddenly the zeitgeist shifted to become that Pandora’s box had opened, that the world is not at all ordered, reasoned, or sensible.

In science, Michaelson’s assertion that physics was essentially a solved problem was destroyed by the discovery of radiation. This changed everything by bringing in quantum theory, which has at its heart that there are events that are completely random, that included Heisenberg’s uncertainty principle, and that of course brought with it the atom bomb.

With Einstein we see most directly the shift in wider intellectual culture away from a sense of unlimited progress. After experiments during a solar eclipse in 1919 provided strong support for his theories, Einstein became an overnight celebrity (“Einstein Theory Triumphs” was the headline in The New York Times). He was seen by the public 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 everyday perceptions break down: nothing can travel faster than light, time bends, and even the commonsense idea of two things happening at the same instant falls apart.

In the general culture there were many reflections of this sense of loss of certainty. For example, the generation of writers and artists who came of age in World War I—including Eliot, Fitzgerald, Hemingway, Pound, and Stein—became known as the Lost Generation. They expressed their experience through themes of alienation, isolation, and

† Below we will cite some things as turning points that occur before 1930; how can the times not line up? For one thing, cultural shifts always involve muddled timelines. Also, the zeitgeist is difficult to define and address so we can take it as a lagging view. Finally, in Mathematics the shift occurred decades before the world generally noticed. We can mark that shift with the announcement of Gödel’s Incompleteness Theorem. Remarkably, that announcement came at the same meeting as Hilbert’s speech, on the day before. Gödel was in the audience for Hilbert and he said to O Taussky-Todd, “He doesn’t get it.”
dismay at the corruption they saw around them. In music, composers such as Debussy, Mahler, and Strauss broke with the traditional expressive forms, in ways that were often hard for listeners to understand — Stravinsky’s *Rite of Spring* caused a near riot at its premiere in 1913. As for art, the painting here dramatically shows that visual artists also picked up on these themes.

In mathematics, much the same inversion of the standing order happened in 1930 with K Gödel’s announcement of the 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 that system, cannot be proved in that system. The theorem is clearly about what cannot be done — there are things that are true that we shall never prove. This statement of hard limits seemed to many observers to be especially striking in the field of mathematics.

Gödel’s Theorem is closely related to the Halting problem. In a mathematical proof, each step must be verifiable as either an axiom or as a deduction that is valid from the prior steps. So proving a mathematical theorem is a kind of computation. Thus, Gödel’s Theorem and other uncomputability results are in the same vein. To people at the time these results were deeply shocking, revolutionary. And while we work in an intellectual culture that has absorbed this shock, that is part of the background of what we all have come to expect, we must still recognize them as a bedrock.

**Extra D. Self Reproduction**

**II.D 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 simply expands while retaining its essential features (one head, two arms, etc.). Projecting backwards, they posited a *homunculus*, a small human-like figure that, when given the breath of life, swells to be a person.

One awkwardness with this hypothesis is that the person may become a parent. So inside each homunculus are its children? And inside them the grandchildren? That is, one problem is the potential infinite regress. Of course today we know that sperm and egg don’t contain future bodies, they

---

† 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, from the axioms you can in principle computably enumerate the theorems.
contain DNA, the code to create the future bodies.

**Paley’s watch** In 1802, W Paley gave an argument for the existence of a god using as data 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.

Paley continues, giving his strongest argument, that the most incredible thing in the natural world, that which distinguishes organic things, 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 watchlike 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 was a very influential text 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 people 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 because the car is less complex than the robot. In that line of reasoning, machines are only able to produce things that are less complex than themselves.

That contention is wrong. We see next that the Fixed Point Theorem gives us self-reproducing mechanisms.

**Quines** A quine is a program that outputs its own source code.†

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 name of the function. This function outputs its name. Since we can go effectively from the index \( m \) to the machine source, this

†The easiest such program finds its source file on the disk and prints it. That is cheating.
function knows its source, in a sense. Said another way, \( P_m \) is a program whose name is its behavior. This is self-reference; machine \( m \) names itself.

To make this more concrete we will step through the nitty-gritty of making a quine. We will use the C language since it is low-level.

A first try might include the source as a string within itself.

```c
main() {
    printf("main(){\n ... }\")
}
```

But is obviously naive; it has infinite regress since included in the string would be another strings, etc. This is the same problem the homoculus theory had.

A less naive approach, one that employs the thinking behind our discussion of the Fixed Point Theorem, is to mention the code before use.

```c
char *e="main(){printf(e);}";
main(){printf(e);};
```

Here is the printout.

```c
main(){printf(e);};
```

This idea is summarized by the sentence “quine ‘quine’”. Here, the verb to quine (invented by D Hofstadter) 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, if we quine ‘say’ then we get “say ‘say’”. Thus, if we quine ‘quine’, we get “quine ‘quine’,” so that the sentence “quine ‘quine’” is a quine. . . In this linguistic analogy, the verb “to quine,” plays the role of the code and “quine” in quotation marks plays the role of the data.

We are closer but not quite there yet. Ratcheting up this approach gives the next try.

```c
char *e="char*e="%c %s %c; %c main() {printf(e,34,e,34,10,10);}%c";
main(){printf(e,34,e,34,10,10);}
```

A quine is a fixed point of an execution environment, when the execution environment is viewed as a function. They are possible in any complete model of computation; the exercises ask for them in a few languages. (In some languages an empty source file is a valid program that produces no output but typically quine contests disallow that.)

**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.

---

Note that 10 is the ASCII encoding for newline and 34 is ASCII for a double quotation mark.
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.

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.D Exercises

D.1 Produce a Scheme quine.

D.2 Produce a Python quine.

D.3 Consider a Scheme function $\text{diag}$ 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 $\text{diag}("hello x world")$ returns $\text{hello 'hello x world' world}$. Show that $\text{print(diag('print(diag(x))'))}$ is a quine.

D.4 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.E 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 of that plan, 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 $D: \mathbb{N} \to \mathbb{N}$, 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 $P_e(e)$ for $D(n)$-many steps. If $P_e(e)$ 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.†

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

\[ \Delta(\text{state}, \text{tape symbol}) = (\text{state}, \text{tape symbol}, \text{head shift}). \]

**Busy Beaver is unsolvable** Write $\Sigma(n)$ for the largest number of 1’s that any $n$ state machine, when started on a blank tape, leaves on the tape after halting. Write $S(n)$ for the most moves, that is, transitions.

Why isn’t $\Sigma$ 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.

**E.1 Theorem (Radó, 1962)** The function $\Sigma$ is not computable.

*Proof* Let $f : \mathbb{N} \to \mathbb{N}$ 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 $M_j$ having $j$ many states that writes $j$-many 1’s to a blank tape. For instance, here is $M_4$.

![Diagram of Turing Machine M_4](image)

†Note the connection with the Ackermann function: we showed that it is not primitive recursive because it grows faster than any primitive recursive function.
Also note that we can compose two Turing machines. The illustration below shows two machines on the left. On the right, we have combined the final states of the first machine with the start state of the second.

Let $F : \mathbb{N} \to \mathbb{N}$ be this function.

$$F(m) = (f(0) + 0^2) + (f(1) + 1^2) + (f(2) + 2^2) + \cdots + (f(m) + m^2)$$

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 $M_F$ that computes it. Let that machine have $n_F$ many states.

Now consider the Turing machine $P$ that performs $M_j$ and follows that with the machine $M_F$, and then follows that with another copy of the machine $M_F$. 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.

**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 $\Sigma(3) = 6$.

**E.2 Example** This is the three state Busy Beaver machine.

```
<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1, 1, R$</td>
<td>$q_4, 1, R$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_2, 0, R$</td>
<td>$q_1, 1, R$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3, 1, L$</td>
<td>$q_0, 1, L$</td>
</tr>
</tbody>
</table>
```

In 1983 A Brady showed that $\Sigma(4) = 107$. As to $\Sigma(5)$, even today no one knows.

Here are the current world records.

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma(n)$</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>13</td>
<td>$\geq 4098$</td>
<td>$\geq 1.29 \times 10^{865}$</td>
</tr>
<tr>
<td>$S(n)$</td>
<td>1</td>
<td>6</td>
<td>21</td>
<td>107</td>
<td>$\geq 47 176 870$</td>
<td>$\geq 3 \times 10^{1730}$</td>
</tr>
</tbody>
</table>

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.E Exercises

✓ E.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.

✓ E.4 (a) How many Turing machines with tape alphabet $\{B, 1\}$ are there having one state? (b) Two? (c) How many with $n$ states?

E.5 How many Turing machines are there, with a tape alphabet $\Sigma$ of $n$ characters and having $n$ states?

E.6 Show that there are noncomputable function that grow slower than some computable function. *Hint:* There are uncountably many functions with output in the set $\mathbb{B}$.

E.7 Give a diagonal construction of a function that is greater than any computable function.
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. In this book we take a symbol (sometimes called a token) to be an atomic unit that a machine can read and write.† An alphabet is a nonempty and finite set of symbols. A string over an alphabet is a sequence of symbols from that alphabet. We usually denote an alphabet with the upper case greek letter Σ although an exception is \( B = \{0, 1\} \), the alphabet from which we make bit strings. We use lower case greek letters such as \( \sigma \) and \( \tau \) to denote strings. We use \( \epsilon \) to denote the empty string, the length zero sequence of symbols. The set of all strings over \( \Sigma \) is \( \Sigma^* \).‡

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 bit strings that begin with 1 is \( \mathcal{L} = \{1, 10, 11, 100, \ldots\} \).

1.3  Example  Another language over \( B \) 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 \( \mathcal{L} = \{\epsilon, aab, aba, baa, aaaaabb, \ldots\} \).

1.5  Example  Let \( \Sigma = \{a, b, c\} \). The language of length-two strings over that alphabet is \( \mathcal{L} = \Sigma^2 = \{aa, ab, ba, \ldots, cc\} \). Over the same alphabet this is the language of length-three strings that are sorted in ascending order.

\[ \{aaa, aab, aac, abb, abc, acc, bbb, bbc, bcc, ccc\} \]

Image: The Tower of Babel, by Pieter Bruegel the Elder (1563) † We can imagine Turing’s clerk calculating without reading and writing symbols, 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 the restriction to readability and writeability is not essential. We include it in the definition of symbols because our mechanisms will use them. For this purpose, elephants would be an inconvenience. ‡ For more on strings see the Appendix on page 316.
1.6 Definition A palindrome is a string that reads the same forwards as backwards.

Some words from English that are plaindromes are ‘kayak’, ‘noon’, and ‘racecar’.

1.7 Example The language of palindromes over $\Sigma = \{a, b\}$ is $L = \{\sigma \in \Sigma^* \mid \sigma = \sigma^R\}$. A few members are abba, aaabaaa, and a.

1.8 Example Let $\Sigma = \{a, b, c\}$. 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.

$$L = \{a^i b^j c^k \in \Sigma^* \mid i, j, k \in \mathbb{N} \text{ and } i^2 + j^2 = k^2 \}$$

$$= \{a a a b b b b b c c c c c c c \ldots \}$$

1.9 Example The empty set is a language $L = \{\}$ over any alphabet. So is the set whose single element is the empty string $\hat{L} = \{\varepsilon\}$. 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; an 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 $P_e$ be a Turing machines, using the input alphabet $\Sigma = \{B, 1\}$. The set of strings $L_e = \{\sigma \in \Sigma^* \mid P_e \text{ halts on input } \sigma\}$ is a language. The collection of all such languages, of the $L_e$ 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 $\{a\}^*$ with a language over $\{b\}^*$ 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 $\Sigma_0 \cup \Sigma_1$. A similar thing happens for intersection; see Exercise 1.31.

Other operations on languages are extensions of operations on strings. We define the concatenation of languages to be the language of concatenations, etc.
1.13 **Definition (Operations on Languages)** The concatenation of languages \( L_0 \circ L_1 \) or \( L_0 \cdot L_1 \) is the set of concatenations, \( \{ \sigma_0 \circ \sigma_1 \mid \sigma_0 \in L_0 \) and \( \sigma_1 \in L_1 \} \).

For any language, the **power** \( L^k \) is the language consisting of the concatenation of \( k \)-many strings \( L^k = \{ \sigma_0 \circ \cdots \circ \sigma_{k-1} \mid \sigma_i \in L \} \). In particular, \( L^1 = L \) and \( L^0 = \{ \varepsilon \} \). The **Kleene star** of a language \( L^* \) is the language consisting of the concatenation of any number of strings.

\[
L^* = \{ \sigma_0 \circ \cdots \circ \sigma_{k-1} \mid k \in \mathbb{N} \text{ and } \sigma_0, \ldots, \sigma_{k-1} \in L \}
\]

This includes the concatenation of 0-many strings, so that \( \varepsilon \in L^* \).

The **reversal** \( L^R \) of a language \( L \) is the language of reversals, \( L^R = \{ \sigma^R \mid \sigma \in L \} \).

1.14 **Example** Where the language is the set of bit strings \( L = \{ 1000001, 1100001 \} \) then the reversal is \( L^R = \{ 1000001, 1000001 \} \).

1.15 **Example** If the language \( L \) consists of two strings \( \{ a, bc \} \) then the second power of that language is \( L^2 = \{ aa, abc, bca, bcba \} \). Its Kleene star is this.

\[
L^* = \{ \varepsilon, a, bc, aa, abc, bca, bcba, aaa, \ldots \}
\]

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.

A final comment about Kleene star. We could define the operation of repeatedly choosing strings from the language in two ways. We could choose a string \( \sigma \) from the language and then replicate, getting the set of \( \sigma^k \)'s. Or, we could repeat choosing strings from the language, getting \( \sigma_0 \circ \sigma_1 \circ \cdots \sigma_{k-1} \)'s. The second case is more useful and that's the definition of \( L^* \).

### III.1 Exercises

1.16 List five of the shortest strings in each language.

(A) \( \{ \sigma \in \mathbb{B}^* \mid \text{the number of 0's plus the number of 1's equals 3} \} \)

(B) \( \{ \sigma \in \mathbb{B}^* \mid \sigma \)'s first and last characters are equal \)

✓ 1.17 Is the set of real numbers a language? More precisely, is the set of decimal representations of real numbers a language?

✓ 1.18 Show that if \( \beta \) is a string then \( \beta^R \) is a palindrome. Do all palindromes have that form?

✓ 1.19 Let \( L_0 = \{ \varepsilon, a, aa, aaa \} \) and \( L_1 = \{ \varepsilon, b, bb, bbb \} \). (A) List all the members of \( L_0 \circ L_1 \). (B) List all the members of \( L_1 \circ L_0 \). (C) List all the members of \( L_0^2 \).

(d) List ten members, if there are ten, of \( L_0^* \).

✓ 1.20 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) \( \{ 1^n \theta^{n+1} \in \mathbb{B}^* \mid n \in \mathbb{N} \} \)

(D) \( \{ 1^n \theta^{2n} 1 \in \mathbb{B}^* \mid n \in \mathbb{N} \} \)
1.21 Where \( L = \{ a, ab \} \), compute each. (A) \( L^2 \) (B) \( L^3 \) (C) \( L^0 \)

\( L = \{ a, ab \} \) and \( L_1 = \{ b, bb \} \) find each. (A) \( L_0 \setminus L_1 \) (B) \( L_1 \setminus L_0 \) (C) \( L_0^2 \) (D) \( L_1^2 \) (E) \( L_0^2 \setminus L_1^2 \) (F) \( L_1^2 \setminus L_0^2 \)

1.22 Where \( L_0 = \{ a, ab \} \) and \( L_1 = \{ b, bb \} \) find each.

(a) \( L_0 \upharpoonright L_1 \) (b) \( L_1 \upharpoonright L_0 \) (c) \( L_0^2 \) (d) \( L_1^2 \) (e) \( L_0^2 \upharpoonright L_1^2 \) (f) \( L_1^2 \upharpoonright L_0^2 \)

1.23 Does \( L^* \) differ from \((L \cup \{ \epsilon \})^*\)?

1.24 Why is \( L_0 \) defined to be \( \{ \epsilon \} \)? Why not \( \{ \} \)?

1.25 Is the \( k \)-th power of a language the same as the language of \( k \)-th powers?

1.26 A person might have guessed that the definition of \( L_k \) would be \( \{ \sigma^k \mid \sigma \in L \} \). Is this set the same at the one defined in Definition 1.13? If so, prove it. If not, provide a counterexample.

1.27 We can ask how many elements are in \( L^2 \).

(a) Prove that where \( L \) has \( k \)-many elements, \( L^2 \) has at least \( k \)-many.
(b) Provide an example, for each \( k \in \mathbb{N} \), of a language that achieves this lower bound.
(c) Prove that where \( L \) has \( k \)-many elements, \( L^2 \) has at most \( k^2 \)-many.
(d) Provide an example, for each \( k \in \mathbb{N} \), of a language that achieves this upper bound.

1.28 Prove that \( L^* = L_0 \cup L_1 \cup L_2 \cup \cdots \).

1.29 What is the language that is the Kleene star of the empty set, \( \emptyset^* \)? Hint: it is not the empty language.

1.30 Consider the empty language \( L_0 = \emptyset \). For any language \( L_1 \), describe \( L_1 \setminus L_0 \).

1.31 We’ve observed that the union of a language over \( \Sigma_0 \) with a language over \( \Sigma_1 \) is a language over \( \Sigma_0 \cup \Sigma_1 \).

(a) Prove this.
(b) Formulate and prove the matching statement for intersection.

1.32 A language \( L \) over some \( \Sigma \) is finite if \( |L| < \infty \), so that there is some bound \( B \in \mathbb{N} \) where \( |\sigma| \leq B \) for all \( \sigma \in L \)

(a) If the language is finite must the alphabet be finite?
(b) Show that the class of finite languages is closed under finite union. That is, show that if \( L_0, \ldots, L_k \) are finite languages over a shared alphabet for some \( k \in \mathbb{N} \) then their union is also finite.
(c) Show that the class of finite languages is also closed under finite intersection and finite concatenation.
(d) Show that the class of finite languages is not closed under complementation or Kleene star.

1.33 What is the difference between the languages \( L = \{ \sigma \in \Sigma^* \mid \sigma = \sigma^R \} \) and \( \hat{L} = \{ \sigma^R \sigma \mid \sigma \in \Sigma^* \} \)?

1.34 For any language \( L \subseteq \Sigma^* \) we can form the set of prefixes.

\[
\text{Pref}(L) = \{ \tau \in \Sigma^* \mid \sigma \in L \text{ and } \tau \text{ is a prefix of } \sigma \}
\]

Where \( \Sigma = \{ a, b \} \) and \( L = \{ abaaba, bba \} \), find \( \text{Pref}(L) \).
1.35 Does $L_0^* = L_1^*$? You must either prove it or provide a counterexample.

1.36 Prove these for any alphabet $\Sigma$. (A) For any natural number $n$ the language $\Sigma^n$ is countable. (B) The language $\Sigma^*$ is countable.

1.37 Another way of defining the powers of a language is $L_0^* = \{\varepsilon\}$, and $L_{k+1}^* = L_k^* \cdot L$. Show this is equivalent to the one given in Definition 1.13.

1.38 Prove that if $L \cdot L = L$ then either $L = \emptyset$ or $\varepsilon \in L$.

1.39 The operations of languages form an algebraic system. Assume these languages are over the same alphabet.

(A) Show that language union and intersection are commutative: $L_0 \cup L_1 = L_1 \cup L_0$ and $L_0 \cap L_1 = L_1 \cap L_0$.

(B) Show that the language consisting of the empty string is the identity element with respect to language concatenation, so $L \cdot \{\varepsilon\} = L$ and $\{\varepsilon\} \cdot L = L$.

(C) Give an example showing that language concatenation is not commutative, that there are languages such that $L_0 \cdot L_1 \neq L_1 \cdot L_0$.

(D) Show that language concatenation is associative: $(L_0 \cdot L_1) \cdot L_2 = L_0 \cdot (L_1 \cdot L_2)$.

(E) Show that $(L_0 \cdot L_1)^R = L_1^R \cdot L_0^R$.

(F) Show that concatenation is left distributive over union: $(L_0 \cup L_1) \cdot L_2 = (L_0 \cdot L_2) \cup (L_1 \cdot L_2)$. Show that it is also right distributive.

(G) Show that the empty language is an annihilator for concatenation: $\emptyset \cdot L = L \cdot \emptyset = \emptyset$.

(H) Show that $\emptyset^* = \{\varepsilon\}$ and that $\{\varepsilon\}^* = \{\varepsilon\}$.

(I) Show that $(L^*)^* = L^*$.

Equation

**Section 2. Grammars**

Our formal definition of a language is that it is a set of strings. This allows for any willy-nilly set. But almost always in practice a language is 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 language usually also have strict syntax rules. A grammar is an analysis of the structure of a language, a set of rules.

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 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.

\[
\langle \text{sentence} \rangle \rightarrow \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle
\]
\[
\langle \text{noun phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle
\]
\[
\langle \text{verb phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle
\]
\[
\langle \text{verb phrase} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{noun phrase} \rangle
\]
\[
\langle \text{article} \rangle \rightarrow \text{the}
\]
\[
\langle \text{adjective} \rangle \rightarrow \text{young}
\]
\[
\langle \text{verb} \rangle \rightarrow \text{caught}
\]
\[
\langle \text{noun} \rangle \rightarrow \text{man} | \text{ball}
\]

Each line is a production or rewrite rule, each with an arrow, \( \rightarrow \).† On the left of each arrow is a head and on 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.‡

We make rules using two different kinds of components. The kind written in typewriter type, such as young, are from the alphabet \( \Sigma \) of the language. These are terminals. The kind written with angle brackets and in italics, such as \( \langle \text{article} \rangle \), are nonterminals. These act as variables and are used for intermediate steps.

The two symbols \( \rightarrow \) and \( | \) are neither terminals nor nonterminals. They are metacharacters, part of the syntax of the rules themselves.

The rewrite rules govern the derivation of strings in the language. Under the English grammar every derivation starts with \( \langle \text{sentence} \rangle \). Along the way, intermediate strings contain a mix of nonterminals and terminals. The rules all have a head with a single nonterminal so to derive the next string, pick a nonterminal in the present string and substitute an associated rule body.

\[
\langle \text{sentence} \rangle \Rightarrow \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle
\]
\[
\Rightarrow \langle \text{article} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle \langle \text{verb phrase} \rangle
\]
\[
\Rightarrow \text{the} \langle \text{adjective} \rangle \langle \text{noun} \rangle \langle \text{verb phrase} \rangle
\]
\[
\Rightarrow \text{the young} \langle \text{noun} \rangle \langle \text{verb phrase} \rangle
\]
\[
\Rightarrow \text{the young man} \langle \text{verb phrase} \rangle
\]
\[
\Rightarrow \text{the young man} \langle \text{verb} \rangle \langle \text{noun phrase} \rangle
\]
\[
\Rightarrow \text{the young man caught} \langle \text{noun phrase} \rangle
\]
\[
\Rightarrow \text{the young man caught the} \langle \text{noun} \rangle
\]
\[
\Rightarrow \text{the young man caught the ball}
\]

Note that the single line arrow \( \rightarrow \) is for rules while the double line \( \Rightarrow \) is for derivations.§

† Read the arrow aloud as “may produce,” or “may expand to,” or “may be constructed as.” § Read aloud as “or.” § Read \( \Rightarrow \) aloud as “derives” or “expands to.”
The above derivation always substitutes for the leftmost nonterminal so it is a leftmost derivation. However, in general we could substitute for any nonterminal. The derivation tree or parse tree is an alternative representation.†

2.2 Definition A context-free grammar, which in this book we will just call a grammar, is a four-tuple \( G = (\Sigma, N, S, P) \). 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.

2.3 Example This grammar describes algebraic expressions that involve only addition, multiplication, and parentheses.

\[
\begin{align*}
\langle expr \rangle & \rightarrow \langle term \rangle + \langle expr \rangle \mid \langle term \rangle \\
\langle term \rangle & \rightarrow \langle term \rangle * \langle factor \rangle \mid \langle factor \rangle \\
\langle factor \rangle & \rightarrow ( \langle expr \rangle ) \mid a \mid b \mid \ldots \mid z
\end{align*}
\]

Here is a derivation of the string \( x*(y+z) \).

\[
\begin{align*}
\langle expr \rangle & \Rightarrow \langle term \rangle \\
& \Rightarrow \langle term \rangle * \langle factor \rangle \\
& \Rightarrow \langle factor \rangle * \langle factor \rangle \\
& \Rightarrow x * \langle factor \rangle \\
& \Rightarrow x * ( \langle expr \rangle ) \\
& \Rightarrow x * ( \langle term \rangle + \langle expr \rangle ) \\
& \Rightarrow x * ( \langle term \rangle + \langle term \rangle ) \\
& \Rightarrow x * ( \langle factor \rangle + \langle term \rangle ) \\
& \Rightarrow x * ( \langle factor \rangle + \langle factor \rangle ) \\
& \Rightarrow x * ( y + \langle factor \rangle ) \\
& \Rightarrow x * ( y + z )
\end{align*}
\]

†The words ‘terminal’ and ‘nonterminal’ come from where the components lie in this tree.
In that example the rules for $\langle expr \rangle$ and $\langle term \rangle$ are recursive. This doesn’t mean that we get stuck in infinite regress. The question is not whether you could perversely keep expanding $\langle expr \rangle$ forever; the question is whether, given a string such as $x^*(y+z)$, you can find a terminating derivation.

In examples and exercises 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 $\langle sentence \rangle \rightarrow \langle noun phrase \rangle \langle verb phrase \rangle$ used above 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, $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaa\epsilon bbb \Rightarrow aaabbb$ is a derivation of $a^3b^3$. For this grammar, derivable strings have the form $a^n b^n$ for $n \in \mathbb{N}$.

We next give a complete description of how the production rules govern the derivations. Each rule has the form ‘head $\rightarrow$ body’, where the head consists of a single nonterminal. The body is a sequence of terminals and nonterminals, (such a string is a sentential form). Each step of a derivation has the form below, where $\tau_0$ and $\tau_1$ are sequences of terminals and non-terminals.

$$\tau_0 \backsim \text{head} \backsim \tau_1 \Rightarrow \tau_0 \backsim \text{body} \backsim \tau_1$$

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.†

This description is reminiscent of 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, shaping a derivation. However, one difference is that we have defined Turing machines as deterministic, so that from a given input string there is a determined sequence of

†This definition of rules and derivations suffices for this book but it is not the most general one. One more general definition allows heads of the form $\sigma_0X\sigma_1$, 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 with something followed by a then you can replace that string with a followed by that thing. For instance, in the derivation $S \Rightarrow aBSc \Rightarrow aBabc \Rightarrow aabBbcc \Rightarrow aabbc$ the third step uses (ii) and the fourth step uses (iii). Grammars with heads of the form $\sigma_0X\sigma_1$ 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 appear most often in practice, because it is a good balance of convenience and power.
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 beginning with the start symbol.

2.6 **Example** This grammar’s language is the set of strings representing natural numbers.

\[
\langle \text{natural} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\
\langle \text{digit} \rangle \rightarrow 0 \mid \ldots \mid 9
\]

This is a derivation for the string 321, along with its parse tree.

\[
\langle \text{natural} \rangle \Rightarrow \langle \text{digit} \rangle \langle \text{natural} \rangle \\
\Rightarrow 3 \langle \text{natural} \rangle \\
\Rightarrow 3 \langle \text{digit} \rangle \langle \text{natural} \rangle \\
\Rightarrow 32 \langle \text{natural} \rangle \\
\Rightarrow 32 \langle \text{digit} \rangle \\
\Rightarrow 321
\]

2.7 **Example** This grammar’s language is the set of strings representing natural numbers in unary.

\[
\langle \text{natural} \rangle \rightarrow \epsilon \mid 1 \langle \text{natural} \rangle
\]

2.8 **Example** Any finite language is derived from a grammar. This one gives the language of all length 2 bit strings, using the brute force approach of just listing all the member strings.

\[
S \rightarrow 00 \mid 01 \mid 10 \mid 11
\]

This grammar gives the length 3 bit strings, using a somewhat more refined approach than force.

\[
A \rightarrow 0B \mid 1B \\
B \rightarrow 0C \mid 1C \\
C \rightarrow 0 \mid 1
\]

2.9 **Example** For this grammar

\[
S \rightarrow aSb \mid T \mid U \\
T \rightarrow aS \mid a \\
U \rightarrow Sb \mid b
\]

the generated language is \( L = \{ a^i b^j \in \{a, b\}^* \mid i \neq 0 \text{ or } j \neq 0 \} \). One way to see that is to replace T and U by their expansions to get this.
This is the first example 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 $L$ and that it is also a superset. The rule that eliminates $T$ and $U$ shows that any derivation step $\tau_0 \leadsto \text{head} \tau_1 \Rightarrow \tau_0 \leadsto \text{body} \tau_1$ 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 $L$.

For containment the other way, we will prove that every $\sigma \in L$ has a derivation. We will use induction on the length $|\sigma|$. The smallest case, the base case, is $|\sigma| = 1$ by the definition of $L$. 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 $L$ of length $k = 1, \ldots, k = n$ has a derivation (where $n \geq 1$) 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 $S \Rightarrow \cdots \Rightarrow \hat{\sigma}$. Prefixing that derivation with a $S \Rightarrow aS$ step will put an additional $a$ on the left, as required: $S \Rightarrow aS \Rightarrow \cdots \Rightarrow \sigma$. The $j > 1$ case is similar, and the case of $\sigma = a^1 b^1$ 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.

$$
\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{else } \langle \text{stmt} \rangle
$$

Ambiguity appears when we give a derivation for this code line.

```plaintext
if enrolled(s) if studied(s) grade='P' else grade='F'
```

Here are the first two lines of one derivation

$$
\langle \text{stmt} \rangle \Rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \\
\quad \Rightarrow \text{if } \langle \text{bool} \rangle \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{else } \langle \text{stmt} \rangle
$$

and here are the first two of another.

$$
\langle \text{stmt} \rangle \Rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{else } \langle \text{stmt} \rangle \\
\quad \Rightarrow \text{if } \langle \text{bool} \rangle \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{else } \langle \text{stmt} \rangle
$$

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 the indented code snippets.

Obviously, those two will give different program behaviors. This is known as a dangling else.

2.11 Example This grammar for elementary algebra expressions

\[
\langle expr \rangle \rightarrow \langle expr \rangle + \langle expr \rangle \\
| \langle expr \rangle \cdot \langle expr \rangle \\
| ( \langle expr \rangle ) | a | b | \ldots | z
\]

is ambiguous because \(a+b\cdot c\) has two leftmost derivations.

\[
\langle expr \rangle \Rightarrow \langle expr \rangle + \langle expr \rangle \Rightarrow a + \langle expr \rangle \\
\Rightarrow a + \langle expr \rangle \cdot \langle expr \rangle \Rightarrow a + b \cdot \langle expr \rangle \Rightarrow a + b \cdot c
\]

\[
\langle expr \rangle \Rightarrow \langle expr \rangle \cdot \langle expr \rangle \Rightarrow \langle expr \rangle + \langle expr \rangle \cdot \langle expr \rangle \\
\Rightarrow a + \langle expr \rangle \cdot \langle expr \rangle \Rightarrow a + b \cdot \langle expr \rangle \Rightarrow a + b \cdot c
\]

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.

\[
\langle expr \rangle \rightarrow \langle expr \rangle + \langle term \rangle \\
\mid \langle term \rangle
\]
\[
\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle \ast \langle \text{factor} \rangle \\
| \langle \text{factor} \rangle \\
\langle \text{factor} \rangle \rightarrow ( \langle \text{expr} \rangle ) \\
| a | b | \ldots | z
\]

Choosing grammars that are not ambiguous is important in practice.

### III.2 Exercises

✓ 2.12 Use the grammar of Example 2.3. (A) What are the variables? (B) What is the start variable? (C) What are the terminals? (D) What are the nonterminals? (E) How many rewrite rules does it have? (F) Give three strings derived from the grammar, besides the string in the example. (G) Give three strings in the language \{ +, *, (, a ..., z) \} \ast that cannot be derived.

2.13 Use the grammar of Exercise 2.14. (A) What are the variables? (B) What is the start variable? (C) What are the terminals? (D) What are the nonterminals? (E) How many rewrite rules does it have? (F) Give three strings derived from the grammar besides the ones in the exercise, or show that there are not three such strings. (G) Give three strings in the language \{ +, *, (, a ..., z) \} \ast that cannot be derived from this grammar, or show there are not three such strings.

✓ 2.14 From this grammar

\[
\langle \text{sentence} \rangle \rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle \\
\langle \text{subject} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun1} \rangle \\
\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{direct-object} \rangle \\
\langle \text{direct-object} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun2} \rangle \\
\langle \text{article} \rangle \rightarrow \text{the} | a | \varepsilon \\
\langle \text{noun} \rangle \rightarrow \text{car} | \text{wall} \\
\langle \text{verb} \rangle \rightarrow \text{hit}
\]

derive each of these: (A) the car hit a wall (B) the car hit the wall (C) the wall hit a car.

2.15 In the language generated by this grammar.

\[
\langle \text{sentence} \rangle \rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle \\
\langle \text{subject} \rangle \rightarrow \langle \text{article} \rangle \langle \text{nouns} \rangle \\
\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{direct-object} \rangle \\
\langle \text{direct-object} \rangle \rightarrow \langle \text{article} \rangle \langle \text{nouns} \rangle \\
\langle \text{article} \rangle \rightarrow \text{the} | a | \varepsilon \\
\langle \text{nouns} \rangle \rightarrow \text{dog} | \text{flea} \\
\langle \text{nouns} \rangle \rightarrow \text{man} | \text{dog} \\
\langle \text{verb} \rangle \rightarrow \text{bites} | \text{licks}
\]
(A) Give a derivation for dog b ites man.
(B) Show that there is no derivation for man b ites dog.

✓ 2.16 Use this grammar

\[
\begin{align*}
S & \rightarrow TB
T & \rightarrow aT | \varepsilon
U & \rightarrow aT | bT | \varepsilon
\end{align*}
\]

for each part. (A) Give a derivation of abba. (B) Give one of baab. (C) Show that there is no derivation of aabab.

✓ 2.17 Give the parse tree for the derivation of Example 2.3.

2.18 Verify that the language derived from the grammar in Example 2.4 is \( L = \{ a^n b^n \mid n \in \mathbb{N} \} \).

2.19 What is the language generated by this grammar?

\[
\begin{align*}
A & \rightarrow aA | B \\
B & \rightarrow bB | cA
\end{align*}
\]

✓ 2.20 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.

2.21 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.

2.22 What is the language generated by a grammar with a set of production rules that is empty?

2.23 Create a grammar for each of these languages.

(A) the language of all character strings \( \mathcal{L} = \{ a, \ldots, z \}^* \)

(B) the language of strings of at least one digit \( \{ \sigma \in \{0, \ldots, 9\}^* \mid |\sigma| \geq 1 \} \)

✓ 2.24 This is a grammar for postal addresses. Note the use of the empty string \( \varepsilon \) to make \( \langle \text{opt suffix} \rangle \) optional.

\[
\begin{align*}
\langle \text{postal address} \rangle & \rightarrow \langle \text{name} \rangle \langle \text{street address} \rangle \langle \text{town} \rangle \\
\langle \text{name} \rangle & \rightarrow \langle \text{personal part} \rangle \langle \text{last name} \rangle \langle \text{opt suffix} \rangle \langle \text{EOL} \rangle \\
& \quad | \langle \text{personal part} \rangle \langle \text{name part} \rangle \langle \text{EOL} \rangle \\
\langle \text{personal part} \rangle & \rightarrow \langle \text{initial} \rangle \ . \ | \langle \text{first name} \rangle \\
\langle \text{street address} \rangle & \rightarrow \langle \text{house num} \rangle \langle \text{street name} \rangle \langle \text{apt num} \rangle \langle \text{EOL} \rangle \\
\langle \text{town} \rangle & \rightarrow \langle \text{town name} \rangle \ , \ \langle \text{state or region} \rangle \langle \text{EOL} \rangle \\
\langle \text{opt suffix} \rangle & \rightarrow \langle \text{Sr.} \rangle \ | \langle \text{Jr.} \rangle \ | \langle \text{roman numeral} \rangle \ | \varepsilon \\
\langle \text{apt num} \rangle & \rightarrow \langle \text{natural} \rangle \ | \varepsilon \\
\langle \text{initial} \rangle & \rightarrow \langle \text{char} \rangle
\end{align*}
\]
The nonterminal \( \langle EOL \rangle \) expands to whatever marks an end of line, while \( \langle \text{white space} \rangle \) expands to a space or tab.

(A) Give a derivation for this address.

Sherlock Holmes
221B Baker Street
London, UK

(B) Show that there is no derivation for this address.

President
1600 Pennsylvania Avenue
Washington, DC

Suggest a modification of the grammar to include this address.

(c) Give three reasons why this grammar is inadequate for general use. (Probably no reasonable-sized grammar would suffice that is less general than one that just accepts any character string; the other obvious possibility is the grammar that lists as separate rules every addresses in the world.)

✓ 2.25 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, \( \{1^{3n} \mid n \in \mathbb{N}\} \).

✓ 2.26 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.

2.27 Here is a grammar for LISP.

\[
\langle s \text{ expression} \rangle \rightarrow \langle \text{atomic symbol} \rangle \\
| ( \langle s \text{ expression} \rangle \cdot \langle s \text{ expression} \rangle ) \\
| \langle \text{list} \rangle
\]

\[
\langle \text{list-entries} \rangle \rightarrow \langle s \text{ expression} \rangle \\
| \langle s \text{ expression} \rangle \langle \text{list-entries} \rangle \\
\langle \text{list} \rangle \rightarrow ( \langle \text{list-entries} \rangle )
\]

\[
\langle \text{atomic symbol} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{atom part} \rangle
\]

\[
\langle \text{atom part} \rangle \rightarrow \langle \text{empty} \rangle \\
| \langle \text{letter} \rangle \langle \text{atom part} \rangle \\
| \langle \text{number} \rangle \langle \text{atom part} \rangle
\]
Here, ‘ ’ means a space character. Give a derivation for each string.

(a) \((a \cdot b)\)
(b) \((a \cdot (b \cdot c))\)
(c) \(((a \cdot b) \cdot c)\)
(d) \(((a \cdot b) \cdot (c \cdot b))\)

2.28 Using the Example 2.11’s unambiguous grammar, produce a derivation for \(a+(b*c)\).

2.29 Both of these are grammars for the language of bit strings \(L = \{0, 1\}^*\).

(A) Show that this is ambiguous.
\[
\langle \text{bit-string} \rangle \rightarrow 0 \mid 1 \mid \langle \text{bit-string} \rangle \langle \text{bit-string} \rangle
\]

(B) Show that this is unambiguous.
\[
\langle \text{bit-string} \rangle \rightarrow \langle \text{bit-string} \rangle 0 \mid \langle \text{bit-string} \rangle 1 \mid 0 \mid 1
\]

2.30 (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.
\[
E \rightarrow E - T \mid T
\]
\[
T \rightarrow a \mid b
\]

2.31 Use the grammar from the footnote on 144 to derive \(aaabbbccc\).

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. Graphs are so natural in this area that we have also already used them. Here are two examples.
Both have vertices, the circles and the symbols, that are connected by edges. The edges represent a relationship between the vertices.

**Definition** We start with the simplest type of graph.

3.1 **Definition** A simple graph is an ordered pair \( G = \langle V, E \rangle \) where \( V \) is a set of vertices or nodes and \( E \) is a set of edges. Each edge is a set of two distinct vertices.

3.2 **Example** This simple graph \( G \) has five vertices \( V = \{ v_0, \ldots, v_4 \} \) and eight edges.

\[
E = \{ \{ v_0, v_1 \}, \{ v_0, v_2 \}, \ldots, \{ v_3, v_4 \} \}
\]

Important: a graph is not its picture. A graph is about how the vertices are connected. Both of the pictures below show the same graph because they show the same vertices and the same connections.

Instead of writing \( e = \{ v, \hat{v} \} \) we often use a string \( e = v \hat{v} \). Since sets are unordered we could also write the same edge as \( e = \hat{v}v \).

There are many variations of Definition 3.1, used for modelling circumstances that a simple graph cannot cover. One variant allows a vertex to connect to itself, forming a loop. Another is a multigraph, which allows two vertices to have more than one edge between them. Another is a directed graph or digraph, where edges have a direction, as in a road map that includes one-way streets. Still another extension 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.

**Traversal** Many problems involve moving through a graph.

3.3 **Definition** Two graph edges are *adjacent* if they share a vertex, so that the first edge is \( u v \) and the second is \( v w \). A *walk* is a sequence of adjacent edges \( \langle u_0 v_1, v_1 v_2, \ldots, v_{n-1} v_n \rangle \). Its **length** is the number of edges, \( n \). If the initial vertex \( u_0 \) equals the final vertex \( v_n \) then it is a 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 that has at least one edge is a *cycle*. A *circuit* is a closed walk that either contains all of the edges, in which case it is an *Euler circuit*, or all of the vertices, making it a *Hamiltonian circuit*. A graph is *connected* if there is a path between any two vertices.
3.4 Example  On the left is a path from $u_0$ to $u_3$. (it is also an example of a trail and a walk). On the right is a Hamiltonian circuit.

3.5 Definition  Where $G = \langle V, E \rangle$ is a graph, a subgraph $\hat{G} = \langle \hat{V}, \hat{E} \rangle$ satisfies $\hat{V} \subseteq V$ and $\hat{E} \subseteq E$. A subgraph with every possible edge, with the property that for every $e = v_i v_j \in E$ such that $v_i, v_j \in \hat{V}$ then $e \in \hat{E}$, is an induced subgraph.

3.6 Example  In the graph $G$ from Example 3.4, consider the highlighted path \{$u_0 u_1, u_1 u_3$\}. Taking those edges, along with the vertices that they contain $\hat{E} = \{u_0, u_1, u_3\}$, gives a subgraph $\hat{G}$.

In the same graph, the induced subgraph involving the set of vertices \{$u_0, u_2, u_3$\} is the outer triangle.

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.

$$M(G) = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 & v_4 \\ v_0 & 0 & 1 & 1 & 0 & 0 \\ v_1 & 1 & 0 & 1 & 1 & 1 \\ v_2 & 1 & 1 & 0 & 1 & 1 \\ v_3 & 0 & 1 & 1 & 0 & 1 \\ v_4 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

3.7 Definition  For a graph $G$, the adjacency matrix $M(G)$ representing the graph has $i,j$ entries equal to the number of edges from $v_i$ to $v_j$.

This covers the graph variants listed earlier. For instance, the graph represented in (*) is simple because the matrix has only 0 and 1 entries, and because the matrix is symmetric (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 nonzero diagonal entry. If the graph 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.

3.8 Lemma  Let the matrix $M(G)$ represent the graph $G$. 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 $v_j$. 

**Colors** We sometimes partition a graph’s vertices.

3.9 **Definition** A \( k \)-coloring of a graph, for \( k \in \mathbb{N} \), is a partition of vertices to \( k \)-many classes such that adjacent vertices do not come from 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: every vertex is connected to every other vertex and there are four vertices. So for not two connected vertices to be the same color we must use four colors.

3.10 **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 no members must be in two places at once?

<table>
<thead>
<tr>
<th>Committee</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armis</td>
</tr>
<tr>
<td>Crump</td>
</tr>
<tr>
<td>Burke</td>
</tr>
<tr>
<td>India</td>
</tr>
<tr>
<td>Burke</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Edwards</td>
</tr>
<tr>
<td>Frank</td>
</tr>
<tr>
<td>Harris</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Smith</td>
</tr>
<tr>
<td>Robinson</td>
</tr>
<tr>
<td>Ke</td>
</tr>
<tr>
<td>Smith</td>
</tr>
<tr>
<td>Robinson</td>
</tr>
</tbody>
</table>

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.

**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

<table>
<thead>
<tr>
<th>Vertex on left</th>
<th>v_0</th>
<th>v_1</th>
<th>v_2</th>
<th>v_3</th>
<th>v_4</th>
<th>v_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex on right</td>
<td>w_0</td>
<td>w_2</td>
<td>w_4</td>
<td>w_1</td>
<td>w_3</td>
<td>w_5</td>
</tr>
</tbody>
</table>

then as a consequence the edges also correspond.

<table>
<thead>
<tr>
<th>Edge on left</th>
<th>{v_0, v_3}</th>
<th>{v_0, v_4}</th>
<th>{v_0, v_5}</th>
<th>{v_1, v_3}</th>
<th>{v_1, v_4}</th>
<th>{v_1, v_5}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge on right</td>
<td>{w_0, w_1}</td>
<td>{w_0, w_3}</td>
<td>{w_0, w_5}</td>
<td>{w_2, w_1}</td>
<td>{w_2, w_3}</td>
<td>{w_2, w_5}</td>
</tr>
</tbody>
</table>

| 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\} |

3.11 Definition Two graphs \(G\) and \(\hat{G}\) are isomorphic if there is a one-to-one and onto map \(f: V \rightarrow \hat{V}\) such that \(G\) has an edge \(\{v_i, v_j\} \in E\) if and only if \(\hat{G}\) has the associated edge \(\{f(v_i), f(v_j)\} \in \hat{E}\).

To verify that two graphs are isomorphic the most natural thing is to produce the map \(f\). Examples are in the exercises.

Showing that graphs are not isomorphic usually entails finding some graph 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 (that is, where that vertex is an element of the set that defines the edge) except that a loop from the vertex to itself counts as two. The degree sequence is the non-increasing sequence of its vertex degrees. Thus, the graph in Example 3.10 has degree sequence \(\langle 3, 2, 1, 1, 1\rangle\). Exercise 3.24 shows that if graphs are isomorphic then associated vertices have the same degree. Thus graphs with different degree sequences are not isomorphic. And, if the the degree sequences are equal then they help us construct an isomorphism, if there is one. Examples of this also are in the exercises. (There are graphs with the same degree sequence that are not isomorphic.)

III.3 Exercises

✓ 3.12 Draw a picture of a graph illustrating each relationship. Some graphs will be digraphs, or may have loops or multiple edges between some pairs of vertices.

(A) Maine is adjacent Massachusetts and New Hampshire. Massachusetts is adjacent to every other state. New Hampshire is adjacent to Maine, Massachusetts, and Vermont. Rhode Island is adjacent to Connecticut and Massachusetts. Vermont is adjacent to Massachusetts and New Hampshire. Give the graph describing the adjacency relation.
(b) 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 all natural numbers less than 30.


(d) The river Pregel cuts the town of Königsberg into four land masses. There are two bridges from mass 0 to mass 1 and one bridge from mass 0 to mass 2. There is one bridge from mass 1 to mass 2, and two bridges from mass 1 to mass 3. Finally, there is one bridge from mass 2 to 3. Consider masses related by bridges. Give the graph.

(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.

3.13 Let a graph \( G \) have vertices \( \mathcal{V} = \{v_0, \ldots, v_5\} \) and these edges.

\[ \mathcal{E} = \{v_0v_1, v_0v_3, v_0v_5, v_1v_4, v_3v_4, v_4v_5\} \]

(A) Draw \( G \).

(b) Give its adjacency matrix.

(c) Find all subgraphs with four nodes and four edges.

(d) Find all subgraphs with four nodes and three edges.

3.14 Fill in the table's blanks.

<table>
<thead>
<tr>
<th></th>
<th>Closed or open?</th>
<th>Vertices can repeat?</th>
<th>Edges can repeat?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trail</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circuit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.15 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.

3.16 Cell tower must get different frequencies if they are in line of sight. 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.

✓ 3.17 Blood cell transfusion is a common medical procedure. But the recipient must be compatible with the donor’s blood type, or 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 which 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, which can be either present (+) or absent (–). Thus there are eight common blood types, A+, A-, B+, B-, O+, O-, AB+, and AB- (there are rare blood types that don’t apply here). To have a safe transfusion, if the donor has the A antigen then the recipient must also have it. 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 if transfusion works from the starting vertex donor type to the ending vertex recipient type. Produce the adjacency matrix.

3.18 Find the degree sequence of the graph in Example 3.2 and of the two graphs of Example 3.4.

3.19 Give the array representation, like that in equation (*), for the two graphs of Example 3.4.

3.20 Draw a graph for this adjacency matrix.

\[
\begin{pmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0
\end{pmatrix}
\]

✓ 3.21 These two graphs are isomorphic.

(A) Define the function giving the correspondence.

(b) Verify that under that function the edges then also correspond.

✓ 3.22 A tree is an undirected graph in which any two vertices are connected by one and only one path. Show that the parse graph given at this subsection’s start is a tree (first relabel the vertices so that they have distinct names).
3.23 Consider building a simple graph by starting with with \( n \) vertices. (A) How many potential edges are there? (B) How many such graphs are there?

3.24 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. Conclude that if they do not have the same number of vertices then they are not isomorphic.

(B) Show that if two graphs are isomorphic then they have the same number of edges. Conclude that if they do not have the same number of edges then they are not isomorphic.

(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.

(E) Use the prior result to show that the two graphs of Example 3.4 are not isomorphic.

3.25 For these two graphs verify that they have the same degree sequence, and that they are not isomorphic. *Hint:* one graph has a longer path starting at the degree 3 vertex.

3.26 Prove Lemma 3.8.

(A) Think of an edge as a length-one walk from \( v_i \) to \( v_j \). Show that the product of the matrix with itself \( (\mathcal{M}(G))^2 \) will have that entry \( i, j \) contains the number of length-two walks.

(B) Use induction to show that for \( n \geq 1 \), the power \( (\mathcal{M}(G))^n \), has that the \( i, j \) entry is the number of length \( n \) walks from \( v_i \) to \( v_j \).

3.27 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.
We shall introduce some widely used notation conveniences for grammars. 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 putting their stamp on it by 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 effort to systematize the early computer language ALGOL60. Since then these rules have become a widely used way to express grammars.

One difference from the prior subsection is a minor typographical change. (These adjustments were made by P Naur; 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 ‘→’, BNF uses ‘::=’.

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 italic type or a different color.‡

BNF is both clear and concise, it can express the range of languages that we ordinarily want to express and it smoothly translates to a parser.‡ That is, BNF is an impedance match—it fits with what we typically want to do. Here we will incorporate some extensions for grouping and replication that is like what you will see in the wild.

A.1 Example  This is a grammar for real numbers having a finite decimal part. Take the rules for ‹natural› from Example 2.7.

\[
\langle \text{start} \rangle ::= -\langle \text{fraction} \rangle \mid +\langle \text{fraction} \rangle \mid \langle \text{fraction} \rangle \\
\langle \text{fraction} \rangle ::= \langle \text{natural} \rangle \mid \langle \text{natural} \rangle . \langle \text{natural} \rangle
\]

This derivation for 2.718 is rightmost.

\[
\langle \text{start} \rangle \Rightarrow \langle \text{fraction} \rangle \Rightarrow \langle \text{natural} \rangle . \langle \text{natural} \rangle \\
\Rightarrow \langle \text{natural} \rangle . \langle \text{digit} \rangle \langle \text{natural} \rangle \Rightarrow \langle \text{natural} \rangle . \langle \text{digit} \rangle \langle \text{digit} \rangle \langle \text{natural} \rangle \\
\Rightarrow \langle \text{natural} \rangle . \langle \text{digit} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle \Rightarrow \langle \text{natural} \rangle . \langle \text{digit} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle 8 \\
\Rightarrow \langle \text{natural} \rangle . \langle \text{digit} \rangle 18 \Rightarrow \langle \text{natural} \rangle . 718 \Rightarrow 2.718
\]

†For example, syntax coloring is a common feature of programming editors. ‡BNF is only loosely defined. Several variants do have standards but what you see often does not conform to any published standard.
Here is a derivation for \( 0.577 \) that is neither leftmost nor rightmost.

\[
\begin{align*}
\text{⟨start⟩} & \Rightarrow \text{⟨fraction⟩} \Rightarrow \text{⟨natural⟩}.\text{⟨natural⟩} \\
& \Rightarrow \text{⟨natural⟩}.\text{⟨digit⟩}\text{⟨natural⟩} \Rightarrow \text{⟨natural⟩}.5\text{⟨natural⟩} \\
& \Rightarrow \text{⟨natural⟩}.5\text{⟨digit⟩}\text{⟨natural⟩} \Rightarrow \text{⟨digit⟩}.5\text{⟨digit⟩}\text{⟨natural⟩} \\
& \Rightarrow \text{⟨digit⟩}.5\text{⟨digit⟩}\text{⟨digit⟩} \Rightarrow \text{⟨digit⟩}.5\text{⟨digit⟩}7 \Rightarrow 0.5\text{⟨digit⟩}7 \\
& \Rightarrow 0.577
\end{align*}
\]

### A.2 Example

Time is a complicated 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. Some nonterminals are from Exercise 2.23. This grammar includes some metacharacter extensions that we will discuss below.

\[
\begin{align*}
\text{⟨date-fullyear⟩} & ::= \text{⟨4-digits⟩} \\
\text{⟨date-month⟩} & ::= \text{⟨2-digits⟩} \\
\text{⟨date-mday⟩} & ::= \text{⟨2-digits⟩} \\
\text{⟨time-hour⟩} & ::= \text{⟨2-digits⟩} \\
\text{⟨time-minute⟩} & ::= \text{⟨2-digits⟩} \\
\text{⟨time-second⟩} & ::= \text{⟨2-digits⟩} \\
\text{⟨time-secfrac⟩} & ::= .\text{⟨1-or-more-digits⟩} \\
\text{⟨time-numoffset⟩} & ::= (+ | -) \text{⟨time-hour⟩} : \text{⟨time-minute⟩} \\
\text{⟨time-offset⟩} & ::= Z | \text{⟨time-numoffset⟩} \\
\text{⟨partial-time⟩} & ::= \text{⟨time-hour⟩} : \text{⟨time-minute⟩} : \text{⟨time-second⟩} \\
& \quad [\text{⟨time-secfrac⟩}] \\
\text{⟨full-date⟩} & ::= \text{⟨date-fullyear⟩} - \text{⟨date-month⟩} - \text{⟨date-mday⟩} \\
\text{⟨full-time⟩} & ::= \text{⟨partial-time⟩} \text{⟨time-offset⟩} \\
\text{⟨date-time⟩} & ::= \text{⟨full-date⟩} T \text{⟨full-time⟩}
\end{align*}
\]

One common BNF notation extension is in that grammar’s \( \text{⟨time-numoffset⟩} \) rule, where the parentheses are used as metacharacters to group a choice between the terminals + and -.

The \( \text{⟨partial-time⟩} \) rule includes square brackets as metacharacters. They deonte that the \( \text{⟨time-secfrac⟩} \) is optional. This is a common construct: another example is this syntax for if ... then ... with an optional else ....

\[
\begin{align*}
\text{⟨if-stmt⟩} & ::= \text{if} \ \text{⟨boolean-exp⟩} \ \text{then} \ \text{⟨stmt-sequence⟩} \\
& \quad [\text{else} \ \text{⟨stmt-sequence⟩}] \ \text{⟨end if⟩};
\end{align*}
\]

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.

\[
\begin{align*}
\text{⟨identifier⟩} & ::= \text{⟨letter⟩} (\text{⟨letter⟩} \mid \text{⟨digit⟩})^*
\end{align*}
\]

Each of these extension constructs is not necessary since we can express them
in plain BNF. For instance, we could replace the prior rule with this.

\[
\langle \text{identifier}\rangle ::= \langle \text{letter}\rangle \mid \langle \text{letter}\rangle \langle \text{atoms}\rangle
\]
\[
\langle \text{atoms}\rangle ::= \langle \text{letter}\rangle \langle \text{atoms}\rangle \mid \langle \text{digit}\rangle \langle \text{atoms}\rangle \mid \varepsilon
\]

But these constructs come up often enough that adopting an abbreviation is convenient.

A.3 \textbf{Example} This grammar for floating point numbers shows all three abbreviations.

\[
\langle \text{floatnumber}\rangle ::= \langle \text{pointfloat}\rangle \mid \langle \text{exponentfloat}\rangle
\]
\[
\langle \text{pointfloat}\rangle ::= \left[\langle \text{intpart}\rangle \langle \text{fraction}\rangle \right] \mid \langle \text{intpart}\rangle .
\]
\[
\langle \text{exponentfloat}\rangle ::= (\langle \text{intpart}\rangle \mid \langle \text{pointfloat}\rangle) \langle \text{exponent}\rangle
\]
\[
\langle \text{intpart}\rangle ::= \langle \text{digit}\rangle +
\]
\[
\langle \text{fraction}\rangle ::= . \langle \text{digit}\rangle +
\]
\[
\langle \text{exponent}\rangle ::= (e \mid E) [+ \mid -] \langle \text{digit}\rangle +
\]

As part of the \langle \text{pointfloat}\rangle rule, the first \langle \text{intpart}\rangle is optional. An \langle \text{intpart}\rangle, what we have earlier called a \langle \text{natural}\rangle, consists of one or more digits. And an expansion of \langle \text{exponent}\rangle must start with a choice between e or E.

Each of these extension constructs is not necessary since we can express them in plain BNF. For instance, we could replace the \langle \text{pointfloat}\rangle rule with this.

\[
\langle \text{pointfloat}\rangle ::= \langle \text{intpart}\rangle \langle \text{fraction}\rangle \mid \langle \text{fraction}\rangle \mid \langle \text{intpart}\rangle .
\]

But these constructs come up often enough that adopting an abbreviation is worthwhile.

A.4 \textbf{Remark} Passing from the grammar to a parser for that grammar is mechanical. That is, it is natural to write a program that takes as input a grammar in BNF and gives as output the source code of a program that will parse files following that grammar's format. Such a program is a \textit{compiler-compiler}.

At one point there were many compiler-compiler available, with a variety of different features. They were amalgamated into one with the production of YACC, Yet Another Compiler Compiler. In recent years, YACC has itself been supplanted with a version that is Free from the GNU project, obviously called Bison.

III.A \textbf{Exercises}

A.5 Write a grammar in BNF for the language of palindromes.

✓ A.6 US ZIP codes have five digits, and may have a dash and four more digits at the end. Give a BNF grammar.

✓ 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.

✓ 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 IIVIII 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.

\[
\begin{align*}
\langle \text{program} \rangle & ::= \{ \langle \text{statement-list} \rangle \} \\
\langle \text{statement-list} \rangle & ::= [ \langle \text{statement} \rangle ; ]^* \\
\langle \text{statement} \rangle & ::= = \langle \text{data-type} \rangle \langle \text{identifier} \rangle \\
& \quad \mid \langle \text{identifier} \rangle = \langle \text{expression} \rangle \\
& \quad \mid \text{print} \langle \text{identifier} \rangle \\
& \quad \mid \text{while} \langle \text{expression} \rangle \{ \langle \text{statement-list} \rangle \} \\
\langle \text{data-type} \rangle & ::= \text{int} \mid \text{boolean} \\
\langle \text{expression} \rangle & ::= \langle \text{identifier} \rangle \mid \langle \text{number} \rangle \mid ( \langle \text{expression} \rangle \langle \text{operator} \rangle \langle \text{expression} \rangle ) \\
\langle \text{identifier} \rangle & ::= \langle \text{letter} \rangle [ \langle \text{letter} \rangle ]^* \\
\langle \text{number} \rangle & ::= \langle \text{digit} \rangle [ \langle \text{digit} \rangle ]^* \\
\langle \text{operator} \rangle & ::= + \mid == \\
\langle \text{letter} \rangle & ::= A \mid B \mid \ldots \mid Z \\
\langle \text{digit} \rangle & ::= 0 \mid 1 \mid \ldots \mid 9
\end{align*}
\]

(A) Give a derivation and parse tree for this program.

```c
{ int A ;
  A = 1 ;
  print A ;
}
```

(B) Give a parse tree for this program.

```c
{ int A ;
  A = 5 ;
  while ( A == 5 ) { int B ; B = 6 ; A = ( B + A ) ;
    print A ; } ;
  boolean C ; print C ;
}
```
(c) Must all programs be surrounded by curly braces?

A.11 Here is a grammar for LISP.

\[
\langle s\text{-expression} \rangle ::= \langle \text{atomic-symbol} \rangle \\
| ( \langle s\text{-expression} \rangle . \langle s\text{-expression} \rangle ) \\
| \langle \text{list} \rangle \\
\langle \text{list} \rangle ::= ( \langle s\text{-expression} \rangle^* ) \\
\langle \text{atomic-symbol} \rangle ::= \langle \text{letter} \rangle \langle \text{atom-part} \rangle \\
\langle \text{atom-part} \rangle ::= \langle \text{empty} \rangle \\
| \langle \text{letter} \rangle \langle \text{atom-part} \rangle \\
| \langle \text{number} \rangle \langle \text{atom-part} \rangle \\
\langle \text{letter} \rangle ::= a | b | \ldots z \\
\langle \text{number} \rangle ::= 1 | 2 | \ldots 9 \\
\]

Derive the s-expression (cons (car x) y).

A.12 Python 3’s Format Specification Mini-Language is used to describe string substitution.

\[
\langle \text{format-spec} \rangle ::= \\
[ [\langle \text{fill} \rangle] \langle \text{align} \rangle [\langle \text{sign} \rangle][#][0][\langle \text{width} \rangle][\langle \text{gr} \rangle]\.\langle \text{precision} \rangle][\langle \text{type} \rangle] \\
\langle \text{fill} \rangle ::= \langle \text{any character} \rangle \\
\langle \text{align} \rangle ::= < | > | = | ^ \\
\langle \text{sign} \rangle ::= + | - | \\
\langle \text{width} \rangle ::= \langle \text{integer} \rangle \\
\langle \text{gr} \rangle ::= - | , \\
\langle \text{precision} \rangle ::= \langle \text{integer} \rangle \\
\langle \text{type} \rangle ::= b | c | d | e | E | f | F | g | G | n | o | s | x | X | % \\
\]

Give a derivation of these strings: (a) 03f (b) +#02X.

A.13 This is the grammar of BNF, written in BNF.

\[
\langle \text{syntax} \rangle ::= \langle \text{rule} \rangle | \langle \text{rule} \rangle \langle \text{syntax} \rangle \\
\langle \text{rule} \rangle ::= \langle \text{opt-ws} \rangle < \langle \text{rule-name} \rangle > \langle \text{opt-ws} \rangle ::= \langle \text{opt-ws} \rangle \langle \text{expression} \rangle \\
\langle \text{line-end} \rangle ::= \langle \text{rule} \rangle \langle \text{line-end} \rangle \\
\langle \text{opt-ws} \rangle ::= \langle \text{opt-ws} \rangle | \\
\langle \text{expression} \rangle ::= \langle \text{list} \rangle | \langle \text{list} \rangle \langle \text{opt-ws} \rangle | \langle \text{opt-ws} \rangle \langle \text{expression} \rangle \\
\langle \text{line-end} \rangle ::= \langle \text{opt-ws} \rangle \langle \text{EOL} \rangle | \langle \text{line-end} \rangle \langle \text{line-end} \rangle \\
\langle \text{list} \rangle ::= \langle \text{term} \rangle | \langle \text{term} \rangle \langle \text{opt-ws} \rangle \langle \text{list} \rangle \\
\langle \text{term} \rangle ::= \langle \text{literal} \rangle | < \langle \text{rule-name} \rangle > \\
\langle \text{literal} \rangle ::= " \langle \text{text1} \rangle " | ' \langle \text{text2} \rangle ' \\
\langle \text{text1} \rangle ::= \varepsilon | \langle \text{character} \rangle \langle \text{text1} \rangle \\
\]


(text2) ::= ε | ⟨character2⟩ ⟨text2⟩

⟨character⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨symbol⟩

⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨symbol⟩ ::= | | ε | - | ! | # | $ | % | & | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [ | \ | ] | ^ | - | ‘ | | | | | ~

⟨character1⟩ ::= ⟨character⟩ | '

⟨character2⟩ ::= ⟨character⟩ | "

⟨rule-name⟩ ::= ⟨letter⟩ | ⟨rule-name⟩ ⟨rule-char⟩

⟨rule-char⟩ ::= ⟨letter⟩ | ⟨digit⟩ | -

(A) Give a derivation of the rule ⟨ab⟩ ::= ⟨cd⟩ e.
(b) Give a derivation of the rule ⟨ab⟩ ::= e ⟨cd⟩.
COMPUT ECCLESIASTIQUE
Our touchstone model of mechanical computation is the Turing machine. A Turing machine has only two components, a CPU and a tape. We will now take the tape away and study the CPU alone.

Alternatively stated, while a Turing Machine has unbounded memory, the devices that we use every day do not. We can ask what jobs can be done by a machine with bounded memory.

Section IV.1 Finite State Machines

We produce a new model of computation by modifying the definition of Turing Machine. We will strip out the capability to write, changing the tape head from read/write to read-only. This gives us insight into what can be done with states alone. It will turn out that this type of machine can do many things, but not as many as a Turing machine.

Definition We will use the same type of transition tables and transition graphs as with Turing machines.

1.1 Example A power switch has two states, $q_{off}$ and $q_{on}$ and its input alphabet has one symbol, toggle.

1.2 Example Operate this turnstile by putting in two tokens and then pushing through. It has three states and its input alphabet is $\Sigma = \{ \text{token, push} \}$.

As we saw with Turing machines, the states are a limited form of memory. For instance, $q_{one}$ is how the turnstile “remembers” that it has so far received one token.

Image: The astronomical clock in Notre-Dame-de-Strasbourg Cathedral, for computing the date of Easter. Easter falls on the first Sunday after the full moon on or after the spring equinox. Calculation of this date was a great challenge for mechanisms of that time, 1843.
1.3 Example  This vending machine dispenses items that cost 30 cents. The picture is complex so we will show it in three layers. First are the arrows for nickels 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.

1.4 Example  This machine, when started in state $q_0$ and fed bit strings, will keep track of the remainder modulo 4 of the number of 1’s.

1.5 Definition  A Finite State machine, or Finite State automata, is composed of five things $M = \langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$. They are 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 or transition function $\Delta: Q \times \Sigma \rightarrow Q$.

This may not immediately appear to be like our definition of a Turing Machine. Some of that is because we have already defined the terms ‘alphabet’ and ‘transition function’. The other differences follow from the fact that that Finite State machines cannot write. For one thing, because Finite State machines cannot write they don’t

† US coin denominations are: 1 cent coins that are not used here, nickles are 5 cents, dimes are 10 cents, and quarters are 25 cents.
need to move the tape for scratch work, so we’ve dropped the tape action symbols L and R.

The other difference between Finite State machines and Turing machines is the presence of the accepting states. Consider, in the vending machine of Example 1.3, the state $q_{30}$. It is an accepting state, meaning that the machine has seen in the input what it is looking for. The same goes for Example 1.2’s turnstile state $q_{\text{ready}}$ and Example 1.1’s power switch state $q_{\text{on}}$. While we can design a Turing Machine to indicate a choice by arranging so that for each input the machine will halt and the only thing on the tape will be either a 1 or 0, a Finite State machine gives a decision by ending in one of these designated states. Above we’ve pictured that the accepting states are connected to a red light so that we know when a computation succeeds. In the transition graphs below we denote the final states with double circles and in the transition function tables we mark them with a ‘+’.

To work a Finite State machine device, put the finite-length input on the tape and press Start. The machine consumes the input, at each step reading in the next tape character.

Consequently there is no Halting problem for Finite State machines—they always halt after a number of steps equal to the length of the input. At the end either the Accept light is on or it isn’t.

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. (The + next to $q_2$ signifies that it is an accepting state.)

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.”
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 $q_1$.

$\Delta$

<table>
<thead>
<tr>
<th>+, −</th>
<th>0, ... 9</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$e$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>+</td>
<td>$q_2$</td>
<td>$e$</td>
</tr>
<tr>
<td>$e$</td>
<td>$e$</td>
<td>$e$</td>
</tr>
</tbody>
</table>

Any bad input character sends the machine to the error state. $e$, which is a sink state, meaning that the machine never leaves that state.

Often our Finite State machine descriptions will assume that a reader can infer the alphabet from the context. For instance, the prior example just says ‘else’. In practice we would take the alphabet to be the set of characters that someone could conceivably enter, including letters such as a and A or characters such as exclamation point or open parenthesis. Thus, design of a Finite State machine up to a modern standard would use all of Unicode. But for the examples and exercises here that volume is not relevant so we will use small alphabets.

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. The drawing below shows the extra edges for the top branch; 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 and have only a few states and edges 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 to put the strings in alphabetical order, and for ones that have common prefixes have the machine step 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, Finite State machines can accomplish useful work such as some kinds of arithmetic. This machine accepts strings representing a natural number that is a multiple of three, such as 15 and 5013.

Because $q_0$ is an accepting state, this machine accepts the empty string. Exercise 1.20 asks for a modification of this machine to accept only non-empty strings.

After the definition of Turing Machine we gave a formal description of the action of such a machine. We can do the same here. As there, 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.

A configuration of a Finite State machine $\mathcal{M}$ is a pair $C = \langle q, \tau \rangle$, where $q$ is a state and $\tau$ is a (possibly empty) string of elements from the tape alphabet, $\tau \in \Sigma^*$. For the input string $\tau_0$, the initial configuration is $C_0 = \langle q_0, \tau_0 \rangle$.

The action of the machine is to transition from one configuration to another. For $s \in \mathbb{N}^+$ we say that the machine's configuration at step $s$ is its configuration after the $s$-th transition, $C_s$. For $s = 0$, at step 0, its configuration is its initial configuration, $C_0$.

We must describe a transition. Assume that at step $i$ the machine $\mathcal{M}$ is in the configuration $C_i = \langle q, \tau_i \rangle$. If $\tau_i$ is not empty then pop the leading input character and call it $c$. That is, where $c = \tau_i[0]$, take $\tau_{i+1} = \langle \tau_i[1], \ldots, \tau_i[|\tau_i| - 1] \rangle$. The machine's next state is $\hat{q} = \Delta(q, c)$ and its next configuration is $C_{i+1} = \langle \hat{q}, \tau_{i+1} \rangle$. 
The other case is that at step $i$ the string $\tau_i$ is empty. This is the halting configuration $C_h$. If the ending state is a final state $q \in F$ then the machine accepts the input $\tau$, otherwise it rejects $\tau$.

We denote that two configurations are related by being separated by a single transition with $C_i \vdash C_{i+1}$. A Finite State machine computation is a sequence $C_0 \vdash C_1 \vdash C_2 \cdots C_h$ where $C_0$ is the initial configuration. Abbreviate such a sequence with $\vdash^*$ so we can write $C_0 \vdash^* C_h$.

1.10 Example The multiple of three machine of the prior example gives this computation.

$$\langle q_0, 5013 \rangle \vdash \langle q_2, 013 \rangle \vdash \langle q_2, 13 \rangle \vdash \langle q_0, 3 \rangle \vdash \langle q_0, \epsilon \rangle$$

Since $q_0$ is an accepting state, the machine accepts 5013.

1.11 Definition For a Finite State machine $M$, the language accepted by that machine $L(M)$ is the set of strings that it accepts.

1.12 Example Finite State machines are easy to translate to code. Here is a Scheme version of the multiple of three machine.$^\dagger$

```scheme
;; 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)))
    (if (null? tau-list)
      state
      (multiple-of-three-fsm-helper (delta state (car tau-list))
        (list->string (cadr 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))))))

$^\dagger$One of the great things about the Scheme programming languages is that, because the last thing called in `multiple-of-three-fsm-helper` is itself, the compiler converts the recursion to iteration. So we get the expressiveness of recursion with the space conservation of iteration.
1.13 Example  This is a simplified version of how phone numbers used to be handled in North America. Consider 1-802-555-0101. The 1 takes us along the second line so the call leaves the local exchange office to go to the long lines. The 0 between the 8 and the 2 determines that the call is to a different area code, instead of a different local exchange within the same area code. Finally the machine processes the local exchange number of 555, and the line number of 0101, and makes the connection.

Today’s picture is much more complicated. For one thing, area codes no longer are required to have a middle digit of 0 or 1. This additional complication is possible because instead of switching with physical devices, we now do it in software.

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 that the machine will end in after starting in the start state and consuming the given string.

1.15 Example  The extended transition function $\hat{\Delta}$ of this machine extends $\Delta$ in that it repeats the first row of $\Delta$’s table.

\[
\hat{\Delta}(a) = q_1 \quad \hat{\Delta}(b) = q_0
\]
(We disregard that that \(\Delta\)'s inputs are characters while \(\hat{\Delta}\)'s input are strings of characters, so that in the line above ‘a’ and ‘b’ are length one strings.) Here is \(\hat{\Delta}\)'s action on the length two strings:

\[
\hat{\Delta}(aa) = q_1 \quad \hat{\Delta}(ab) = q_2 \quad \hat{\Delta}(ba) = q_1 \quad \hat{\Delta}(bb) = q_0
\]

The definition of the extended transition function brings us back to thinking about determinism. That's because \(\hat{\Delta}\) would not be well-defined without determinism; \(\Delta\) has one and only one next state for all input configurations and so, by induction, for all input strings \(\hat{\Delta}\) has one and only 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 True or false: a Finite State machine’s language must be finite.

✓ 1.17 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?

✓ 1.18 Produce the transition function table for Example 1.6.

✓ 1.19 Rebut “no Finite State machine can accept the language \(\{a^n b \mid n \in \mathbb{N}\}\) because \(n\) is infinite.”

1.20 Modify the machine of Example 1.9 so that it accepts only non-empty strings.

✓ 1.21 For the machines of Example 1.7, Example 1.6, Example 1.8, and Example 1.9, answer these. (A) What are the accepting states? (B) Does it accept the empty string \(\varepsilon\)? (C) What is the shortest string that it accepts? (D) Is the language of accepted strings infinite?

1.22 Produce the transition graph picturing this transition function.

<table>
<thead>
<tr>
<th>(\Delta)</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0)</td>
<td>(q_2)</td>
<td>(q_1)</td>
</tr>
<tr>
<td>+ (q_1)</td>
<td>(q_0)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(q_2)</td>
<td>(q_2)</td>
</tr>
</tbody>
</table>

What is the associated language of accepted strings?

✓ 1.23 For each language, name five strings in the language and five that are not in the language (if there are not five, name as many as there are). Then give a transition graph and table for a Finite State machine accepting the language. Use \(\Sigma = \{a,b\}\).

<table>
<thead>
<tr>
<th>(A) (A)</th>
<th>(B) (B)</th>
<th>(C) (C)</th>
<th>(D) (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sigma \in \Sigma^*)</td>
<td>(\sigma) has at least two (a)'s</td>
<td>(\sigma) has exactly two (a)'s</td>
<td>(\sigma) has less than three (a)'s</td>
</tr>
</tbody>
</table>
1.24 Produce a Finite State machine over the alphabet $\Sigma = \{A, \ldots, Z, \theta, \ldots, 9\}$ that accepts only the string 911, and a machine that accepts any string but that one.

1.25 Apply the extended transition function of Example 1.15 to all of the length three and length four string inputs.

✓ 1.26 What language is accepted by this machine?

✓ 1.27 Give a Finite State machine over $\Sigma = \{a, b, c\}$ that accepts any string containing the substring abc.

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 nonelements. Give a Finite State machine accepting this language.

✓ 1.29 Consider a language of comment strings that begin with /#, end with #/, and have no #/’s in the middle. Give a Finite State machine to accept those.

1.30 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 accepts the language. Use $\Sigma = \{a, b\}$.

(A) $L = \{\sigma \in \{a, b\}^* \mid \sigma \text{ ends in } aa\}$
(B) $\{\sigma \in \{a, b\}^* \mid \sigma = \epsilon\}$
(C) $\{\sigma \in \{a, b\}^* \mid \sigma = a^3b \text{ or } \sigma = ba^3\}$
(D) $\{\sigma \in \{a, b\}^* \mid \sigma = a^n \text{ or } \sigma = b^n \text{ for } n \in \mathbb{N}\}$

1.31 What happens when the input to the extended transition function is the empty string?

✓ 1.32 Produce a Finite State machine that accepts the language over $\Sigma = \{\theta, \ldots, 9\}$.

(A) $\{\sigma \in \Sigma^* \mid \sigma \text{ has either no } \theta \text{’s or no } 2 \text{’s}\}$
(B) $\{\sigma \in \Sigma^* \mid \text{the number of } 1 \text{’s is a multiple of } 5\}$

✓ 1.33 Give a Finite State machine over the alphabet $\Sigma = \{A, \ldots, Z\}$ that accepts only strings in which the vowels occur in ascending order. (The vowels, in ascending order, are traditionally taken to be A, E, I, O, and U.) Although the string AEIA0AUA has an A that occurs after an E, nonetheless the machine should accept it because there exists a subsequence of string elements consisting of the ordered vowels.

✓ 1.34 This grammar

\[
\langle \text{real} \rangle \rightarrow \langle \text{posreal} \rangle \mid + \langle \text{posreal} \rangle \mid - \langle \text{posreal} \rangle \\
\langle \text{posreal} \rangle \rightarrow \langle \text{natural} \rangle \mid \langle \text{natural} \rangle . \mid \langle \text{natural} \rangle . \langle \text{natural} \rangle \\
\langle \text{natural} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\
\langle \text{digit} \rangle \rightarrow \theta \mid \ldots \ 9
\]
gives rise to a language. (A) Give five strings that are in the language and five that are not. (B) Is the string \( 1.12 \) in the language? (C) Briefly describe the language. (D) Give a Finite State machine that accepts the language.

1.35 For each language produce five strings from that language, and produce a Finite State machine to accept it.

(A) \( \{ \sigma \in B^* \mid \) every 1 in \( \sigma \) is both preceded and followed by a 0 \}

(B) \( \{ \sigma \in B^* \mid \sigma \) is the binary representation of a number divisible by 4 \}

(C) \( \{ \sigma \in \{0, \ldots, 9\}^* \mid \sigma \) is the decimal representation of a number that is even \}

(D) \( \{ \sigma \in \{0, \ldots, 9\}^* \mid \sigma \) represents a number divisible by 100 in base ten \}

1.36 Consider \( \{ \sigma \in \{0, \ldots, 9\}^* \mid \sigma \) represents a multiple of 4 in base ten \}. Briefly describe a Finite State machine; you need not give the full graph or table.

1.37 The definition of the extended transition function, Definition 1.14, is conceptual, with some details held back. Here is a constructive definition, one that you could easily translate into a computer program. On the empty input string, \( \hat{\Delta}(\epsilon) = q_0 \). If the input string \( \sigma \) is not empty then peel off the final character, \( \sigma = \sigma_0 x \) for \( x \in \Sigma \), and let \( \hat{\Delta}(\sigma) = \Delta(\hat{\Delta}(\sigma_0), x) \). Now apply this to the machine in Example 1.6.

(A) Use the definition to find \( \hat{\Delta}(\emptyset) \) 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.

✓ 1.38 Produce a Finite State machine that accepts the language over \( \Sigma = \{a, b\} \) containing no more than one occurrence of \( \text{aa} \). Note that \( \text{aaa} \) contains two occurrences.

1.39 Let \( \Sigma = \mathbb{B} \). (A) List all of the different Finite State machines over \( \Sigma \) with \( Q = \{ q_0 \} \). (Ignore whether a state is final or not; we will do that below.) (B) List all the the ones with \( Q = \{ q_0, q_1 \} \). (C) How many machines are there with \( n \) states? (D) What if we distinguish between machines with different sets of final states?

✓ 1.40 Propositiones ad acuendos iuvenes (problems to sharpen the young) is the oldest collection of mathematical problems written in Latin. It teaches logical thinking and problem solving and was written by Alcuin of York (735–804), head of the Frankish court school at Aachen and a royal advisor to Charlemagne. One problem, titled Propositio de lupo et capra et fasiculo cauli, is famous. A man had to transport to the far side of a river a wolf, a goat, and a bundle of cabbages. The only boat he could find was one that could carry only two of them. For that reason, he sought a plan which would enable them all to get to the far side unhurt. Let him, who is able, say how it could be possible to transport them safely. Readers from that time would have understood that a wolf cannot be left alone with a goat nor can a goat be left alone with cabbages. Construct the relevant Finite State machine and use it to solve the problem.
1.41 Consider a variant on a Finite State machine, where the set of input strings is bounded.

(a) In Rock-Paper-Scissors, two players each simultaneously produce one of the three R, P, or S. A player producing R beats a player producing S, and similarly S beats P, and P beats R (if they produce the same then it is a do-over). Encode a game instance with the two-character string \( \sigma = \langle \text{player one's play, player two's play} \rangle \). Make a machine that recognizes a win for player one.

(b) Make a machine that accepts a Tic-Tac-Toe game that is a win for the X's. A board has nine squares so encode a game instance with length nine strings. Both the examples here illustrate particularly well how the machines use their states as a kind of memory — to remember, in a way, what has happened in the computation so far.

1.42 Can a Finite State machine have no states; does the definition allow \( Q = \emptyset \)?

1.43 Show that any language accepted by a Finite State machine is accepted by some Turing machine. (There are lots of ways to define ‘accepted by a Turing machine’. One is that the Turing machine reads the input and halts with the head under the only symbol on the tape, a 1.)

1.44 Fix a Finite State machine \( M \), its set of accepting states \( F \), and the associated language \( L \subseteq \Sigma^* \) of accepted strings. If we flip every accepting state to non-accepting and every non-accepting state to accepting, what happens to the language?

1.45 Show, as suggested by Example 1.8, that for any finite language, there is a Finite State machine that accepts a string if and only if it is a member of that language.

1.46 We can show that there are languages such that no Finite State machine accepts that language. Fix an alphabet \( \Sigma \). (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 Non-determinism

Turing machines, and also Finite State machines, have the property that the next state is completely determined by the current state and current character. Once you lay out an initial tape and push Start then you just walk through the steps like, well . . . , like an automaton. We now consider machines that are non-deterministic, where from any configuration the machine could move to more than one next state, or to just one, or even to no state at all.

Motivation Suppose that you have a grammar with some rules and start symbol \( S \). You are given the string \( \sigma \) and asked if has a derivation. The challenge to these
problems is that the derivation can potentially go more than one way. 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 used intuition to spot that one way or another leads to an answer. If you wrote a program for it then you might instead try every possible case; you might do a breadth-first traversal of the tree of all derivations.

Although the breadth-first technique is familiar from experience with programming classes, don’t think that it is natural. This problem is most naturally done in parallel and breadth-first traversal is a serialization that we do only because it is required by the machine model that we have, that is, it is suited to the single-CPU devices that we have in mind when we first learn programming.

The American philosopher, and Hall of Fame baseball catcher, Y Berra said, “If 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 that applies that rule, 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 \( \sigma \) appears, if it ever does.

We can illustrate this with the celebrated Travelling Salesman problem. Take a map of the forty eight contiguous US states and look at trips that visit every state capital. We want to know, say, if there is a cycle, a trip that visits each capital and returns back to the start, of less than 16 000 kilometers. Starting at Montpelier we could fork forty seven child processes, one for each potential next capital. The process assigned Albany, for instance, would know that the trip so far is 126 kilometers. Each child would then fork its own children, forty six of them. At the end, if there is a trip of less than 16 000 kilometers then some process knows it. There will be lots of processes and many of them have failed to find a short enough trip, but if even one succeeds then we consider the overall search a success.

This computation is nondeterministic in that while it is happening the machine has many process. In that sense the machine is simultaneously in many different states.

This approach suits an unboundedly-parallel machine, a machine that is angelic in that whenever we want more computational resources, such as being able to allocate new children, those resources just appear. Or, a single CPU could simulate the children with time-slicing, which dovetails by running the first process for a few ticks, then running the second for a few, and then back to the first, the second, and then to the third, etc. This echoes our experience with everyday computers, where when we are writing an email in one window and watching a video in another then the machine appears to be in multiple states simultaneously.
This section considers nondeterministic Finite State machines. (We will do nondeterministic Turing machines in the fifth chapter.) We will have two ways to think about nondeterminism, two mental models. The first was introduced above: we can picture that when such a machine is presented with multiple possible next states then the machine forks, so that it is in all of them simultaneously. This illustrates.

2.1 Example The Finite State machine below is nondeterministic because leaving \( q_0 \) are two arrows labelled 0. It also has states with a deficit of edges; e.g., no 1 arrow leaves \( q_1 \).

\[
\begin{array}{c}
q_0 \\
| q_1 \\
| q_2 \\
| q_3
\end{array}
\]

Give it input 00001 and imagine that when there are two next states, then the machine splits in two. As the animation below shows, the computation history is a tree.

2.2 Animation: Steps in the nondeterministic computation.

When we considered the forking approach to string derivations or to the Travelling 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 accepts the input string. There are also branches that are not successful. The one at the bottom dies out after step 2 because this machine has no next state when the present state \( q_2 \) and the input is 0. Another is the branch at the top, which never dies but also does not accept the input. However, we don’t care about unsuccessful branches. We only care that there exists a successful one. So we will define that a nondeterministic machine accepts an input if there is at least one branch in the

\[\text{While these models are helpful in learning and thinking about nondeterminism, they are not part of the formal definitions and proofs.}\]
computation tree that accepts the input.

The machine in the above example accepts a string if it ends in two 0’s and a 1. When we feed it the input 00001 the problem 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, perhaps somewhat fancifully, say that the machine has correctly guessed.

This is our second mental model for nondeterminism. We will imagine programming by calling a function, some $\text{amb}(S, R_0, R_1 \ldots R_{n-1})$, and having the computer somehow guess a successful search sequence.

Saying that the machine is guessing is jarring since it does not fit our intuition of how computers work. Nonetheless, researchers in this area often speak this way. A perhaps less disconcerting way to put it is to imagine that the machine is furnished with the answer (“go around twice, then off to the right”) and only has to check it. This mental model of nondeterminism is demonic because the furnisher is often personified as a supernatural being, a demon, 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 machine could check.

We shall describe nondeterminism in both ways: as a machine being in multiple states at once, and as a machine guessing. To use both paradigms we must reconcile them by showing that there is a nondeterministic machine accepting a string under the first model if and only if there is one accepting that string under the second. Here we will do that for nondeterministic Finite State machines.

**Definition** A nondeterministic Finite State machine differs from a deterministic machine only in that the next-state function does not output single states, it outputs sets of states.

2.3 **Definition** A nondeterministic Finite State machine $M = (Q, q_{\text{start}}, F, \Sigma, \Delta)$ 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), final state!nondeterministic Finite State machine 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 our first encounter with the concept of nondeterminism, which is critical for the book’s final part. Second, we also present them because they are useful in practice; for some examples of jobs that are more easily solved in this way, see the exercises. First, we need them to prove Kleene’s Theorem, Theorem 3.10 below.

2.4 **Example** This is Example 2.1’s nondeterministic Finite State machine, along with its transition function.
The change from deterministic Finite State machines is that here the entries of the array are sets of states.

2.5 Definition A nondeterministic Finite State machine accepts an input string if for that string there is some sequence of transitions allowed by the machine that ends in an accepting state. The language of the machine $L(M)$ is the set of accepted strings.

Here is a more precise definition of string acceptance. Fix a nondeterministic $M$ with start state $q_0$ and transition function $\Delta : Q \times \Sigma \rightarrow Q$. Its extended transition function $\hat{\Delta} : \Sigma^* \rightarrow Q$ gives the state that the machine will be in after starting in the start state and consuming the given string. That is, $\hat{\Delta}(\epsilon) = \{q_0\}$, and where $\hat{\Delta}(\tau) = \{q_{i_0}, q_{i_1}, \ldots, q_{i_k}\}$ for $\tau \in \Sigma^*$ we have that $\hat{\Delta}(\tau \cdot t) = \Delta(q_{i_0}, t) \cup \Delta(q_{i_1}, t) \cup \cdots \cup \Delta(q_{i_k}, t)$. Then $\sigma \in \Sigma^*$ is accepted if any state in $\hat{\Delta}(\sigma)$ is a final state.

2.6 Example The language accepted by this machine

is the set of strings containing the substring $aa$ or $bb$. For instance, the machine accepts $abaaba$, where the matching substring starts at index 2, because there is a sequence of allowed transitions ending in an accepting state, namely this.

$\langle q_0, abaaba \rangle \vdash \langle q_0, baaba \rangle \vdash \langle q_0, aaba \rangle \vdash \langle q_1, aba \rangle \vdash \langle q_2, ba \rangle \vdash \langle q_2, a \rangle \vdash \langle q_2, \epsilon \rangle$

2.7 Example This machine, with $\Sigma = \{a, b, c\}$,

accepts the language $\{(ac)^n \mid n \in \mathbb{N}\} = \{\epsilon, ac, acac, \ldots\}$. The symbol $b$ isn’t attached to any arrow so it won’t play a part in any accepting string.

One advantage of nondeterministic Finite State machines is that they are often easier to write than a deterministic machine that does the same job.

2.8 Example This machine accepts any string in $\{a, b\}^*$ whose next to last character is $a$. 
2.9 Example This machine accepts \( \{ \sigma \in \mathbb{B}^* \mid \sigma = 0^r 1 \text{ where } r \in \mathbb{B}^* \} \).

2.10 Example This is a garage door opener listener, that waits to hear the remote control send the signal \( 0101110 \).

That is, it accepts the language \( \{ \sigma \in \mathbb{B}^* \mid \sigma \sim 0101110 \} \) of strings with that suffix.

2.11 Remark Note again that something in that example is jarring. The listener will end in an accepting state if it starts in state \( q_0 \) and hears a string \( 010101110 \). That string starts with three \( 01 \)'s while the machine is listening for \( 0101110 \), with two. How does this mechanism guess that it should ignore the first \( 01 \) but pay attention to the second? A person trained in programming may well react that “guessing” is a not-mechanically-accomplishable task.

In mathematics we can consider whatever we can precisely define, and we have precisely defined nondeterministic Finite State machines. However, in this book we have so far emphasized devices that are physically realizable in principle so this may seem like a change in direction. It is not a change because below we will show how to convert a nondeterministic Finite State machine into deterministic one. So we can think of the above machine as an abbreviation for a deterministic device, which obviates the paradox of guessing for at least this class of machine.

\( \varepsilon \) transitions Another extension, beyond nondeterminism, is to allow \( \varepsilon \) transitions, also known as \( \varepsilon \) moves. We alter the definition of a nondeterministic Finite State machine Definition 2.3 so that the transition function’s signature goes from \( \Delta : Q \times \Sigma \to \mathcal{P}(Q) \) to \( \Delta : Q \times (\Sigma \cup \{ \varepsilon \}) \to \mathcal{P}(Q) \) (where \( \varepsilon \) is not an element of \( \Sigma \)). The associated behavior is that the machine can transition spontaneously, without consuming any input.†

2.12 Example This machine accepts valid integers.

It can accept strings that do not start with a sign: if we give it input 123 then it 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, so the machine accepts the input.

† Or you could think of it as transitioning on consuming the empty string \( \varepsilon \).
2.13 **Example** A machine may follow two or more \( \varepsilon \) transitions in a row. If this machine is in state \( q_0 \) then it may stay in that state, or spontaneously transition to \( q_2 \), or \( q_3 \), or \( q_5 \).

![Diagram showing machine transitions]

That is, the language of this machine is the four element set \( \mathcal{L} = \{ \text{abc, abd, ac, ad} \} \).

To precisely describe how Finite State machines with \( \varepsilon \) transitions work, we first define a function \( E: Q \times \mathbb{N} \to Q \) so that \( E(q, i) \) is the set of states \( q \) reachable within exactly \( i \) transitions. That is, for \( q \in Q \) define \( E(q, 0) = \{ q \} \), and where \( E(q, i) = \{ q_{i_0}, \ldots q_{i_k} \} \) define \( E(q, i + 1) = E(q, i) \cup \{ \Delta(q_{i_0}, \varepsilon) \} \cup \cdots \cup \{ \Delta(q_{i_k}, \varepsilon) \} \). Observe that these are nested \( E(q, 0) \subseteq E(q, 1) \subseteq \cdots \) and are subsets 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 \( E(q, \hat{i}) \subseteq E(q, \hat{i} + 1) \subseteq \cdots \). Define the \( \varepsilon \) closure function \( \hat{E}: Q \to \mathcal{P}(Q) \) by \( \hat{E}(q) = E(q, \hat{i}) \).

With that, we will modify the definition of the extended transition function \( \hat{\Delta}: \Sigma^* \to Q \) from section 2. Start with \( \hat{\Delta}(\varepsilon) = \hat{E}(q_0) \). And, where \( \hat{\Delta}(\tau) = \{ q_{i_0}, q_{i_1}, \ldots q_{i_k} \} \), for \( \tau \in \Sigma^* \) define the next step to be \( \hat{\Delta}(\tau \triangleleft t) = \hat{E}(\Delta(q_{i_0}, t)) \cup \cdots \cup \hat{E}(\Delta(q_{i_k}, t)) \). As earlier, the machine accepts \( \sigma \in \Sigma^* \) if any one of the states in \( \hat{\Delta}(\sigma) \) is a final state.

As with nondeterministic machines above, one reason \( \varepsilon \) transitions are widely used is that they often make solving a complex job much easier. Some examples follow and more are in the exercises.

2.14 **Example** A \( \varepsilon \) 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.

![Diagram showing parallel connection]

The top nondeterministic machine’s language is \( \{ \sigma \in \Sigma^* \mid \sigma \text{ ends in } ab \} \) and the bottom machine’s language is \( \{ \sigma \in \Sigma^* \mid \sigma = (ac)^n \text{ for some } n \in \mathbb{N} \} \), where \( \Sigma = \{ a, b, c \} \). These two are unioned to make the language for the entire machine.

\[
\{ \sigma \in \Sigma^* \mid \text{either } \sigma \text{ ends in } ab \text{ or } \sigma = (ac)^n \text{ for } n \in \mathbb{N} \}
\]
2.15 Example  An \( \varepsilon \) transition can also tie together two machines with a serial connection.

The left side is a nondeterministic machine. If \( q_0 \) were its sole accepting state then the left side's language would be \( \{ (aab)^m \mid m \in \mathbb{N} \} \). The right side accepts strings in \( \{ (a|aba)^n \mid n \in \mathbb{N} \} \). Overall, the machine accepts strings in the concatenation of those languages.

\[
\mathcal{L}(\mathcal{M}) = \{ \sigma \in \{a, b\}^* \mid \sigma = (aab)^m(a|aba)^n \text{ for } m, n \in \mathbb{N} \}
\]

That is, a string is accepted by the overall machine if first comes some number of \( aab \)'s, and then comes some number of: either \( a \)'s or \( aba \)'s. For example, the machine accepts these two strings (the brackets are there to help a person's eye pick out the parts).

\[
\underbrace{aab}_\text{aab} \underbrace{aba}_\text{aba} \text{  or  } \underbrace{a}_{\text{aba}} \underbrace{aba}_\text{aba}
\]

2.16 Example  An \( \varepsilon \) transition edge can also take a nondeterministic machine and give it the Kleene star effect. That is, without the \( \varepsilon \) edge this machine's language is \( \{ \varepsilon, ab \} \) while adding it makes the language \( \{ (ab)^n \mid n \in \mathbb{N} \} \).

Equivalence of the machine types  The following result says that for Finite State machines, nondeterminism does not alter the power of the machines. So our first encounter with nondeterminism has it as less a radical innovation and more of a problem-solving convenience.

2.17 Theorem  The class of languages accepted by nondeterministic Finite State machines equals the class of languages accepted by deterministic Finite State machines. This remains true if we allow the nondeterministic machines to have \( \varepsilon \) transitions.

To show that the two classes are equal we will argue that they are subsets of each other. One direction is easy; any deterministic machine is a nondeterministic machine. That is, in a deterministic machine the next-state function outputs single states. To make it a nondeterministic machine just convert those single states into singleton sets. Thus the set of languages accepted by deterministic machines is a subset of the set accepted by nondeterministic machines.
For inclusion the other way we will demonstrate how to start with a nondeterministic machine with \(\epsilon\) transitions and construct a deterministic machine that accepts the same language. We won’t give a complete proof, although certainly one is possible, because a proof is a notational mess and the examples below are convincing.

First consider a nondeterministic machine \(M_N\) that has no \(\epsilon\) transitions. Let its next-state function be \(\Delta_N\). As the next example illustrates, the associated deterministic machine \(M_D\) has states \(s_i\) that are the sets of states of \(M_N\), so \(s_i = \{q_{i_1}, q_{i_2}, \ldots\}\). The next-state function \(\Delta_D\) of \(M_D\) combines the next states associated with each \(q_{ij}\).

\[
\Delta_D(s_i) = \Delta_N(q_{i_1}) \cup \Delta_N(q_{i_2}) \cup \cdots \quad (*)
\]

2.18 Example  Consider this nondeterministic machine with \(\epsilon\) transitions, \(M_N\).

The deterministic machine \(M_D\) that has the same behavior as \(M_N\) is below.

The deterministic \(M_D\)’s states \(s_i\) are the sets of states of \(M_N\). For instance, suppose that this machine is in \(s_5 = \{q_0, q_2\}\) and is reading \(a\). Apply equation (*): combine the set of next states due to \(q_0\), the set \(\Delta_N(q_0, a) = \{q_0, q_1\}\), with the next states due to \(q_2\), the set \(\Delta_N(q_2, a) = \{\}\). That gives \(\Delta_D(s_5, a) = \{q_0, q_1\}\), which is the state \(s_4\).

\[
\begin{array}{c|cc}
\Delta_D & a & b \\
\hline
s_0 = \{\} & s_0 & s_0 \\
+ s_1 = \{q_0\} & s_4 & s_0 \\
s_2 = \{q_1\} & s_0 & s_3 \\
+ s_3 = \{q_2\} & s_0 & s_0 \\
+ s_4 = \{q_0, q_1\} & s_4 & s_3 \\
+ s_5 = \{q_0, q_2\} & s_4 & s_0 \\
+ s_6 = \{q_1, q_2\} & s_0 & s_3 \\
+ s_7 = \{q_0, q_1, q_2\} & s_4 & s_3 \\
\end{array}
\]

A state of \(M_D\) is accepting if any of its elements are accepting states in \(M_N\). The start state of \(M_D\) is \(s_1 = \{q_0\}\) because the start state of \(M_N\) is \(q_0\).

Besides the notational convenience, naming the sets of states as \(s_i\)’s makes clear that \(M_D\) is a deterministic Finite State machine. So does exhibiting its transition graph.
If the nondeterministic machine has \(k\)-many states then under this construction the deterministic machine has \(2^k\)-many states. Typically many of them can be eliminated. For instance, in the above machine the state \(s_6\) is unreachable since there are no arrows into \(s_6\). The next section covers minimizing the number of states in a machine.

The next example expands the above construction to cover nondeterministic machines that have \(\varepsilon\) transitions. Basically, we have to follow those transitions. For instance, suppose that we have a nondeterministic machine and we are constructing the associated deterministic machine’s next-state function \(\Delta_D\), that the current configuration is \(s_i = \{ q_{i_1}, q_{i_2}, \ldots \}\) and the machine is reading \(a\). If there is a \(\varepsilon\) transition from \(q_{i_j}\) to some \(q\) then we must add to equation (\(\ast\)) the term \(\Delta_N(\{ q \}, a)\). Further, we have to iterate: if there is a \(\varepsilon\) transition leading out of \(q\) then we must handle its endpoint the same way. (That is, we must find the \(\varepsilon\) closure; see Exercise 2.44.) Finally, the start state of the deterministic machine will be the set consisting of all the states of \(M_N\) that are reachable by \(\varepsilon\) transitions from \(q_0\) (that is, it is the \(\varepsilon\) closure of the set \(\{ q_0 \}\)).

2.19 Example  Consider this nondeterministic machine.

Suppose it is in state \(q_0\) and is reading \(a\). The arrow on the left takes the machine from \(q_0\) to \(q_2\). Alternatively, starting at \(q_0\), following the \(\varepsilon\) transition to \(q_3\), and then reading the \(a\) gives \(q_3\). So the machine is next in the set of states \(\{ q_0, q_3 \}\).

This is the full deterministic machine.
For instance this repeats the computation above.

\[ \Delta_D(s_1, a) = \Delta_N(q_0, a) \cup \Delta_N(q_3, a) = \{ q_2 \} \cup \{ q_3 \} = \{ q_2, q_3 \} = s_{10} \]

Another example is that the machine is in \( s_2 = \{ q_1 \} \) and is reading \( a \). Following the \( \epsilon \) to \( q_0 \) and then reading \( a \) leads to \( q_2 \). Again starting in \( q_1 \), following the \( \epsilon \) to \( q_0 \) and following the other \( \epsilon \) to \( q_3 \), and then reading the \( a \) leads to \( q_3 \).

\[ \Delta_D(s_2, a) = \Delta_N(q_1, a) \cup \Delta_N(q_0, a) \cup \Delta_N(q_3, a) \]

\[ = \{ \} \cup \{ q_2 \} \cup \{ q_3 \} = s_{10} \]

A state is accepting if it contains \( q_3 \) but is also accepting if it contains \( q_0 \) or \( q_1 \) since from those we can reach \( q_3 \) by \( \epsilon \) transitions. This machine's start state is \( s_7 = \{ q_0, q_3 \} \), because from \( q_0 \) we can reach \( q_3 \) via an \( \epsilon \) transition.

IV.2 Exercises

2.20 Give the transition function for the machine of Example 2.6, and of Example 2.7.

✓ 2.21 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.

2.22 Your friend says, “Epsilon transitions don’t make any sense because this machine will never get its first step done; it just endlessly following the epsilon transitions.”

![Diagram of a machine with epsilon transitions]

Straighten them out.

2.23 This machine has $\Sigma = \{a, b\}$.

![Diagram of a machine with transitions]

(A) Does it accept the empty string? (B) a? b? (C) List five strings of minimal length that it accepts. (D) List five of minimal length that it does not accept.

2.24 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 $\epsilon$.

✓ 2.25 Give diagrams for nondeterministic Finite State machines that accept the given language, and that have the given number of states. Use $\Sigma = \mathbb{B}^*$.

(A) $L_0 = \{\sigma \mid \sigma$ ends in $\emptyset\}$, having three states
(B) $L_1 = \{\sigma \mid \sigma$ has the substring $0110\}$, with five states
(C) $\{\sigma \mid \sigma$ contains an even number of 0's or exactly two 1's\}, with six states
(D) $\{\emptyset\}^*$, with one state

2.26 This table

<table>
<thead>
<tr>
<th>$\Delta$</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${q_0}$</td>
<td>${q_1, q_2}$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${q_3}$</td>
<td>${q_3}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>${q_1}$</td>
<td>${q_3}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${q_3}$</td>
<td>${q_3}$</td>
</tr>
</tbody>
</table>

gives the next-state function for a nondeterministic Finite State machine. (A) Draw the transition graph. (B) What is the accepted language? (C) Give the next-state table for a deterministic machine that accepts the same language.

✓ 2.27 Draw the graph of a nondeterministic Finite State machine over $\mathbb{B}$ that accepts strings with the suffix 111000111.

✓ 2.28 Give a Finite State machine, which may be nondeterministic, that accepts the set of strings from $\{a, b\}^*$ whose second character is a and whose next to last character is b. Draw its transition graph.
2.29 Find a nondeterministic Finite State machine over \( \Sigma = \{a, b\}\) that accepts the language of strings whose second character is a and whose next to last character is also a. Draw a graph and also give its transition function as a table.

✓ 2.30 Find the nondeterministic Finite State machine that accepts all bitstrings that begin with 10. Use the algorithm given above to produce a deterministic machine that does the same.

✓ 2.31 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.

2.32 Find a nondeterministic Finite State machine that accepts this language of three words \(L = \{\text{cat, cap, carumba}\}\).

2.33 Draw the transition graph of a nondeterministic machine that accepts the language \(\{\sigma = \tau_0 \tau_1 \tau_2 \in B^* \mid \tau_0 = 1, \tau_2 = 1, \text{ and } \tau_1 = (00)^k \text{ for some } k \in \mathbb{N}\}\).

2.34 Give a nondeterministic Finite State machine over \(\Sigma = \{a, b, c\}\) that accepts the language of strings that omit at least one of the characters in the alphabet.

✓ 2.35 What is the language of this machine?

\[
\begin{array}{c}
\begin{array}{ccc}
q_0 & \xrightarrow{0} & q_1 \\
q_1 & \xrightarrow{1} & q_2 \\
q_2 & \xrightarrow{0} & q_0
\end{array}
\end{array}
\]

2.36 Find a deterministic machine and a nondeterministic machine that accepts each language over \(B\). You need not construct the deterministic machine from the other; if you like then you can just construct it using your native wit.

(A) The set of bitstrings containing an even number of 0’s.

(B) The set of bitstrings containing the substring 11.

(C) The set of bitstrings containing both an even number of 0’s and the substring 11.

✓ 2.37 For both of these follow the construction above to make a deterministic machine with the same language.

\[
\begin{array}{c}
\begin{array}{ccc}
q_0 & \xrightarrow{0} & q_1 \\
q_0 & \xrightarrow{1} & q_2 \\
q_1 & \xrightarrow{0,1} & q_2 \\
q_2 & \xrightarrow{1} & q_1
\end{array}
\end{array}
\]

✓ 2.38 For each give a nondeterministic Finite State machine over \(\Sigma = \{0, 1, 2\}\).

(A) The machine accepts the language of strings whose final character appears exactly twice in the string.

(B) The machine accepts the language of strings whose final character appears exactly twice in the string, but in between those two occurrences is no higher digit.
2.39 For each give a nondeterministic Finite State machine over \( \mathbb{B} \) that accepts each language.

(A) In each string, every 0 is followed immediately by a 1.

(B) Each string contains \( 000 \) followed, possibly with some intermediate characters, by \( 001 \).

(C) In each string the first two characters equals the final two characters, in order. (*Hint:* what about \( 000? \))

(D) There is either an even number of 0’s or an odd number of 1’s.

2.40 Give a nondeterministic Finite State machine over \( \Sigma = \{ a, b, c \} \) that accepts only the empty string. Give one that accepts any string except the empty string. For both, give the transition graph and table.

2.41 A grammar is right linear if every production rule has the form \( A \rightarrow bC \), where the right side has a single terminal \( b \in \Sigma \) followed by a single nonterminal. From this right linear grammar

\[
S \rightarrow aA \\
A \rightarrow aA \mid bB \\
B \rightarrow bB \mid b
\]

we can get this nondeterministic Finite State machine.

2.42 Decide whether each problem is solvable or unsolvable (by a Turing machine).

(A) \( \mathcal{L}_{DFA} = \{ \langle M, \sigma \rangle \mid \text{the deterministic Finite State machine } M \text{ accepts } \sigma \} \)

(B) \( \mathcal{L}_{NFA} = \{ \langle M, \sigma \rangle \mid \text{the nondeterministic machine } M \text{ accepts } \sigma \} \)

(C) \( \mathcal{L} = \{ M \mid \text{the Finite State machine } M \text{ accepts at least one string} \} \)

2.43 Given a nondeterministic Finite State machine without \( \epsilon \) transitions, and with two accepting states, show how to add \( \epsilon \) transitions so that the extended machine will have only one accepting state, without changing the accepted language.
Example 2.19 illustrated following $\varepsilon$ transitions. We can give a rigorous definition. For each state $q \in Q$ we will define the $\varepsilon$ closure $E(q) \subseteq Q$. The definition is inductive, so we get a sequence of sets $E_0(q) \subseteq E_1(q) \subseteq \cdots E_k(q) = E(q)$. The intuition is that $E_i(q)$ is the set of states reachable from $q$ by traversing $i$ or fewer $\varepsilon$ edges. Formally, the definition’s base step is that for each state, $E_0(q) = \{ q \}$. The inductive step starts with $E_i(q) = \{ q_{i_0}, \ldots q_{i_j} \}$ for some $i \in \mathbb{N}$ and produces $E_{i+1}(q)$

$$E_{i+1}(q) = E_i(q) \cup \Delta(q_{i_0}, \varepsilon) \cup \cdots \Delta(q_{i_j}, \varepsilon)$$

(A) For the machine of Example 2.19, for each $q \in Q$ produce $E_0(q)$, $E_1(q)$, $E_2(q)$, and $E_3(q)$. List $E(q)$ for each $q \in Q$.

(b) Do the same for Exercise 2.23’s machine.

(c) This machine has loops of $\varepsilon$’s.

Does the definition just keep going; what is $E(q)$ for each $q \in Q$?

(d) Prove that for every machine, $E(q)$ is defined for every $q \in Q$.

Section IV.3 Regular expressions

In 1951, S Kleene was studying a mathematical model of neurons. Neurons do not have any scratch memory. They just respond to input signals arriving from the other neurons to which they are connected. Kleene abstracted some aspects of the definition of other researchers to get a definition of Finite State machines.

He noted patterns to the languages that are accepted by such machines. One of these patterns is the star operator that was later named after him. He gave the definition in the first subsection below, and mathematically justified that with the theorem in the second subsection.

**Definition** This machine

accepts strings that have some number of a’s, followed by one b, followed by some number of a’s. We now give a convenient way, called regular expressions, to denote...
constructs such as “any number of” and “followed by.” We will introduce these with a few preliminary examples so that the definition will be clearer when it comes. These examples use the alphabet $\Sigma = \{a, \ldots, z\}$.

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, $\mathcal{L} = \{hat, het, hit, hot, hut\}$.

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 regular expression $ab*c$ describes the language whose words begin with an $a$, followed by any number of $b$’s (including possibly zero-many $b$’s), and ending with a $c$. So ‘$*$’ means ‘repeat the prior thing any number of times’. This regular expression describes the language $\mathcal{L} = \{ac, abc, abbc, \ldots\}$.

3.3 Example There is an interaction between pipe and star. The language described by the regular expression $a(b|c)*$ consists of words starting with $a$ and ending with any number of $b$’s or $c$’s. The described language mixes the two letters, $\mathcal{L} = \{a, ab, ac, abb, abc, acb, acc, \ldots\}$.

In contrast, to describe the language whose members begin with $a$ and end with any number of $b$’s or any number of $c$’s, $\hat{\mathcal{L}} = \{a, ab, abb, \ldots, ac, acc, \ldots\}$, use the regular expression $a(b*|c*)$.

We first describe the syntax of regular expressions.

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

\[
\langle regex \rangle \rightarrow \langle concat \rangle \\
| \langle regex \rangle '!' \langle concat \rangle \\
\langle concat \rangle \rightarrow \langle simple \rangle \\
| \langle concat \rangle \langle simple \rangle \\
\langle simple \rangle \rightarrow \langle char \rangle \\
| \langle simple \rangle * \\
| ( \langle regex \rangle ) \\
\langle char \rangle \rightarrow \emptyset \mid \varepsilon \\
| x_0 \mid x_1 \mid \ldots
\]

where the $x_i$ characters are members of the alphabet $\Sigma$.†

†As we often do in grammars, we use the pipe symbol $|$ as a metacharacter, to collapse rules with the same left side. But pipe also appears in regular expressions themselves. For that usage it is wrapped in single quotes.
To cut down on parentheses we must establish operator precedence: star binds most tightly, then concatenation, then the pipe alternation operator, |. We use parentheses when needed, as in Example 3.3.

As to their semantics, what they mean, we will define that recursively. The language described by the single-character regular expression $\varnothing$ is the empty set, $L(\varnothing) = \varnothing$. The language described by the regular expression consisting of only the character $\varepsilon$ is the one-element language consisting of only the empty string, $L(\varepsilon) = \{\varepsilon\}$. If a regular expression consists of just one character from the alphabet then the language it describes by contains only one string and that string has only that single character: where $a \in \Sigma$, we have $L(a) = \{a\}$.

We finish by defining the semantics of the operations. We denote the language associated with the regular expression $R$ as $L(R)$. Then the pipe symbol means the union of the languages $L(R | \hat{R}) = L(R) \cup L(\hat{R})$. Concatenation of the regular expressions means concatenation of the languages $L(R \triangleright \hat{R}) = L(R) \triangleright L(\hat{R})$. And, the Kleene star of the regular expression describes the star of the language, $L(R^*) = L(R)^*$.

### Example 3.5
Let $\Sigma = \{a, b\}$. Consider the regular expression $aba^*$. It is the concatenation of $a$, $b$, and $a^*$. The first describes the single-element language $L(a) = \{a\}$. Likewise the second describes $L(b) = \{b\}$. Together, the string $ab$ describes the concatenation of the languages.

$L(a) \triangleright L(b) = \{ \sigma \in \Sigma^* \mid \sigma = \sigma_0 \triangleright \sigma_1 \text{ where } \sigma_0 \in L(a) \text{ and } \sigma_1 \in L(b) \} = \{ab\}$

The regular expression $a^*$ describes $L(a^*) = \{a^n \mid n \in \mathbb{N}\}$. Concatenating it to the language in the prior equation gives this.

$L(aba^*) = \{ \sigma \in \Sigma^* \mid \sigma = \sigma_0 \triangleright \sigma_1 \text{ where } \sigma_0 \in L(ab) \text{ and } \sigma_1 \in L(a^*) \} = \{ab, aba, abaa, aba^3, \ldots\} = \{aba^n \mid n \in \mathbb{N}\}$

We will illustrate with some constructs that appear often in building regular expressions. These examples use $\Sigma = \{a, b, c\}$.

### Example 3.6
Describe the language consisting of strings of a's whose length is a multiple of three, $L = \{a^{3k} \mid k \in \mathbb{N}\} = \{\varepsilon, aaa, aaaaaa, \ldots\}$, with the regular expression $(aaa)^*$.

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.

### Example 3.7
To match any character we can list them all. The language consisting of three-letter words ending in $bc$ is $\{abc, bbc, cbc\}$. The regular expression $(a|b|c)bc$ describes it. (If the alphabet is large then listing all characters is impractical; see the Extra section on Extended Regular Expressions.)
3.8 Example Describe the language of strings that have any number of a’s and optionally end in one b, \( L = \{ \varepsilon, b, a, ab, aa, aab, \ldots \} \), with the regular expression \( a^* (\varepsilon | b) \).

Similarly, to describe the language consisting of words with between three and five a’s, \( L = \{ aaa, aaaa, aaaaa \} \), use \( aaa (\varepsilon | a | aa) \).

3.9 Example The language \( \{ b, bc, bcc, ab, abc, abcc, aab, \ldots \} \) consisting of words starting with any number of a’s (including possibly zero-many a’s), followed by a single b, and then ending in fewer than three c’s is described by \( a^* b (\varepsilon | c | cc) \).

**Kleene’s Theorem** Our study of regular expressions is justified by the next result, that they describe the languages of interest.

3.10 Theorem (Kleene’s Theorem) A language is accepted by a Finite State machine if and only if that language is described by a regular expression.

We will prove this in separate halves. These halves use nondeterministic machines but since we can convert those to deterministic machines, the result holds for those machines.

3.11 Lemma If a language is described by a regular expression then there is a Finite State machine that accepts 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 \( L(R) = \{ \} \) and the machine on the left below accepts \( L(R) \). If \( R = \varepsilon \) then \( L(R) = \{ \varepsilon \} \) and the machine in the middle accepts this language. If the regular expression is a character from the alphabet, such as \( R = a \), then the machine on the right works.

![Diagram](image)

To finish, we handle the three operations. First consider alternation; suppose that \( R = R_0 | R_1 \), where \( R_0 \) and \( R_1 \) are regular. By the inductive assumption there is a machine \( M_0 \) whose language is described by \( R_0 \) and a machine \( M_1 \) whose language is described by \( R_1 \). Create the machine accepting 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 \( M_0 \) and \( M_1 \). See Example 2.14.

The next case is that \( R = R_0 \sim R_1 \). The inductive hypothesis gives a machine for each, \( M_0 \) and \( M_1 \). Join those two serially: for each accepting state in \( M_0 \), make an \( \varepsilon \) transition to the start state of \( M_1 \) and then convert all those accepting states of \( M_0 \) to be non-accepting states. See Example 2.15.

The final case is that \( R = (R_0)^* \). Take the machine \( M_0 \) accepting \( R_0 \) and for each accepting state that is not the start state, make an \( \varepsilon \) transition to the start state. Make the start state an accepting state. See Example 2.16.
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 accepted by a Finite State machine is described by a regular expression.

Our strategy is to start with a Finite State machine and eliminate its states one at a time, keeping the accepted language the same. Below is an illustration, before and after pictures of part of a larger machine where we eliminate the state $q_1$.

In the after picture the edge is labelled $ab$, 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 until there are only two left, with one edge between them, and that edge will be labelled with the desired expression.

Before the proof, an example. Consider the machine on the left below. Take $A$, $B$, $C$, and $D$ to be regular expressions.

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 the accepted language here is the same as that of 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 first transform the machine to a new one $\hat{\mathcal{M}}$, 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 an edge connecting it to $f$ (by the prior paragraph there is at least one such accepting state). Label those edges with $\epsilon$. Convert all the accepting states to non-accepting states and make $f$ accepting.

Next introduce a new start state $e$. Give it an edge leading to $\mathcal{M}$'s $q_0$, labelled with $\epsilon$. (Ensuring that $\hat{\mathcal{M}}$ has at least two states is a technical convenience to handle machines of all sizes uniformly.)

Because the edge labels are regular expressions, we can arrange $\hat{\mathcal{M}}$ so that leading from any $q_i$ to any $q_j$ is at most one edge because if $\mathcal{M}$ has more than one edge then use the pipe $|$ to combine their labels.

Do the same with loops, that is, cases where $i = j$. Like the prior ones, this transformation clearly 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 $q \neq f$ has no outgoing edges then drop it, along with the edges that come into it. The language of the machine will not change because this state cannot lead to a string being accepted — it doesn't lead anywhere so it cannot lead to an accepting state, and $q$ cannot itself be an accepting state as only $f$ is an accepting state.

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. (Construct the set of reachable states, and with it the set of unreachable states, in the natural way. Begin with the start state $S_0 = \{e\}$. Find all states reachable from $e$ in a single transition by following every edge out of $e$, $\epsilon$ or not, to get $S_1 = \{e, q_{1,0}, \ldots q_{1,k}\}$. Then iterate: starting with a set $S_i$ of nodes that are reachable in $i$-many steps, for each $q \in S_i$ follow each outbound edge for a single step. The collection of the nodes reached in this way is $S_{i+1}$. Stop the iteration when $S_i = S_{i+1}$, at which point it is the set of ever-reachable states. The unreachable ones are the others.)

With that, we have $\hat{\mathcal{M}}$ and are ready for state elimination.

Below are the before and after pictures of the elimination of one state. (This is the general version of the example before the proof.) The before picture applies to any $q \in \mathcal{M}$ with $q \neq e$ and $q \neq f$. It has at least one incoming and at least one outgoing edge, by the setup work above. It may have a loop.
There is a tricky point here: possibly some of the states shown on the left of each of the two pictures equal some shown on the right. For example, in the before picture possibly $q_{i0}$ equals $q_{o0}$. If so then the shown edge $R_{i0, o0}$ 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 accepted by the machine is unchanged.

Perform this elimination for every state that is not $e$ or $f$. There are only finitely many such states so this procedure must eventually stop. When it does, all that remains are $e$, $f$, and an edge. The edge label is the desired regular expression.

3.14 Example  Consider $M$ on the left. Introduce $e$ and $f$ to get $\hat{M}$ on the right.

Start by eliminating $q_2$. In the terms of the proof’s key step, $q_1 = q_{i0}$ and $q_0 = q_{o0}$. The regular expressions are $R_{i0} = a$, $R_{o0} = b$, $R_{i0, o0} = b$, and $R_{\ell} = b$. That gives this machine.

Next eliminate $q_1$. There is one incoming node $q_0 = q_{i0}$ and two outgoing nodes $q_0 = q_{o0}$ and $f = q_{o1}$. (Note that $q_0$ is both an incoming and outgoing node; this is the tricky point mentioned in the proof.) The regular expressions are $R_{i0} = a$, $R_{o0} = b | (ab^*b)$, and $R_{o1} = \varepsilon$. 
Almost done. All that remains is to eliminate $q_0$. The sole incoming node is $e = q_0$, and the sole outgoing node is $f = q_0$, and so $R_{i_0} = \varepsilon, R_{o_0} = \varepsilon | a \varepsilon$, and $R_{f} = \varepsilon | a(b|ab * b)$. (That regular expression simplifies. For instance, $a \varepsilon = a$.)

IV.3 Exercises

3.15 Decide if the string $\sigma$ matches the regular expression $R$. (A) $\sigma = 0010$, $R = \emptyset * 10$ (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, if possible, five bitstrings that match and five that do not. (A) $01 *$ (B) $(01) *$ (C) $1(0|1)1$ (D) $(0|1)(\varepsilon|1)0*$ (E) $\emptyset$

✓ 3.17 For these regular expressions, for each element of $\{a,b\}^*$ that is of length at most 3, decide if it is a match. (A) $a * b$ (B) $a *$ (C) $\emptyset$ (D) $\varepsilon$ (E) $b(a|b) a$

✓ 3.18 For these regular expressions, decide if each element of $\mathbb{B}^*$ of length at most 4 is a match. (A) $0 * 1$ (B) $1 * 0$ (C) $\emptyset$ (D) $\varepsilon$ (E) $0(0|1)*$ (F) $(100)(\varepsilon|1)0*$

✓ 3.19 A friend says to you, “Kleene star means ‘match the inside and repeat’ so the regular expression $(0 * 1)^*$ matches the strings 001001 and 010101 but not 01001 and 00000101, where the substrings are unequal.” Straighten them out.

3.20 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.

✓ 3.21 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.

✓ 3.22 Give a regular expression for the set of bitstrings whose number of 0’s is even. Give a regular expression for the set of bitstrings whose number of 1’s is greater than 2. Give a regular expression for the set of bitstrings whose number of 0’s is even and whose number of 1’s is greater than 2.

3.23 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.
3.24 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. (B) Each string has at least two b’s that are not followed by an a. (C) Each string has no more than one pair of consecutive a’s and no more than one pair of consecutive b’s.

3.25 Produce a regular expression for the language of bitstrings such that the number of 1’s is a multiple of four.

3.26 Give a regular expression to accept each language.
   (A) \( \{ \sigma \in \{a, b\}^* \mid \sigma \text{ ends with the same symbol it began with, and } \sigma \neq \epsilon \} \)
   (B) \( \{ a^i b a^j \mid i \text{ and } j \text{ leave the same remainder on division by three} \} \)

3.27 Give a regular expression for each language over \( \mathbb{B}^* \).
   (A) The strings representing a binary number that is a multiple of eight.
   (B) The bitstrings where the first character differs from the final one.
   (C) The bitstrings where no two adjacent characters are equal.

3.28 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.

3.29 Fix a Finite State machine \( M \). Kleene’s Theorem shows that the set of strings taking \( M \) from the start state to the set of final states is regular.
   (A) Show that for any set of states \( S \subseteq Q_M \) the set of strings taking \( M \) from the start state to one of the states in \( S \) is regular.
   (B) Show that the set of strings taking \( M \) from any single state to any other single state is regular.

3.30 Part of the proof of Lemma 3.13 involves removing unreachable states. For each of these machines, follow the procedure described there to find the set of unreachable states.

3.31 Show that the set of languages over \( \Sigma \) that are described by a regular expression is countable. Conclude that there are languages not accepted by any Finite State machine.

3.32 Construct the parse tree for these regular expressions over \( \Sigma = \{a, b\} \).
   (A) \( a(b | c) \)  (B) \( ab^*(a | c) \)

3.33 Construct the parse tree for Example 3.3’s \( a(b | c)^* \) and \( a(b^* | c^*) \).

3.34 Get a regular expression by applying the method of Lemma 3.13’s proof (A) to the machine on the left (B) to the machine on the right.
3.35 Find the regular expression accepted by the machine by applying the method of Lemma 3.13's proof (A) to the machine on the left (B) to the machine on the right.

3.36 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 tricky point.

3.37 An alternative proof of Lemma 3.11 reverses the steps of Lemma 3.13. This is the subset method. Start by labelling the single edge on a two-state machine with the given regular expression.

Then instead of eliminating nodes, introduce them.

(This decomposes the regular expression while the proof of Lemma 3.11 builds it up from atomic parts, but they are alike in connecting the structure of the machine with the structure of the expression.) Use this approach to get a machine that accepts the language described by the following regular expressions. (A) $a|b$ (B) $ca^*$ (C) $(a|b)c^*$ (D) $(a|b)(b^*|a^*)$

**SECTION IV.4 Regular languages**

**Definition** Finite State machines are interesting because, at least, they give us insight into what a machine can do without scratch memory. We can get an idea of
how powerful such a machine is by determining the set of jobs that they can and cannot do.

4.1 Definition A regular language is one that is accepted by some Finite State machine.

There are infinitely many regular languages. For one thing, every finite language \( L_0 = \{ \} \), \( L_1 = \{ 0 \} \), \( L_2 = \{ 00 \} \), \( \ldots \) is regular; just list all the cases as in Example 1.8.

If the language is finite but large then constructing the machine by listing cases may not be practical. For example, there are finitely many people in the world and each has finitely many active phone numbers so strictly speaking the set of all currently-active phone numbers is a regular language but constructing a Finite State machine for it is silly.†

Closure properties In the proof of Kleene’s Theorem we saw that if two languages are regular then their union is also regular. We have also seen, in Example 2.15, that if two languages are regular then their concatenation is regular. We say that a structure is closed under an operation if performing that operation on its members always yields another member.

4.2 Lemma The collection of regular languages is closed under union, concatenation, and Kleene star.

Proof In proving Lemma 3.11, the first half of Kleene’s Theorem, we did the regular expression operations of pipe, concatenation, and Kleene star. Those correspond to these set operations: for instance, if \( R_0 \) is a regular expression describing the language \( L_0 \), and \( R_1 \) describes \( L_1 \), then the regular expression \( R_0 \mid R_1 \) describes \( L_0 \cup L_1 \).

4.3 Theorem The collection of regular languages is closed under set complement, intersection, and set difference.

Proof The equation \( S \cap \bar{S} = (S^c \cup \bar{S}^c)^c \) describes set intersection in terms of union and complement. The prior lemma shows that regular languages are closed under union, so if we show they are closed under complement then this equation gives that they are also closed under intersection. Similarly, the equation \( S - \bar{S} = S \cup \bar{S}^c \) means that showing closure under complement will show also closure under set difference.

So to show that the complement of a regular language is also regular, fix a regular language \( L \). It is accepted by some Finite State machine \( M \).

† A finite regular language doesn’t have to be large for it to be difficult, in some sense. For example, consider Goldbach’s conjecture, that every even number greater than 2 is the sum of two primes, as in \( 4 = 2 + 2 \), \( 6 = 3 + 3 \), \( 8 = 3 + 5 \), \( \ldots \). Computer testing shows that this pattern continues to hold, up to very large numbers, but no one knows if it is true for all evens. Now consider the set consisting of the string \( \sigma \in \{0, \ldots, 9\}^* \) representing 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.
Define a new machine $\hat{M}$ with the same states and transition function as $M$ but whose accepting states are the complement, $F_{\hat{M}} = Q_M - F_M$. Because $M$ is deterministic, each input string $\tau$ is associated with a unique last state, $q_\tau$, the state the machine is in after it consumes $\tau$’s last character. Moving from $M$ to $\hat{M}$ by taking the complement of the accepting states will complement the set of accepted strings, the language of the machine, because $q_\tau \in F_M$ if and only if $q_\tau \notin F_{\hat{M}}$. Thus the language of $\hat{M}$ is the complement of the language of $M$ and since this language is accepted by a Finite State machine, it too is regular.

4.4 Example  This language is regular.

$$\{ \sigma \in \mathbb{B}^* | \sigma \text{ has an even number of 0's and more than two 1's} \}$$

One way to show that is to produce a Finite State machine. Another is to spot that $L$ is the intersection of $L_0 = \{ \sigma \in \mathbb{B}^* | \sigma \text{ has an even number of 0's} \}$ and $L_1 = \{ \sigma \in \mathbb{B}^* | \sigma \text{ has more than two 1's} \}$. Producing machines for those two is easy.

We now know that regular expressions, regular grammars, deterministic finite state machines, and nondeterministic finite state machines all describe the same set of languages, namely, the regular languages. The fact that we can describe these languages in so many completely different ways suggests that there’s something natural and important about these languages. This is in analogy with the fact that the equivalence of Turing machines, the lambda calculus, and all kinds of other things suggests that the computable languages are natural and important. They’re not just an artifact of whatever random decisions the original definer made.

IV.4 Exercises

✓ 4.5 True or false? Of course you must justify each answer.
   (A) Every regular language is finite.
   (B) Over $\mathbb{B}^*$, the empty language is not regular.
   (C) Over $\mathbb{B}^*$, the language of all strings, $\mathbb{B}^*$, is not regular.
   (D) Every regular language is countable.
   (E) Every Finite State machine accepts at least one string.
   (F) For every Finite State machine there is one that has fewer states but accepts the same language.

✓ 4.6 One of these is true and one is false. Which is which? (A) Any finite language is regular. (b) Any regular language is finite.

✓ 4.7 Show that each language over $\Sigma = \{a, b\}$ is regular.
   (A) $\{ \sigma \in \Sigma^* | \sigma \text{ starts and ends with } a \}$
   (B) $\{ \sigma \in \Sigma^* | \text{ the number of a's is even} \}$

4.8 True or false? Justify your answer.
   (A) If $L_0$ is a regular languages and $L_1 \subseteq L_0$ then $L_1$ is also a regular languages.
(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.

\( \checkmark \) 4.9 Suppose that the language \( \mathcal{L} \) over \( \mathbb{B} \) is regular. Show that the language \( \hat{\mathcal{L}} = \{ 1^\sim \sigma \mid \sigma \in \mathcal{L} \} \), also over \( \mathbb{B} \), is also regular.

\( \checkmark \) 4.10 Prove that the collection of regular languages over \( \Sigma \) is closed under each of the operations.
   (A) \( \text{pref}(\mathcal{L}) = \{ \sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \sigma \sim \tau \in \mathcal{L} \} \)
   (B) \( \text{suff}(\mathcal{L}) = \{ \sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \tau \sigma \in \mathcal{L} \} \)

4.11 The proof of Theorem 4.3 works with deterministic Finite State machines. Find a nondeterministic Finite State machine \( \mathcal{M} \) such that producing another machine \( \hat{\mathcal{M}} \) by taking the complement of the accepting states, \( F_{\hat{\mathcal{M}}} = (F_\mathcal{M})^c \), will not result in the language of the second machine being the complement of the language of the first.

\( \checkmark \) 4.12 Show that the set of regular languages is countable.

4.13 Prove that the language accepted by a Finite State machine with \( n \) states is infinite if and only if that machine accepts at least one string of length \( k \), where \( n \leq k < 2n \).

4.14 Fix an alphabet \( \Sigma \neq \emptyset \). Is the set of regular languages over \( \Sigma \) infinite? If so, is it countable? Is the set of languages over \( \Sigma \) that are not regular infinite? If so, is it countable?

4.15 Fix two alphabets \( \Sigma_0, \Sigma_1 \). A function \( h : \Sigma_0 \rightarrow \Sigma_1^* \) induces a homomorphism on \( \Sigma_0^* \) via the operation \( h(\sigma \sim \tau) = h(\sigma) \sim h(\tau) \) (and \( h(\varepsilon) = \varepsilon \)).
   (A) Take \( \Sigma_0 = \mathbb{B} \) and \( \sigma_1 = \{ a, b \} \). Fix a homomorphism \( \hat{h}(\emptyset) = a \) and \( \hat{h}(1) = ba \).
      Find \( \hat{h}(\emptyset 0), \hat{h}(10), \) and \( \hat{h}(101) \).
   (B) Define \( \hat{h}(\mathcal{L}) = \{ h(\sigma) \mid \sigma \in \Sigma_0^* \} \). Let \( \hat{\mathcal{L}} = \{ \sigma \sim 1 \mid \sigma \in \mathbb{B}^* \} \); 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 \( h(\mathcal{L}) \).

4.16 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 \( (\sigma_0 \sim \sigma_1)^R = \sigma_1^R \sim \sigma_0^R \).
      Hint: you can use induction on the length of \( \sigma_1 \).
   (B) One way to prove the result is to define a reversal operation on regular expressions. Fix an alphabet \( \Sigma \). Let \( \mathcal{R} \) be the set of regular expressions over that alphabet. Then we can define each of these: (i) \( \emptyset^R = \emptyset \) (ii) \( \varepsilon^R = \varepsilon \) (iii) \( (\mathcal{R} \times \Sigma) = R \times x \) for any \( x \in \Sigma \) (iv) \( (R_0 \sim R_1)^R = R_1 R_0^R \) (v) \( R_0 | R_1^R = R_0^R | R_1^R \) (vi) \( R^* R = R R^* \). (Note the relationship between (iv) and the prior exercise.
item.) Now show that this operation turns a regular expression for a language into a regular expression for the language’s reversal.

(c) Another way to prove this is to reverse the machine. Take the language to be accepted by some machine $M$. Construct a new nondeterministic Finite State machine $\hat{M}$ by: (i) giving $\hat{M}$ all of $M$’s states, so that $Q_{\hat{M}} = Q_M$. (ii) adding to $\hat{M}$ a new state $s$, make $\epsilon$ transitions from it to all of $M$’s accepting states, and make $s$ the start state of $\hat{M}$ (iii) reverse all of the transitions in the $M$’s graph, and (iv) turn $M$’s accepting states to non-accepting states, and turn $M$’s start state to an accepting state. Show that $\hat{M}$ accepts $L^R$.

4.17 Consider the set of powers of two $\{2^n \mid n \in \mathbb{N}\}$.

(A) Show that there is a Finite State machine that accepts the collection of strings representing powers of 2 in binary.

(b) Show that there is no Finite State machine that accepts the collection of strings representing these numbers in decimal.

**Section IV.5 Languages that are not regular**

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 cause a pattern in what a machine accepts. The diagram shows a machine that accepts $aabb\,b$ (it only shows some of the states, those that the machine traverses in processing this input).

Besides $aabb\,b$, this machine must also accept $a(\text{abb})^2b\,c$ because that string takes the machine through the loop twice, and then to the accepting state. Likewise, this machine accepts $a(\text{abb})^3b\,c$, and looping more times pumps out more accepted strings.

**Theorem (Pumping lemma)** Let $L$ be a regular language. Then there is a constant $p \in \mathbb{N}$, the pumping length for the language, such that every string $\sigma \in L$ with $|\sigma| \geq p$ decomposes into three components $\sigma = \alpha \gamma$ satisfying: (1) the first two components are short, $|\alpha \gamma| \leq p$, (2) $\beta$ is not empty, and (3) all of the strings $\alpha \gamma, \alpha \gamma^2, \alpha \gamma^3, \ldots$ are also members of the language $L$. 
Proof  Suppose that $L$ is accepted by the Finite State machine $M$. Denote the number of states in $M$ by $p$. Consider $\sigma = \langle s_0, \ldots, s_k \rangle$ 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 (not necessarily distinct) states that the machine visits is one more than the number of transitions. Thus, in processing the first $p$ characters of the input string $\sigma$, the machine must visit some state more than once, that is, it must loop, by the Pigeonhole Principle.

Fix just such a repeated state $q$. Fix the first two substrings of $\sigma$, $\langle s_0, \ldots, s_i \rangle$ and $\langle s_0, \ldots, s_j \rangle$ with $i < j$ that take the machine to state $q$, meaning that $j$ is minimal such that the extended transition function gives $\hat{\Delta}(\langle s_0, \ldots, s_i \rangle) = \hat{\Delta}(\langle s_0, \ldots, s_j \rangle) = q$. Let $\alpha$ be the string that first brings the machine to state $q$, namely $\langle s_0, \ldots, s_i \rangle$. Let $\beta$ be the string that brings the machine around the loop, $\langle s_{i+1}, \ldots, s_j \rangle$. Finally, let $\gamma$ be the rest, $\langle s_{j+1}, \ldots, s_k \rangle$. (Possibly one or both of $\alpha$ and $\gamma$ is empty.) These strings satisfy conditions (1) and (2). (Choosing $q$ to be a state that is repeated within the initial segment of $\sigma$, and choosing $i$ 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 pick an $\alpha$ that includes the first ninety nine loops, and that therefore is longer than $p$.)

For condition (3),

$$\alpha^2 \gamma = \langle s_0, \ldots, s_i, s_{j+1}, \ldots, s_k \rangle$$

brings the machine from the start state $q_0$ to $q$, and then to the same ending state as did $\sigma$. That is, $\hat{\Delta}(\alpha \gamma) = \hat{\Delta}(\alpha \beta \gamma)$ and so is an accepting state. The other strings in (3) are similar. For instance, for

$$\alpha \beta^2 \gamma = \langle s_0, \ldots, s_i, s_{i+1}, \ldots, s_{j-1}, s_{i+1}, \ldots, s_{j+1}, \ldots, s_k \rangle$$

the substring $\alpha$ brings the machine from $q_0$ to the state $q$, the first $\beta$ brings it around to $q$ again, then the second $\beta$ makes the machine loop to $q$ yet again, and finally $\gamma$ brings it to the same ending state as did $\sigma$.  

Typically we use the Pumping lemma to show that a language is not regular, by assuming that it is regular and then using the lemma to get a 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 the two parentheses $\Sigma = \{ \), ( \}.

$$L = \{(n^n \in \Sigma^* | n \in \mathbb{N}\} = \{ \epsilon, ( ), (()), ((()), (^{4})^4, \ldots \}$$

For contradiction, assume that it is regular. Then the Pumping lemma says that $L$ has a pumping length, $p$.

Consider the string $\sigma = (p)^p$. It is an element of $L$ and its length is greater than or equal to $p$ so the Pumping lemma says that it 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 $p$. Thus both of the substrings $\alpha$ and $\beta$ are
composed exclusively of open parentheses, (’s. Condition (2) is that $\beta$ is not the empty string, so it contains at least one (.

Condition (3) is that all of the strings $\alpha \gamma$, $\alpha \beta^2 \gamma$, $\alpha \beta^3 \gamma$, … are members of $L$. For the contradiction consider $\alpha \beta^2 \gamma$. Compared with $\sigma = \alpha \beta \gamma$, this string has an extra $\beta$. That adds at least one open parenthesis, without adding any balancing closed parentheses. In short, $\alpha \beta^2 \gamma$ has more (’s than )’s and so it is not a member of the language $L$. But this contradicts the Pumping lemma, and therefore the assumption that $L$ is regular is incorrect.

Matching parentheses, and other types of matching, is something that we often do. For instance, compilers verify that parentheses match. So that example shows that for many common computing tasks, regular languages don’t suffice.

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 $L = \{ \sigma \in \Sigma^* \mid \sigma^R = \sigma \}$ of all palindromes over $\Sigma = \{a, b\}$ is not regular.

For contradiction assume that this language is regular. The Pumping lemma says that $L$ has a pumping length. Call it $p$.

Consider $\sigma = a^pba^p \in L$. That string has more than $p$ characters so 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) is that the strings in the list $\alpha \gamma$, $\alpha \beta^2 \gamma$, $\alpha \beta^3 \gamma$, … are also members of the language. We will get the contradiction by showing that the first one, $\alpha \gamma$, is not a palindrome. Compared to $\sigma = \alpha \beta \gamma$, in $\alpha \gamma$ the $\beta$ is gone. The substring $\beta$ does not have the $b$, because $\alpha$ and $\beta$ consist entirely of $a$’s, $\gamma$ has the $b$. By the definition of $\sigma$, after that $b$ comes $a^p$. But, for $\alpha \gamma$ to be a palindrome, the $b$ must be preceded by $p$-many $a$’s and by omitting $\beta$ we’ve omitted at least one $a$. So $\alpha \gamma$ is not a palindrome, which is a contradiction because the Pumping lemma says that the strings in the list are all members of $L$.

5.4 Remark Don’t make the mistake of thinking that the parts of $\sigma$ must match with the parts of its decomposition $\alpha \beta \gamma$. For instance, in the prior example a person might think that the Pumping lemma says that the decomposition of $a^pba^p$ into $\alpha \beta \gamma$ puts $\alpha$ as the initial $a^p$, etc. But the Pumping lemma doesn’t say that. It only says that the first two strings together, $\alpha \beta$, consists exclusively of $a$’s. As far as the statement goes, the suffix $\gamma$ could well start with some $a$’s, then followed by $ba^p$. Likewise, in the matched parentheses example, the Pumping lemma does not say that $\alpha \beta \gamma$ gets all of the open parentheses, and $\gamma$ could well have some.

5.5 Example Consider $L = \{ \emptyset^m1^n \in \mathbb{B}^* \mid m = n + 1 \} = \{ \emptyset, 001, 00011, \ldots \}$. Its members have one fewer 1 than $\emptyset$. We will prove that it is not regular.

For contradiction assume otherwise, that it is regular, and set $p$ as its pumping length. Consider $\sigma = \emptyset^{p+1}1^p \in L$. Because $|\sigma| \geq p$, the Pumping lemma gives the 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 $\emptyset$’s. Condition (2) says that $\beta$
Section 5. Languages that are not regular

has at least one character, necessarily a \( \theta \). Condition (3) says that each of these is in the language: \( \alpha \gamma \), \( \alpha \beta_2 \gamma \), \( \alpha \beta_3 \gamma \), \ldots Consider the first, \( \alpha \gamma \). By (1) all of the 1's are in \( \gamma \). With that, by (2) the string \( \alpha \gamma \) has fewer \( \theta \)'s then does \( \sigma \) but the same number of 1's. That means \( \alpha \gamma \not\in L \), which contradicts the Pumping lemma.

We can interpret the prior example to say that Finite State machines cannot correctly recognize a predecessor-successor relationship. We can also use the Pumping lemma to show Finite State machines cannot accept other arithmetic relations.

5.6 Example The language \( L = \{ a^n \mid n \text{ is a perfect square} \} = \{ \epsilon, a, a^4, a^9, a^{16}, \ldots \} \) is not regular. For, suppose otherwise. Fix a pumping length \( p \) and consider \( \sigma = a^{(p^2)} \), so that \( |\sigma| = p^2 \).

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 \( 0 < |\beta| \). Condition (3) say that all of the strings \( \alpha \gamma \), \( \alpha \beta \gamma \), \( \alpha \beta \gamma \), \ldots are members of the language.

Consider the gap between the length \( |\sigma| = |\alpha \beta \gamma| \) and the length \( |\alpha \beta^2 \gamma| \). Because \( 0 < |\beta| \leq p \), that gap is at most \( p \). But the definition of the language is that after \( \sigma \) the next-longest string has length \( (p + 1)^2 = p^2 + 2p + 1 \), which is strictly greater than \( p \). Thus the length of \( \alpha \beta^2 \gamma \) is not a perfect square, which contradicts the Pumping lemma’s assertion that \( \alpha \beta^2 \gamma \in L \).

Sometimes we can use the Pumping lemma in conjunction with the closure properties of regular languages to solve problems.

5.7 Example The language \( L = \{ \sigma \in \{ a, b \}^* \mid \sigma \text{ has as many } a \text{'s as } b \text{'s} \} \) is not regular. To prove that, note that the language \( \hat{L} = \{ a^m b^n \in \{ a, b \}^* \mid m, n \in \mathbb{N} \} \) is clearly regular, described by the regular expression \( a*b* \), and recall that the intersection of two regular languages is regular. But \( L \cap \hat{L} \) is the set \( \{ a^n b^n \mid n \in \mathbb{N} \} \) and Example 5.2 shows it 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 accepts 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 accepting 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

5.8 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 the first string on the list, \( \alpha \gamma \), is not a member of the language.
5.9 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) \( L_0 = \{a^nb^m \mid n + 2 = m \} \) 
(B) \( L_1 = \{a^nb^mc^n \mid n, m \in \mathbb{N} \} \) 
(c) \( L_2 = \{a^nb^m \mid n < m \} \)

5.10 Show that the language over \( \{a, b\}^* \) consisting of strings having more a’s than b’s is not regular.

5.11 For each language, 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. Assume that \( \Sigma = \{a, b\} \).

(A) \( \{\sigma \in \Sigma^* \mid \sigma \in \Sigma^* \} \) 
(B) \( \{a^nb^m \in \Sigma^* \mid n, m \in \mathbb{N} \} \)
(c) \( \{a^nb^m \in \Sigma^* \mid n + 3 = m \} \)
(d) \( \{a^nb^m \in \Sigma^* \mid m - n > 12 \} \)

5.12 One of these is regular and one is not. Which is which? Of course, you must prove your assertions. (A) \( \{a^nb^m \in \{a, b\}^* \mid n = m^2 \} \) 
(B) \( \{a^nb^m \in \{a, b\}^* \mid 3 < m, n \} \)

5.13 Use the Pumping lemma to prove that \( L = \{a^{m-1}cb^m \mid m \in \mathbb{N}^+ \} \) is not regular. (It may help to first produce five strings from the language.)

5.14 Is this \( \{\sigma \in \mathbb{B}^* \mid \sigma = \alpha\beta\alpha^R \text{ for } \alpha, \beta \in \mathbb{B}^* \} \) regular? Either way, prove it.

5.15 Prove that \( L = \{\sigma \in \{1\}^* \mid |\sigma| = n! \text{ for some } n \in \mathbb{N} \} \) is not regular. Hint: the differences \((n + 1)! - n!\) grow without bound.

5.16 One of these is regular, one is not: \( \{0^m10^n \mid m, n \in \mathbb{N} \} \) and \( \{0^n10^n \mid n \in \mathbb{N} \} \). Which is which? Of course, you must prove your assertions.

5.17 Show that there is a Finite State machine that accepts this language of all sums totalling less than four, \( L_4 = \{a^ib^jc^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k \text{ and } k < 4.\} \). Use the Pumping lemma to show that no Finite State machine accepts the language of all sums, \( L = \{a^ib^jc^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k.\} \).

5.18 Rebut someone who says to you, “Sure, for the machine in the introduction on section 5, a single loop will cause \( \sigma = \alpha \beta \gamma \). But if you had a double loop like below then you need a longer decomposition.”

5.19 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.

5.20 Show that \( \{0^m1^n \in \mathbb{B}^* \mid m \neq n \} \) is not regular. Hint: use the closure properties of regular languages.

5.21 Show that \( \{\sigma \in \mathbb{B}^* \mid \sigma = 1^n \text{ where } n \text{ is prime} \} \) is not a regular language. Hint: in a sequence of numbers with a constant positive difference, some term is not prime.
5.22 Consider \{ a^i b^j c^i \mid i, j \in \mathbb{N} \}.
   (A) Give five strings from this language.
   (B) Use the Pumping lemma to show that it is not regular.

5.23 The language \( L \) described by the regular expression \( a^* b b b^* \) 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 accepts the language. Here that gives \( p = 4 \).
   (A) Consider \( \sigma = a b b b \). Give a decomposition \( \sigma = \alpha \beta \gamma \) that satisfies the three conditions.
   (B) Do the same for \( \sigma = b^{15} \).

5.24 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 \( L \) described by \((\epsilon 0 1)^*\). Construct a deterministic Finite State machine with three states that accepts this language.
   (B) Show that the minimal pumping length for \( L \) is 1.

5.25 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 accepts the finite language \( L_1 = \{ \epsilon, a \} \) must have at least two accepting states.
   (B) Show that any Deterministic Finite State machine that accepts \( L_2 = \{ \epsilon, a, a a \} \) must have at least three accepting states.
   (C) Show that for any natural number \( n \) there is a regular language that is not recognized by any Deterministic Finite State machine with at most \( n \) final states.

Section IV.6 Minimization

Contrast these two Finite State machines. For each, the language of accepted strings is \( \{ \sigma \in B^* \mid \sigma \text{ has at least one } 0 \text{ and at least one } 1 \} \).

Our experience from making machines is that the states should have a meaning. For instance, on the left \( q_2 \) means something like, “have seen the needed at least
one 1 but still waiting for a 0.” In a properly designed machine, input puts the machine in state \( q_2 \) if indeed that input has so far had at least one 1 but no 0.

The machine on the right doesn’t satisfy this design principle because the meaning of \( q_4 \) is the same as that of \( q_2 \), and \( q_3 \)'s meaning is the same as \( q_5 \)'s. That is, the pairs of states have the same future. This machine has redundant states.

We will give an algorithm that starts with a Finite State machine and from it finds the smallest machine that accepts the same language. The algorithm works by by collapsing 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 \( \sigma = \emptyset \) ends in the non-accepting state \( q_1 \), while starting it in \( q_2 \) and processing the same input ends in the accepting state \( q_3 \). So \( q_0 \) and \( q_2 \) are \( n = 1 \)-distinguishable, and therefore are distinguishable.

Also, \( q_2 \) and \( q_3 \) are 0-distinguishable, via \( \sigma = \varepsilon \). Any state that is not accepting is 0-distinguishable from any state that is accepting.

6.3 **Example** For the machine on the right, the states \( q_2 \) and \( q_4 \) are indistinguishable. The set of input strings that bring the machine from \( q_2 \) to an accepting state is \( \{ \sigma \in B^* \mid \sigma \text{ has the form } \emptyset^* | (1^*0^*) \} \). For state \( q_4 \) the set of strings taking it to an accepting state satisfies the same regular expression.

The states that the machine is brought to need not be equal, it is a matter of whether they are accepting or not. For instance, if \( \sigma = \emptyset \) then starting at \( q_2 \) takes the machine to \( q_3 \) while starting at \( q_4 \) takes the machine to \( q_5 \). This is not a distinguishing string because while the states aren’t equal, both are accepting.

6.4 **Lemma** The \( \sim \) relation and the \( \sim_n \) relations are equivalences.

**Proof** Exercise 6.15.

Our algorithm first finds all states that are distinguishable by the length zero string, next finds all states distinguishable by length zero or one strings, etc. At the end the machine’s states are broken into classes where inside each class the states are indistinguishable by strings of any length. Those classes serve as the states of the minimal machine. We first outline the steps, then we will work through two complete examples.

So consider again the machine with redundant states that we saw in (\(*\)) above.

†This is Moore’s algorithm. It is easy to understand and suitable for small calculations but if you are writing code then be aware that another algorithm, Hopcroft’s algorithm, is better, albeit more complex.
The states that are indistinguishable by the length zero string $\varepsilon$ fall into two classes, the non-accepting states $Q - F = \{q_0, q_1, q_2, q_4\}$ accepting states $F = \{q_3, q_5\}$. We use this notation for the equivalence classes of the $\sim_0$ relation.

$$\mathcal{E}_{0,0} = Q - F = \{q_0, q_1, q_2, q_4\} \quad \mathcal{E}_{0,1} = F = \{q_3, q_5\}$$

Next we find the classes of states that are 1-indistinguishable, that are indistinguishable by either length zero or length one strings. One way to proceed is: for each state we find where putting the machine in that state and feeding it the length zero string $\varepsilon$ and the length one strings 0 and 1 will take the machine. In 'take the machine' we only care whether the resulting state is a member of $\mathcal{E}_{0,0}$ or of $\mathcal{E}_{0,1}$. (The first column repeats the prior paragraph. We'll streamline the algorithm below.)

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$\mathcal{E}_{0,0}$</td>
<td>$\mathcal{E}_{0,0}$</td>
<td>$\mathcal{E}_{0,0}$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$\mathcal{E}_{0,0}$</td>
<td>$\mathcal{E}_{0,0}$</td>
<td>$\mathcal{E}_{0,1}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\mathcal{E}_{0,0}$</td>
<td>$\mathcal{E}_{0,1}$</td>
<td>$\mathcal{E}_{0,0}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$\mathcal{E}_{0,1}$</td>
<td>$\mathcal{E}_{0,1}$</td>
<td>$\mathcal{E}_{0,1}$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$\mathcal{E}_{0,0}$</td>
<td>$\mathcal{E}_{0,1}$</td>
<td>$\mathcal{E}_{0,0}$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$\mathcal{E}_{0,1}$</td>
<td>$\mathcal{E}_{0,1}$</td>
<td>$\mathcal{E}_{0,1}$</td>
</tr>
</tbody>
</table>

Some states are distinguishable, some are not. For instance, $q_0$ and $q_1$ are 1-distinguishable by the input string 1, while and $q_1$ and $q_2$ are 1-distinguishable by 0. But $q_2$ and $q_4$ are 1-indistinguishable. These are the $\sim_1$ equivalence classes.

$$\mathcal{E}_{1,0} = \{q_0\} \quad \mathcal{E}_{1,1} = \{q_1\} \quad \mathcal{E}_{1,2} = \{q_2, q_4\} \quad \mathcal{E}_{1,3} = \{q_3, q_5\} \quad (**)$$

The states that we spotted by eye as redundant, $q_2, q_4$ and $q_3, q_5$ continue to be together.

To get the $\sim_2$ classes we may think to expand the table to add four columns, one for each of $\emptyset, 01, 10, \text{ and } 11$. But proceeding in this way, at the next step doing all strings of length three strings, etc., would make the time to do the job grow exponentially since there are $2^k$ strings of length $k$.

The way to streamline the algorithm is to ask how two states $q, \hat{q}$ could be 2-distinguishable but not 1-distinguishable. This happens only if, where the distinguishing string is $\sigma$, the two states are distinguished by $\sigma$'s final character, simply because if an earlier character would do then they would be 1-distinguishable.

In general, let the states $q$ and $\hat{q}$ be $n+1$-distinguishable but not $n$-distinguishable. Suppose that a distinguishing string is $\sigma = \langle s_0, s_1, \ldots s_{n-1}, s_n \rangle = \tau \sim s_n$. Because the states are not $n$-distinguishable, where the prefix $\tau$ brings the machine from $q$ to a state $r$ in some class $\mathcal{E}_{n,i}$, then $\tau$ must bring the machine from $\hat{q}$ to some $\hat{r}$ in the same class, $\mathcal{E}_{n,i}$. So the distinguishing must involve $\sigma$'s final character, $s_n$, taking $r$ to a state in one class and taking $\hat{r}$ to a state in another.
Therefore, at each step we don’t need to test whole strings, we need only test single characters to see if they split the equivalence classes, the $E_{n,i}$’s. And we need only test classes with more than one character, because singleton classes cannot be split. These are the relevant single-character computations.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_2$</td>
<td>$E_{1,3}$</td>
<td>$E_{1,2}$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$E_{1,3}$</td>
<td>$E_{1,2}$</td>
<td>$q_5$</td>
</tr>
</tbody>
</table>

In neither case is there a split so the algorithm stops. The examples below add a table notation that simplifies the computation.

**6.5 Example** We will find a machine that accepts the same language as this one but that has a minimimum number of states.

![Diagram showing a DFA with states $q_0, q_1, q_2, q_3, q_4, q_5$, transitions labeled with 'a' and 'b'.]

To do bookkeeping we will use triangular tables like the one below. They have an entry for every two-element set of state indices $\{i, j\}$ with $i, j \in \{0, \ldots, 5\}$ and $i \neq j$.

Start by marking the $i, j$ entries for state pairs that are 0-distinguishable, that is, where one of $q_i, q_j$ is accepting while the other is not.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The blanks entries mark state pairs that are 0-indistinguishable. So $\sim_0$, which as a binary relation is a set of pairs, is below. (The table doesn’t bother with the $q_i, q_i$ boxes, because a state is indistinguishable from itself. In the relation below these are written $\{q_0, q_0\}$. Of course in a set repeats collapse so we could rewrite, e.g., $\{q_0, q_0\}$ as $\{q_0\}$, but as written they are correct and also fit with Lemma 6.7.)

$\sim_0 = \{\{q_0, q_0\}, \{q_0, q_3\}, \{q_0, q_4\}, \{q_3, q_3\}, \{q_3, q_4\}, \{q_4, q_4\}, \\
\{q_1, q_1\}, \{q_1, q_2\}, \{q_1, q_5\}, \{q_2, q_2\}, \{q_2, q_5\}, \{q_5, q_5\}\}$

Form the $\sim_0$-equivalence classes by finding entries $i, j$ that are blank, and then $q_i$ is 0-equivalent to $q_j$. For instance, the equivalence class containing $q_0$ also contains $q_3$ and $q_4$ because the boxes 0, 2 and 0, 3 are blank. Similarly, the equivalence class
Section 6. Minimization

containing \( q_2 \) also contains \( q_1 \) and \( q_5 \). Here are the two \( \sim_0 \)-equivalence classes.

\[
\mathcal{E}_{0,0} = \{0, 3, 4\} \quad \mathcal{E}_{0,1} = \{1, 2, 5\}
\]

Iterate. Look at the table's blanks, the 0-indistinguishable pairs of states, to see if they can be 1-distinguished. On the left below is each 0-indistinguishable pair along with the pair to which the machine takes it with the listed input character. For instance, the input \( a \) takes the machine's state \( q_0 \) to \( q_1 \), and takes \( q_3 \) to \( q_5 \). If the resulting pair is 0-distinguishable then mark it.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>0</th>
<th>( \mathcal{E}_{1,0} = {0, 3, 4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 3}</td>
<td>{1, 5}</td>
<td>{2, 5}</td>
<td>1</td>
</tr>
<tr>
<td>{0, 4}</td>
<td>{1, 5}</td>
<td>{2, 5}</td>
<td>2</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>{3, 4}</td>
<td>{4, 3}</td>
<td>3</td>
</tr>
<tr>
<td>{1, 5}</td>
<td>{3, 5}</td>
<td>{4, 5}</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>{2, 5}</td>
<td>{4, 5}</td>
<td>{3, 5}</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>{3, 4}</td>
<td>{5, 5}</td>
<td>{5, 5}</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

So this round finds that \( q_1 \) is 1-distinguishable from \( q_5 \) and that \( q_2 \) is 1-distinguishable from \( q_5 \). The triangular table shows the additional marks, and the 1-equivalence classes also reflect that.

The next iteration subdivides the \( \sim_1 \)-equivalence classes, the \( \mathcal{E}_{1,i} \)'s, to compute the \( \sim_2 \)-equivalence classes.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>0</th>
<th>( \mathcal{E}_{2,0} = {0} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 3}</td>
<td>{1, 5}</td>
<td>( \checkmark ) {2, 5} ( \checkmark )</td>
<td>1</td>
</tr>
<tr>
<td>{0, 4}</td>
<td>{1, 5}</td>
<td>( \checkmark ) {2, 5} ( \checkmark )</td>
<td>2</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>{3, 4}</td>
<td>{4, 3}</td>
<td>3</td>
</tr>
<tr>
<td>{3, 4}</td>
<td>{5, 5}</td>
<td>{5, 5}</td>
<td>4</td>
</tr>
</tbody>
</table>

Running through one more iteration gives no more splitting so the process stops. Here is the minimized machine with \( s_0 \) for \( \mathcal{E}_{2,0} \), etc.

This machine's start state is the one containing the original machine's start, \( q_0 \). This machine's final states are the ones containing the final states of the original machine.

As to the arrows, for instance consider this machine's arrow labelled \( a \) from \( s_1 \) to \( s_3 \). On this machine \( s_1 \) represents the class \( \mathcal{E}_1 = \{q_3, q_4\} \). Look on the starting machine at the arrows leaving those states and labelled \( a \). Both arrows go to \( q_5 \), which is contained in the equivalence class \( s_3 = \mathcal{E}_3 \).

The algorithm has one more step, which was not needed in the prior example. If the machine has any states that are unreachable from \( q_0 \) then we omit those.
6.6 Example Minimize this machine.

First, \(q_5\) cannot be reached from the start state. Drop it. That leaves this initial triangular table.

\[
\begin{array}{c|ccc}
0 & 1 & 2 & 3 \\
\hline
0 & ✓ & ✓ & ✓ \\
1 & ✓ & ✓ & ✓ \\
2 & ✓ & ✓ & ✓ \\
3 & ✓ & ✓ & ✓ \\
\end{array}
\]

\(\mathcal{E}_{0,0} = \{0, 1, 2\}\)

\(\mathcal{E}_{0,1} = \{3, 4\}\)

Split the 0-equivalence classes to make the 1-equivalence classes.

\[
\begin{array}{c|ccc|c|c}
\emptyset & 0 & 1 & & \emptyset & \mathcal{E}_{1,0} = \{0\} \\
\{0, 1\} & \{1, 2\} & \{2, 3\} & ✓ & \{1, 2\} & \mathcal{E}_{1,1} = \{1, 2\} \\
\{0, 2\} & \{1, 2\} & \{2, 4\} & ✓ & \{3, 4\} & \mathcal{E}_{1,2} = \{3, 4\} \\
\{1, 2\} & \{2, 2\} & \{3, 4\} & ✓ & \{3, 4\} & \\
\{3, 4\} & \{3, 4\} & \{3, 4\} & & \end{array}
\]

No additional splitting happens on the next iteration so the process stops. The minimized machine has three states.

This result verifies that the algorithm computes what we want, the \(\sim\) equivalence classes.

6.7 Lemma Let \(M\) be a Finite State machine over the alphabet \(\Sigma\). For this sequence of binary relations among the states

\[
\mathcal{R}_0 = \left\{ (q, \hat{q}) \in Q_M^2 \mid \text{both are accepting or neither is accepting} \right\}
\]

\[
\vdots
\]

\[
\mathcal{R}_{k+1} = \left\{ (q, \hat{q}) \in \mathcal{R}_k \mid \{\Delta(q, x), \Delta(\hat{q}, x)\} \in \mathcal{R}_k \text{ for all } x \in \Sigma \right\}
\]

these properties hold: (1) the binary relation \(\mathcal{R}_k\) equals the binary relation \(\sim_k\) (2) \(\mathcal{R}_0 \supseteq \mathcal{R}_1 \supseteq \cdots\) (3) there is an \(n \in \{0, \ldots, |Q_M|\}\) such that \(\mathcal{R}_{n+1} = \mathcal{R}_n\), and furthermore, \(\mathcal{R}_k = \mathcal{R}_n\) for all \(k \geq n\) (4) if \(\mathcal{R}_{n+1} = \mathcal{R}_n\) then \(\mathcal{R}_n\) is the \(\sim\) relation.

Proof Half of property (1) is to show that if two states are \(\mathcal{R}_k\)-distinguishable then they are \(\sim_k\)-distinguishable. We do induction on \(k\). For the \(k = 0\) base step,
Definition 6.1 says that states are 0 indistinguishable if and only if either both are accepting states or neither is, which is the definition of $R_0$. The inductive step supposes that $R_i$ equals $\sim_i$ for all $i \leq k$ and considers the $k + 1$ case. Let $q$ and $\hat{q}$ be $R_{k+1}$ distinguishable. Then there is a character $x \in \Sigma$ such that the pair $\{ \Delta(q, x), \Delta(\hat{q}, x) \}$ was marked at some previous iteration, $i$. The inductive hypothesis says there is a string $\tau$ of length at most $k$ that $\sim_k$ distinguishes $\Delta(q, x)$ from $\Delta(\hat{q}, x)$. Then the string $\sigma = x \sim \tau$ has length at most $k + 1$ and will $\sim_{k+1}$ distinguish $q$ from $\hat{q}$.

The other half of (1) is to show that if two states are $\sim_{k+1}$-distinguishable, then they are $R_{k+1}$-distinguishable. We will use induction on the length $k$ of the distinguishing string. The $k = 0$ base step is again that $\sim_0$ and $R_0$ are equal by definition. For the inductive step suppose that the length of $\sigma$ is $\ell + 1$ and suppose also the hypothesis that for all $j \leq \ell$ if two states are $\sim_j$-distinguishable then they are $R_j$-distinguishable. Let $q$ and $\hat{q}$ be $\sim_{k+1}$-distinguishable, via the string $\sigma$. Decompose $\sigma$ as $\sigma = x \sim \sigma_0$, for $x \in \Sigma$. Apply the inductive hypothesis, giving that the pair $\{ \Delta(q, x), \Delta(\hat{q}, x) \}$ is $R_k$-distinguishable, and so they are in different $R_k$ classes. That implies that the pair $q, \hat{q}$ is $R_{k+1}$-distinguishable, via the character $x$.

Property (2) holds because all of the elements of $R_{k+1}$ are elements of $R_k$, by definition.

For the first half of (3), as the examples illustrate, each iteration splits equivalence classes and that can only happen each states is alone in its class. The worst case starts with a single class and at each iteration pulls out one state. The “furthermore” of (3) holds because if none of the two-element sets of $R_{n+1}$ differ from those of $R_n$ then they must be the same at the next iteration, simply because it is the same operation.

Property (4) follows from the prior properties, and Definition 6.1. See Exercise 6.14.

IV.6 Exercises

6.8 What happens when you minimize a machine that is already minimal?
✓ 6.9 Perform the minimization algorithm using the triangular tables on the machine with redundant states at the start of this section.
✓ 6.10 Minimize these two.
6.11 What happens if you perform the minimization procedure in Example 6.6 without first omitting the unreachable state?

6.12 Definition 6.1 defines a binary relation \( \sim \) between states. Show that it is an equivalence relation.

6.13 Minimize.

Note that the algorithm takes, roughly, a number of steps that are equal to the number of states in the machine.

6.14 Verify the final sentence of Lemma 6.7 by showing that the \( \sim \) relation is equal to \( \sim_n \) for some \( n \) less than or equal to the number of states in the machine. Hint: to show that there must be a distinguishing string of that size you can use the Pumping lemma.

6.15 Verify Lemma 6.4.

(A) Verify that each \( \sim_n \) is an equivalence relation between states.

(B) Verify that \( \sim \) is an equivalence.

6.16 For each language \( L \) accepted by some Finite State machine, let \( \text{rank}(L) \) be the smallest number \( n \in \mathbb{N} \) such that \( L \) 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

A Finite State machine cannot accept the language of balanced parentheses. So it is not powerful enough to use, for instance, if you want to decide whether input strings are valid programs in a modern programming language. The natural data structure to handle nested parentheses is a stack. We will next see a machine type consisting of a Finite State machine attached to a pushdown stack.

A stack is like a list in that the data is lined up in a row. However, with a stack you don’t have random access to the elements. It is like the restaurant dish dispenser shown below on the left. When you take one dish off the top then a spring pushes the remaining dishes up, so you get access to the next one. The reverse happens when you pop a new dish on; the spring gets compressed so all the old dishes side down and the new dish becomes the only one that is accessible.
Similarly, you can push a piece of data onto the top of our stack device, or you can pop the top element. 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. It starts with three characters on the stack, g3, g0, and g2. We push g1 on the stack, and then g0. At this point, although g1 is on the stack we don’t have immediate access to it. To get at g1 we must first pop off g0, as in the last stack shown.

Once something is popped, it is gone. The machine could have a state whose intuitive meaning is, say, that we just popped g0, but because there are finitely many states that strategy has limits.

So, like a Turing machine tape, a stack is unbounded storage. But it has restrictions that the tape does not.

**Definition** To define these machines we will extend the definition of Finite State machines, starting with deterministic machines.

7.1 **Definition** A Pushdown machine has a finite set of states $Q = \{ q_0, \ldots, q_{n-1} \}$ including a start state $q_0$ and a subset $F \subseteq Q$ of accepting states, a nonempty input alphabet $\Sigma$ and a nonempty stack alphabet $\Gamma$, as well as a transition function $\Delta: Q \times (\Sigma \cup \{B, \epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$.

We assume that the stack alphabet $\Gamma$ contains the character that we use to mark the stack bottom, $\bot$. The rest of $\Gamma$ is $g0$, $g1$, etc. We also assume that the tape alphabet $\Sigma$ does not contain the blank, $B$, or the character $\epsilon$.

The transition function describes how these machines act. For the input $(q_i, s, g_j) \in Q \times (\Sigma \cup \{B, \epsilon\}) \times \Gamma$ there are two cases. When the character $s$ is

---

† Read that character aloud as “bottom.” ‡ The definition allows $\epsilon$ 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.
an element of $\Sigma \cup \{B\}$ then an instruction $\Delta(q_i, s, g_j) = (q_k, \gamma)$ 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 $\gamma = \langle g_{i_0}, \ldots, g_{i_m} \rangle$ 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 $\bot$. 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.†

### 7.2 Example

This grammar generates the language of balanced parentheses.

$$S \rightarrow [S] | SS | \varepsilon \quad \mathcal{L}_{BAL} = \{ \varepsilon, [], [()], [[]], [[][]], [[][][]], [[][][][]], \ldots \}$$

The Pumping lemma shows that no Finite State machine accepts this language. But it is accepted by a Pushdown machine. This machine has states $Q = \{q_0, q_1, q_2\}$ with accepting states $F = \{q_1\}$, and languages $\Sigma = \{[, ]\}$ and $\Gamma = \{0\}$, and $\{\bot\}$. The table gives its transition function $\Delta$, with the instructions numbered for ease of reference.

<table>
<thead>
<tr>
<th>Instr no</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0, [, \bot$</td>
<td>$q_0, 'g0, \bot'$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0, [, g0$</td>
<td>$q_0, 'g0g0'$</td>
</tr>
<tr>
<td>2</td>
<td>$q_0, ], g0$</td>
<td>$q_0, \varepsilon$</td>
</tr>
<tr>
<td>3</td>
<td>$q_0, ], \bot$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>4</td>
<td>$q_0, B, \bot$</td>
<td>$q_1, \varepsilon$</td>
</tr>
</tbody>
</table>

It keeps a running tally of the number of $[$’s minus the number of $]$’s, as the number of $g0$’s on the stack. This computation starts with the input $[[[][]]]$ and ends in an accepting state.

† 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.
### 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.

$$L_{MM} = \{ \sigma \in \{a, b, c\}^* \mid \sigma = \tau c \tau^R \text{ for some } \tau \in \{a, b\}^* \}$$

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 $q_3$.

<table>
<thead>
<tr>
<th>Instr no</th>
<th>Input</th>
<th>Output</th>
<th>Instr no</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0, a, \bot$</td>
<td>$q_0, 'g0' \bot$</td>
<td>9</td>
<td>$q_1, a, g0$</td>
<td>$q_1, \varepsilon$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0, b, \bot$</td>
<td>$q_0, 'g1' \bot$</td>
<td>10</td>
<td>$q_1, a, g1$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>2</td>
<td>$q_0, \varepsilon, \bot$</td>
<td>$q_3, \varepsilon$</td>
<td>11</td>
<td>$q_1, a, \bot$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>3</td>
<td>$q_0, a, g0$</td>
<td>$q_0, 'g0g0'$</td>
<td>12</td>
<td>$q_1, b, g0$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>4</td>
<td>$q_0, a, g1$</td>
<td>$q_0, 'g0g1'$</td>
<td>13</td>
<td>$q_1, b, g1$</td>
<td>$q_1, \varepsilon$</td>
</tr>
<tr>
<td>5</td>
<td>$q_0, b, g0$</td>
<td>$q_0, 'g1g0'$</td>
<td>14</td>
<td>$q_1, b, \bot$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>6</td>
<td>$q_0, b, g1$</td>
<td>$q_0, 'g1g1'$</td>
<td>15</td>
<td>$q_1, B, g0$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>7</td>
<td>$q_0, c, g0$</td>
<td>$q_1, 'g0'$</td>
<td>16</td>
<td>$q_1, B, g1$</td>
<td>$q_2, \varepsilon$</td>
</tr>
<tr>
<td>8</td>
<td>$q_0, c, g1$</td>
<td>$q_1, 'g1'$</td>
<td>17</td>
<td>$q_1, B, \bot$</td>
<td>$q_3, \varepsilon$</td>
</tr>
</tbody>
</table>
This computation has the machine accept bacab.

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ q_0 \end{array}] ⊥</td>
</tr>
<tr>
<td>1</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ g_1 \end{array}] ⊥</td>
</tr>
<tr>
<td>2</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ g_0 \ g_1 \end{array}] ⊥</td>
</tr>
<tr>
<td>3</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ g_0 \ g_1 \end{array}] ⊥</td>
</tr>
<tr>
<td>4</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ g_1 \end{array}] ⊥</td>
</tr>
<tr>
<td>5</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ g_1 \end{array}] ⊥</td>
</tr>
<tr>
<td>6</td>
<td>[\begin{array}{c} b \ a \ c \ a \ b \ g_1 \end{array}] ⊥</td>
</tr>
</tbody>
</table>

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 very fast and makes efficient use of the machine resources. But it is suited to situations where the code is written by a program, such as a word processor or \TeX, 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 \[B\] as 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 chooses one of the outputs.
This grammar generates the language of all palindromes over $\mathbb{B}^*$.

$$P \rightarrow \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1 \quad L_{\text{PAL}} = \{\epsilon, 0, 1, 00, 11, 000, 010, 101, 111, \ldots\}$$

This language is not accepted by any Finite State machine, but it is accepted 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 $\Gamma = \{g0, g1, \bot\}$.

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 $g1$.

<table>
<thead>
<tr>
<th>Instr no</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0, 0, \bot$</td>
<td>$q_0, 'g0' \bot'$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0, 1, \bot$</td>
<td>$q_0, 'g1' \bot'$</td>
</tr>
<tr>
<td>2</td>
<td>$q_0, \epsilon, \bot$</td>
<td>$q_2, \epsilon$</td>
</tr>
<tr>
<td>3</td>
<td>$q_0, 0, g0$</td>
<td>$q_0, 'g0g0'$</td>
</tr>
<tr>
<td>4</td>
<td>$q_0, 1, g0$</td>
<td>$q_0, 'g1g0'$</td>
</tr>
<tr>
<td>5</td>
<td>$q_0, 0, g1$</td>
<td>$q_0, 'g0g1'$</td>
</tr>
<tr>
<td>6</td>
<td>$q_0, 1, g1$</td>
<td>$q_0, 'g1g1'$</td>
</tr>
<tr>
<td>7</td>
<td>$q_0, 0, g0$</td>
<td>$q_1, 'g0g0'$</td>
</tr>
<tr>
<td>8</td>
<td>$q_0, 1, g1$</td>
<td>$q_1, 'g0g1'$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0$</td>
</tr>
<tr>
<td>2</td>
<td>$q_1$</td>
</tr>
<tr>
<td>3</td>
<td>$q_1$</td>
</tr>
<tr>
<td>4</td>
<td>$q_2$</td>
</tr>
</tbody>
</table>
Here is the machine accepting 01010 using instructions 0, 4, 12, 16, and 17.

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 0 1 0 ⊥</td>
</tr>
<tr>
<td>1</td>
<td>0 1 0 1 0 g0 ⊥</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0 1 0 g1 g0 ⊥</td>
</tr>
<tr>
<td>3</td>
<td>0 1 0 1 0 g1 g0 ⊥</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 1 0 g0 ⊥</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1 0 ⊥</td>
</tr>
</tbody>
</table>

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 g0. 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 g1 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 accept
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 proved 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 \( \{ \sigma \sigma \mid \sigma \in \mathbb{B}^* \} \). 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.\(^\dagger\)

The diagram below summarizes. The box encloses all languages of bitstrings,
all subsets of \( \mathbb{B}^* \). The nested sets enclose those languages accepted by some Finite
State machine, or some Pushdown machine, etc.

<table>
<thead>
<tr>
<th>Class</th>
<th>Machine type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Finite State, including nondeterministic</td>
</tr>
<tr>
<td>B</td>
<td>Pushdown</td>
</tr>
<tr>
<td>C</td>
<td>nondeterministic Pushdown</td>
</tr>
<tr>
<td>D</td>
<td>Turing</td>
</tr>
</tbody>
</table>

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
accepted by some nondeterministic Pushdown machine if and only if it has a
context free grammar.

\(^\dagger\) Another way to tell that the set of languages accepted by an nondeterministic Pushdown machine
is a strict subset of the set of languages accepted 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.
IV.7 Exercises

✓ 7.6 Produce a Pushdown Automata that does not halt.

✓ 7.7 Produce a Pushdown machine to accept each language over $\Sigma = \{a, b, c\}$.
   (A) $\{a^ncb^{2n} \mid n \in \mathbb{N}\}$
   (B) $\{a^ncb^{n-1} \mid n > 0\}$

7.8 Give a Pushdown machine that accepts $\{\theta \tau \theta \mid \tau \in \mathbb{B}^*\}$.

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 [\]'s. Why is it not needed?
   (B) Prove that this machine accepts the language of balanced parentheses defined by the grammar.

7.10 Give a Pushdown machine that accepts $\{a^{2n}b^n \mid n \in \mathbb{N}\}$.

✓ 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 $\{a^nbc^n \in \{a, b, c\}^* \mid n \in \mathbb{N}\}$, 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 $\{\sigma b \sigma R \mid \sigma \in \{a, b\}^*\}$.

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 accepted 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 accepted by a pushdown automata is accepted by some Turing machine.

7.21 There is a Pumping lemma for Context Free languages: if $L$ is Context Free then it has a pumping length $p \geq 1$ such that any $\sigma \in L$ with $|\sigma| \geq p$ decomposes into five parts $\sigma = \alpha \beta \gamma \delta \zeta$ subject to the conditions
   (i) $|\alpha \beta| \leq p$,
   (ii) $|\beta \delta| \geq 1$, and
   (iii) $\alpha \beta^n \gamma \delta^n \zeta \in L$ for all $n \in \mathbb{N}$.

   (A) Use it to show that $\{a^nbc^n \mid n \in \mathbb{N}\}$ 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 ⊢ and a transition step.

(C) Define when a machine accepts a string.

Extra

IV.A Regular expressions in the wild

Regular expressions are often used in practice. For instance, imagine that you need to search a web server log for the names of all the PDF's downloaded from a subdirectory. A user on a Unix-derived system might type this.

```bash
grep "/linearalgebra/.*˙pdf" /var/log/apache2/access.log
```

The `grep` utility looks through the file line by line, and if a line matches the pattern then `grep` prints that line. That pattern, starting with the subdirectory `/linearalgebra/`, is an extended regular expression.

That is, in practice we often need text operations, and regular expressions are an important tool. Modern programming languages such as Python and Scheme include capabilities for extended regular expressions, sometimes called regexes, that go beyond the small-scale theory examples we saw earlier. These extensions fall into two categories. The first is convenience constructs that make easier something that would otherwise be doable, but awkward. The second is that some of the extensions to regular expressions in modern programming languages go beyond mere abbreviations. More on this later.

First, the convenience extensions. Many of them are about sheer scale: our earlier 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. It may may even contain all of Unicode’s more than one hundred thousand characters. We need managable ways to describe larger sets of characters.

Consider matching a digit. The regular expression 

\((0|1|2|3|4|5|6|7|8|9)\)

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 extended regular expressions are metacharacters. Or, because the digit characters are contiguous in the character set,† we

† This is true in both ASCII and Unicode.
can shorten it further to [0-9]. Along the same lines, [A-Za-z] matches a single 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, [^0-9] matches a non-digit and [^A-Za-z] matches a character that is not an ASCII letter.

The most common lists have short abbreviations. Another abbreviation for the digits is \d. Use \D for the ASCII non-digits, \s for the whitespace characters (space, tab, newline, formfeed, and line return) and \S for ASCII characters that are non-whitespace. Cover the alphanumeric characters (upper and lower case ASCII letters, digits, and underscore) with \w and cover the ASCII non-alphanumeric characters with \W. And — the big kahuna — the dot ‘.’ is a metacharacter that matches any member of the alphabet at all.† We saw the dot in the grep example that began this discussion.

A.1 Example  Canadian postal codes have seven characters: the fourth is a space, the first, third, and sixth are letters, and the others are digits. The regular expression [a-zA-Z]\d[a-zA-Z]\d[a-zA-Z]\d describes them.

A.2 Example  Dates are often given in the ‘dd/mm/yy’ format. This matches: \d\d/\d\d/\d\d.

A.3 Example  In the twelve hour time format some typical times strings are ‘8:05 am’ or ‘10:15 pm’. You could use this (note the empty string at the start).

(\|0|1)\d:\d\s(\d|pm)

Recall that in the regular expression a(b|c)d the parentheses and the pipe are not there to be matched. They are metacharacters, part of the syntax of the regular expression. Once we expand the alphabet Σ to include all characters, we run into the problem that we are already using some of the additional characters as metacharacters.

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.

Quantifiers  In the theoretical cases we saw earlier, to match ‘at most one a’ we used ε|a. In practice we can write something like (|a), as we did above for the twelve hour times. But depicting the empty string by just putting nothing there can be confusing. Modern languages make question mark a metacharacter and allow you to write a? for ‘at most one a’.

†Programming languages in practice by default have the dot character match any character except newline. They have a way to make it also match newline if you want.
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. We can match them with \d{5}.

A.5 Example North American phone numbers match \d{3} \d{3}-\d{4}.

A.6 Example The regular expression (-|\+)?\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; it is escaped because + is a metacharacter.

A.7 Example The expression [a-fA-F0-9]+ matches a natural number representation in hexadecimal. Programmers often prefix such a representation with \x so the expression becomes (\x)?[a-fA-F0-9]+.

A.8 Example A C language identifier begins with an ASCII letter or underscore and then can have arbitrarily many more letters, digits, or underscores: [a-zA-Z_]\w*.

A.9 Example Match a user name of between three and twelve letters, digits, underscores, or periods with [\w\.]{3,12}. Use {.8,} to match a password that is at least eight characters long.

A.10 Example Match a valid username on Reddit: [\w-]{3,20}. The hyphen, because it comes last in the square brackets, matches itself. And no, Reddit does not allow a period in a username.

A.11 Example For email addresses, \S+@\S+ is a commonly used extended expression.†

A.12 Example Match the text inside a single set of parentheses with \([\^[\()]*\).

A.13 Example This matches a URL, a web address such as http://joshua.smcvt.edu/computing. This regex is more intricate than prior ones so it deserves some explanation. It is based on breaking URL’s into three parts: a scheme such as http followed by a colon and two forward slashes a host such as joshua.smcvt.edu, and a path such as /computing (the standard also allows a query string that follows a question mark but this regex does not handle those).

$$(https?|ftp)://([^\s/\?\.#]+\.?){1,4}(/[\s]*)?$$

Notice the https?, so the scheme can be http or https, as well as ftp. After

†This is naive in that there are elaborate rules for the syntax of email addresses (see below). But it is a reasonable sanity check.
a colon and two forward slashes comes the host part, consisting of some 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 a path. The specification allows paths to be case sensitive but the regex here has only lower case.

**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\*}$`.

The regex engines in modern languages let you specify that the match is case insensitive (although they differ in the syntax).

**A.15 Example** An HTML document tag for an image, such as `<img src="logo.jpg">`, uses either of the keys `src` or `img` to give the name of the file containing the image that will be served. 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 ‘s’).

**Beyond convenience** The regular expression engines that come with recent programming languages have capabilities beyond matching only those languages that accepted by Finite State machines.

**A.16 Example** The web document language HTML uses tags such as `<b>boldface text</b>` and `<i>italicized text</i>`. Matching any one 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, Racket’s 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. (As you might guess from 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 imore power than the theoretical regular expressions that we studied earlier.

**A.17 Example** This is the language of squares over $\Sigma = \{a, b\}$.

$$\mathcal{L} = \{ \sigma \in \Sigma^* \mid \sigma = \tau \tilde{\tau} \text{ for some } \tau \in \Sigma^* \}$$

Some members are aabaab, baaabaaa, and aa. The Pumping lemma shows that the language of squares is not regular; see Exercise A.35. Accept this language with the regex `(\.+)%`; note the back reference.
**Downsides** Regular expressions are powerful tools, and this goes double for enhanced regexes. As illustrated by the examples above, some of their uses are: to validate usernames, to search text files, and to filter results. But they can come with costs also.

For instance, the regular expression for twelve hour time from Example A.3

\[(ε|0|1):d|d|\s(am|pm)\] 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 \((ε|\s am|\s pm)\), which is a bit more complex but does solve the issue.

Another issue is that it also matches some strings that you don’t want, such as 13:00 am or 9:61 pm. We can solve this as with the prior paragraph, by listing the cases.

\[(01|02|\ldots|11|12):(01|02|\ldots|59|60)(\s am|\s pm)\]

This is like the prior fixup, in that it does indeed fix the issue but it does so at a cost of complexity, since it amounts to a list of the allowed substrings.

Another example is that not every string matching the Canadian postal expression in Example A.1 has a corresponding post office—for one thing, no valid codes begin with Z. And 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 little more lists of strings, (which would change frequently).

The canonical extreme example is the regex for valid email addresses. We show here just five lines out of its 81 but that’s enough to make the point about its complexity.

And, even if you do have an address that fits the standard, you don’t know if there is an email server listening at that address.

At this point regular expressions may be starting to seem a little 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

† Some substrings are elided so it fits in the margins, .
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.

IV.A Exercises

✓ A.18 Which of the strings matches the regex ab+c? (A) abc (B) ac (C) abbb (D) bbc

A.19 Which of the strings matches the regex [a-z]+[\.\? !]? (A) battle! (B) Hot (C) green (D) swamping. (E) jump up. (F) undulate? (G) is.?

✓ A.20 Give 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.21 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.22 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.23 Give an extended regular expression matching strings that contain an open square bracket and an open curly brace.

✓ A.24 Every lot of land in New York City is denoted by a string of digits called BBL, for Borough (one digit), Block (five digits), and Lot (four digits). Give a regex.

(A) They are sometimes written with parentheses around the area code. Extend the regex to cover this case.
(B) Sometimes phone numbers do not include the area code. Extend to cover this also.

A.26 Most operating systems come with file that has a list of words, which can be used for spell-checking, etc. For instance, on Linux it may be at /usr/share/dict/words but in any event you can find it by running locate
words | grep dict. Use that file to find how many words fit the criteria.
(A) contains the letter a (B) starts with A (C) contains a or A (D) contains X
(E) contains x or X (F) contains the string st (G) contains the string ing
(H) contains an a, and later a b (I) contains none of the usual vowels a, e, i, o or u
(J) contains all the usual vowels (K) contains all the usual vowels, in ascending order
✓ A.27 Give a regex to accept time in a 24 hour format. It should match times of the
form ‘hh:mm:ss.sss’ or ‘hh:mm:ss’ or ‘hh:mm’ or ‘hh’.
A.28 Give a regex describing a floating point number.
✓ A.29 Give a suitable 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.30 Postal codes in the United Kingdom have six possible formats. They are: (i) A11 1AA, (ii) A1 1AA, (iii) A1A 1AA, (iv) AA11 1AA, (v) AA1 1AA, and (vi) AA1A 1AA, where A stands for a capital ASCII letter and 1 stands for a digit.
   (A) Give a regex.
   (B) Shorten it.
✓ A.31 You are stuck on a crossword puzzle. You know that the first letter (of eight) is an g, the third is an n and the seventh is an i. You have access to a file that contains all English words, each on its own line. Give a suitable regex.
A.32 In the Downsides discussion of Example A.3, we change the ending to (ε | \s am| \s pm). Why not \s(ε | am| pm), which factors out the whitespace?
A.33 Give an extended regular expression that matches no string.
✓ A.34 The Roman numerals taught in 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 VIIII with IX. Give a regular expression for valid Roman numerals less than 5000.
A.35 Example A.17 says that the language of squares over \( \Sigma = \{ a, b \} \)
\[
\mathcal{L} = \{ \sigma \in \Sigma^* \mid \sigma = \tau \cdot \tau \text{ for some } \tau \in \Sigma^* \}
\]
is not regular. Verify that.
A.36 Consider \( \mathcal{L} = \{ \emptyset^n1^n \mid n > 0 \} \). (A) Show that it is not regular. (B) Find a regex.
A.37 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.38 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.

(A) \[^SPEAK\]+ HE|LL|O+ [PLEASE]+

(B) (A|B|C)\1 (AB|OE|SK) .*M?O.* (AN|FE|BE)

Extra IV.B The Myhill-Nerode Theorem

We defined regular languages in terms of Finite State machines. Here we will give a characterization that does not depend on that.

This Finite state machine accepts strings that end in \textit{ab}.

Consider other strings over \(\Sigma = \{a, b\}\), not just the accepted ones, and see where they bring the machine.

\[
\begin{array}{c|ccccccccccc}
\text{Input string } \sigma & \epsilon & a & b & aa & ab & ba & bb & aaa & aab & aba & abb \\
\hline
\text{Ending state } \hat{\Delta}(\sigma) & q_0 & q_1 & q_0 & q_1 & q_2 & q_1 & q_0 & q_1 & q_2 & q_1 & q_0 \\
\end{array}
\]

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 \(q_2\).
B.1 Definition  Let $M$ be a Finite State machine with alphabet $\Sigma$. Two strings $\sigma_0, \sigma_1 \in \Sigma^*$ are $M$-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 $M$-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 $M$ 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 $\sigma_2$.

So a machine gives rise to a partition. Does it go the other way?

B.3 Definition  Suppose that $L$ is a language over $\Sigma$. Two strings $\sigma, \hat{\sigma} \in \Sigma^*$ are $L$-related (or $L$-indistinguishable), denoted $\sigma \sim_L \hat{\sigma}$, when for every suffix $\tau \in \Sigma^*$ we have $\sigma \sim \tau \in L$ if and only if $\hat{\sigma} \sim \tau \in L$. Otherwise, the two strings are $L$-distinguishable.

Said another way, the two strings $\sigma$ and $\hat{\sigma}$ can be $L$-distinguished when there is a suffix $\tau$ that separates them: of the two $\sigma \sim \tau$ and $\hat{\sigma} \sim \tau$, one is an element of $L$ while the other is not.

B.4 Lemma  For any language $L$, the binary relation $\sim_L$ is an equivalence, and thus gives rise to a partition of all strings.

Proof  Reflexivity, that $\sigma \sim_L \sigma$, is trivial. So is symmetry, that $\sigma_0 \sim_L \sigma_1$ implies $\sigma_1 \sim_L \sigma_0$. For transitivity suppose $\sigma_0 \sim_L \sigma_1$ and $\sigma_1 \sim_L \sigma_2$. If $\sigma_0 \sim \tau \in L$ then by the first supposition $\sigma_1 \sim \tau \in L$, and the second supposition in turn gives $\sigma_2 \sim \tau \in L$. Similarly $\sigma_0 \sim \tau \notin L$ implies that $\sigma_2 \sim \tau \notin L$. Thus $\sigma_0 \sim_L \sigma_2$.

B.5 Example  Let $L$ be the set $\{ \sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 1's \}$. We can find the parts of the partition. If two strings $\sigma_0, \sigma_1$ both have an even number of 1’s then they are $L$-related. That’s because for any $\tau \in \mathbb{B}^*$, if $\tau$ has an even number of 1’s then $\sigma_0 \sim \tau \in L$ and $\sigma_0 \sim \tau \in L$, while if $\tau$ has an odd number of 1’s then the concatenations will not be members of $L$. Similarly, two strings both have an odd number of 1’s then they are $L$-related. So the relationship $\sim_L$ gives rise to this partition of $\mathbb{B}^*$.

\[
E_{L, 0} = \{ \epsilon, 0, 00, 11, 000, 011, 101, 110, \ldots \} \quad E_{L, 1} = \{ 1, 01, 10, 001, 010, \ldots \}
\]
B.6 Example Let $L$ be $\{ \sigma \in \{a, b\}^* \mid \sigma$ has the same number of a’s as b’s $\}$. Then two members of $L$, two strings $\sigma_0, \sigma_1 \in \Sigma^*$ with the same number of a’s as b’s, are $L$-related. This is because for any suffix $\tau$, the string $\sigma_0 \sim \tau$ is an element of $L$ if and only if $\sigma_1 \sim \tau$ is an element of $L$, 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 $L$-related because for any suffix $\tau$, the string $\sigma_0 \sim \tau$ is an element of $L$ if and only if $\sigma_1 \sim \tau$ is an element of $L$, namely if and only if $\tau$ has one fewer a than b.

Following this reasoning, $\sim_L$ partitions $\{a, b\}^*$ into the infinitely many parts $E_{L,i} = \{ \sigma \in \{a, b\}^* \mid$ the number of a’s minus the number of b’s equals $i \}$, where $i \in \mathbb{Z}$.

B.7 Example This machine $M$ accepts $L = \{ \sigma \in \{a, b\}^* \mid$ $\sigma$ has even length $\}$. 

We will compare the partitions induced by the two relations introduced above.

The $M$-related relation breaks $\{a, b\}^*$ into five parts, one for each state (since each state in $M$ is reachable).

$$E_{M,0} = \{ \varepsilon \}$$
$$E_{M,1} = \{ a, aaa, aab, aba, abb, aaaaa, aaaaab, \ldots \}$$
$$E_{M,2} = \{ b, baa, bab, bba, bbaa, bbaab, \ldots \}$$
$$E_{M,3} = \{ aa, ab, aaaa, aaab, abaa, abba, abba, aaaaa, \ldots \}$$
$$E_{M,4} = \{ ba, bb, baaa, baab, baba, babb, bbba, bbbba, bbaaaa, \ldots \}$$

The $L$-related relation breaks $\{a, b\}^*$ into two parts.

$$E_{L,0} = \{ \sigma \mid \sigma$ has even length $\}$$
$$E_{L,1} = \{ \sigma \mid \sigma$ has odd length $\}$$

Verify this by noting that if two strings are in $E_{L,0}$ then adding a suffix $\tau$ will result in a string that is a member of $L$ if and only if the length of $\tau$ is even, and the same reasoning holds for $E_{L,1}$ and odd-length $\tau$’s.

The sketch below shows the universe of strings $\{a, b\}^*$, partitioned in two ways. There are two $L$-related parts, the left and right halves. The five $M$-related parts are subsets of the $L$-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 accepts $\mathcal{L}$. 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 $\sigma_0 \downarrow \tau$ causes it to end in the same state as does the input $\sigma_1 \downarrow \tau$. In particular, $\sigma_0 \downarrow \tau$ takes $\mathcal{M}$ to a final state if and only if $\sigma_1 \downarrow \tau$ does. So the two strings are $\mathcal{L}$-related. 

**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, $\sigma_0 \sim \beta \sim_\mathcal{L} \sigma_1 \sim \beta$. (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 $\sigma_0 \downarrow \tau$, $\sigma_1 \downarrow \tau$ is in $\mathcal{L}$ while the other is not. Taking $\beta \downarrow \hat{\tau}$ for $\tau$ gives that for the two strings $\sigma_0 \downarrow \beta$ and $\sigma_1 \downarrow \beta$, 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.

**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 $\sigma \downarrow a$ 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 $\sigma \downarrow b$ 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 $\sigma \downarrow x$ 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 accepted 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 \(L\)-related is finite. We will show that \(L\) is regular by producing a Finite State machine that accepts \(L\).

The machine's states are the partition's elements, the \(E_{L,i}\)'s. That is, \(s_i = E_{L,i}\). The start state is the part containing the empty string \(\varepsilon\). A state is final if that part contains strings from the language \(L\) (Lemma B.9 (2) says that each part contains either no strings from \(L\) or consists entirely of strings from \(L\)).

The transition function is: for any state \(s_i = E_{L,i}\) and alphabet element \(x\), compute the next state \(\Delta(s_i, x)\) by starting with any string in that part \(\sigma \in E_{L,i}\), appending the character to get a new string \(\hat{\sigma} = \sigma \uparrow x\), and then finding the part containing that string, the \(E_{L,j}\) such that \(\hat{\sigma} \in E_{L,j}\). 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 \(s_i = E_{L,i}\), 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 \(\sigma_0, \sigma_1 \in E_{L,i}\) and make a common extension by the one-element string \(\beta = \langle x \rangle\) so the results are in the same part \(\sigma_0 \sim_L \sigma_1\).

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 \(\Lambda([\sigma], x) = [\sigma \uparrow x]\). With that, a simple induction shows that the extended transition function in the new machine is \(\hat{\Lambda}(a) = [a]\).

Finally, we must verify that the language accepted by this machine is \(L\). 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 \(\sigma \in L\).

### IV.B Exercises

✓ B.12 Find the \(L\) equivalence classes for each regular set. The alphabet is \(\Sigma = \{a, b\}\).

(A) \(L_0 = \{a^n b \mid n \in \mathbb{N}\}\)

(b) \(L_1 = \{a^2 b^n \mid n \in \mathbb{N}\}\)

✓ B.13 For each language describe the \(L\) 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 string with the substring \(0110\)

(d) The set of strings without the substring \(0110\)

✓ B.14 The language of palindromes \(L = \{\sigma \in a^* b^* \mid \sigma^R = \sigma\}\) is not regular. Find infinitely many \(L\) equivalence classes.

✓ B.15 Use the Myhill-Nerode Theorem to show that the language \(L = \{a^n b^n \mid n \in \mathbb{N}\}\) is not regular.
Part Three

Computational Complexity
Chapter V  Computational Complexity

In the first part of this book we asked what can be done mechanically, at all. Once you know that a task can be done, then a natural next step is to look to do it efficiently, in a practical time or practical space, or by being parsimonious with some other computational resource.

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 an abstract feel.

When actual computers became widely available, people in many fields began working on algorithms to solve their problems. The Theory of Computing blossomed. Starting in the 1970’s and gaining strength during the 1980’s and 1990’s, the subject has incorporated many more questions that at least originate in applied fields. These fields need that the answers are feasible so concerns about the practicality of algorithms have grown in importance.

We start this part by reviewing how we measure algorithm practicality, the orders of growth of functions. As this is a review, we will not give a detailed development, just enough for our needs. We will then see a collection of the kinds of problems that drive the field today. With that, we will explore the themes that those problems bring out, and in particular consider the famous P and NP question.

Section V.1  Big \(O\)

An algorithm uses resources at a rate that depends on the size of its input. We have tools to describe rates.

We will review the basics of the growth rate of functions. We will not give a full description, just enough for our needs, and we will leave the proofs as exercises.

First, an anecdote. Here is a multiplication problem.

\[
\begin{align*}
678 \\
\times 42 \\
\hline
1356 \\
+27120 \\
\hline
28476
\end{align*}
\]

Image: Water 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 far more important than gravity. Similarly, changing an algorithm from taking \(n^2\) time to taking time that is \(n \log n\) can make some things easy that were previously not practical.
In grade school we learn the algorithm that for each digit of the multiplier 42, we multiply each digit of the multiplicand 678. That’s a nested loop. So the natural hypothesis is that multiplication of two \( n \)-bit numbers requires, at worst, a time of about \( n^2 \) ticks.

In 1960 A Kolmogorov organized a seminar at Moscow State University aimed at proving this. Before the next meeting one of his students, A Karatsuba, had found that it is false. Karatsuba produced a clever algorithm that used about \( n^\log_3(3) \approx n^{1.585} \) ticks. At the next meeting Kolmogorov explained the result and closed the seminar.

This story illustrates a theme in the Theory of Computation: every day researchers produce results saying, “here is a job and here is a way to do it in less time, or less space, etc.” We are good at finding clever ways to solve a problem and thereby reducing the upper bound rate at which we consume some computation resource, such as time. But we are not as good at proving nontrivial lower bounds, at saying “no algorithm can do this job in less than this number of ticks.” That’s one reason why in the development below we will mostly compare the growth rates of functions using a measure that is about ‘less than or equal to’.

**Motivation** Suppose that we have a job to do and two algorithms with which to do it. These have the property that when the input is size \( n \in \mathbb{N} \), in the worst case the first algorithm takes about \( \sqrt{n} \) many ticks while the second takes about \( 10 \cdot \log(n) \).\(^\dagger\)

For small \( n \) the first algorithm’s square root seems preferable to the second’s logarithm. For instance, when the input is 1000 the two output values are \( \sqrt{1000} \approx 31.62 \) and \( 10 \log(1000) \approx 99.66 \).

\[\begin{array}{c|c|c|c}
\text{Input} & \sqrt{n} & 10 \log(n) \\
1000 & 31.62 & 99.66 \\
\end{array}\]

However, in the long run the values of \( \sqrt{n} \) are much larger than the values of \( 10 \log(n) \). For instance, \( \sqrt{1000000} = 1000 \) while \( 10 \log(1000000) \approx 199.32 \).

\[\begin{array}{c|c|c|c}
\text{Input} & \sqrt{n} & 10 \log(n) \\
1000000 & 1000 & 199.32 \\
\end{array}\]

\(^\dagger\)Recall that \( \log(n) = \log_2(n) \) is defined by: start with \( n \) and then find the power of 2 that produces it. So, if \( n = 8 \) then \( \log(n) = 3 \). If \( n = 10 \) then \( \log(n) = 3.32 \).
Thus, giving the best definition for comparing the growth of two functions is more complicated than just seeing which always has smaller output values. Before that definition we will see two additional motivating 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, $f/g$.†

On the left your eye is struck that $n^2 + 5n + 6$ is ahead of $n^2$. But on the right the ratios demonstrate that your eye is misleading you. For large inputs the $5n$ and the $6$ are swamped by the highest order term, the $n^2$. These two functions track together — by far the biggest factor in the behavior of these two is that they are both quadratic — and for our purposes they are basically the same.

1.2 Example Next compare the quadratic $f(n) = n^2 + 5n + 6$ with the cubic $g(x) = n^3 + 2n + 3$. Although $f(0) = 6$ is larger than $g(0) = 3$ and $f(1) = 12$ is larger than $g(1) = 6$, soon enough because of $g$’s outputs are larger than $f$’s, because $g$ is a cubic. In contrast to the the prior two, they don’t track together. The values of $g$ speed ahead of those of $f$, so much so that as shown the values of $f$ don’t seem to rise above the $n$-axis — although $f(100) = 10,506$, it is no match for $g(100) = 2,000,304$.

†These graphs are discrete; they picture functions of natural numbers, not of real numbers. The earlier graphs of $\sqrt{n}$ and $10\log n$ are also discrete but there the many dots blend together to look like a continuous curve.
The ratio graph underlines this; the cubic polynomial accelerates away from the quadratic in the sense that the ratios grow without bound. In this sense, $g$ is a faster growing function than $f$. They both go to infinity but $g$ goes there faster.

1.3 Example Now compare the quadratics $f(x) = 2n^2 + 3n + 4$ and $g(n) = n^2 + 5n + 6$. While $f(0) = 4$ is less than $g(0) = 6$ and $f(1) = 9$ is less than $g(1) = 12$, eventually $f$’s values are above $g$’s. This we’ve seen before, that the function comparison definition needs to discount initial behavior.

What’s different in this comparison is that, $f$’s leading 2 keeps its graph ahead but is not enough to make it accelerate away. Instead, the ratio between the two is bounded. So the function comparison definition discounts constant multiples.

1.4 Example Let $b : \mathbb{N} \to \mathbb{N}$ 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 $n$.

<table>
<thead>
<tr>
<th>Input $n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>$b(n)$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$\lg(n)$</td>
<td>$-\infty$</td>
<td>0</td>
<td>1</td>
<td>1.58</td>
<td>2</td>
<td>2.32</td>
<td>2.58</td>
<td>2.81</td>
<td>3</td>
<td>3.17</td>
</tr>
</tbody>
</table>

This graph shows $b(n)$ for $n \in \{1, \ldots, 30\}$.

The formula is $b(n) = 1 + \lfloor \lg(n) \rfloor$, except that if $n = 0$ then $b(n) = 1$. The graph below compares $b(n)$ with $f(n) = \lg(n)$. (Note the change in both the horizontal and vertical scales, and that the domain of $f$ does not include 0 so $f$’s dots start with $n = 1$.)
The floor function does not matter much and the ‘+1’ does not matter much — over the long run, \(b(n)\) and \(\lg(n)\) give essentially the same values. We may summarize, without wandering too far from the strict truth, that the function giving the number of bits required to express a number is the base 2 logarithm.

We have mentioned that the function comparison definition given below disregards constant multiplicative factors. The formula for converting among logarithmic functions with different bases, \(\log_c(x) = (1/\log_b(c)) \cdot \log_b(x)\), shows that they differ only by a constant factor. So even the base does not matter; we can say that the number of bits is “a” logarithmic function.

**Definition** We are studying how algorithms use machine resources. The sizes of a machine’s resources, such as the size of inputs and outputs, are natural numbers. So a first guess may be that we should study the growth of natural number functions. However, above we have already found useful a function that takes reals to reals, \(\lg\), that takes reals to reals so we will consider real functions. (When we have a function that takes only natural number arguments then there is an easy way to extend it to take real number arguments. For instance we can extend the factorial function \(f(n) = n!\) with \(\hat{f}(x) = \lfloor x \rfloor!\) so the domain is the set of nonnegative reals.) Note that the function \(f(x) = \lg(x)\) gives negative values for some arguments, and is not even defined for arguments less than or equal to 0, so we will extend beyond those real functions with positive values.

1.5 **Definition** A function \(f\) that inputs real number arguments and outputs real number values is a **complexity function** if it satisfies (1) \(f\) has an **unbounded domain** in that there is a number \(N\) such that \(x \geq N\) implies that \(f(x)\) is defined, and (2) \(f\) is **eventually nonnegative** in that there exists a number \(M \geq 0\) so that \(x \geq M\) implies that \(f(x) \geq 0\).

1.6 **Definition** Let \(g\) be a complexity function with domain \(D\). Then \(\mathcal{O}(g)\) is the collection of complexity functions \(f\) satisfying: there are numbers \(N, C \in D\) so that whenever \(x \geq R\) then \(C \cdot g(x) \geq f(x)\) (and \(f(x)\) is defined). We say that \(f \in \mathcal{O}(g)\), or that \(f\) is \(\mathcal{O}(g)\), or that \(f = \mathcal{O}(g)\).

1.7 **Remark** (1) The term ‘complexity function’ is not standard. We shall find it convenient for stating the results below. (2) We use the letter ‘\(\mathcal{O}\)’ because this is about the order of growth. Read \(\mathcal{O}(g)\) aloud as “big-O of \(g\).” (3) The ‘\(f = \mathcal{O}(g)\)’ notation is awkward but common.⁠† The notation ‘\(f \in \mathcal{O}(g)\)’ is better but not as common. (4) Some authors allow negative real outputs and write the inequality with absolute values, \(f(x) \leq C \cdot |g(x)|\). (5) We often say something like ‘\(n^2 + 5n + 6\) is \(\mathcal{O}(n^2)\)’ instead of ‘\(f\) is \(\mathcal{O}(g)\) where \(f(n) = n^2 + 5n + 6\) and \(g(n) = n^2\).’ (6) Sometimes you see ‘\(f\) is \(\mathcal{O}(g)\)’ stated as ‘\(f(n)\) is \(\mathcal{O}(g(n))\).’ Strictly speaking, this is wrong because \(f(n)\) and \(g(n)\) are numbers, not functions.

---

⁠† One way that it is awkward is that it does not follow the usual rules of equality; for instance you cannot rewrite it as ‘\(\mathcal{O}(g) = f\)’. Another is that \(n = \mathcal{O}(n^2)\) and \(n^2 = \mathcal{O}(n^2)\) does not imply that \(n = n^2\).
Think of ‘$f$ is $O(g)$’ as meaning that $f$’s growth rate is less than or equal to $g$’s rate. These illustrate; the sketch on the left suggests that $g$ is accelerating away from $f$, that $g$’s growth rate is greater than $f$’s. The sketch on the right has the two tracking together, so their rates are alike.

1.8 Example To apply the definition to two functions we must produce suitable $N$ and $C$ and verify that they work. Let $f(n) = n^2$ and $g(n) = n^3$. Then $f$ is $O(g)$, as witnessed by $N = 2$ and $C = 1$. The verification is that $n > 2$ implies that $g(n) = n^3 = n \cdot n^2$ is greater than $2 \cdot n^2$, which is greater than $n^2 = 1 \cdot f(n)$.

If $f(n) = 5n^2$ and $g(n) = n^4$ then to show $f$ is $O(g)$ take $N = 2$ and $C = 2$. The verification is that $n > 2$ implies that $C \cdot n^4 = 2 \cdot n^2 \cdot n^2 \geq 8n^2 > 5 \cdot n^2$.

Don’t confuse a function having values that are strictly smaller than another with its having a growth rate that is strictly smaller.

1.9 Example Let $f(n) = n^2 + 1$ and $g(n) = n^2$, so that $g$’s values are smaller than $f$’s. But $f$ is $O(g)$. To verify, take $N = 2$ and $C = 2$. Then for $n \geq N$ write $n = j + 2$, giving $C \cdot g(n) = 2n^2 = 2(j + 2)^2 = 2j^2 + 8j + 8$, which is greater than $f(n) = (j + 2)^2 + 1 = j^2 + 4j + 5$. Similarly, $h(n) = 2n^2$ is $O(n^2)$ despite that $h$’s values are larger for $n > 0$.

1.10 Example The zero function $Z$ is $O(g)$ for every complexity function $g$. Verify with $N = 0$ and $C = 1$.

1.11 Example Some pairs of functions aren’t comparable. Let $g(n) = n^3$. The function

$$f(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n^4 & \text{if } n \text{ is odd} \end{cases}$$

is not $O(g)$ because when $n$ is odd, no $C$ will suffice. For instance, $C = 3$ does not suffice since there is no number $N$ such that $n \geq N$ guarantees $3n^4 \leq n^3$; on the contrary, if $n \geq 1$ then $3n^4 = 3n \cdot n^2 \geq 3 \cdot n^2 > n^2$. Similar reasoning works for any $C$. Likewise, $g$ is not $O(f)$ because of $f$’s behavior on even inputs.
1.12 **Lemma (Algebraic properties)** Let \( f, g, f_0, f_1, g_0 \) and \( g_1 \) be complexity functions. (1) If \( f \) is \( O(g) \) then for any positive constant \( a \in \mathbb{R}^+ \), the function \( af \) is \( O(g) \). (2) If \( f_0 \) is \( O(g_0) \) and \( f_1 \) is \( O(g_1) \) then the sum \( f_0 + f_1 \) is \( O(g) \), where \( g(n) = \max(g_0(n), g_1(n)) \). In particular, if there is a function \( g_0 \) where both \( f_0 \) and \( f_1 \) are \( O(g_0) \) then \( f_0 + f_1 \) is also \( O(g_0) \). (3) If \( f_0 \) is \( O(g_0) \) and \( f_1 \) is \( O(g_1) \) then the product \( f_0 f_1 \) is \( O(g_0 g_1) \).

1.13 **Example** The \( O(f) \)'s are sets. To illustrate, we will show that \( O(n \cdot \lg(n)) \) equals \( O(\lg(n!)) \) by mutual containment.

Recall that logarithmic functions turn multiplication into addition: \( \lg(a \cdot b) = \lg(a) + \lg(b) \). The \( O(\lg(n!)) \subseteq O(n \lg(n)) \) direction follows from this.

\[
\lg(n!) = \lg(n) + \lg(n-1) + \cdots + \lg(2) + \lg(1) < n \cdot \lg(n)
\]

The \( O(n \lg(n)) \subseteq O(\lg(n!)) \) direction is trickier.

\[
\lg(n!) = \lg(n) + \lg(n-1) + \cdots + \lg(1) > \lg(n) + \lg(n-1) + \cdots + \lg(\lfloor n/2 \rfloor) > (n/2) \cdot \lg(n/2) = (1/2) \cdot n \lg(n) - (1/2) \cdot n \lg(2)
\]

1.14 **Lemma** The big-O relation is reflexive, so \( f \) is \( O(f) \). It is also transitive, so if \( f \) is \( O(g) \) and \( g \) is \( O(h) \) then \( f \) is \( O(h) \).

1.15 **Definition** Two complexity functions have **equivalent growth rates** if \( f \) is \( O(g) \) and also \( g \) is \( O(f) \). We say \( f \) is \( \Theta(g) \) or, what is the same thing, \( g \) is \( \Theta(f) \).

1.16 **Lemma** The relation between functions of having equivalent growth rates is an equivalence.

1.17 **Figure**: The bean encloses the set of complexity functions. More quickly growing functions are higher up, so that \( f_0(n) = n^5 \) would be shown above \( f_1(n) = n^4 \). On the left is the cone \( O(g) \) for some \( g \). This contains functions with growth rate less than or equal to \( g \)'s. At the top is \( \Theta(g) \), containing the functions with growth rate equivalent to \( g \)'s. The bean on the right adds a cone \( O(f) \) where \( f \) in \( O(g) \).

Directly applying the definition of \( O \), as in Example 1.8, is sometimes the best way to go. But there is a result from Calculus that very often makes the job easier.
1.18 **Theorem** Let $f, g$ be complexity functions. Suppose that $\lim_{x \to \infty} f(x)/g(x)$ exists and equals $L \in \mathbb{R} \cup \{\infty\}$.

1. If $L = 0$ then $g$ grows faster than $f$, that is, $f$ is $O(g)$ but $g$ is not $O(f)$.

2. If $L = \infty$ then $f$ grows faster than $g$, that is, $g$ is $O(f)$ but $f$ is not $O(g)$.

3. If $0 < L < \infty$ then the two functions have equivalent growth rates, so that $f$ is $\Theta(g)$ and $g$ is $\Theta(f)$.

That theorem pairs well with L'Hôpital’s Rule.

1.19 **Theorem (L'Hôpital’s Rule)** Suppose that $f, g$ are complexity functions such that both $f(x) \to \infty$ and $g(x) \to \infty$ as $x \to \infty$, and such that both are differentiable for large enough inputs. If $\lim_{x \to \infty} f'(x)/g'(x)$ exists and equals $L \in \mathbb{R} \cup \{\pm \infty\}$ then $\lim_{x \to \infty} f(x)/g(x)$ also exists and also equals $L$.

1.20 **Example** L'Hôpital’s Rule works great for polynomial functions. Where $f(x) = x^2 + 5x + 6$ and $g(x) = x^3 + 2x + 3$.

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{x^2 + 5x + 6}{x^3 + 2x + 3} = \lim_{x \to \infty} \frac{2x + 5}{3x^2 + 2} = \lim_{x \to \infty} \frac{2}{6x} = 0$$

Then Theorem 1.18 says that $f$ is $O(g)$ but $g$ is not $O(f)$. That is, $f$'s growth rate is less than $g$'s.

Next consider $f(x) = x^2 + 5x + 6$ and $g(x) = x^2$.

$$\lim_{x \to \infty} \frac{2x^2 + 3x + 4}{x^2} = \lim_{x \to \infty} \frac{4x + 3}{2x} = \lim_{x \to \infty} \frac{4}{2} = 2$$

So the growth rates of the two are comparable, $f$ is $\Theta(g)$ and $g$ is $\Theta(f)$.

For $f(x) = 5x^4 + 15$ and $g(x) = x^2 - 3x$, this

$$\lim_{x \to \infty} \frac{5x^4 + 15}{x^2 - 3x} = \lim_{x \to \infty} \frac{20x^3}{2x - 3} = \lim_{x \to \infty} \frac{60x^2}{2} = \infty$$

shows that $f$'s growth rate is strictly greater than $g$'s rate, that is, $g$ is $O(f)$ but $f$ is not $O(g)$.

1.21 **Example** The logarithmic function $f(x) = \log_b(x)$ grows very slowly: $\log_b(x)$ is $O(x)$, and $\log_b(x)$ is $O(x^{0.1})$, and is $O(x^{0.01})$, and in fact $\log_b(x)$ is $O(x^d)$ for any positive $d$, no matter how small, by this equation.

$$\lim_{x \to \infty} \frac{\ln(x)}{x^d} = \lim_{x \to \infty} \frac{(1/x)}{dx^{d-1}} = \frac{1}{d} \cdot \lim_{x \to \infty} \frac{1}{x^d} = 0$$

Exercise 1.45 shows that $x^d$ is not $O(\log_b(x))$.

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 \(O(x)\).

\[
\lim_{x \to \infty} \frac{(\ln(x))^2}{x} = \lim_{x \to \infty} \frac{2 \cdot \ln(x) \cdot (1/x)}{1} = 2 \cdot \lim_{x \to \infty} \frac{\ln(x)}{x} = 2 \cdot \lim_{x \to \infty} \frac{1/x}{1} = 0
\]

Exercise 1.50 shows that for every power \(k\) the function \((\log_b(x))^k\) is \(O(x^d)\) 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 \(d > 1\).

\[
\lim_{x \to \infty} \frac{x \ln(x)}{x^d} = \lim_{x \to \infty} \frac{x \cdot (1/x) + 1 \cdot \ln(x)}{dx^{d-1}} = \lim_{x \to \infty} \frac{1 + \ln(x)}{dx^{d-1}} = \frac{1}{d(d-1)} \cdot \lim_{x \to \infty} \frac{(1/x)}{x^{d-2}} = \frac{1}{d(d-1)} \cdot \lim_{x \to \infty} \frac{1}{x^{d-1}} = 0
\]

**Example**  We can compare the polynomial \(f(x) = x^2\) to the exponential \(g(x) = 2^x\).

\[
\lim_{x \to \infty} \frac{2^x}{x^2} = \lim_{x \to \infty} \frac{2^x \cdot \ln(2)}{2x} = \lim_{x \to \infty} \frac{2^x \cdot (\ln(2))^2}{2} = \infty
\]

Thus, \(f\) is in \(O(g)\) but \(g\) is not in \(O(f)\).

**Lemma**  Logarithmic functions grow more slowly than polynomial functions: if \(f(n) = \log_b(n)\) for some base \(b\) and \(g(n) = a_m n^m + \cdots + a_0\) then \(f\) is \(O(g)\) but \(g\) is not \(O(f)\). Polynomial functions grow more slowly than exponential functions: where \(g\) is a polynomial function and \(h(n) = b^n\) for some base \(b > 1\) then \(f\) is \(O(h)\) but \(h\) is not \(O(f)\).

**Proof**  See Exercise 1.55.

Being able to use L’Hôpital’s Rule is one motivation for defining complexity functions to have a continuous domain (after the point \(N\)), although it is more natural to consider functions with discrete domains, because we are working with mechanical computation. The next result assures us that conclusions drawn in continuous contexts still apply in discrete ones. (As given, this lemma does not say that the functions are only defined for numbers larger than some value \(N\). But that version is harder to state and this one makes the same point.)

**Lemma**  Let \(f_0, f_1 : \mathbb{R} \to \mathbb{R}\), and consider the restrictions to a discrete domain \(g_0 = f_0|_N\) and \(g_1 = f_1|_N\). Where \(L, a \in \mathbb{R}, (1)\) if \(L = \lim_{x \to \infty} (af_0)(x)\) then \(L = \lim_{n \to \infty} (ag_0)(n)\), (2) if \(L = \lim_{x \to \infty} (f_0 + f_1)(x)\) then \(L = \lim_{n \to \infty} (g_0 + g_1)(n)\), (3) if \(L = \lim_{x \to \infty} (f_0 \cdot f_1)(x)\) then \(L = \lim_{n \to \infty} (g_0 \cdot g_1)(n)\), and (4) where the expressions are defined, if \(L = \lim_{x \to \infty} (f_0/f_1)(x)\) then \(L = \lim_{n \to \infty} (g_0/g_1)(n)\).

**Tractable and intractable**  Some orders of growth appear often in practice. This table lists them. They are in ascending order, meaning that \(O(f)\) is listed before \(O(g)\) if \(f\) is \(O(g)\) but \(g\) is not \(O(f)\).
There are about $10^{10}$ years old and a billion is relative change. It is also a huge absolute change. The universe is about 248 Chapter V. Computational Complexity

Another way to dramatize that is to consider an algorithm that takes a time
that grows at a rate of $2^b$, where the input is $b$-many bits. Adding one more bit, say bringing the input from $110100101$ to $1101001010$, doubles the time taken.

Cobham’s thesis is that the tractable problems — those that are at least conceivably solvable in practice — are those for which there is an algorithm whose resource consumption is polynomial.† 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 We use Big $\mathcal{O}$ to classify algorithms according to how much they use computing resources. The bottom line is that Big $\mathcal{O}$ is about scalability — something that is $\mathcal{O}(n^2)$ scales worse than something that is $\mathcal{O}(n \lg n)$, but better than something that is $\mathcal{O}(n^3)$. Is it any more than that?

Yes, that is the essence. Nonetheless there are some points about what $\mathcal{O}$ results mean that traditionally puzzle folks, and some elaboration may help.

First, Big $\mathcal{O}$ is for algorithms, not programs. Compare these snippets.

```python
for i in range(0,10):
    x=4
    find_clique(G,x,i)
```

The left one puts $x=4$ inside the loop, which costs the time for nine repeated assignments. But Big $\mathcal{O}$ disregards this constant time difference, and so is not suited analyzing fine coding details. Rather, it is for an abstraction level one up from implementations. Big $\mathcal{O}$’s level is that of algorithms.

That fits with our second point. Algorithms are tied to an underlying computing model.‡ They can have different costs on different machine models. (This definition speaks to the time used; a definition for the cost of other resources is similar.)

A model besides the Turing machine that is widely used in the Theory of Computing is the Random Access machine (RAM). Whereas a Turing machine cell store only a single symbol, in each register the RAM model stores an entire

---

† 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 cites the common experience that when someone announces an algorithm with a large exponent or large constant then over time the approach gets refined and those two shrink. And, if we were to show that a problem is not solvable in polynomial time then we have a strong argument that it is not feasible. In any event, polynomial time is better than exponential time. Here we accepting the thesis; it gives a technical meaning to the informal ‘acceptably fast’.

‡ More discussion of this is in Section 3 below.
integer. And whereas to get to a cell a Turing machine may spend a lot of steps moving the tape, the RAM model gets the contents in one step.

Close analysis shows that if we start with an algorithm intended for a RAM and translate it to run on a Turing machine then this may add as much as $n^3$ extra time, so that if the algorithm is $O(n^2)$ on the RAM then on the Turing machine it is $O(n^5)$. Thus, the best way to proceed is to settle on a model first, and only then use Big $O$ to help pick an algorithm.

Our third point concerns that the definitions of Big-$O$ and Big-$\Theta$ ignore constant factors; does that potentially make Big $O$ useless for judging the practicality of an algorithm? Suppose that our analysis of an algorithm shows that for inputs $n$ larger than $N$ it takes time given by the quadratic function $C \cdot n^2$, so we describe this algorithm as $O(n^2)$. Don’t we need to know $C$ and $N$? After all, an enormous coefficient such as $C = 2^{100}$ would make the algorithm completely impractical and similarly if $N$ is huge then we may need to analyze what happens before that point.

And there are other constants that we ignore. Imagine that analysis of an algorithm on a RAM model finds that the time it takes is $4n^2 \cdot T + 3n \cdot W + 2n \cdot R + \lg M$ where the constants are: $T$ is the time for a clock tick, $W$ and $R$ are the times for a disk write and read, and $M$ is the time for a memory access. Besides ignoring the $4$ in $4n^2$, in citing the algorithm we would typically also not put a value to the constants, instead for instance saying that the algorithm is “logarithmic in memory usage.” Doesn’t discounting constants considerably reduce Big $O$’s value choosing among an array of available algorithms?

One answer is that finding these constants is very hard. They involve the hardware’s exact details such as the memory addressing and paging, cache hits, and whether the CPU can do some operations in parallel. And, even if you do the work to find the constants, often you get very little benefit—we rarely find that they are so enormous that they change what algorithm we pick by much. As Table 1.26 illustrates, knowing that the algorithm grows logarithmically in memory usage is in the long run vastly more influential that the exact value of the constant.

A second, related, answer is that we have no reliable standard. Machines vary widely in their low-level details and these details can make a tremendous difference. This holds even if we forget about such models as Turing machines and just focus on the kinds of machines that we have on our desks and in our pockets. We could try to agree on a specification for a standard reference model — for instance, this is the approach taken in D Knuth’s *Art of Computer Programming* series — but we would have to periodically update the standard. So published results from some ago may be problematic because they refer to an old standard. Being reference independent is an advantage, and in a quickly changing field, almost a necessity.

---

† 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 $O(n^2)$ on a RAM model, but on an unboundedly parallel machine model will take constant time $O(1)$.‡ People do sometimes note the order of magnitude of these constants.
But being reference independent requires that we ignore constants and instead make relative statements. Not having to worry about the difference between the cost of a division and the cost of a memory write, as long as those costs lie between reasonable limits, means that constant factors are meaningless since changing units is inseparable from changing constant factors. Here is an analogy: absolute measurement of distance involve units such as miles or kilometers, but being able to make statements irrespective of the units requires making relative statements such as, “from here, New York City is twice as far as Boston.”

So, as this discussion began by saying, Big $\mathcal{O}$ notation expresses how an algorithm performs, on a particular machine model, as the input size grows. Thus we say that a function that takes $3n$ steps 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, and $\mathcal{O}(\lg n)$ means that doubling the input size will increase the number of steps by at most some constant. This notation ignores constants because that is inherent in being a unit free, relative measurement.

But certainly, as noted above, Big $\mathcal{O}$ is only a rough comparison; it cannot say with great precision when two algorithms which will be absolutely better when they are cast into code and run on a particular platform.†

That leads to the fourth point, a concern involving the definition of Big $\mathcal{O}$ and Big $\Theta$. Understanding how an algorithm performs as the input size grows requires that we define input size.

Consider a naive algorithm for factoring numbers that, given input $n$, tests each $k \in \{2, \ldots, n-1\}$ to see if it divides $n$. If $n$ is prime then it tests all those $k$'s, which is roughly $n$-many divisions. We can take the size of an input $n$ to be the number of bits needed to represent $n$ in binary, approximately $\lg n$. So for this algorithm the input is of size about $\lg n$ and the number of operations is about $n$. That makes the number of operations exponential in the input size — in this model this algorithm is $\mathcal{O}(2^b)$, where $b$ is the number of input bits.

However, most people’s experience from programming class is that this algorithm would be described as linear, as $\mathcal{O}(n)$, because for the number $n$ there are about $n$-many divisions. How to explain the difference?

This is another example that an algorithm depends on an underlying computing model. In many application situations the programmer makes the engineering judgement that for every use of their program the input will fit into a fixed-sized computer word, such as a 64 bit word. This means that we are selecting

† For that, use benchmarks.
a computation model, like the RAM model, where larger numbers take the same
time to read as smaller numbers. Then the prior paragraph applies and algorithm
associated with this model is linear.

So this is in part an application versus theory thing: in the common programming
practice situation where the bit size of the inputs is bounded, the runtime behavior
is a function of the number of inputs \( n \). In a theoretical conversation, accepting
input that is arbitrarily large, it is a function of the bit size of those inputs. 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
\textit{pseudopolynomial}.

One final point about Big \( \mathcal{O} \). When we are analyzing an algorithm we can look
at different behaviors. For instance, we can analyze the worst case behavior or the
average behavior. An example is that the widely used sorting algorithm quicksort
takes quadratic time \( \mathcal{O}(n^2) \) at worst but on average it is fast, \( \mathcal{O}(n \lg n) \). However,
worst-case analysis is by far the most common. So, algorithm analysis can happen
on a number of fronts.

\section*{V.1 Exercises}

1.28 True or false: if a function is \( \mathcal{O}(n^2) \) then it is \( \mathcal{O}(n^3) \).

✓ 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 order of growth hierarchy 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 Find which function on the order of growth hierarchy has the same rate of
growth as each of these.

(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 of functions 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) \( f(n) = 2n^2, g(n) = \sqrt{n} \)
Section 1. Big $\mathcal{O}$

1.47 There are orders of growth between polynomial and exponential. Let

\[ f(n) = n^{1.2} + \log n, \quad g(n) = n^{\sqrt{2}} + 2n \]

1.35 Which of these are $\mathcal{O}(n^2)$? (A) $\log n$ (B) $3 + 2n + n^2$ (C) $3 + 2n + n^3$

1.36 For each, state true or false. (A) $5n^2 + 2n$ is $\mathcal{O}(n^3)$ (B) $2 + 4n^3$ is $\mathcal{O}(\log n)$

1.37 For each find the smallest $k \in \mathbb{N}$ so that the given function is $\mathcal{O}(n^k)$.

(A) $n^3 + (n^4/10000000)$ (B) $(n+2)(n+3)(n^2-\log n)$ (C) $5n^3 + 25 + \text{ceiling} \log(n)$

1.38 To Table 1.26 add a column for $3^n$.

1.39 To Table 1.26 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) $\log n$ (B) $n$ (C) $n \log n$ (D) $\sqrt{n}$ (E) $n^2$

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 or equal to $g(n) = n^3$?

1.42 Use L'Hôpital's Rule to show that $(\ln(x))^2$ is $\mathcal{O}(x)$. Show also that $(\ln(x))^2$ is $\mathcal{O}(x^p)$ for any power $p \geq 1$, starting from Example 1.21.

1.43 Where does the restriction $g(n) \leq n^{O(1)}$ place the function $g$ in the growth hierarchy?

1.44 Fix a deterministic Finite State machine. What is the order of growth of its run time?

1.45 Do the converse half of Example 1.21: fix a logarithmic base $b > 0$ and show that for any exponent $d > 0$ the power function $x^d$ is not $\mathcal{O}(\log_b(x))$.

1.46 Let $f(n) = 2n$ and $g(n) = n^2$. Prove directly from Definition 1.6 that $f$ is $\mathcal{O}(g)$ but that $g$ is not $\mathcal{O}(f)$.

1.47 There are orders of growth between polynomial and exponential. Let $f(n) = n^{\log(n)}$.

(A) Verify that $g(m) = m^k$ is in $\mathcal{O}(f)$ for all powers $k \in \mathbb{N}$.

(B) Verify that $f$ is in $\mathcal{O}(2^m)$ by showing that $n^{\log(n)} = 2^{(\log(n))^2}$ (Hint: take the logarithm of both sides), and this function grows more slowly than $2^{mn}$ for any $m > 0$ no matter how small.

1.48 Prove that $2^n$ is $\mathcal{O}(n!)$. (Hint: because of the factorial, rather than use L'Hôpital's Rule, proceed directly from the definition.)
1.49  (A) Show that the function \( f(x) = 7 \) is \( O(1) \).
    (B) Show that the function \( f(x) = 7 + \sin(x) \) is \( O(1) \). So if a function is in \( O(1) \) that does not mean it is constant.
    (C) Show that the function \( f(x) = 7 + (1/x) \) is \( O(1) \).
    (D) Show that a natural number function \( f \) is \( O(1) \) if and only if it is bounded above by a constant, so that there exists \( L \in \mathbb{N} \) so that for all inputs \( n \in \mathbb{N} \), if \( f \) is defined on \( n \) then \( f(n) \leq L \).

1.50  Use L'Hôpital's Rule's to verify these, starting from Example 1.21.
    (A) \( (\log_b(x))^2 \) is \( O(x^d) \) for any \( d \in \mathbb{R}^+ \)
    (B) \( (\log_b(x))^3 \) is \( O(x^d) \) for any \( d \in \mathbb{R}^+ \)

1.51  Name three functions that are \( O(Z) \) where \( Z: \mathbb{N} \to \mathbb{N} \) is the zero function.

1.52  We will show that there are functions with order of growth strictly between the polynomial and exponential functions. Specifically, \( f(x) = x^{\lg x} \) is one.
    (A) Argue that for any fixed \( k \), \( x^k \in O(x^{\lg x}) \).
    (B) Show that \( x^{\lg x} = 2^{(\lg n)^2} \). Hint: take the logarithm of both sides.
    (C) Use L'Hôpital's rule to show that \( x^{\lg x} \) is in \( O(2^{cn}) \) for any \( c > 0 \).

1.53  Verify these clauses of Lemma 1.12. Let \( f_0 \) be \( O(g_0) \) and let \( f_1 \) be \( O(g_1) \). (A) If \( a \in \mathbb{R}^+ \) then \( af \) is also \( O(g) \). (B) The function \( f_0 + f_1 \) is \( O(g) \), where \( g \) is defined by \( g(n) = \max(g_0(n), g_1(n)) \). (C) The product \( f_0 f_1 \) is \( O(g_0 g_1) \). (The product of two functions \( h_0, h_1 \) is defined by \( h_0 h_1(n) = h_0(n) \cdot h_1(n) \).)

1.54  Assume that \( f: \mathbb{N} \to \mathbb{N} \) is increasing, so that \( n_1 \geq n_0 \) implies \( f(n_1) \geq f(n_0) \). Assume also that \( c: \mathbb{N} \to \mathbb{N} \) is a constant function. Show that \( c \) is \( O(f) \).

1.55  Prove Lemma 1.23.

✓ 1.56  What is the connection between computable functions and feasibly computable functions? (A) Show that there is a computable function whose output values grow at a rate that is \( O(1) \), one whose values grow at a rate that is \( O(n) \), one for \( O(n^2) \), etc. (B) The Halting problem function \( K \) is uncomputable. Give a \( O \) bound on the rate of growth of its output. (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 strictly larger, only that the output values are larger.)

1.57  Verify these clauses of Lemma 1.14.
    (A) The big-\( O \) relation is reflexive.
    (B) It is also transitive.

1.58  Verify Lemma 1.16. Hint: the prior exercise does some of the work.

1.59  Verify these clauses of Theorem 1.18. Assume that \( f, g \) are natural number functions and suppose that \( \lim_{n \to \infty} f(n)/g(n) \) exists and equals \( L \in \mathbb{R} \cup \{ \infty \} \).
    (A) If \( L = \infty \) then \( f \) grows faster than \( g \).
    (B) If \( L = 0 \) then \( g \) grows faster than \( f \).
    (C) If \( L \) is between 0 and \( \infty \) then the two functions have the same growth rate.
Section 2. A problem miscellany

1.60 Show that $O(2^n) \neq O(3^n)$.

Section V.2 A problem miscellany

Much of the present work in the Theory of Computation is driven by problems. Often these come from a field outside the subject and are taken up by computation researchers who try to classify problems by their computational difficulty. Here we will list some problems, to get a sense of the ones that appear in the subject and also to use for examples and exercises. These are all well known in the field.

Problems with stories We start with a few problems that come with stories. Besides being fun, these stories 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.

One of his ventures was a game, Around the World. As below, the vertices were holes labelled with the names of world cities. Players put pegs in the holes, looking for a circuit that visited 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.

As stated, this is a decision problem, to say ‘yes’ or ‘no’ whether the input graph has a circuit. A related decision problem is to input a pair and decide if it is a member of $\{ (G, C) \mid C$ is a circuit of $G \}$. Some of the other problems described in this section are stated as decision problems but some are other types. We will expand on problem types in the next section.
A special case of Hamiltonian Circuit is the **Knight's Tour**, 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, two in one direction and then one perpendicular to that direction.)

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** (Travelling Salesman) Given a set \( S = \{ c_0, \ldots, c_{k-1} \} \) whose elements are called cities, and an array of integer distances \( d(c_i, c_j) \in \mathbb{N}^+ \) for all \( c_i \neq c_j \) (subject to \( d(c_i, c_2) = d(c_j, c_i) \)), find the cheapest circuit that visits every city.

An instance starts with a map of the forty eight contiguous US states, including the state capitals along with 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, but 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 less than \( B \) long’. If we had an algorithm to quickly solve this decision problem then we could 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 shortest trip.

The next problem sounds much like Hamilton Circuit, in that it involves walking around a graph, exhaustively. But it turns out to act very differently.

Today the city of Kaliningrad is in a Russian exclave between Poland and Lithuania. But in 1727 it was Prussian 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’s walk cross each bridge once and only once, and

---

**Leonhard Euler**  
1707–1783
arrive back at the start? No one could think of a way but no one could think of a reason that there was no way. A local mayor wrote to Euler, who proved that no circuit is possible. This paper founded Graph Theory.

Euler’s summary sketch is in the middle and the graph is on the right.

2.4 **PROBLEM (Euler Circuit)** Given a graph, find a circuit that traverses each edge once and only once.

Next is a problem that sounds like the prior ones, hard. But all of us see it solved every day.

2.5 **PROBLEM (Shortest Path)** Given a weighted graph and two vertices, find the shortest path between them.

For instance, with this graph we could look for the cheapest path from A to F.

This is the problem that you have when you ask your phone for the shortest-time driving directions to a distant city. There is an algorithm that solves it quickly.†

The next problem was discovered in 1852 by a young mathematician, F Guthrie, who was coloring a map of the counties of England. He wanted to make adjacent counties different colors and he wanted to use as few colors as possible. His map required only four colors and he conjectured that for any map at all, four colors suffice.

Guthrie imposed the condition that the map’s regions must be contiguous, and he defined ‘adjacent’ to mean sharing a border that is an interval, not just a point; see Exercise 2.41. On the right below is a graph version of the problem. Counties are represented as vertices and edges connect them if they are adjacent. Because this graph arises from a map, it is planar—we can draw it in the plane so that its edges do not cross. So Four Color inputs a planar graph and tries separate the vertices into no more than four categories, the colors, such that adjacent vertices are in different categories.

†Dijkstra’s algorithm is at worst quadratic in the number of vertices.
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 mathematical friends, including Hamilton. It remained unsolved until 1976 when K Appel and W Haken announced that they reduced the proof to 1,936 cases, and gotten a computer to check those cases. This was the first major proof that was done on a computer and it was controversial, although now computer work is routine.

2.7 **Problem (Graph Colorability)** Given a graph and a number $k \in \mathbb{N}$, decide whether the graph is $k$-colorable, whether we can partition its vertices into $k$-many parts $\mathcal{V} = \mathcal{C}_0 \cup \cdots \cup \mathcal{C}_{k-1}$ such that no two same-part vertices are connected by an edge.

2.8 **Problem (Chromatic Number)** Given a graph, find the smallest number $k \in \mathbb{N}$ such that the graph is $k$-colorable.

We close with a problem that will serve as a benchmark to which we compare other problems. In 1847 by G Boole outlined the subject that we today call Boolean algebra in *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. It provided a systematic mathematical formalism for simple logical relations.

A variable is **Boolean** if the values it takes on are $T$ or $F$. A **Boolean function** inputs and outputs tuples of those. **Boolean formulas** connect variables using the binary **and** operator $\land$, the binary **or** operator $\lor$, and the unary **not** operator $\neg$. Here is a formula with three variables.

$$f(P, Q, R) = (P \lor Q) \land (P \lor \neg Q) \land (\neg P \lor Q) \land (\neg P \lor \neg Q \lor \neg R)$$

We will assume that the formulas are in **conjunctive normal form**, so they consist of clauses of $\lor$’s connected with $\land$’s. A Boolean formula is **satisfiable** if there is some combination of input $T$’s and $F$’s so that the formula evaluates to $T$. This **truth table** shows the input-output behavior of the function defined by that formula.

---

**Image:** Counties of England and the derived planar graph.
That $T$ in the final column witnesses that this formula is satisfiable.

### Problem (Satisfiability, SAT)

Given a propositional logic formula, decide if it is satisfiable.

### Problem (3-Satisfiability)

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 the number of rows in the truth table is $2^v$ where $v$ is the number of input variables. So solving this problem appears to take exponential time. Whether that appearance is right is a very important problem, as we will see in later sections.

### More problems

We will list some more problems and examples, albeit without stories.

### Problem (Vertex-to-Vertex Path)

Given a graph and two vertices, find if there is a path from the first vertex to the second.†

### Example

These are two Western-tradition constellations, Ursa Minor and Draco.

Here we can solve the Vertex-to-Vertex Path problem by eye. For any two vertices in Ursa Minor there is a path and for any two vertices in Draco there is a path. But if one vertex is in the one and the other is in the other then there is no path.

†There are lots of problems about paths, so calling this just the Path problem is confusing. Some authors call this $st$-Path, $st$-Connectivity, or STCON.
2.13 **Problem (Minimum Spanning Tree)** Given an edge-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.

In this graph all of the edges have weights.

The highlighted subgraph includes all of the vertices, that is, it **spans** the graph. A **tree** is a graph with no cycles. This subgraph’s edge weights total to a minimum from among all spanning subgraphs so it must not have any cycles, or else we could eliminate an edge from the cycle and thereby lower the edge weight total without dropping any vertices.

**Minimum Spanning Tree** looks like **Hamiltonian Circuit** in requiring that the subgraph contain all the vertices. However, for this problem we know algorithms that are quick, \( O(n \lg n) \).

2.14 **Problem (Vertex Cover)** Given a graph and a bound \( B \in \mathbb{N} \), decide if the graph has a **\( B \)-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 \( C \).

2.15 **Example** A museum wants to post guards to watch their paintings. They have eight halls, laid out as below and they plan to post guards at corners. What is the smallest number of guards that will suffice to watch all of the hallways?

![Diagram of museum layout](image)

Obviously, one guard will not do. A two-element list that covers 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 **\( B \)-clique**, a set of \( B \)-many vertices such that any two are connected.

If the graph nodes represent people and the edges connect friends then we want to know if there are \( B \)-many mutual friends. Any 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 you can broadcast a message 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 $(x, y, z)$.

<table>
<thead>
<tr>
<th>$z = a$</th>
<th>$y = b$</th>
<th>$y = c$</th>
<th>$z = d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = a$</td>
<td>$(a, b, a)$</td>
<td>$(a, c, a)$</td>
<td>$x = a$</td>
</tr>
<tr>
<td>$x = b$</td>
<td>$(b, b, a)$</td>
<td>$(b, c, a)$</td>
<td>$x = b$</td>
</tr>
</tbody>
</table>
Then the set $\hat{M} = \{\langle a, b, a \rangle, \langle b, c, d \rangle\}$ has the requisite $n = 2$ elements and those two agree in none of their three coordinates.

2.24 **Problem (Subset Sum)** Given a multiset of natural numbers $S = \{n_0, \ldots, n_{k-1}\}$ and a target $T \in \mathbb{N}$, decide if a subset of $S$ sums to the target.

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 $\{1, 2, 3\}$.

2.25 **Example** Decide if there is a subset of $\{911, 22, 821, 563, 405, 986, 165, 732\}$ that adds to $T = 1173$. (This works: $\{165, 986, 22\}$.)

2.26 **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?

<table>
<thead>
<tr>
<th>Item</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Value</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>30</td>
</tr>
</tbody>
</table>

We pack the most value while keeping to the weight limit by taking items (a) and (b). So we cannot meet the value target.

2.27 **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, $\sum_{a \in \hat{A}} s(a) = \sum_{a \notin \hat{A}} s(a)$.

2.28 **Example** The set $A = \{I, a, my, go, rivers, cat, hotel, comb\}$ has eight words. The size of a word is the number of letters. Then $\hat{A} = \{cat, river, I, a, go\}$ gives $\sum_{a \in \hat{A}} s(a) = \sum_{a \notin \hat{A}} s(a) = 12$.

2.29 **Problem (Crossword)** Given an $n \times n$ grid, and a set of $2n^2$-many strings, each of length $n$, decide if the words can be packed into the grid.

2.30 **Example** Can we pack the words AGE, AGO, BEG, CAB, CAD, and DOG into a $3 \times 3$ grid?
Section 2. A problem miscellany

2.32 Problem (15 Game) Given an $n \times n$ grid holding tiles numbered $1, \ldots, n - 1$, 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 may seem inextricably linked, but as we understand them today, they seem to have different behavior.

2.33 Problem (Divisor) Given a number $n \in \mathbb{N}$, find a nontrivial divisor.

When the numbers are sufficiently large, we know of no efficient algorithm to find divisors.† However, it must be said that at this time we also have no proof that no efficient algorithm exists.‡ 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.

2.34 Problem (Prime Factorization) Given a number $n \in \mathbb{N}$, produce its decomposition into a product of primes.

Factoring seems, as far as we know today, to be hard. What about if you only want to know whether a number is prime or composite, and don’t care about its factors?

2.35 Problem (Composite) Given a number $n \in \mathbb{N}$, determine if it has any nontrivial factors; that is, decide if there is a number $a$ that divides $n$ and such that $1 < a < n$.

For many years the consensus among experts was that Composite was probably quite hard.§ After all, for centuries many very smart people had worked on composites and primes, and none of them had produced a fast test. But in 2002 M Agrawal, N Kayal, and N Saxena proved that primality testing can be done in time polynomial in the number of digits of the number. At this point refinements of their technique run in $O(n^6)$. This is known as the AKS primality test.¶

On thing this test does is dramatically illustrate the lesson that although experts are expert and their opinions have value, nonetheless they can be wrong. People producing a result that

†No efficient algorithm is known on a non-quantum computer. ‡There is no proof despite centuries of ingenious attacks on the problems 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. §There are a number of probabilistic algorithms that are often used in practice that can test primality very quickly, with an extremely small chance of error. ¶At the time they did most of the work, Kayal and Saxena were undergraduates.
gainsays established orthodoxy has happened before and will no doubt happen again — one correct proof outweighs any number of opinions.

V.2  Exercises

✓ 2.36  Find a divisor of each number. (A) 31221  (B) 52 424  (C) 9600

2.37  Name the prime numbers less than one hundred.

✓ 2.38  Decide if each formula is satisfiable.
  (A) \((P \land Q) \lor (\neg Q \land R)\)
  (B) \((P \rightarrow Q) \land \neg((P \land Q) \lor \neg P)\)

✓ 2.39  A tetrahedral pyramid and a cube are two of the five Platonic solids. Each has a Hamiltonian circuit, as shown.

Hamilton used a third Platonic solid, the dodecahedron, for his game. Find a Hamiltonian circuit for the two other Platonic solids, the octahedron and the icosahedron. (To make them easier to use, we have expanded one face until we can squash the entire shape down into the plane without any edges crossing.)

2.40  Give a map that requires four colors.

2.41  (A) In Four Color we require that the countries be contiguous, that they not consist of separated regions. Give a 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 map that would require five colors if we drop that requirement.

✓ 2.42  Solve Example 2.25.

✓ 2.43  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).
2.44 A popular game extends Vertex-to-vertex Path by counting the separation. Below is a portion of the movie connection graph, where vertices are actors and they are connected if they have ever been together in a movie.†

Let someone’s Bacon number be 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?

✓ 2.45 This Knapsack instance has no solution when the weight bound is \( B = 73 \) and the value target is \( T = 140 \).

<table>
<thead>
<tr>
<th>Item</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>21</td>
<td>33</td>
<td>49</td>
<td>42</td>
<td>19</td>
</tr>
<tr>
<td>Value</td>
<td>50</td>
<td>48</td>
<td>34</td>
<td>44</td>
<td>40</td>
</tr>
</tbody>
</table>

Verify that by brute force, by checking every possible packing attempt. (Probably the fastest way would be to write a program.)

2.46 Find the shortest path in this graph

† Information from https://oracleofbacon.org/
2.47 Imagine a directed graph where the edges are prone to failure. For instance, it may be a road network in a region prone to flooding. For each edge we know the probability that the edge will fail. Show how to use a solution to the Shortest Path problem to find the path between two vertices least likely to fail.

2.48 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. (You may find that the easiest way is to check it with a program.)

2.49 The Course Scheduling problem starts with a list of students and the classes they wish to take and then finds how many time slots are needed to schedule the classes. Of course, it assumes that if there is a student taking two classes then those two will not 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 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.

2.50 Some authors define Satisfiability as: given a finite set of propositional logic statements, find if there is a single input tuple \( b_0, ..., b_{j-1} \) (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.

2.51 What shape is a 3-clique? A 2-clique?

2.52 How many edges does a \( k \)-clique have?

2.53 Find all 3-cliques in this graph.

2.54 Is there a 3-clique in this graph? A 4-clique? A 5-clique?
2.55 Recall that Vertex Cover inputs a graph $G = \langle V, E \rangle$ 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 $\hat{S} = V - S$ is an independent set.

2.56 Consider Three Dimensional Matching, Problem 2.22. Let $X = \{a, b, c\}$, $Y = \{b, c, d\}$, and $Z = \{a, d, e\}$.

(A) List all 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?

2.57 In Example 2.21 the broadcast takes four steps. Can it be done in fewer steps?

Section V.3 Problems, algorithms, and programs

Now, with a good number of examples in hand, we will reflect on a number of aspects of problems, and solutions. We will discuss this on an intuitive level only—
indeed, some of these things have no widely accepted precise definition.

A problem is a job, a task. We are most focused on problems that can be solved with a mechanism, although we continue to be interested to find that a problem we are studying cannot be solved mechanically at all.

Often a problem encompasses a family of tasks. Contrast the general problem of Shortest Path with the instance of finding the shortest path between Los Angeles and New York. We are more likely to talk about the first than the second, both because the second is a special case and because the first feels more natural.†

An algorithm describes at a high level an effective way to solve the problem.‡ An algorithm is not an implementation, although it should be described in a way that is detailed enough that implementing it is routine for an experienced professional.

One subtle point about algorithms is that while they are abstractions, they are nonetheless based on an underlying 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 a desktop computer, such as the RAM model.

An example of a very different computing model that an algorithm could target is distributed computation. For example, the SETI@home project is a scientific experiment, where volunteers install on their computer a program which downloads and analyzes blocks of data from a radio telescope. This is massively parallel computation.§

A program is an implementation of an algorithm, typically expressed in a formal computer language and designed to be executed on a specific computing platform.

To illustrate the differences between the problems, algorithms, and programs, consider the problem Prime Factorization. One algorithm is to use brute force, that is, to check every number less than the input. We could implement that with a program written in Scheme.

Statements and representations  Strictly speaking, a complete 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 might be strings representing decimal numbers or might be in unary, a bit string where the number $n$ is represented with $n$-many 1’s. One reason that this matters is that the form of the input and output can change the algorithm or its runtime behavior. Consider the problem of determining whether an input number is divisible by four: a number is divisible

†There are interesting problems with only one task, such as computing the digits of $\pi$. ‡There is no widely-accepted formal definition of ‘algorithm’. Whatever it is, it fits between ‘mathematical function’ and ‘computer program’. For example, ‘sort’ is a function that takes in a set of items and gives out the sorted sequence. This function could be implemented using different algorithms: merge sort, heap sort, etc. In turn, each algorithm could be implemented in many programs. So the best handle that we have is that an ‘algorithm’ is an equivalence class of programs (i.e., Turing machines), where two programs are equivalent if they do essentially the same thing. §It searches for extraterrestrial intelligence. It is Free and you can easily run it on your computer. Visit http://setiathome.berkeley.edu/.
by four if and only if in its binary representation the final two bits are 00, so this input form makes the job trivial.

To provide some uniformity of representation, we will adopt the point of view which we can call Lipton’s Thesis that everthing of interest can be represented, reasonably efficiently, by bitstrings.† That is, compared to the description of Shortest Path given as Problem 2.5, we will understand that a more completely specification is: “given a reasonable bit string representation of a weighted graph \( \tilde{G} \) and two vertices, \( \tilde{v}_0 \) and \( \tilde{v}_1 \), return a reasonable bit string representation of the shortest path between them.” This includes all of the mathematically-flavored problems described earlier. But it also includes things that seem less natural, such as that we can with bit strings faithfully represent Beethoven’s 9th Symphony or an exquisite Old Master.

![3.1 Figure: Basket of Fruit by Caravaggio (1571–1610)](image)

Bit strings are convenient, straightforward, and familiar. But the assertion that they always suffice to give a representation that is efficient enough is in some ways like Church’s Thesis: we cannot prove it but experience—particularly very recent experience—argues that it is so.

However, having observed the need for representations and that we will use bit strings, from now on we will usually not mention it. We will usually leave detailed specifications for the programmer, because they will typically not affect the outcome of our analysis much, as long as they are reasonable.

**Types of problems** There are patterns in which problems we see in the Theory of Computation. One problem type that we have already seen is a function problem.

---

† Here, ‘reasonable’ means that it is not so inefficient as to greatly change the big-O behavior.
This asks that the algorithm have a single output for each input. Two examples are **Prime Factorization** and the problem of finding the greatest common divisor of two natural numbers.

A second type is the **optimization problem**, which looks for the best solution according to some metric. **Shortest Path** is of this type. So is **Minimal Spanning Tree**.

Another type is a **search problem**. Here, while there may be many solutions in the search space, we stop when we have found any one. An example is the problem, “Given a weighted graph, and two vertices, and a bound $B \in \mathbb{N}$, find a path between the vertices that costs less than the bound.” Another example is that of finding a $B$-coloring for a graph; there may be many colorings but we just want one. Two more are the Knapsack problem and the Subset Sum problem.

A problem with a ‘Yes’ or ‘No’ answer is a **decision problem**. The first problem we saw, the **Entscheidungsproblem**, is one of these. Another is the problem of deciding whether a given natural number is prime, and another is **Clique**. We have also seen decision problems in conjunction with the Halting problem, such as the problem of determining, given an index $e$, whether $\phi_e$ will output a seven for any input.

Although a decision problem calls for producing a function of a kind, a Boolean function, they are common enough to be a separate category.

Often a decision problem is expressed as a **language decision problem** or **language recognition problem**, where we are given some language and asked for an algorithm to decide if the input is a member of that language. We did lots of these in the Automata chapter, such as producing a machine that decides if an input string is a member of $L = \{ \sigma \in \{a,b\}^* \mid \sigma \text{ contains at least two } b's\}$, or proving that no Finite State machine can determine membership in $\{ a^n b^n \mid n \in \mathbb{N} \}$.

One reason for our interest in problems of language membership arises from practice; for instance, a language compiler must recognize whether a given source file is a member of the language. Another is that Finite State machines can only do one thing, decide languages, and so to compare these with other machines we must do so via 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.

From this point on, by ‘problem’ we will usually mean a decision problem for a language. (We will be sloppy about the distinction between the decision problem for a language and the language itself; for instance, we often use the letter $L$ for a problem.) Often, that will mean recasting problems to put them into language decision terms.

### 3.2 Example

Recall **Satisfiability**. As stated, it is a yes-or-no decision problem. We can recast it as the problem of determining membership in this language $SAT = \{ F \mid F \text{ is a satisfiable propositional logic statement } \}$.

---

<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 the statement is true or false.
3.3 Example Similarly, we can recast the problem of determining whether a given number is prime to the problem of determining whether a given representation of \( n \) is a member of \( \{ r \in \mathbb{B}^* \mid r \text{ represents a prime number} \} \).

Those recastings are straightforward. For optimization problems there is a standard technique to reword the optimization as a sequence of language decision problems.

3.4 Example Recall that Chromatic Number inputs a graph and returns a minimal number \( B \in \mathbb{N} \) such that the graph is \( B \)-colorable. Recast it by: for each \( B \in \mathbb{N} \) define the language \( L_B = \{ G \mid G \text{ has a } B\text{-coloring} \} \). Then, given a graph, if we could solve the language decision problem for each \( B \) then we can find the minimal chromatic number by testing whether it has a \( B = 1 \)-coloring, then testing whether it has a \( B = 2 \)-coloring, etc., until we find the smallest such \( B \).

3.5 Example Travelling Salesman, as stated in the prior section, is an optimization problem. Recast it as a language decision problem in a way like the prior example: set a bound \( B \in \mathbb{N} \) and consider this language.

\[
\mathcal{T}S_B = \{ G \mid G \text{ is a weighted graph with a circuit } C \text{ where } |C| \leq B \}
\]

Now, given a weighted graph, if we could solve the language decision problem for each \( B \) then to find the least integer for which there is a circuit we can ask if that graph is a member of \( \mathcal{T}S_1 \), if it is a member of \( \mathcal{T}S_2 \), etc. When we find a \( B \) then that gives the length of the shortest circuit.

The recasting must capture the essential difficulty in the problem. Note that the above optimization problems will be solvable in polytime if and only if the associated parametrized problems are solvable in polytime—searching through \( B = 0, B = 1 \), etc, only adds a linear factor.

3.6 Remark Because we are isolating language decision problems, some authors adopt a modification of the definition of a Turing machine, to have it come with a subset of accepting states. Such a machine solves a problem if it halts on all inputs \( \sigma \in \mathbb{B}^* \), and when it halts it is in an accepting state exactly if \( \sigma \) is in the language.

V.3 Exercises

✓ 3.7 What is the difference — speaking informally, since some of these do not have formal definitions — between an algorithm and: (a) a heuristic, (b) pseudocode, (c) a Turing machine (d) a flowchart, and (e) a process?

3.8 Solve the decision problem for (a) the empty language, and (b) the language \( \mathbb{B} \).

3.9 Name something important that cannot be represented in binary.

✓ 3.10 Identify each as a function problem, a decision problem, a search problem, or an optimization problem. (A) The problem Connectedness, 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 Nearest Neighbor problem 
that inputs a weighted graph and a vertex returns the vertex nearest the given one 
(that does not equal the given one). (e) 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 \( a^k \equiv b \pmod{p} \). (f) The problem that inputs a bitstring and returns the 
number that it represents in decimal. (g) The problem that takes a string and 
decides if there is a person in the address book with that name.

3.11 Express each decision problem as a one of accepting members of a language. 
(A) Decide whether a number is a perfect square. 
(B) Decide whether any triple \( \langle x, y, z \rangle \in \mathbb{N}^3 \) is a Pythagorean triple, 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.

✓ 3.12 Recast each to a language decision problem. (A) Euler Circuit, Problem 2.4. 
(B) Shortest Path, Problem 2.5. (C) Graph Colorability, Problem 2.7.

✓ 3.13 Show how to use an algorithm that solves Shortest Path to solve Vertex-to-vertex Path. 
How to use it on graphs that are not weighted?

✓ 3.14 Express each optimization problem as a language decision problem. 
(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.

3.15 Give a function problem version of Hamiltonian Circuit. Also give an optimiza-
tion version.

3.16 Let \( \mathcal{L}_F = \{ \langle n, B \rangle \in \mathbb{N}^2 \mid \text{there is an } m \in \{1, \ldots, B\} \text{ that divides } n \} \) and con-
sider 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 of, given a number, returning a prime factor of that 
number.

✓ 3.17 Show that with an algorithm that solves Subset Sum, Problem 2.24, we can 
quickly solve the associated function problem of finding the subset.

3.18 Give an example where you can solve the decision problem instantaneously 
but you cannot do so for the search problem.

3.19 Convert each function problem to an equivalent decision problem.
Section 4. \( P \)

(A) The problem that inputs two natural numbers and returns their product.
(B) The problem of Nearest Neighbor, that inputs a weighted graph and a vertex and returns the vertex nearest the given one but not equal to it.

✓ 3.20 Sketch an algorithm for each language recognition problem.

(A) \( L = \{ \sigma \in \{0, \ldots, 9\}^* \mid \sigma \text{ is the decimal representation of a multiple of } 100 \} \)

(B) \( \{ \sigma \in \{0, 1\}^* \mid \sigma \text{ has more } 1 \text{'s than } 0 \text{'s} \} \)

(C) \( L = \{ \sigma \in B^* \mid \sigma^R = \sigma \} \)

✓ 3.21 Convert each optimization problem into a family of decision problems.

(A) Travelling Salesman (b) Shortest Path

3.22 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.

3.23 Show how to use an algorithm that solves Vertex-to-vertex Path to solve the problem Graph Connectedness, 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 a class is a collection of languages.

4.1 **Definition** A complexity class is a collection of languages that can be computed within some resource bound.

4.2 **Example** One complexity class is the collection of languages for which there is a Turing machine that runs in time \( \mathcal{O}(n^{10}) \). Restated, a language \( L \) is in that class if there is a Turing machine \( P \) that runs in time \( t: \mathbb{N} \to \mathbb{N} \) where \( t \in \mathcal{O}(n^{10}) \), such that the associated function \( \phi_P: \mathbb{N} \to \mathbb{B} \) has \( \phi_P(n) = 1 \) if \( n \in L \) and \( \phi_P(n) = 0 \) otherwise.

4.3 **Example** We have already studied the class \( \text{RE} \) of languages \( L \) that are computably enumerable, that is, for which there is a computable function \( \phi_e \) so that the set \( \{ k = f(i) \mid k = \phi_e(i) \text{ for some input } i \in \mathbb{N} \} \) equals the set \( L \). Here the resource bound is that there is no finite time bound.

Since a language is a set, we are studying the characteristic functions of sets, and grouping them together based on how hard they are to compute.\(^\dagger\) Recall that in complexity theory we do not distinguish between problems — that is, decision problems for a language — and the language itself. So we will often say that a complexity class is a collection of problems, or that problem is a member of a class.

\(^\dagger\)There are many ways to do this. It may involve Turing machines or other machine types, such as nondeterministic Turing machines or Turing machines with access to an oracle for numbers that are truly random. It may also involve grouping problems by how long they take, or by their use of computational resources other than time, such as space.
Definition  The complexity class that we introduce now is the most important one. It is the collection of language decision problems that we take to be tractable under Cobham’s Thesis.

4.4 Definition  A problem is a member of the class $P$ if there is an algorithm for it that runs in polynomial time on a deterministic Turing machine.

4.5 Figure: The bean encloses all decision problems, $\mathcal{L} \subseteq \mathbb{B}^*$. Shaded is the class $P$. The shading is in layers because $P$ contains problems such that there is a solution algorithm that is $O(1)$, problems for which there is an algorithm that is $O(n)$, ones with a $O(n^2)$ algorithm, etc.

One problem that is a member of $P$ is that of deciding the language of perfect squares $\mathcal{L}_0 = \{ n \in \mathbb{N} \mid \text{there is } k \in \mathbb{N} \text{ with } k^2 = n \}$ since given a natural number, no matter how large, we can quickly decide if it is a square by at worst squaring all smaller numbers. Here are some other languages whose decision problems are in $P$: (1) deciding the language of correct sums $\mathcal{L}_1 = \{ \langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c \}$, (2) the language of sorted strings $\mathcal{L}_2 = \{ \sigma \in a, \ldots, z^* \mid \sigma \text{ is in alphabetical order} \}$, (3) the language of primes $\mathcal{L}_3 = \{ k \mid k \text{ is decimal representation of a prime} \}$, (4) For a regular expression $R$, the language of strings satisfying that expression $\mathcal{L}_4 = \{ \sigma \in \{a, b\}^* \mid \sigma \text{ satisfies } R \}$, (5) the language of pairs of connected nodes $\mathcal{L}_5 = \{ \langle G, v, \hat{v} \rangle \mid \text{graph } G \text{ has a path } v\hat{v} \}$ (this is the Vertex-to-Vertex Path problem).

4.6 Remark  Again, note that the members of $P$ are problems, essentially languages, for which there exists a decision algorithm that is sufficiently fast. Thus, it is wrong to say that an algorithm is in $P$.

Effect of the model of computation  A problem is in $P$ if it has an algorithm that is polytime. But algorithms are based on an underlying computing model. Is membership in $P$ dependent on the model that we use for computation?

In particular, experience with the Turing machine model gives a person the sense that they involve a lot of tape moving. So a person may expect that they are slow. However, close analysis with a wide range of alternative computational models proposed over the years shows that while Turing machines algorithms are often slower than related algorithms for other natural models, it is only by a factor of between $n^2$ and $n^4$. That is, we have a problem for which there is a $O(n)$

\footnote{We can take a model to be ‘natural’ if it was not invented in order to be a counterexample to this.}
algorithm on another model then we might find that on a Turing machine model it may be $O(n^3)$, or $O(n^4)$, or $O(n^5)$. So it is still in $P$.

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 in polytime on a probabilistic Turing machine. This thesis does not enjoy anything like the support of the original Church’s Thesis. But under the extended thesis, a problem that falls in the class $P$ using Turing machines also falls in that class using other natural models.

4.7 Remark Breaking news! There is a challenge to the extended thesis. For some time we have known of several problems, including the familiar Prime Factorization, 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.

What’s more, quite recently engineers at Google Labs claimed to have achieved Quantum Supremacy, to have solved a problem using an algorithm that runs on the Quantum Computing model that is not solvable on a machine on a Turing machine or RAM model in less than centuries. See this YouTube video from Google and S Aaronson’s Quantum Supremacy FAQ.

But there are reservations. For one, the claim is the subject of controversy; see this posting from IBM Research and A Quantum Supremacy Skepticism FAQ from researcher G Kalai. For another, this is not general purpose computing; the problem solved is exotic. In addition, whether quantum computers will ever be practical physical devices used for everyday problems is not at this moment perfectly clear, although scientists and engineers are making great progress developing them. For the purposes of this book we put this event aside, as these devices are not now available, but we will be watching these events with great interest.

Naturalness We will give the class $P$ a lot of attention. There are several reasons why it is natural to study.

The first was described in the prior subsection. Over the past century researchers have devised many models of computation. All of them compute the same set of functions as Turing machines — otherwise Church’s Thesis would be invalidated — but they may do so at different speeds. However, all these models run within polytime of each other. So $P$ is invariant under the choice of computing model: if a problem is in $P$ based on any familiar model including Turing machines, RAM machines, and Racket programs under Linux, then it is in $P$ for all of them. The differences between models washes out in the general polynomial sloppiness. (Standard practice defines $P$ via Turing machines.)

Another reason that $P$ is a natural class of problems to study is that it has a number of appealing closure properties. Fix a total function $f: \mathbb{N} \rightarrow \mathbb{N}$ and consider the set $L_f = \{ \langle \bar{n}, f(\bar{n}) \rangle \in \mathbb{B}^* \mid n \in \mathbb{N} \}$, where the bars mean that we represent

---

1 One definition of 'reasonable' is “in principle physically realizable” (Bernstein and Vazirani 1997).
2 A Turing machine with a random oracle.
3 All of the non-quantum natural models, anyway.
the numbers in binary. This set essentially turns the problem of computing \( f \) into
the decision problem of accepting the language \( L_f \). With that, \( P \) is closed under
function addition, scalar multiplication by an integer, subtraction, multiplication,
and composition. It is also closed under concatenation, so if \( L \in P \) and \( \hat{L} \in P \) then
the concatenation language \( L \odot \hat{L} \) is also an element of \( P \), and the same holds for
Kleene star.

But the main reason we are interested in \( P \) is Cobham's Thesis, that a reasonable
formalization of the intuitive idea that a problem is tractable is that it is in \( P \). Insofar as
theory should be a guide for practice, this is a compelling reason.

### Exercises

✓ **4.8** True or False: every problem that involves determining membership in a finite
language is in \( P \).

✓ **4.9** This sentence is mistaken; describe what is wrong with it: “I've got a problem
whose algorithm is in \( P \).”

✓ **4.10** What is the difference between a complexity class and an order of growth?

✓ **4.11** Is the Halting problem in \( P \)?

4.12 Show that any Regular language is in \( P \).

4.13 For any regular language, it is accepted by some Finite State machine. A
Finite State machine is a kind of Turing machine. Each Finite State machine
runs in linear time, since \( t(\sigma) = n \) where \( |\sigma| = n \). Linear is polynomial, so each
Regular language can be accepted in polytime.

✓ **4.14** Prove that the problem of deciding if two natural numbers is relatively prime
is in \( P \).

✓ **4.15** Prove that each problem is in \( P \) by citing the runtime of a particular algorithm.
   (A) Deciding the language of correct sums, \( \{ \langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c \} \).
   (B) Deciding the language \( \{ \sigma \in a, \ldots, z^* \mid \sigma \text{ is in alphabetical order} \} \).
   (C) Deciding \( \{ \langle A, B, C \rangle \mid \text{the matrices are such that } AB = C \} \).
   (D) Deciding the language of primes, \( \{ 1^k \mid k \text{ is prime} \} \).
   (E) Reachable nodes: \( \{ \langle G, v_0, v_1 \rangle \mid \text{the graph } G \text{ has a path from } v_0 \text{ to } v_1 \} \).

4.16 Find which of these is currently known to be in \( P \) and which is not. *Hint:* you
may need to look up what is the fastest known algorithm. (A) **Shortest Path**
(b) **Knapsack** (c) **Euler Path** (d) **Hamiltonian Circuit**

4.17 The problem of **Graph Connectedness** is: given a graph, decide if there is a
path from any vertex to any other. Sketch a proof that this problem is in \( P \).

4.18 Prove that if \( P_1 \in P \) and if \( P_0 \leq_P P_1 \) then \( P_0 \in P \).

4.19 Sketch a proof that each problem is in \( P \).
   (A) The \( \tau^3 \) problem: given a bitstring \( \sigma \), decide if it has the form \( \sigma = \tau \odot \tau \odot \tau \).
   (B) The problem of deciding which Turing machines halt within ten steps.

✓ **4.20** Prove that \( P \) is closed under complement.
4.21 Show that $P$ is closed under concatenation and Kleene star.

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 Clique?
   (B) Sketch a proof that this problem is in $P$.

4.23 As in the section body, where $f : \mathbb{N} \to \mathbb{N}$ is a computable function consider the decision problem $L_f$ for the language $\{ \langle \bar{n}, f(n) \rangle \in \mathbb{B}^* \mid n \in \mathbb{N} \}$, where the bar means the numbers are represented in binary. In this way we can consider a function to be a member of $P$.
   (A) Prove that the class $P$ is closed under function addition
   (B) Prove that $P$ is closed under multiplication by a scalar $r \in \mathbb{R}$.
   (C) Prove it is closed under function subtraction.
   (D) Prove $P$ is closed under function multiplication.
   (E) Prove it is closed under function composition.

4.24 Prove that the class of languages $P$ is closed under each operation. (A) Reversal
   (B) Union (C) Concatenation (D) Kleene star

4.25 We will show that making polynomially many calls to a subroutine that is polynomial may take exponential time. Consider a routine that may call a subroutine $S$. This subroutine is perfectly reasonable: $S$ runs in time linear in the input length, and given input of length $n$ it returns output of length $2n$.
   (A) Show that if the routine calls $S$ once then it costs a time that is polynomial in the length of the input.
   (B) Show that if the routine calls $S$ twice, as $S(S(x))$, then it costs a time that is polynomial in the length of the the input.
   (C) Ditto for a call that is nested $k$ deep for fixed $k \in \mathbb{N}$.
   (D) In contrast, for variable $k$ show that the call

$$S(S(\cdots S(x) \cdots))$$

$k$ times

costs time exponential in the length of the input.

---

**Section V.5 NP**

Recall that a machine is nondeterministic if from a present configuration and input it may pass to a next configuration with zero, or one, or more, next states. Recall also that we have two mental models of how these devices operate. In the first, the machine is unboundedly parallel: its computation history is a tree and it computes all of the branches. This fits with our experience on everyday computers where, for instance, we may have lots of windows open at one time, running lots of processes. In the second model the machine guesses which next state to follow, or is told which
one by an oracle, and then deterministically checks that branch of the computation tree.

With the first model we say an input string is accepted if at least one of the branches on the tree is an accepting branch. With the second model we say the input is accepted if there is a sequence of guesses that the machine could make, or a sequence of hints that the demon could produce, that leads to an accepting state. With those definitions, the two models are logically equivalent in that they accept the same languages.

We have already considered nondeterminism for both Finite State machines and Pushdown machines. For Finite State machines, nondeterminism did not make any difference in the sense that for any job done by a nondeterministic Finite State machine there is a deterministic Finite State machine that can do the same job. But Pushdown machines are a different story: there are jobs that can be done by a nondeterministic Pushdown machine that cannot be done by any deterministic machine. One such job is accepting the language of palindromes.

5.1 Lemma

Deterministic and nondeterministic Turing machines solve the same problems. In particular, every problem that can be solved by a nondeterministic machine can be solved by a deterministic machine.

Proof Since a deterministic Turing machine is a special case of a nondeterministic one, one direction is obvious.

For the other, start with a nondeterministic machine and we will describe how a deterministic machine can simulate the same computation. The nondeterministic machine generates a computation tree. The deterministic machine can do a breadth-first traversal of that tree. If there is an accepting branch then the deterministic machine will eventually reach it and it also will accept the input, otherwise it will not accept the input.

That proof is, basically, time-slicing. With the machines that are on our desks and in our pockets, we simulate an unboundedly-parallel computer by having the CPU switch among processes, giving each enough time to make some progress but not so much time that other processes are starved. This is a form of the dovetailing that we have see earlier. To the user it appears that many things are happening at once but actually there is only one, or at least a limited number of, simultaneous physical processes.†

So nondeterminism doesn’t add to what can be computed in principle. But that doesn’t mean that these machines are worthless.

For one thing, we saw with nondeterministic Finite State machines that they can be a good fit for the problems we wanted to solve. There, we could easily go from problem statement to a nondeterministic machine and then get a deterministic machine from that. We can do a similar thing with Turing machines.

Similarly, one interesting thing about nondeterministic Turing machines is that they appear to be a good match for solving some problems that on a serial device

† Depending on how many CPU’s the device has.
are hard to solve. An example is the Travelling Salesman problem. Stated in guessing terms, the machine simply guesses, making a sequence of where-next guesses, to find the best circuit. Or, in supernatural terms, the machine is given a circuit by some oracular demon and then checks whether it is shorter than the given bound. So for some problems, nondeterminism simplifies going from the problem statement to a solution statement.

**Speed** But the real excitement is that a nondeterministic Turing machine, if we had one, might be much faster than a deterministic one.

5.2 **Example** Consider **Satisfiability**. Is this propositional logic formula satisfiable?

\[ f = (P \lor Q) \land (P \lor \neg Q) \land (\neg P \lor Q) \land (\neg P \lor \neg Q \lor \neg R) \land (Q \lor R) \]

The natural approach is to compute a truth table and see whether the final column has any T’s. Here the next to last row has the T so this formula is satisfiable.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>P \lor Q</th>
<th>P \lor \neg Q</th>
<th>\neg P \lor Q</th>
<th>\neg P \lor \neg Q \lor \neg R</th>
<th>Q \lor R</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

The number of rows in a truth table grows exponentially with the number of variables, specifically it is 2 raised to the number of variables. So that approach is very slow, super-polynomial.

On the other hand, this problem seems perfectly suited for unbounded parallelism. For each line of the table we could fork a child process. These children are done quickly, certainly in polytime. Then we just call down the bus, “Any successes?” and if any child is holding a ‘T” then we declare \( f \) satisfiable. (In terms of the mental model: the machine guesses a solution, or is given a solution by the demon, and only has to verify that it works, which is also fast.) In total then, a nondeterministic machine does this job in polytime while it at least appears that a serial machine takes exponential time.

5.3 **Definition** A nondeterministic machine decides the language \( \mathcal{L} \) in time \( t: \mathbb{N} \to \mathbb{N} \) if for any input \( \sigma \in \mathcal{L} \) the machine computes at least one accepting branch in time \( t(|\sigma|) \).

So while adding unbounded parallelism to Turing machines doesn’t allow them to compute any entirely new functions, a person could sensibly conjecture that it may allow them to compute some of functions faster.
Definition  We now name the class of problems associated with nondeterministic Turing machines.

5.4 Definition The complexity class \( \text{NP} \) is the set of languages for which there is a polynomial time nondeterministic Turing machine acceptor.

5.5 Lemma \( \text{P} \subseteq \text{NP} \)

Proof Any deterministic Turing machine is trivially also a nondeterministic one. So a problem that has a polynomial time algorithm on a deterministic machine trivially has a polytime algorithm on a nondeterministic machine.

5.6 Remark Very important: no one knows whether \( \text{P} \) is a strict subset, so that \( \text{P} \neq \text{NP} \), or whether \( \text{P} = \text{NP} \). That is, at this moment no one knows whether, because deterministic machines do not come with unbounded parallelism baked in and must simulate it with time-slicing, they are necessarily slower. We will take up this question again in the next section.

We next see an alternate way to show that a problem is in \( \text{NP} \).†

Consider the above instance of Satisfiability, Example 5.2. Each line of the truth table is easy; the challenge is that there are a lot of lines. Intuitively, we can conjecture that a nondeterministic machine can use unbounded parallelism to simultaneously cover all the lines. Switching to the other way that we conceptualize nondeterminism, if the demon whispers, “Psst! Look at \( \omega = \text{TTF} \).” then a deterministic machine could quickly verify that the formula is satisfiable.

5.7 Definition For a language, a deterministic Turing machine \( \mathcal{P} \) is a verifier if for every \( \sigma \in \mathbb{B}^* \), that string is member of the language if and only if there exists a witness or certificate string \( \omega \in \mathbb{B}^* \) such that \( \mathcal{P} \) accepts \( \langle \sigma, \omega \rangle \).

5.8 Lemma A language is a member of \( \text{NP} \) if and only if it has a polynomial time verifier. That is, \( \mathcal{L} \in \text{NP} \) if and only if there is a deterministic Turing machine \( \mathcal{P} \) so that for every \( \sigma \in \mathbb{B}^* \) there is a witness \( \omega \) where \( \sigma \in \mathcal{L} \) if and only if \( \mathcal{P} \) accepts \( \langle \sigma, \omega \rangle \), in time polynomial in \( \sigma \).

So to show that a language is in \( \text{NP} \) we write a verifier \( \mathcal{P} \), a program that inputs hints \( \omega \) and confirms that they work. Before the proof we will see some examples and discuss some aspects of the definition and lemma.

5.9 Example The Satisfiability problem is to determine membership in this language.

\[ \text{SAT} = \{ \sigma \mid \sigma \text{ represents a propositional logic formula } f \text{ that is satisfiable} \} \]

For the witnesses we can use strings pointing to a line of the truth table, such as \( \omega = \text{TTF} \). This family of witnesses works because we can write the verifier shown below, a Turing machine \( \mathcal{P} \) that inputs a representation of a propositional logic

†A common pattern in mathematical presentations is to have a a definition that is conceptually clear and to follow it with results that make it easier to compute whether the definition applies.
If the statement $\sigma$ is satisfiable then such an $\omega$ exists, and the machine can verify the truth-table line quickly, certainly in polytime. If $\sigma$ is not satisfiable then it doesn’t matter what $\omega$ the machine inputs, it will not be able to verify a $T$ for that line. So $\sigma \in SAT$ if and only if there exists an $\omega$ such that the verifier $P$ accepts $\langle \sigma, \omega \rangle$.

5.10 Remark Before more examples we pause for a couple of comments. The most striking thing about Definition 5.7 is that it says ‘there exists’ a witness $\omega$. It does not say where the witness comes from. A person with programming experience may well ask, “but how will we compute $\omega$?” The question is not how to find it. The question is instead: can we write a verifier program that takes $\omega$’s, say from a demon, and uses them to verify that the $\sigma$’s are in $\mathcal{L}$, in polytime?

A second point is that the definition does not require that there be a witness in the case that $\sigma \notin \mathcal{L}$. Instead, in this case there just is no witness $\omega$ for the verifier to accept.

Third, a comment on the running time of the verifier. Contrast the above example with the problem of chess. Imagine that a demon hands you some papers and tells you that they contain a perfect chess strategy. Verifying this strategy by having a computer step through the responses to each move and responses to those responses, etc., at least appears to be an exponential-time job. So at least at first glance, chess does not seem to be verifiable in polytime. Thus, the fact that the definition requires that any verifier runs in time polynomial in $|\sigma|$ makes that verification tractable.

In addition, because the verifier runs in time polynomial in $|\sigma|$, the witness $\omega$ must have length polynomial in $|\sigma|$. This is simply because if the witness were too long then the machine would not even be able to read it in the allotted time.

Finally, above we described $\sigma$ as a string ‘representing’ a propositional logic statement and $\omega$ as ‘representing’ a table line. That gets tedious. In practice authors usually omit it and from now on we’ll do the same. For instance, the next example says that $\sigma$ is a triple instead of saying that it represents a triple.

5.11 Example The Hamiltonian Path problem is like Hamiltonian Circuit except that it inputs two vertices instead of requiring that the starting vertex equal the ending
Chapter V. Computational Complexity

\[ \mathcal{L} = \{ \langle G, v, \hat{v} \rangle \mid \text{some path in } G \text{ between } v \text{ and } \hat{v} \text{ visits every vertex} \} \]

We will show that this problem is in \( \text{NP} \). To do that, we will describe a family of witness \( \omega \)'s, hints from the demon that will be inputs for a suitable Turing machine verifier \( \mathcal{P} \). We take the witness to be a path, \( \omega = \langle v, v_1, \ldots, \hat{v} \rangle \). Our verifier inputs \( \sigma = \langle G, v, \hat{v} \rangle \) and \( \omega \) and then it tries to confirm that the list of vertices visited by the path is all of the vertices in the graph.

We finish with the promised proof of Lemma 5.8

**Proof** Suppose first that the language \( \mathcal{L} \) is accepted by the nondeterministic Turing machine \( \mathcal{P} \) in polynomial time; we will construct a polynomial time verifier \( \mathcal{V} \). Let \( p : \mathbb{N} \to \mathbb{N} \) be the polynomial such that the time taken by \( \mathcal{P} \) on input \( \sigma \in \mathbb{B}^{*} \) is \( p(|\sigma|) \). For any \( \sigma \in \mathcal{L} \) the computation tree has an accepting branch. We can make a certificate \( \omega \) of the branchings. With the certificate, a deterministic verifier can retrace \( \mathcal{P} \)'s accepting branch. Because the branch’s length must be less than the length of the computation, \( p(|\sigma|) \), the certificate will have length less than \( p(|\sigma|) \). So the verifier \( \mathcal{V} \) can do the retracing in polynomial time.
For the other direction suppose that the language $\mathcal{L}$ is accepted by a verifier $\mathcal{V}$ that runs in time that is bounded by some polynomial $p$. We will describe a process in guessing terms. Given $\sigma \in \mathbb{B}^*$, produce a sequence of $|\sigma|$-many guesses of branches in the computation tree to make a witness string $\omega$. Use this witness along with $\sigma$ as inputs to $\mathcal{V}$. Because of the verifier step, there is a way for this guessing process to succeed if and only if $\sigma \in \mathcal{L}$. Because the verifier runs in polynomial time, the entire process of guessing and then verifying does also.

So one way to think of $\textbf{NP}$ is that it is the class of problems that have efficient verifiers: there is a polytime algorithm that, given a witness $\omega$, can use it to verify membership in the language.

In summary, we are interested in knowing for which problems are there relatively good algorithms and for which are there not. Cobham’s Thesis identifies the problems that have a good algorithm as the problems in $\textbf{P}$. In this section we defined the class of problems for which there is a good way to verify a solution, in contrast with the problems for which there is a good algorithm to generate a solution. In the next section we will consider whether these two classes differ.

V.5 Exercises

✓ 5.14 Your friend says, “$\textbf{NP}$ is the set of problems with deterministic verifiers.” That’s not quite right; gently correct them.

✓ 5.15 Decide if each formula is satisfiable.

(a) $(P \land Q) \lor (\neg Q \land R)$
(b) $(P \rightarrow Q) \land \neg((P \land Q) \lor \neg P)$

✓ 5.16 A friend of yours asks, “In Lemma 5.8, 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 Turing machine is easy: just ignore $\sigma$ and follow the bit.” They are confused. Straighten them out.

✓ 5.17 Sketch a nondeterministic algorithm for each. State it both in terms of a computation tree and in terms of guessing.

(a) Three Dimensional Matching: where $X, Y, Z$ are sets of integers having $n$ elements, given as input $M \subseteq X \times Y \times Z$ then decide if there is an $n$-element $\hat{M} \subseteq M$ so that no two triples agree on any coordinates.

(b) Partition: given a finite multiset $\mathcal{A}$ such that each of its elements has a size $s(a) \in \mathbb{N}^+$, decide if the set splits into $\hat{\mathcal{A}}, \mathcal{A} - \hat{\mathcal{A}}$ so the total of the sizes is the same, $\sum_{a \in \hat{\mathcal{A}}} s(a) = \sum_{a \notin \hat{\mathcal{A}}} s(a)$.

5.18 Sketch a nondeterministic algorithm to solve each problem. Give both a computation tree formulation and a guessing formulation.

(a) Prime Factorization: given a number $n \in \mathbb{N}$, produce the prime factorization.

(b) The problem of $k$-Coloring: given an undirected graph, find a $k$-coloring for its vertices.

(c) The Marriage problem: given equal numbers of men and women, each person
ranks the members of the other sex. We must then assign to each person a partner. An assignment is unstable if there are two people of opposite sex would rather have each other than their assigned partners, otherwise that assignment is stable.

(d) The Integer Linear Programming problem: maximize a linear objective function \( f(x_0, \ldots, x_n) = d_0x_0 + \cdots + d_nx_n \) subject to constraints \( a_{i,0}x_0 + \cdots + a_{i,n}x_n \leq b_i \) and \( x_j \in \mathbb{N} \).

✓ 5.19 A simple substitution cipher encrypts text by substituting one letter for another. Thus, we start by fixing a permutation of the letters, for example \( \langle \text{F}, \text{P}, \ldots \rangle \) and then the cipher is that any \( \text{A} \) is replaced by a \( \text{F} \), any \( \text{B} \) is replaced by a \( \text{P} \), etc. Sketch two algorithms for decoding a substitution cipher, one deterministic and one nondeterministic. Describe the second in both computation tree terms, and in guessing terms.

✓ 5.20 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 \( \text{NP} \).

5.21 Describe the problem of determining whether a number is odd as the problem of deciding membership in a language. Show it is in \( \text{NP} \).

✓ 5.22 Recall how we recast Travelling Salesman as a language decision problem by using the bound technique. Show that the result is in \( \text{NP} \) by sketching a verifier that works with a suitable witness.

✓ 5.23 The problem Double-SAT is: given a propositional logic statement, decide whether it has at least two different substitutions of Boolean values that make the statement true. Show that this problem is in \( \text{NP} \) by producing a suitable witness.

5.24 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. Show that this problem is in \( \text{NP} \).

✓ 5.25 Outline a nondeterministic algorithm that inputs a planar graph and outputs a four-coloring. Describe it both in terms of computation tree and in terms of a demon giving the machine a witness.

✓ 5.26 Check that Knapsack is in \( \text{NP} \) by checking that for every ‘yes’ there is a polynomial-length witness that can be verified by a deterministic machine in polytime.

5.27 Each of these problems is in \( \text{NP} \). For each, give a witness.

(a) The problem of finding is a given number is divisible by four.

(b) The problem of finding the product of two matrices.

✓ 5.28 Give a witness for each of these problems from \( \text{NP} \). (A) Hamiltonian Circuit (B) Composite

✓ 5.29 Recast this as a decision problem and then show it is in \( \text{NP} \): We are going to put numbers into boxes. If three numbers \( x, y, \) and \( z \) are in a box then it cannot be the case that any two of them add to the third. If you put all of the numbers \( 1, 2, \ldots, n \) into boxes, what is the smallest number of boxes that you will need?
5.30 For each problem, recast it as a language decision problem by describing the associated language, and then show it is in \( \text{NP} \) by finding a witness.

(a) The **Linear Divisibility** problem inputs \( a, b \in \mathbb{N} \) 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 an \( n(n + 1)/2 \)-member multiset. The reverse of this problem, starting with the distances and reconstructing the positions, is the **Turnpike** problem.

5.31 Is the Halting problem in \( \text{NP} \)?

✓ 5.32 Show that this problem is in \( \text{NP} \). A company has two delivery trucks. So each morning they are presented with a set of locations, and the distances between those. Some location is distinguished as the start and finish. They must decide if there are two cycles such that every location is on at least one of the two cycles and both cycles have length at most \( K \).

5.33 Sketch a nondeterministic algorithm that inputs a planar graph and a bound \( B \in \mathbb{N} \) and decides whether the graph is \( B \)-colorable. Describe it in terms of computation tree and also in terms of the machine guessing.

✓ 5.34 Two graphs \( G_0, G_1 \) are isomorphic if there is a one-to-one and onto function \( f : V_0 \rightarrow V_1 \) such that \( \{ v, \hat{v} \} \) is an edge of \( G_0 \) if and only if \( \{ f(v), f(\hat{v}) \} \) is an edge of \( G_1 \). Consider the problem of computing wheter two graphs are isomorphic.

(a) Define the appropriate language.

(b) Show that the problem of determining membership in that language is a member of the class \( \text{NP} \).

5.35 For each problem, sketch a deterministic and a nondeterministic algorithm for solving that problem. Phrase the nondeterministic algorithm in both computation tree terms, and in guessing terms.

(a) The **Hamiltonian Path** problem: given a graph and two vertices, decide if there is a path between these two that visits each vertex once and only once.

(b) **Euler Circuit**

5.36 For each problem, sketch a nondeterministic algorithm. State it both in terms of a computation tree and in terms of guessing.

(a) **Clique**

(b) **Vertex Cover**

5.37 Prove that the class of languages \( \text{NP} \) is closed under (A) union (B) intersection (C) reversal, (D) concatenation, (E) Kleene star. No one knows if \( \text{NP} \) is closed under complementation.

Section V.6 Problem reduction

When we studied incomputability we considered whether there could be a \texttt{halts_on_three_checker} routine that, given \( x \), decides whether Turing ma-
chine $x$ halts on input 3. We showed that with such a checker we could solve the Halting problem, and denoted that with $K \leq_T \text{halts\_on\_three\_checker}$. The intuition behind the notation is that solving the Halts-on-three problem is at least as hard as solving the Halting problem. Formally, we write $B \leq_T A$ if there is a computable function $f$ such that $x \in B$ if and only if $f(x) \in A$ and we then say that $B$ is Turing-reducible to $A$, because to solve $B$ it suffices to solve $A$. An instance is that the Halting problem is Turing-reducible to the Halts-on-three problem.

In general, we say that problem $B$ is reduces to problem $A$ if we can answer questions about $B$ by accessing answers to questions about $A$. A reduction is thus a way to translate from one to the other. The translation uses arbitrary computable functions for $\leq_T$, and in other contexts uses other types of functions.

In computational complexity we want to solve problems efficiently. In this context the appropriate idea of reduction, of some version of “$L_1 \leq L_0$” expressing that $L_0$ is harder than $L_0$, is that $L_1 \leq L_0$ if a method for solving $L_0$ efficiently gives us a method to solve $L_1$ efficiently.

For instance, if we have an efficient algorithm that solves Shortest Path then with it we can efficiently solve Vertex-to-vertex Path, simply by throwing away the path length information. So Vertex-to-vertex Path reduces to Shortest Path; we shall write Vertex-to-vertex Path $\leq_p$ Shortest Path.

6.1 Definition Let $L_0, L_1$ be languages, subsets of $\mathbb{B}^*$. Then $L_1$ is polynomial time reducible to $L_0$ (or polynomial time mapping reducible, or polynomial time many-one reducible, or Karp reducible), written $L_1 \leq_p L_0$, if there is a computable function $f : \mathbb{B}^* \rightarrow \mathbb{B}^*$ that runs in polynomial time such that $\sigma \in L_1$ if and only if $f(\sigma) \in L_0$.

6.2 Lemma If $L_1 \leq_p L_0$ and $L_0 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$.

Proof Suppose that $L_1 \leq_p L_0$ via the function $f$ and also suppose that there is an algorithm for determining membership in $L_0$. To determine membership in $L_1$ we can do this: given input $\sigma$, find $f(\sigma)$ and apply the $L_0$-algorithm to determine if $f(\sigma) \in L_0$. Where the $L_0$ algorithm runs in time that is $O(n^i)$, and where $f$ runs in polytime for a polynomial that has degree $j$, then determining $L_1$ membership in this way runs in time that is $O(n^{\max(i,j)})$.

6.3 Figure: The bean encloses all problems. The ones with fast algorithms are at the bottom. Problems are connected if the one underneath reduces to the one above.
6.4 Example We will sketch an argument that **Graph Colorability** reduces to **Satisfiability**, that **Graph Colorability** \( \leq_p \) **Satisfiability**. We’ll go through the \( k = 3 \) construction; other \( k \)'s go the same way.

Recall that a graph is \( k \)-colorable if we can partition the vertices into \( k \) many classes, called ‘colors’ because that’s how they are pictured, so that there is no edge between two same-colored vertices.

6.5 Animation: A 3-coloring of a graph.

And, a propositional logic statement \( S \) is satisfiable if there is an assignment of the variables in the statement that make the statement as a whole evaluate to \( T \).

Write the set of all graphs as \( \mathcal{L}_1 \) and write the set of propositional logic statements as \( \mathcal{L}_0 \). (As discussed earlier, these are actually sets of bit strings. One is the set of strings \( \gamma \in \mathcal{L}_1 \) representing graphs \( \mathcal{G} \) with reasonable efficiency. The other is the set of reasonably efficient string representations \( \sigma \in \mathcal{L}_0 \) of statements \( S \). For the rest of the discussion we will not mention representations.)

To show that \( \mathcal{L}_1 \leq_p \mathcal{L}_0 \) we will produce a translation function \( f \) that inputs a graph \( \mathcal{G} \) and a outputs a propositional logic statement \( S = f(\mathcal{G}) \) such that if the graph is \( 3 \)-colorable if and only if the statement is satisfiable. The function must run in polynomial time.

Let \( \mathcal{G} \) have vertices \( \{v_0, ..., v_{n-1}\} \). The statement will have \( 3n \)-many Boolean variables: \( a_0, ..., a_{n-1} \), and \( b_0, ..., b_{n-1} \), and \( c_0, ..., c_{n-1} \). The idea is that if the \( i \)-th vertex \( v_i \) gets the first color then the associated variable \( a_i \) evaluates to True, \( a_i = T \), while if it gets the second color then \( b_i = T \), and if it gets the third color then \( c_i = T \). Thus, for each vertex \( v_i \) create a clause saying that it gets at least one color.

\[(a_i \lor b_i \lor c_i)\]

In addition, for each edge \( \{v_i, v_j\} \) create three clauses that together ensure that the edge does not connect two same-color vertices.

\[(-a_i \lor -a_j) \quad (-b_i \lor -b_j) \quad (-c_i \lor -c_j)\]

Now the statement is the conjunction of those clauses; it joins them with \( \land \)'s. The graph has a 3-coloring if and only if that statement is satisfiable.

This illustrates. The graph has four vertices so the statement starts with four clauses, saying that for each vertex \( v_i \) at least one of the associated variables \( a_i, b_i, \) or \( c_i \) has the value \( T \). The graph has four edges, \( v_0v_1, v_0v_3, v_1v_2, \) and \( v_2v_3 \). The
statement continues with three clauses for each edge, together ensuring that the variables associated with the edge’s vertices do not both have the value $T$.

\[ S = (a_0 \lor b_0 \lor c_0) \land (a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land (a_3 \lor b_3 \lor c_3) \]
\[ \land (\neg a_0 \lor \neg a_1) \land (\neg b_0 \lor \neg b_1) \land (\neg c_0 \lor \neg c_1) \]
\[ \land (\neg a_0 \lor \neg a_3) \land (\neg b_0 \lor \neg b_3) \land (\neg c_0 \lor \neg c_3) \]
\[ \land (\neg a_1 \lor \neg a_2) \land (\neg b_1 \lor \neg b_2) \land (\neg c_1 \lor \neg c_2) \]
\[ \land (\neg a_1 \lor \neg a_3) \land (\neg b_1 \lor \neg b_3) \land (\neg c_1 \lor \neg c_3) \]

Thus, $S$ is satisfiable if and only if $G$ has a 3-coloring.

Completing the proof means checking that the translation function, which inputs a bit string representation of $G$ and outputs a bitstring representation of $S$, is polynomial. That’s clear, although the argument is messy so we’ll skip it.

The significance of the reduction is that we now know that if we could solve Satisfiability in polynomial time then we could solve $k$-Colorability in polynomial time. Given a $k$-coloring problem $G$, translate it to a SAT problem $f(G) = S$. The translation takes polytime and $L_1$ takes polytime, so in total this is a polytime scheme.

That example gives some sense of why SAT can be a convenient problem to reduce to, a benchmark problem, since describing the conditions in a problem in logical statement terms is natural. Another kind of natural reduction is when we reduce to, a benchmark problem, since describing the conditions in a problem in logical statement terms is natural. Another kind of natural reduction is when we can see that one problem as a special case of another, or at least closely related.

6.6 EXAMPLE We will show that Subset Sum $\leq_p$ Knapsack, so that a quick way to solve the Knapsack problem would yield a quick way to solve the Subset Sum problem.

Recall that Subset Sum starts with a multiset $S = \{b_0, \ldots, b_{k-1}\} \subset \mathbb{N}$ and a target $T \in \mathbb{N}^+$. It asks if there is a subset whose elements add to the target.

Knapsack starts with a multiset of objects $\hat{S} = \{s_0, \ldots, s_{n-1}\}$, along with a bound $W \in \mathbb{N}$ and a target $V \in \mathbb{N}$. Each $s_i$ has both a weight $w(s_i) \in \mathbb{N}^+$ and a value $v(s_i) \in \mathbb{N}^+$. The problem is to decide if there is a subset $K \subseteq S$ such that the sum of its element weights is less than or equal to $W$ while the sum of the values is greater than or equal to $V$.

The reduction function $f$ inputs instances of Subset Sum, pairs $(S, T)$, and outputs instances of Knapsack, five-tuples $(\hat{S}, v, w, V, W)$, and must run in polytime. For the reduction, convert to the special case of Knapsack where each element has the same weight as value, $w(s_i) = v(s_i)$, and where $W = V = T$.

As an illustration, suppose that we want to know if there is a subset of $S = \{18, 23, 31, 33, 72, 86, 94\}$ that adds to $T = 126$. Let the Knapsack instance $f(S) = \hat{S}$ have the same number of elements as $S$, so $\hat{S} = \{s_0, \ldots, s_6\}$, let $w(s_0) = v(s_0) = 18$, $w(s_1) = v(s_1) = 23$, etc., and set the weight and value targets $W, V$ to be $T = 126$, A Subset Sum instance has a solution $A \subseteq S$ if and only the Knapsack
instance has a solution \( K = f(A) \subseteq \hat{S} \). Showing that the function \( f \) to convert the bitstring representations of problem instances is polytime is easy.

Above we showed that \( k \) Coloring reduces to Satisfiability, so doing Satisfiability fast is at least as hard as doing \( k \) Coloring fast. There, we commented that Satisfiability is often convenient for reductions and in the next section we will give a theorem saying that it is at least as hard as every problem in \( \text{NP} \). But we close by giving a problem that is at least as hard as Satisfiability.

6.7 Example. Recall Clique: given a graph \( \mathcal{G} \) and a bound \( B \in \mathbb{N} \), decide if the graph has a \( B \)-clique, a size \( B \) set of vertices such that every vertex in the set is connected to every other vertex. We will suggest how to construct the polytime function \( f \) giving that Satisfiability \( \leq_p \) Clique.

The reduction translates each Satisfiability instance, a propositional logic statement \( S \), to a Clique instance \( \langle \mathcal{G}, B \rangle = f(S) \). Consider this example propositional logic statement, \( S = (x_0 \lor x_1) \land (\neg x_0 \lor \neg x_1) \land (x_0 \lor \neg x_1) \). The three components \( x_0 \lor x_1, \neg x_0 \lor \neg x_1, \) and \( x_0 \lor \neg x_1 \) are called ‘clauses’. In a clause, an atom is either a variable \( x_i \) or its negation \( \neg x_i \). Put a vertex in \( \mathcal{G} \) for each occurrence of an atom in a clause. The statement \( S \) has three clauses with two atoms each so the graph has six vertices, as shown below. Put an edge in \( \mathcal{G} \) if the associated atoms are in different clauses and are not negations (that is, not \( x_i \) and \( \neg x_i \)). This is the graph arising from \( S \).

![Graph](image)

Observe that \( S \) is satisfiable if and only if the graph has a 3-clique. Showing that the translation function \( f \) is polytime is routine.

**NP complete** Because \( \text{P} \subseteq \text{NP} \), the class \( \text{NP} \) contains lots of easy problems, ones with a fast algorithm. For instance, one member of \( \text{NP} \) is the problem of determining whether a number is odd. Of course, the interest in the class is that it also contains lots of problems that seem to be hard. Can we prove that those problems are indeed hard?

In 1971 this question was raised by S Cook. He noted the significance of polynomial time reducibility, so that an efficient solution to one problem results in an efficient solution for the other. And, he showed that among the problems in \( \text{NP} \), there are ones that are maximally hard.†

†This was also shown by L Levin who was behind the Iron Curtain so that knowledge of his work did not spread west for some time.
6.8 **Theorem (Cook-Levin Theorem)** Satisfiability has the property that $L \leq_p SAT$ for any $L \in \text{NP}$.

First, $SAT \in \text{NP}$ because a nondeterministic machine can guess which line of the truth table to verify. Said another way: given a Boolean formula, use as a certificate $\omega$ a sequence giving an assignment of truth values that satisfies the formula.

The rest of the proof is long so it is in an Extra section. But its basic idea is reachable. We are given a problem $L \in \text{NP}$ and we must show that $L \leq_p SAT$. To show that $L$ is reducible to $SAT$ we must translate membership questions for $L$ into a Boolean formulas so that the membership answer is 'yes' if and only if the formula evaluates to $T$. The only thing we know about the problem $L$ is that its input $\sigma$ is accepted by some nondeterministic machine $P$ in time given by a polynomial $p$. So the proof constructs, from $P$, $\sigma$, and $p$, a Boolean formula that yields $T$ if and only if $P$ accepts $\sigma$. The Boolean formula encodes the constraints that a Turing machine operates under, such as that the only tape symbol that can be changed in any one step is the one under the machine's Read/Write head.

Our main interest in the theorem is that it shows there are problems in $\text{NP}$ that are at least as hard as any problem in that class.

6.9 **Definition** A problem $L$ is $\text{NP}$-hard if $\hat{L} \leq_p L$ for every $\hat{L} \in \text{NP}$. A problem $L$ is $\text{NP}$-complete if it is $\text{NP}$-hard and is a member of $\text{NP}$.

In general, a problem $L$ is hard for a class $C$ if all problems in that class reduce to it: if $\hat{L} \in C$ then $\hat{L} \leq_p L$. A problem is complete for a class if it is hard for that class and is also a member of that class.

6.10 **Figure**: This shows all problems, the collection of all $L \in \mathbb{B}^*$. In the bottom right is $\text{NP}$, drawn with $P$ as a strict subset (although we don't know that is true). In the top right, shaded, are the $\text{NP}$-hard problems. The highlighted intersection of the two is the set of $\text{NP}$ complete problems.

6.11 **Lemma** If $L_0$ is $\text{NP}$ complete, and $L_0 \leq_p L_1$, and $L_1 \in \text{NP}$ then $L_1$ is $\text{NP}$ complete.

**Proof** Exercise 6.36.

The $\text{NP}$ complete problems are to the class $\text{NP}$ as the problems Turing-equivalent to $K$, the solution to the Halting problem, are to the computably enumerable sets.
They are at the top level of their class: we could solve every other problem in
that class if we could solve that one problem. (However, we know that we cannot
solve \( K \). The situation is different with the \textbf{NP} complete problems; more below.)

Soon after Cook raised the question of \textbf{NP} completeness, R Karp
brought it to widespread attention. He produced a list of twenty one
problems that were well known to be difficult, ones where many smart
people had for years been unable find an efficient algorithm. Karp
showed that they were all \textbf{NP} complete and so if any one could be
solved by an efficient algorithm then they all could be.

Karp’s paper made clear that the collection of \textbf{NP} complete prob-
lems encompassed many problems, spread across Computer Science,
Mathematics, and the natural sciences. These are hard problems, ones
that researchers have struggled to find a fast algorithm to solve. Not
every hard problem is \textbf{NP} complete but many, many thousands of problems have
been shown to be \textbf{NP} complete and so whatever it is that makes these problems
hard, they all share it.

The next list contains the problems most often used to prove that something
is \textbf{NP} complete. These descriptions appeared earlier; they are copied here for
convenience. Typically, proving that a problem \( L \) is \textbf{NP} complete has two halves.
The first half shows that \( L \in \text{NP} \). Usually this is easy, just show that a witness
exists that could be verified in polytime. The second half shows that your problem
is \textbf{NP}-hard. Often this involves showing that some problem on the list reduces to \( L \);
for instance, that \( 3\text{-SAT} \leq_p L \).

\begin{theorem}[Basic \textbf{NP} Complete Problems]
Each of these problems is \textbf{NP}-complete.
\begin{description}
\item[3-Satisfiability, 3-\texttt{SAT}]
Given a propositional logic formula in conjunctive normal
form in which each clause has at most 3 variables, decide if it is satisfiable.
\item[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.
\item[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 \( C \).
\item[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 such that any two are connected.
\item[Hamiltonian Circuit]
Given a graph, decide if it contains a Hamiltonian circuit, a
cyclic path that includes each vertex.
\item[Partition]
Given a finite multiset \( S \), decide if there is a division of the set into
two parts \( \hat{S} \) and \( S - \hat{S} \) so the total of the elements in the two is the same,
\[ \sum_{s \in \hat{S}} s = \sum_{s \notin \hat{S}} s. \]
\end{description}
\end{theorem}
6.13 **Example**  Travelling Salesman is NP complete because Hamiltonian Circuit reduces to it: Hamiltonian Circuit $\leq_p$ Travelling Salesman. To prove this we must produce a reduction function $f$ and check that it is polytime.

The function inputs an instance of Hamiltonian Circuit, a graph $G = \langle V, E \rangle$ whose edges are unweighted. It returns the instance of Travelling Salesman that uses the vertex set $V$ as cities and that takes the distances between the cities to be: $d(v_i, v_j) = 1$ if $v_ivj$ is an edge of $G$ and $d(v_i, v_j) = 2$ if $v_i v_j \notin E$, and such that the bound $B$ is the number of vertices $|V|$.

The bound means that there will be a Travelling Salesman solution if and only if there is a Hamiltonian Circuit solution, namely the Travelling Salesman solution uses the edges of the Hamiltonian circuit. What remains is to show that $f$ is polytime in the size of the input problem instance, which we take to be the number of vertices since the number of edges is less than twice the number of vertices (so that polytime in the number of vertices is also polytime in the number of vertices plus the number of edges). To output the Travelling Salesman instance, the algorithm examines all the pairs of vertices, which is a nested loop and so takes time that is quadratic in the number of vertices.

One of Karp’s points was the practical importance of NP completeness. Many problems from applications fall into this class. The next example illustrates. (It also illustrates that reductions can be complex.)

6.14 **Example**  Most colleges make a schedule by putting classes into time slots and then students pick which classes they will take. Imagine instead that students first select the classes, and then the college must solve the problem of deciding if there is a non-conflicting time schedule for those classes.

Consider a college with 2 time slots, $k$-many classes, each with some capacity for enrolled students, and $n$-many total students. Every student $s$ has two disjoint lists of classes $\ell_s, \hat{\ell}_s$ and must choose one class from each list $c_s \in \ell_s$ and $\hat{c}_s \in \hat{\ell}_s$. The problem $L$ is to decide if there is a way to partition the set of classes into two such that every class is in a time slot and every student is in both of the classes they selected, and no student is in two classes in the same time slot. We will show that $L$ is NP complete.

Observe that $L$ is a member of NP because checking a solution is fast: a witness $\omega$ is a sequence of assignments of students to classes and classes to time slots, which has length polynomial in the number of students, and we can in polytime check that each student is assigned one class from each of their lists, that each class is given a time slot, etc.

We will show that $L$ is NP-hard by showing that $3$-$SAT \leq_p L$, that there is a reduction function that runs in polytime. An input instance to this function consists of $m$-many clauses that are joined by $\land$'s, each clause with 3 or fewer of the Boolean variables $x_0, \ldots x_{n-1}$ or their negations, joined by $\lor$'s. The reduction function gives as output an instance of $L$. The output instance will have a successful schedule if and only if the input formula is satisfiable.
Section 6. Problem reduction

For each of the input instance’s Boolean variables the output instance will have an associated course: \( c_0, \ldots, c_{n-1} \) (of capacity equal to the total number of students). There will be more courses also, including one named ‘T’ and one named ‘F’ (the idea is that if the input is satisfied by an assignment where \( x_i = T \) then the output can allot \( c_i \) to the same slot as course T). These will each have capacity \( 2m + 1 \). Also start by giving the output \( L \) instance a student \( s_0 \) whose two lists are \( \ell_{s_0} = T \) and \( \hat{s}_0 = \{ F \} \); this puts them in separate slots.

Now consider the clauses in the input formula. The easiest clauses are the ones without negations, of the form \( x_{i_0} \), or \( x_{i_0} \lor x_{i_1} \), or \( x_{i_0} \lor x_{i_1} \lor x_{i_2} \). If the clause is \( x_{i_0} \) then in the output instance we create a student \( s \) with the two lists \( \ell_s = \{ c_{i_0} \} \) and \( \hat{s} = \{ F \} \). If the clause is \( x_{i_0} \lor x_{i_1} \) then the student gets \( \ell_s = \{ c_{i_0}, c_{i_1} \} \) and \( \hat{s} = \{ F \} \). Similarly, for \( x_{i_0} \lor x_{i_1} \lor x_{i_2} \) the created student’s lists are \( \ell_s = \{ c_{i_0}, c_{i_1}, c_{i_2} \} \) and \( \hat{s} = \{ F \} \).

Clauses with only negations, of the form \( \neg x_{i_0} \), or \( \neg x_{i_0} \lor \neg x_{i_1} \), or \( \neg x_{i_0} \lor \neg x_{i_1} \lor \neg x_{i_2} \), are similar. For instance, for \( \neg x_{i_0} \) we create a student with lists \( \ell_s = \{ T \} \) and \( \hat{s} = \{ c_{i_0} \} \).

The messiest clauses have a mixture of positive and negative atoms. We will do an example: suppose that the \( j \)-th clause is \( x_{i_0} \lor x_{i_1} \lor \neg x_{i_2} \). In addition to the classes \( c_{i_0}, c_{i_1}, \) and \( c_{i_2} \) created earlier, we also create three new classes \( t_j, f_j \) and \( z_j \), with capacity of one student each. And we don’t just create one student, we create three: \( s_j, u_j, \) and \( v_j \). Here are their lists.

<table>
<thead>
<tr>
<th>Student</th>
<th>( \ell )</th>
<th>( \hat{s} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_j )</td>
<td>{ ( c_{i_0}, c_{i_1}, t_j ) }</td>
<td>{ ( c_{i_2}, f_j ) }</td>
</tr>
<tr>
<td>( u_j )</td>
<td>{ ( t_j, z_j ) }</td>
<td>{ F }</td>
</tr>
<tr>
<td>( v_j )</td>
<td>{ T }</td>
<td>{ ( f_j, z_j ) }</td>
</tr>
</tbody>
</table>

To confirm that this reduction function suffices, we must show that if there is an assignment of truth values to the Boolean variables that satisfies the formula then there is a course schedule solution, and we must also show the converse.

So first suppose that there a way to give each Boolean variable a value of \( T \) or \( F \) such that the entire propositional logic formula evaluates to \( T \). Then we get a non-conflicting course time slot allotment by: where \( x_i = T \) put the course \( c_i \) in the same time slot as \( T \), and where \( x_i = F \) put the course \( c_i \) in the F slot. This clearly works for the all-positive atom or all-negative atoms cases. The interesting cases are the mixed positive and negative atom clauses, such as \( x_{i_0} \lor x_{i_1} \lor \neg x_{i_2} \). We can check that for each possible assignment making this clause evaluate to \( T \) there is a non-conflicting way to arrange the other courses. For instance, suppose that \( x_{i_0} = T, x_{i_1} = F, \) and \( x_{i_2} = T \). We have put course \( c_{i_0} \) in the time slot when \( T \) is offered and student \( s_j \) can take it then, and similarly can also take course \( c_{i_2} \) when \( F \) is offered. Then student \( u_j \) takes \( t_j \) when \( T \) is offered and also takes course \( F \), and student \( v_j \) takes \( T \) and \( f_j \).

Conversely, suppose that for any assignment of truth values to the Boolean
variables the input formula is not satisfied. We will show that implies that for any allocation of classes into time slots there is a conflict.

So fix an allocation of classes into time slots, say with the classes \( c_{i_0}, c_{i_1}, \ldots \) being offered in the same slot as class T and the rest with class F. Associate with that allocation of classes the assignment of Boolean variables that sets \( x_{i_0} = T \), \( x_{i_1} = T \), etc. and the rest to \( F \). This assignment causes the formula to yield a value of \( F \), as this formula is not satisfiable. So for this assignment the formula has some clause, clause \( j \), that does not evaluate to \( T \).

The first possibility is that clause \( j \) has either all positive atoms or all negative atoms. An example is the clause \( x_{i_0} \lor x_{i_1} \lor x_{i_2} \). Then the assignment must have \( x_{i_0} = F \), \( x_{i_1} = F \), and \( x_{i_2} = F \), and so the allocation of classes must be that all three of \( c_{i_0}, c_{i_1}, \) and \( c_{i_2} \) go with class F. This will cause a conflict in course assignments because the output instance was created with a student \( s_j \) having the lists \( \ell_{s_j} = \{ c_{i_0}, c_{i_1}, c_{i_2} \} \) and \( \hat{\ell}_{s_j} = \{ F \} \).

The other case is that clause \( j \) has a mixed form, such as \( x_{i_0} \lor x_{i_1} \lor \neg x_{i_2} \), so that the Boolean variable assignment is \( x_{i_0} = F \), \( x_{i_1} = F \), and \( x_{i_2} = T \). This association comes from the allocation of classes that puts \( c_{i_0} \) and \( c_{i_1} \) with class F, and puts \( c_{i_2} \) with T. We claim that there is no non-conflicting way to assign students to courses. Refer to the table above. This allocation of classes could only be non-conflicting if student \( s_j \) selected classes \( t_j \) and \( f_j \). But then because of the capacity of one in these classes, student \( u_j \) must select \( z_j \) and \( F \), and student \( v_j \) has a conflict.

Whew! To close, observe that the reduction function that creates the output \( L \) instance from the input propositional logic formula runs in polytime.

Another way to use the basic list to show that a problem \( L \) is \( \text{NP} \) complete is to find a special case of \( L \) on that list.

6.15 Example Knapsack starts with a multiset of objects \( S = \{ s_0, \ldots, s_{k-1} \} \) where the elements have a weight \( \omega(s_i) \in \mathbb{N}^+ \) and a value \( v(s_i) \in \mathbb{N}^+ \), along with a weight bound \( B \in \mathbb{N}^+ \) and value target \( T \in \mathbb{N}^+ \). It asks if there is \( K \subseteq S \) whose elements have total weight less than or equal to the bound and total value greater than or equal to the target.

This is an \( \text{NP} \) problem. The witness \( \omega \), where \( \omega[i] = 1 \) if \( s_i \) is in the knapsack, and \( \omega[i] = 0 \) if it is not, is only \( k \) bits long and so is \( O(n) \) in the number of elements in \( S \). A deterministic machine can verify this witness in polynomial time since it only has to total the weights and values of the elements of \( K \).

To show that Knapsack is \( \text{NP} \)-hard, we show that a special case is \( \text{NP} \)-hard. Consider the case where \( w(s_i) = v(s_i) \) for all \( s_i \in S \), and where \( B \) and \( K \) each equal half of the total of all the weights. This is a Partition problem, which is in the above list.

There are problems that are members of \( \text{NP} \) but are not \( \text{NP} \) complete. Most experts believe that Factoring is hard for classical computers\(^{\dagger} \) but that it is not

\(^{\dagger}\)In 1994, P. Shor discovered an algorithm for a quantum computer that solves the Factoring problem in polynomial time. This will have significant implications if quantum computation proves to be possible to engineer.
NP complete. Experts also suspect that Graph Isomorphism is not NP complete. As always with such judgements, though, without proof they could be mistaken.

**EXP** In this chapter’s first section we included in the list of commonly encountered orders of growth $O(2^n)$ and $O(3^n)$, and by extension other exponentials.

6.16 **Definition** A problem is an element of the complexity class EXP if there is an algorithm for solving it that runs in time $O(b^{p(n)})$ for some constant base $b$ and polynomial $p$.

Satisfiability can be solved in exponential time by checking each row of the truth table, and any NP problem can be solved from Satisfiability with only an addition of polytime. So EXP has this relationship to the classes we have already studied.

6.17 **Lemma** $P \subseteq NP \subseteq EXP$

*Proof* Fix $L \in NP$. We can verify $L$ on a deterministic Turing machine $P$ 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 $L$ in exponential time by brute-forcing it: we will use $P$ 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 $O(n^c 2^{n^c})$ operations. Finish by observing that $n^c 2^{n^c}$ is in $O(2^{n^c})$.

We don’t know whether there are any NP problems that absolutely require exponential time. Conceivably NP is contained in a smaller determinstic time complexity class—for instance, maybe Satisfiability can be solved in less than exponential time. But we just don’t know.

Whereas a first take on polytime is “can conceivably be used,” a first approximation of EXP is that for some of its problems the best algorithms are just too slow to imagine using. However, the big take-away from EXP is that it contains nearly every problem, and every class, that we concern ourselves with in practice. We can construct theories about still harder problems, but EXP is big enough to contain most problems we ever seriously hope to attack.
\( P = \text{NP?} \) Every deterministic Turing machine is trivially a nondeterministic machine, and so \( P \subseteq \text{NP} \). What about the other direction?

Posing the \( P \) versus \( \text{NP} \) question asks: does adding parallelism add speed? Can unbounded parallelism bring problems from super-polynomial to polynomial?

The short answer is that no one knows. One of these two pictures is right but we do not know which one.

6.19 Figure: Which is it: \( P \subseteq \text{NP} \) or \( P = \text{NP} \)?

There is a simple way to settle the question. By Lemma 6.11, if someone shows that any \( \text{NP} \) complete problem is a member of \( P \) then \( P = \text{NP} \). If someone shows that there is an \( \text{NP} \) problem that is not a member of \( P \) then \( P \neq \text{NP} \). But despite nearly a half century of effort by many extremely bright people, no one has done either one.

Many people have worked on this question because this is an extremely important problem.\(^\dagger\) True, as formulated in Karp’s original paper, the question of whether \( P \) equals \( \text{NP} \) may 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 \ldots 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 demonstrated that many of the problems people have been struggling with in practical applications fall into this category. Researchers who have been struggling with \text{Vertex Cover} and those who have been struggling with \text{Clique} found that they are in some sense working on the same problem. By now the list of these 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.

In practice, proving that a problem is a member of \( \text{NP} \) is often an ending point;

\(^\dagger\) One indication of its importance is that it is on a list of problems in mathematics for which there is a one million dollar prize; see \url{http://www.claymath.org/millennium-problems}.\)
a researcher may well reason that continuing to try to find an algorithm is likely to be not fruitful since many of the best minds of mathematics, science, and computer science have failed at it. They may instead turn their attention elsewhere, perhaps to approximations.

In the book’s first part we studied problems that are unsolvable. That was a black and white situation; either a problem is mechanically solvable in principle or it is not. We here study that many problems of interest are infeasible. That is, the class of NP complete problems form a kind of transition between the possible and the impossible.

We can use this to engineering advantage. For instance, schemes for holding elections are notoriously prone to manipulation and there are theorems that say they must be. But we can hope to use system that, while in principle is manipulatable, are constructed so that it is in practice infeasible to compute how to do that. Another example of the same thing is the celebrated RSA encryption system that is used to protect Internet commerce; see Extra A.

This returns us to the book’s opening question about mathematical proof. Recall the Entscheidungsproblem that was a motivation behind the definition of a Turing machine. It asks for an algorithm that inputs a mathematical statement $\sigma$ and outputs whether $\sigma$ is true. Maybe it is a caricature but imagine that the job of mathematicians is to prove theorems. The Entscheidungsproblem asks if we can replace them with machines.

In the intervening century we have come to understand, through the work of Gödel and others, the difference between a statement’s being true and its being provable, and that we cannot always decide from axioms whether a statement is true. Consequently, we modify the Entscheidungsproblem to ask for an algorithm that inputs statements and decides whether they are provable.

The plan is in principle simple. A proof is a sequence of statements, $\sigma_0, \sigma_1, \ldots, \sigma_k$, where the final statement is the conclusion, and where each statement either is an axiom or else follows from the statements before it by an application of a rule of logic (a typical rule allows the simultaneous replacement of all $x$’s with $y + 4$’s). In principle a computer could brute-force the question of whether a given statement is provable by doing a dovetail, a breadth-first search of all derivations. If a proof exists then it will appear eventually.

The difficulty is that final word, eventually. This algorithm is very slow. So we ask: Is there a tractable way?

In the terminology we now have, the modified Entscheidungsproblem is a decision problem: given a statement $\sigma$ and bound $B \in \mathbb{N}$, we ask if there is a sequence of statements witnessing a proof ending in $\sigma$ that is shorter than the bound. A computer can quickly check whether a given proof is valid. So 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 one but no one can show there isn’t one either.

As far back as 1956, Gödel raised these issues in a letter to von Neumann (this letter did not become public until years later).

One can obviously easily construct a Turing machine, which for every formula $F$ in first order predicate logic and every natural number $n$, allows one to decide if there is a proof of $F$ of length $n$ (length = number of symbols). Let $\Psi(F, n)$ be the number of steps the machine requires for this and let $\phi(n) = \max_F \Psi(F, n)$. The question is how fast $\phi(n)$ grows for an optimal machine. One can show that $\phi(n) \geq k \cdot n$. If there really were a machine with $\phi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number $n$ so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows that slowly. Since it seems that $\phi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and after all $\phi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from $N$ to $\log N$ (or $(\log N)^2$). . . 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.

So we can compare $\mathbf{P}$ versus $\mathbf{NP}$ 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 $\mathbf{P}$ versus $\mathbf{NP}$ question, in contrast, speaks to what we can ever, in practice, know.

**Discussion** The $\mathbf{P}$ versus $\mathbf{NP}$ question is certainly the sexiest one in the Theoretical of Computing today. It has attracted a great deal of speculation, and gossip. In 2018 a poll of experts found that out of 152 respondents, 88% thought that $\mathbf{P} \neq \mathbf{NP}$ while only 12% thought that $\mathbf{P} = \mathbf{NP}$. This subsection discusses some of the intuition around the question.

First the intuition that $\mathbf{P} \neq \mathbf{NP}$. When we work on a jigsaw puzzle we perceive that checking that the solution is correct is very much easier than finding the solution in the first place. The checking is mechanical, tedious. But the finding, we think, is creative — solving a jigsaw puzzle by brute-force trying every possible piece against every other strikes us as far too much computation to be practical.

Similarly, encryption schemes are engineered so that, given an encrypted message, decrypting it with the key is fast and easy but trying to brute force decrypt it by doing all possible keys is, we think, just not tractable.

This is the distinction between problems in $\mathbf{P}$ and problems that are $\mathbf{NP}$ complete. A problem is in $\mathbf{P}$ if finding a solution is fast, and a problem is in $\mathbf{NP}$ if verifying
the correctness of a given witness $\omega$ is fast. From this point of view, the result that $P \subseteq NP$ becomes the observation that if a problem is fast to solve then it must be fast to verify. But inclusion in the other direction seems to most experts to be unlikely.

Thus, informally, the $P$ versus $NP$ question asks if finding a solution as fast as recognizing one. If $P = NP$ then the two jobs are, essentially, equally difficult. But to most people, checking a proposed solution in polynomial time seems easier than finding a solution in polynomial time. These people guess that $P \neq NP$.

Many commentators, including Cook, S. Cook 2000 have extended this thinking outside the bounds of Theoretical Computer Science. “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 likely — then we could have fast mechanical writers of excellent music.

We finish with a taste of the contrarian view.

Many observers have noted that there are many cases where everyone “knew” that some algorithm was the fastest to solve a certain problem but 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 is obvious, and natural, and runs in time $O(n^3)$. But while trying to prove that Gauss’s Method is optimal, Strassen instead found a method that is $O(n^{\lg 7})$ ($\lg 7 \approx 2.81$). Every day on the Theory of Computing blog feed (Various authors 2017) there are examples of this, of researchers producing algorithms faster than the ones previously known. R J Lipton expresses the same sense (Lipton 2009).

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, (D. Knuth 2014).

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

---

$\dagger$ Here is a small analogy for both the grade school multiplication and Gauss’s Method results: consider the algebra problem of computing $ac + ad + bc + bd$. Someone might claim that to evaluate it “obviously” requires four multiplications. But then someone else counters that it equals $(a + b)(c + d)$, and so only requires one multiplication. The two cited results are a few among many examples evincing that naturalness and obviousness do not guarantee that the answer is correct, that no one will come along with a clever way that does the job with less.
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 $P = \text{NP}$ 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 but in the end these researchers, and all of us, look to settle the question with proof.

V.6 Exercises

✓ 6.20 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 things wrong with it.

✓ 6.21 True or false: NP is a subset of NP complete, which is a subset of NP-hard.

6.22 Suppose that the language $A$ is polynomial time reducible to the language $B$, $A \leq_p B$. Which of these are true?
   (A) A tractable way to decide $A$ can be used to tractably decide $B$.
   (B) If $A$ is tractably decidable then $B$ is tractably decidable also.
   (C) If $A$ is not tractably decidable then $B$ is not tractably decidable too.

✓ 6.23 Assume that $P \neq \text{NP}$. Which of these can we infer from the fact that Primality is in NP but is not known to be NP-complete?
   (A) There exists an algorithm that solves arbitrary instances of Primality.
   (B) There exists an algorithm that efficiently solves arbitrary instances of Primality.
   (C) If we found an efficient algorithm for Primality then we could immediately use it to solve Travelling Salesman.

✓ 6.24 Suppose $L_1 \leq_p L_0$. For each, decide if it is true or false. (A) If $L_0$ is NP complete then so is $L_1$. (B) If $L_1$ is NP complete then so is $L_0$. (C) If $L_0$ is NP complete and $L_1$ is in NP then $L_1$ is NP complete. (D) If $L_1$ is NP complete and $L_0$ is in NP then $L_0$ is NP complete. (E) It cannot be the case that both $L_0$ and $L_1$ are NP complete (F) If $L_1$ is in $P$ then so is $L_0$. (G) If $L_0$ is in $P$ then so is $L_1$.

✓ 6.25 Vertex Cover inputs a graph $G = (V, E)$ 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. We will show both that Independent Set $\leq_p$ Vertex Cover and that Vertex Cover $\leq_p$ Independent Set.

(A) Consider this graph. Find a vertex cover $S$ with $k = 4$ elements.

(B) In that graph find an independent set $\hat{S}$ with $\hat{k} = 6$ elements.

(C) Prove that $S$ is a vertex cover if and only if its complement $\hat{S} = V - S$ is an independent set.

(D) Conclude that Independent Set $\leq_p$ Vertex Cover and that Vertex Cover $\leq_p$ Independent Set.

✓ 6.26 Recall that Subset Sum 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. Recall also that Partition inputs a multiset $S$ and decides whether or not it has a subset $\hat{S} \subset S$ so that the sum or 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.

6.27 If a problem $L$ is in NP and $L \leq_p \hat{L}$ then what can we say about the problem $\hat{L}$?

6.28 Are there any known problems in NP and not in P that aren’t NP complete?

6.29 Show that $\leq_p$ is transitive: if $L_0 \leq_p L_1$ and $L_1 \leq_p L_2$ then $L_0 \leq_p L_2$.

✓ 6.30 The difficulty in settling $P = NP$ is to get lower bounds. That is, the trouble lies in showing that the given problem cannot be solved by any algorithm without such-and-such many steps. A common mistake is to think that any algorithm must visit all of its input and then to produce a problem with lots of input. Show that the successor function can be done on a Turing machine in constant time, in only a few steps, so that the running time if the input is large is the same as the time if the input is small. That is, show that this problem can be done with an algorithm that does not visit all the input: on a Turing machine given input $n$ in unary, with the head under the leftmost 1, end with $n + 1$-many 1’s, with the head under the leftmost 1.

6.31 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 game can reasonably run, for instance a chess game? (Chess has more than two moves at each step but it makes a reasonable game comparison.)

✓ 6.32 Travelling Salesman is NP complete. From P ≠ NP which of the following statements could we infer? (A) No algorithm solves all instances of Travelling Salesman. (b) No algorithm quickly solves all instances of Travelling Salesman. (c) Travelling Salesman is in P. (d) All algorithms for Travelling Salesman run in polynomial time.

6.33 Suppose that L₁ ≤ₚ L₀. Determine if each statement is correct. (A) If L₀ is NP complete then so is L₁. (b) If L₁ is NP complete then so is L₀. (c) If L₀ is NP complete and L₁ ∈ NP then L₁ is also NP complete. (d) If L₁ is NP complete and L₀ ∈ NP then L₀ is also NP complete. (e) Not both of L₀, L₁ are NP complete. (f) If L₁ is in P then L₀ is in P. (g) If L₀ is in P then L₁ is in P.

6.34 Which of these are NP complete? (a) The brute force algorithm for solving Satisfiability. (b) The quicksort algorithm for sorting strings.

6.35 Prove that if P = NP then every P ∈ P is NP complete, except for the problems of determining membership in the empty language and the full language, P = Ø and P = Σ*.

6.36 Prove Lemma 6.11.

6.37 Recast the second half of the proof of Lemma 5.8 in computation tree terms.

6.38 Show that a polynomial time algorithm that calls a polynomial time subroutine is still polynomial time. Show that if the subroutine is exponential then the entire algorithm is exponential.

6.39 Prove that if L ≤ₚ Ă L if and only if Ă L ≤ₚ L.

6.40 We will sketch a proof that the Halting problem is NP hard but not NP. To suit this Part, we take the Halting problem to be the language HP = {⟨P_e, x⟩ | φ_e(x)↓} where P_e and x have reasonably efficient representations as bitstrings.
(A) Show that the Halting problem is not a member of NP.
(b) Sketch an argument that for any problem L ∈ NP, there is a polynomial time computable f : B* → B* such that σ ∈ L if and only if f(σ) ∈ HP.

6.41 A problem is P complete if it maximally hard among problems in P.
(A) Give a formal definition.
(b) Show that if there is a P-complete problem, and if it is not a member of NP then P ≠ NP.
Section 7. Other classes

There are many other classes but we will see one more that delineates, in a sense, our everyday concerns.

Time Complexity  The impediment a programmer runs across first is time.

7.1 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 \( O(f) \).

A problem is an element of \( \text{NTIME}(f) \) if it is decided by a nondeterministic Turing machine that runs in time \( O(f) \).

This does not say that an algorithm is in that class, rather it says that a language is in that class. A language is in that class if there is an algorithm for its decision problem that has the stated big-\( O \) runtime behavior.

7.2 Lemma  A problem is polytime, \( P \), if it is a member of \( \text{DTIME}(n^c) \) for some power \( c \in \mathbb{N} \).

\[
P = \bigcup_{c \in \mathbb{N}} \text{DTIME}(n^c) = \text{DTIME}(n) \cup \text{DTIME}(n^2) \cup \text{DTIME}(n^3) \cup \cdots
\]

The matching statements hold for \( \text{NP} \) and \( \text{EXP} \).

\[
\text{NP} = \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c) = \text{NTIME}(n) \cup \text{NTIME}(n^2) \cup \text{NTIME}(n^3) \cup \cdots
\]

\[
\text{EXP} = \bigcup_{c \in \mathbb{N}} \text{DTIME}(2^{n^c}) = \text{DTIME}(2^n) \cup \text{DTIME}(2^{n^2}) \cup \text{DTIME}(2^{n^3}) \cup \cdots
\]

Proof  The only equality that is not immediate is the last one. Recall that a problem is in \( \text{EXP} \) if an algorithm for it that runs in time \( O(b^{p(n)}) \) for some constant base \( b \) and polynomial \( p \). The equality above only uses the base 2. To cover the discrepancy, we will show that \( 3^n \in O(2^{o(n^2)}) \). Consider \( \lim_{n \to \infty} 2^{(x^2)/3^x} \). Rewrite the fraction as \( (2^x/3)^x \), which when \( x > 2 \) is larger than \( (4/3)^x \), which goes to infinity. This argument works for any base, not just \( b = 3 \).

7.3 Remark  While the above description of \( \text{NP} \) 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 \( \text{NP} \) 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.

7.4 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.

**7.5 Definition** Let \( s : \mathbb{N} \rightarrow \mathbb{N} \). A language decision problem is an element of \( \text{DSPACE}(s) \), or \( \text{SPACE}(s) \), if that languages is decided by a deterministic Turing machine that runs in space \( \mathcal{O}(s) \). A problem is an element of \( \text{NSPACE}(s) \) if the languages 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.

**7.6 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.

**7.7 Definition**

\[
\text{PSPACE} = \bigcup_{c \in \mathbb{N}} \text{DSPACE}(n^c) = \text{DSPACE}(n) \cup \text{DSPACE}(n^2) \cup \text{DSPACE}(n^3) \cup \cdots
\]

\[
\text{NPSPACE} = \bigcup_{c \in \mathbb{N}} \text{NSPACE}(n^c) = \text{NSPACE}(n) \cup \text{NSPACE}(n^2) \cup \text{NSPACE}(n^3) \cup \cdots
\]

So \( \text{PSPACE} \) 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.

**7.8 Lemma** \( \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \)

*Proof* For any problem in \( \text{NP} \), 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’.

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.

7.9 Theorem (Savitch’s Theorem) \( \text{PSPACE} = \text{NPSPACE} \)

We finish with a list of the most natural complexity classes.

7.10 Definition These are the canonical complexity classes

1. \( \text{P} \), deterministic polynomial time and \( \text{NP} \), nondeterministic polynomial time
2. \( \text{E} = \bigcup_{k=1,2,...} \text{DTIME}(k^n) \) and \( \text{NE} = \bigcup_{k=1,2,...} \text{DTIME}(k^n) \)
3. \( \text{EXP} = \bigcup_{k=1,2,...} \text{DTIME}(2^{n^k}) \), deterministic exponential time and \( \text{NEXP} = \bigcup_{k=1,2,...} \text{NTIME}(2^{n^k}) \), nondeterministic exponential time
4. \( \text{L} = \text{DSPACE}(\lg n) \), deterministic log space and \( \text{NL} = \text{NSPACE}(\lg n) \), nondeterministic log space
5. \( \text{PSPACE} \), deterministic polynomial space
6. \( \text{EXPSPACE} = \bigcup_{k=1,2,...} \text{DSPACE}(2^{n^k}) \), 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/.

One way to understand what is happening with these classes is that defining a class is a way to ask a type of Theory of Computing question. For instance, we have already seen that asking whether \( \text{NP} \) equals \( \text{P} \) 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 \( \text{P} \subseteq \text{PSPACE} \). 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 many more, see the Zoo.

The class \( \text{BPP} \), 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 \( \text{BPP} \) equals \( \text{P} \) is asking whether 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.7 Exercises

7.11 Give a naive algorithm for each problem that is exponential.
   (A) Travelling Salesman
   (B) Subset Sum
   (C) $k$ Coloring

Extra

V.A RSA Encryption

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 website using an https address, the site sends you information, called a key, that your browser uses to encrypt your card number. The website then uses a different key to decrypt. This is an important point: the decrypter must differ from 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 bit string 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 the fact that there are things we can do easily, but that we do not know how to 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 engineering Big-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 we shall describe exploits this difference.$^\dagger$

It was invented in 1976 by three graduate students, R Rivest, A Shamir, and L Adleman. Rivest read a paper proposing key pairs and decided to implement the idea. 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 $m^e \mod n$. To decrypt it, to recover $m$, calculate $(m^e)^d \mod n$. 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 $\phi(n) = (p - 1) \cdot (q - 1)$. Next, choose $e$ with $1 < e < \phi(n)$ 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 $\phi(n) = (p - 1)(q - 1) = 11 200$. 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 $d = e^{-1} \mod 11 200 = 6 597$, 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

$^\dagger$Recent versions of the algorithm used in practice incorporate refinements that we shall not discuss. The core idea is unchanged.
two substrings, getting the decimals 72 and 105. Using her public key he computes

\[ 72^{3533} \mod 11413 = 10496 \quad 105^{3533} \mod 11413 = 4861 \]

and sends Alice the sequence \( \langle 10496, 4861 \rangle \). Alice recovers his message by using her private key.

\[ 10496^{6597} \mod 11413 = 72 \quad 4861^{6597} \mod 11413 = 105 \]

**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.

\[
\lim_{x \to \infty} \frac{\text{number of primes less than } x}{(x/\ln x)}
\]

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.

**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.

**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, \(sa + tm\) 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 \(m\).

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.

**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 \(i \in \{1, \ldots, p-1\}\) and reduce modulo \(p\) to get the numbers \(r_i = ia \mod p\).

We will show that the set \(R = \{r_1, \ldots, r_{p-1}\}\) equals the set \(P = \{1, \ldots, p-1\}\). 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 \(\{1, \ldots, p-1\}\).

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 \(r_0 - r_1 = i_0 a - i_1 a = (i_0 - i_1) a\), because \(p\) is prime and does not divide \(i_0 - i_1\) or \(a\) (as each is smaller in absolute value than \(p\)), it 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.

\[
a \cdot 2a \cdots (p-1)a \equiv 1 \cdot 2 \cdots (p-1) \pmod{p}
\]

\[
(p-1)! \cdot a^{p-1} \equiv (p-1)! \pmod{p}
\]

Cancelling the \((p-1)!\)'s gives the result.

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.

<table>
<thead>
<tr>
<th>( a )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^{7-1} )</td>
<td>1</td>
<td>64</td>
<td>729</td>
<td>4096</td>
<td>15625</td>
<td>46656</td>
</tr>
<tr>
<td>( (a^6 - 1)/7 )</td>
<td>0</td>
<td>9</td>
<td>104</td>
<td>585</td>
<td>2232</td>
<td>6665</td>
</tr>
</tbody>
</table>

For instance, \( 15625 = 7 \cdot 2232 + 1 \).

Given \( n \), if we find a base \( a \) with \( 0 < a < n \) so that \( a^{n-1} \mod n \) is not 1 then \( n \) is not prime.

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.\(^\dagger\) 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 \( O(k \cdot (\log n)^2 \cdot \log \log n \cdot \log \log \log n) \). 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.\(^\dagger\) 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 $pq = n$. □

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 $O(b^3)$ time (and $O((b)$ space), called Shor’s algorithm. But it runs only on quantum computers. At this moment there are no such computers built, although there has been progress on that. For the moment, RSA seems safe. (There are schemes that could replace it, if needed.)

V.A Exercises

✓ A.11 There are twenty five primes less than or equal to 100. Find them.

✓ A.12 We can walk through an RSA calculation.
   (a) For the primes, take $p = 11$, $q = 13$. Find $n = pq$ and $\varphi(n) = (p-1) \cdot (q-1)$.
   (b) For the encoder $e$ use the smallest prime $1 < e < \varphi(n)$ that is relatively prime with $\varphi(n)$.
   (c) Find the decoder $d$, the multiplicative inverse of $e$ modulo $n$. (You can use Euclid’s algorithm, or just test the candidates.)
   (d) Take the message to be represented as the number $m = 9$. Encrypt it and decrypt it.

A.13 To test whether a number $n$ is prime, we could just try dividing it by all numbers less than it.
   (a) Show that we needn’t try all numbers less than $n$, instead we can just try all $k$ with $2 \leq k \leq \sqrt{n}$.
   (b) Show that we cannot lower that any further than $\sqrt{n}$.
   (c) For input $n = 10^{12}$ how many numbers would you need to test?
   (d) Show that this is a terrible algorithm since it is exponential in the size of the input.

A.14 Show that the probability that a random $b$-bit number is prime is about $1/b$. 
Extra

V.B Tractability and good-enoughness

Are we taking the right approach to characterizing the behavior of algorithms, to understanding the complexity of computations?

A theory shapes the way that you look at the world. For instance, Newton’s $F = ma$ points to an approach to analyzing physical situations: if you see a change, look for a force. That approach has been fantastically successful, enabling us to build bridges, send people to the moon, etc.

So we should ask if our theory is right. Of course, the theorems are right—the proofs check out, the results stand up to formalization, etc. But it is healthy to examine the current approach to ask whether there is a better way to see the problems in front of us.

In the theory we’ve outlined, Cobham’s Thesis identifies $P$ with the tractable problems. However, the situation today is not so neat.

First, there are some problems known to be in $P$ for which we do not know a practical approach. For one thing, as we discussed when we introduced Cobham’s Thesis, a problem for which the smallest possible algorithm is $O(n^{1000})$ is not practical. True, for problems that are announced with best known algorithms having such huge exponents, over time researchers improve the algorithm and the exponents drop, but nonetheless there are problems in the current literature associated with impractical exponents. And also not practical is when an algorithm is $O(n^2)$ but whose running time on close inspection proves to be something like $2^{1000} n^2$.

On the other side of the ledger we have problems not known to be in $P$ for which we have solutions good enough for practice.

One such problem is the Travelling Salesman problem. Experts believe that it is not in $P$, since it is $NP$ complete, but nonetheless there exist algorithms that can in a reasonable time find solutions for problem instances involving millions of nodes, with a high probability 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.

In May 2004, the Traveling Salesman instance of visiting all 24,978 cities in Sweden was solved, giving a tour of about 72,500 kilometers. The approach was to find a nearly-best solution and then use that to find the best one. The final stages, that improved the lower bound by by 0.000023 percent, required eight years of computation time running in parallel on a network of Linux workstations.

There are many results that give answers that are practical for problems
that our theory suggests are intractible. And many problems that are
attackable in theory but that turn out to be awkward in practice. So much
more work needs to be done.
Part Four

Appendix
Here we recap two areas of prerequisite material, as a reference or quick review and to fix notation.

**Appendix A. Strings**

An alphabet is a nonempty and finite set of symbols. 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 $\varepsilon$. We write symbols in a distinct typeface, as in $1$ or $a$, because the alternative of quoting them would be clunky.†

One potentially surprising aspect of the term 'symbol' is that a symbol may contain more than one character. For instance, a programming language may have `if` as a symbol, indecomposable into separate letters.

For example, the Scheme alphabet contains the symbols `or` and `car`, and allows variable names such as `a`, `x`, or `lastname`. An example of a string is `(or a ready)`, which is a sequence of five alphabet elements $\langle$, or, a, ready, $\rangle$.

Traditionally, alphabets are denoted with the Greek letter $\Sigma$. We also name strings with lower case Greek letters and denote the items in the string with the associated lower case roman letter, as in $\sigma = \langle s_0, \ldots, s_i\rangle$ or $\tau = \langle t_0, \ldots, t_j\rangle$. Note that the subscript indices are set up so that if $i = 0$ or $j = 0$ then this gives the empty string, so we take $i, j \in \mathbb{N}$. Where $\sigma = \langle s_0, \ldots, s_n\rangle$ is a string, its length $|\sigma|$ is the number of symbols that it contains, $n$. In particular, the length of the empty string is $|\varepsilon| = 0$.

The 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 $abc$ instead of $\langle a, b, c \rangle$. This convenience comes with the disadvantage that without the diamond brackets the empty string is just nothing, which is why we use the separate symbol $\varepsilon$.‡

The alphabet consisting of the zero and one characters is $\mathbb{B} = \{0, 1\}$. Strings over this alphabet are bitstrings or bit strings.

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 is the Kleene star, read aloud as “star.”

Strings are simple, so there are only a few operations. Let $\sigma = \langle s_0, \ldots, s_i\rangle$ and $\tau = \langle t_0, \ldots, t_j\rangle$ be strings over an alphabet $\Sigma$. The concatenation $\sigma \tau$ or $\sigma \cdot \tau$ appends the second sequence to the first: $\sigma \tau = \langle s_0, \ldots, s_i, t_0, \ldots, t_j\rangle$. Where

---

† We give them a distinct look because to distinguish the symbol ‘a’ from the variable ‘a’, so that we can tell “let $S = a$” apart from “let $S = a.$” Symbols are not variables—they don’t hold a value, they are themselves a value.‡ To see why when we drop the commas we want the alphabet to consist of single-character symbols, consider $\Sigma = \{a, aa\}$ and the string $aaa$. Without the commas this string is ambiguous: it could mean $\langle a, aa \rangle$, or $\langle aa, a \rangle$, or $\langle a, a, a \rangle$.§ Omitting the diamond brackets and commas also blurs the distinction between a symbol and a one-symbol string, between a and $\langle a \rangle$. However, dropping the brackets it is so convenient that we accept this disadvantage.
Let \( \sigma = \tau_0 \cdots \tau_{k-1} \) then 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, \( \tau_{k-1} \), is a **suffix**.

A **power** or **replication** of a string is an iterated concatenation with itself, so that \( \sigma^2 = \sigma \sigma \) and \( \sigma^3 = \sigma \sigma \sigma \), etc. We write \( \sigma^1 = \sigma \) and \( \sigma^0 = \varepsilon \). The **reversal** \( \sigma^R \) of a string takes the symbols in reverse order: \( \sigma^R = \langle s_{l-1}, \ldots, s_0 \rangle \). 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 abcbaac. The third power \( \sigma^3 \) is abcabcabc, and the reversal \( \tau^R \) is caab. A string that equals its own reversal is a **palindrome**; examples are \( \alpha = abba \), \( \beta = cdc \), and \( \varepsilon \).

### Exercises

**A.1** Let \( \sigma = 10110 \) and \( \tau = 110111 \) be bit strings. Find each. (A) \( \sigma^\sim \tau \) (B) \( \sigma^\sim \tau \sim \sigma \) (C) \( \sigma^R \) (D) \( \sigma^3 \) (E) \( \theta^3 \sim \sigma \)

**A.2** Let the alphabet be \( \Sigma = \{a, b, c\} \). Suppose that \( \sigma = ab \) and \( \tau = bca \). Find each. (A) \( \sigma^\sim \tau \) (B) \( \sigma^2 \sim \tau^2 \) (C) \( \sigma^R \sim \tau^R \) (D) \( \sigma^3 \)

**A.3** Let \( \mathcal{L} = \{ \sigma \in \Sigma^* \mid |\sigma| = 4 \text{ and } \sigma \text{ starts with } \theta \} \). How many elements are in that language?

**A.4** Suppose that \( \Sigma = \{a, b, c\} \) and that \( \sigma = abcabcba \). (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 \( |\sigma^\sim \tau| \)? 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) \( \alpha^\sim \varepsilon = \alpha \) and \( \varepsilon^\sim \alpha = \alpha \) (B) \( \alpha^\sim \beta = \beta^\sim \alpha \) (C) \( \alpha^\sim \beta^R = \beta^R \sim \alpha^R \) (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 \( \sigma^\sim \tau \neq \tau^\sim \sigma \).

**A.8** In defining decomposition above we have \( \sigma = \tau_0 \cdots \tau_{n-1} \), 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 replication is equivalent to the one above.

\[
\sigma^n = \begin{cases} 
\varepsilon & \text{if } n = 0 \\
\sigma^{n-1} \sim \sigma & \text{if } n > 0
\end{cases}
\]
Appendix B. **Functions**

A function is an input/output relationship: for each input value there is an output value, and that output value is unique.

One 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 over a–z with the length of that string. A third is the association of each polynomial $a_n x^n + \cdots + a_1 x + a_0$ with a Boolean value $T$ or $F$, depending on whether $1$ is a root of that polynomial.

For the definition fix two sets, a domain $D$ and a codomain $C$. A function, or map, $f : D \to C$ is a set of pairs $(x, y) \in D \times C$ subject to the restriction that every element $x \in D$ appears in one and only one pair. Where $(x, y)$ is one of the pairs we write $f(x) = y$ or $x \mapsto y$. (Note the difference between the arrow in $f : D \to C$ and the one in $x \mapsto y$). We say that $x$ is an input or argument to the function, and that $y$ is an output or value.

The most important point is what a function isn’t: it isn’t a formula. The function that gives the presidents of the US, $f(0) =$ George Washington, etc., has no sensible formula. Nor can you produce a formula that returns World Series winners, including next year’s. Many functions are described by formulas, such as $f(m) = mc^2$, and many functions are described by computer programs. But what makes something a function is that for each input from the domain there is one and only one output from the codomain, not that you can calculate those output values.

A function may take more than one input, for instance $\text{dist}(x, y) = \sqrt{x^2 + y^2}$. We may call a function 2-ary, or 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 extra 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 $f(x) = \lfloor x \rfloor$.

![Graph of $f(x) = x^3$](image1.png)

![Graph of $f(x) = \lfloor x \rfloor$](image2.png)

We also may illustrate functions using a bean diagram, which does not have the domain and the codomain mixed, but instead separates them. Here it pictures the
action of the exclusive or operator.

On the right is a variant of the bean diagram, showing the absolute value function mapping integers to integers.

**Codomain and range** Where $S \subseteq D$ is a subset of the domain, its image is the set $f(S) = \{ f(s) \mid s \in S \}$. Thus, under the floor function the image of the positive reals is the nonnegative reals. The image of the entire domain is the range of the function, $\text{ran}(f) = f(D) = \{ f(s) \mid s \in S \}$. Note that the range is not the same as the codomain; instead, the codomain is a convenient superset. For instance, for the real function $f(x) = x^2$ we usually write $f : \mathbb{R} \to \mathbb{R}$, although the range is the nonnegative reals.

When we are defining a function and write $f : D \to C$ we need that every $f(x)$ is indeed an element of $C$; usually it is clear but on occasion we need to verify it explicilty.

**Domain** Sometimes it is the domain that is a concern. Examples of real number functions where the domain causes trouble are that $f(x) = 1/x$ is undefined at $x = 0$, and that the infinite series $g(r) = 1 + r + r^2 + \cdots$ diverges when $r$ is outside the interval $(-1..1)$. Formally, when we define the function we must specify the domain to eliminate such problems, say by defining the domain of $f$ as $\mathbb{R} = \{ 0 \}$. However, we are often casual about this and in this subject if there are some elements of the domain where the function is undefined we will say that it is a partial function (otherwise it is a total function).

We sometimes want to cut the domain of a function back to a subset. If $f : D \to C$ is a function and $S \subseteq D$ then the restriction is $f \restriction_S$ is the function with domain $S$ and codomain $D$ defined by $f \restriction_S(x) = f(x)$.

**Well-defined** The definition of a function contains the restriction that each domain element maps to one and only one codomain element, $y = f(x)$. We say that functions are well-defined.

When we are considering a relationship between $x$'s and $y$'s and asking if it is a function, this restriction is typically the point at issue. For instance, consider the relation between $x, y \in \mathbb{R}$ where the square of $y$ is $x$. This is not a functional relationship because if $x = 9$ then both $y = 3$ and $y = -3$ are related to $x$. Similarly, if we ask which real number angle $y$ has a cosine of $x = 1/2$ then there

†Sometimes people say that they are, “checking that the function is well-defined.” In a strict sense this is confused, or at least awkward, since if it is a function then it is by definition well-defined. However, all tigers have stripes but we do sometimes say “striped tiger;”; natural language is funny that way.
are many such numbers. Or, if we are setting up a company's email we may decide to use a person's first initial and last name but then realize that there can easily be more than one, say, r.jones. The problem is that the relation connecting the email address to person is not well-defined — it is ill-defined.

If a function is suitable for graphing on xy axes then visual proof of well-definedness is that for any x in the domain, the vertical line at x intercepts the graph in exactly one point.

**One-to-one and onto** The definition of function has an asymmetry; it requires that each domain element be in one and only one pair, but it does not require the same of the codomain elements.

A function f is one-to-one, or 1-1 or an injection, if each codomain element is in at most one pair in f. This function is one-to-one because every element in the bean on the right has at most one arrow ending at it.

![One-to-one example](image)

The most common way to verify that a function is one-to-one is to assume that \( f(x_0) = f(x_1) \) and then deduce 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.

As the above picture suggests, where both the domain and codomain are finite, if the function is one-to-one then the domain has at most as many elements as its codomain.

Note also that the restriction of a one-to-one function is one-to-one.

A function f is onto, or a surjection, if each codomain element is in at least one pair in f. Thus, a function is onto if its codomain equals its range. This function is onto because every element in the bean on the right has at least one arrow ending at it.

![Onto example](image)

The most common way to verify that a function is onto is to start with a codomain element y and then produce a domain element x that maps to it. 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 intersects the graph in at least one point.

As the above picture suggests, where the domain and codomain are finite, if a function is onto then its domain has at least as many elements as its codomain.
Correspondence A function is a correspondence, or a bijection, if it is both one-to-one and onto. The left picture shows a correspondence between two finite sets, both with four elements, and 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. If the 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 exactly one point.

As the above pictures suggests, where the domain and codomain are finite, if a function is a correspondence then its domain has exactly as many elements as its codomain.

Composition and inverse If $f: D \to C$ and $g: C \to B$ then their composition $g \circ f: D \to B$ is given by $g \circ f(d) = g(f(d))$. For instance, the real functions $f(x) = \sin(x)$ and $g(x) = x^2$ have the composition $g \circ f = (\sin(x))^2$.

Composition does not commute. For instance, with the functions from the prior paragraph the composition $f \circ g = \sin(x^2)$ is different. It can fail to commute even more dramatically: if $f: \mathbb{R}^2 \to \mathbb{R}$ is given by $f(x_0, x_1) = x_0$ and $g: \mathbb{R} \to \mathbb{R}$ is given by $g(x) = 3x$ then $g \circ f(x_0, x_1) = 3x_0$ is perfectly sensible but composition in the other order is not defined.

The composition of one-to-one functions is one-to-one, and the composition of onto functions is onto. Of course then, the composition of correspondences is a correspondence.

An identity function $id: D \to D$ has the action $id(d) = d$ for all $d \in D$. It acts as the identity element in function composition, so that if $f: D \to C$ then $f \circ id = f$ and if $g: C \to D$ then $id \circ g = g$. And, if $h: D \to D$ then $h \circ id = id \circ h = h$.

Given $f: D \to C$, 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 it is an inverse (or two-sided inverse) of $f$, denoted $f^{-1}$. 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 $f: \mathbb{R} \to \mathbb{R}$ be $f(x) = 3x + 1$ and $g: \mathbb{R} \to \mathbb{R}$ be $g(x) = x^2 + 1$. (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.
(A) Let \( g: \mathbb{R}^3 \to \mathbb{R}^2 \) be the projection map \((x, y, z) \mapsto (x, y)\) and let \( f: \mathbb{R}^2 \to \mathbb{R}^3 \) be the injection map \((x, y) \mapsto (x, y, 0)\). Then \( g \) is a left inverse of \( f \) but not a right inverse.

(b) The function \( f: \mathbb{Z} \to \mathbb{Z} \) given by \( f(n) = n^2 \) has no left inverse.

(c) Where \( D = \{0, 1, 2, 3\} \) and \( C = \{10, 11\} \), the function \( f: D \to C \) given by \( 0 \mapsto 10, 1 \mapsto 11, 2 \mapsto 10, 3 \mapsto 11 \) has more than one right inverse.

B.3 (a) Where \( f: \mathbb{Z} \to \mathbb{Z} \) is \( f(a) = a + 3 \) and \( g: \mathbb{Z} \to \mathbb{Z} \) is \( g(a) = a - 3 \), show that \( g \) is inverse to \( f \).

(b) Where \( h: \mathbb{Z} \to \mathbb{Z} \) is the function that returns \( n + 1 \) if \( n \) is even and returns \( n - 1 \) if \( n \) is odd, find a function inverse to \( h \).

(c) If \( s: \mathbb{R}^+ \to \mathbb{R}^+ \) is \( s(x) = x^2 \), find its inverse.

B.4 Let \( D = \{0, 1, 2\} \) and \( C = \{10, 11, 12\} \). Also let \( f, g: D \to C \) be \( f(0) = 10, f(1) = 11, f(2) = 12 \), and \( g(0) = 10, g(1) = 10, g(2) = 12 \). Then: (a) verify that \( f \) is a correspondence (b) construct an inverse for \( f \) (c) verify that \( g \) is not a correspondence (d) show that \( g \) has no inverse.

B.5 (A) A composition of one-to-one functions is one-to-one. (B) A composition of onto functions is onto. With the prior item this gives that a composition of correspondences is a correspondence. (C) If \( g \circ f \) is one-to-one then \( f \) is one-to-one. (D) If \( g \circ f \) is onto then \( g \) is onto. (E) If \( g \circ f \) is onto, is \( f \) onto? If it is one-to-one, is \( g \) one-to-one?

B.6 Prove.

(a) A function has an inverse if and only if that function 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 Let \( D \) and \( C \) be finite sets. Prove that if there is a correspondence \( f: D \to C \) then the two have the same number of elements. \textit{Hint:} for each you can do induction either on \(|C| \) or \(|D|\).

(A) If \( f \) is one-to-one then \(|C| \geq |D|\).

(B) If \( f \) is onto then \(|C| \leq |D|\).
Part Five

Notes
Notes

These are citations or discussions that supplement the text body. Each has a word or phrase in italics from that text body, and then the note is in plain text. Many of the citations or discussions include links for you to read more.

Preface

_in addition to technical detail, also aims for a breadth of knowledge_  https://www.aacu.org/leap/what-is-a-liberal-education

_suits the topics_  John McCarthy, the inventor of Scheme's ancestor Lisp, suggested the language as well-suited for the Theory of Computation. Two arguments in that direction are the language's elegance and expressiveness. This is captured an online obituary of McCarthy written by Bertrand Meyer for the Communications of the Association for Computing Machinery: “The Lisp 1.5 manual, published in 1962, was another masterpiece; as early as page 13 it introduces — an unbelievable feat, especially considering that the program takes hardly more than half a page — an interpreter for the language being defined, written in that very language! The more recent reader can only experience here the kind of visceral, poignant and inextinguishable jealously that overwhelms us the first time we realize that we will never be able to attend the première of Don Giovanni at the Estates Theater in Prague on 29 October, 1787 . . . . What may have been the reaction of someone in ‘Data Processing,’ such as it was in 1962, suddenly coming across such a language manual?” (B. Meyer 2011)

Prologue

_D Hilbert and W Ackermann_  Hilbert was a very prominent mathematician, perhaps the world’s most prominent mathematician, and Ackermann was his student. So they made an impression when they wrote, “[This] must be considered the main problem of mathematical logic” (Hilbert and Ackermann 1950), p 73.

_mathematical statement_  Specifically, the statement as discussed by Hilbert and Ackermann comes from a first-order logic (versions of the Entscheidungsproblem for other systems had been proposed by other mathematicians). First-order logic differs from propositional logic, the logic of truth tables, in that it allows variables. Thus for instance if you are studying the natural numbers then you can have a Boolean function 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 ∈ N, Prime(x) implies PerfectSquare(x).” The modifier “first-order” means that the variables used by the Boolean functions are members of the domain of discourse (for Prime above it is 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 codebreaking 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 2016e) is a fun watch but is not a slave to historical accuracy.) After the war, at the National Physical Laboratory he made one of the first designs for a stored-program computer. In 1952, when it was a crime in the UK, Turing was prosecuted for homosexual acts. He was given chemical castration as an alternative to prison. He died in 1954 from cyanide poisoning which an inquest determined was suicide. In 2009, following an Internet campaign, British Prime Minister Gordon Brown made an official public apology on behalf of the British government for “the appalling way he was treated.”

*Olympic marathon* His time at the qualifying event was only ten minutes behind what was later the winning time in the 1948 Olympic marathon. For more, see https://www.turing.org.uk/book/update/part6.html and http://www-groups.dcs.st-and.ac.uk/~history/Extras/Turing_running.html.

clerk Before the engineering of computing machines had advanced enough to make capable machines widely available, much of what we would today do with a program was done by people, then called “computers.” This book’s cover shows human computers at work.

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 then 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 2016c) about nomograms, (navyreviewer 2010) about a naval firing computer, and (Unknown 1948) about a more general-purpose machine.

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 without resource bounds. An example is that the algorithm for long division that we all learned in grade school puts no bounds on either inputs or outputs, or 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.
a complete description of how these machines act Readers new to this subject sometimes ask why we choose a model in which operations are so basic that programming can be an annoyance. 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 models that other authors use, in only a few paragraphs. A real machine takes a full book.

$q$ is a state, a member of $Q$ We are vague about what ‘states’ are but we assume that whatever they are, the sets $\hat{Q}$ and $Q$ are disjoint from the tape alphabet set $\Sigma$ and from the tape action 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.
gloss over the part about interpreting the strings We do this for the same reason that we might 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).

A number of mathematicians See also (Wikipedia 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 Turings 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.” (StackExchange 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 done by a machine with two or more tapes**  For instance, we can simulate a two-tape machine $P_2$ on a one-tape machine $P_1$. One way to do this is by having $P_1$ use its even-numbered tape positions for $P_2$’s first tape and using its odd tape positions for $P_2$’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 2016a)

the unique solution is not computable  See (Pour-El and Richards 1981).
Three-Body Problem  See https://en.wikipedia.org/wiki/Three-body_problem
we can still wonder  See (Piccinini 2017).

This big question remains open  A sample of readings: frequently cited is (Black 2000), which takes the thesis to be about what is humanly computable, and (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.

Plonk!  See (Wikipedia 2015a).

Suppose that you have infinitely many dollars.  (Joel David Hamkins 2010)

H Grassman produced a more elegant definition  Dedekind used this definition to give the first rigorous proof of the laws of elementary school arithmetic.

logically problematic  The sense that there is something perplexing about recursion is often expressed with an anecdote. After a lecture on cosmology the philosopher W James was approached by an older woman from the audience. “Your theory 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. I know better,” she said. “And what is it that you think, madam?” “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, “But what does this second turtle stand on?” The woman crowed, “It’s no use Mr James—it’s turtles all the way down.” (Wikipedia 2016f)

define the function on later values using only earlier 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 $f(5)$.

One elegant thing about Grassmann’s approach  A Perlis’s epigram, “Recursion is the root of computation since it trades description for time” expresses this elegance. The grade school definition of addition is prescriptive in that it gives a procedure. But Grassman’s definition is descriptive in giving the meaning, the semantics, of the operation. 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.

the first sequence of numbers ever computed on an electronic computer  It was computed on EDSAC, on 1949-May-06.
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)

\[ H_3(4, 4) \text{ is much greater than the number of elementary particles in the universe} \]

The radius of the universe if 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 \( H_3(4, 4) \). (Levin 2016)

An operation is primitive recursive if and only if it is bounded  (A. R. 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.

\[ p(\hat{n} + k \cdot \hat{m}) \equiv p(\hat{n}) \mod \hat{m} \]

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. □

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).

Sin \( x \) may be calculated via its Taylor polynomial  The Taylor series is \( \sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \cdots \).

We might do a practical calculation by deciding that a sufficiently good approximation is to terminate that series at the \( x^5 \) term, giving a Taylor polynomial.


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.

Earliest computer crazes  (Bellos 2014)
a rabbit  Discovered by A. Trevorrow in 1986.

For technical convenience  This presentation is based on that of (Hennie 1977), (Smoryński 1991), and (Robinson 1948).

LOOP program  (A. R. 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 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 ?? suggests, 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 2016g) (Huggett 2010), (Bragg 2016), as well as http://www.smbc-comics.com/comic/zeno and this xkcd.

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 $(0 \ldots \infty)$  A modern version of exploiting open-endedness is the Thomson’s Lamp Paradox: a person

![xkcd comic](https://xkcd.com/1973/)

Courtesy xkcd.com
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 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

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 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 \( f_j : \mathbb{N} \to S_j \) To pass from the original collection of infinitely many onto functions \( f_i : \mathbb{N} \to S_i \) 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 the machine will ever get to.

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 2015b).
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.

too many real numbers to fit in them  This is a Pigeonhole Principle argument.

there are jobs that no computer can do  To a person with training in programming, where all of the focus is on getting the computer to do things, the existence of jobs that cannot be done can be a surprise, perhaps even a shock. One thing that it points out is that the topics introduced here are nontrivial, that formalizing the definition of mechanical computation and the results about infinity leads to interesting conclusions.

Your friend 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 Lichterman program the ENIAC, circa 1946. (US Army Photo)

A pattern in technology is for jobs done in hardware to migrate to software  One story that illustrates the naturalness of this involves the English mathematician C Babbage and his protogee A Lovelace. In 1812 Babbage was developing tables of logarithms. These were calculated by computers—people who computed them by hand. To check the accuracy he was comparing two tables. He became annoyed at the number of discrepencies and had the idea to build a machine to compute the numbers. He got a government grant to design and construct a machine called the difference engine. 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).

That machine was never finished, though, because the thought to make it programmable 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 paper cards  It weaves with hooks whose positions, raised or lowered, are determined by holes punched in the cards

a universal machine is like an operating system  Flowcharts are widely used to sketch algorithms; here is one from XKCD.
n consecutive nines  At the 762-nd decimal point there are six nines in a row. This is call the Feynman point; see https://en.wikipedia.org/wiki/Feynman_point. Most experts guess that for any $n$ the decimal expansion contains a sequence of $n$ consecutive nines but no one has proved or disproved that.

there is a difference between defining this function  This is a little like Schrödinger’s cat paradox (see https://en.wikipedia.org/wiki/Schr%C3%B6dinger’s_cat) in that it seems that one of the two is right but we just don’t know which.

π’s $i$-th decimal place  As we have noted, some real numbers have two decimal representations, one ending in 0’s and one ending in 9’s. But every such number is rational (as “ending in 0’s” implies) and π is not rational, so π is not one of these numbers.

partial application  See also (Wikipedia contributors 2019d).

it must be effective  In fact, careful analysis shows that it is primitive recursive.

halt on some inputs but not on others  A Turing machine could fail to halt because it has an infinite loop. The Turing machine $P_0 = \{q_0 B B q_0, q_0 11 q_0\}$ never halts, cycling forever in state $q_0$. We could patch this problem; we could write a program $\inf_{\text{loop}}$ 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 the sense that they never repeat a configuration. One is $P_1 = \{q_0 B 1 q_1, q_1 1 R q_0\}$ which when started on a blank tape will endlessly move to the right, writing 1’s.

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.
A dovetail joint is often used in woodworking, for example to hold together the sides of a drawer. It weaves the two sides in alternately, as shown here.

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)

For a full treatment see (Rogers 1987).

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.

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)

The fable is due to David Hilbert in 1924. It was popularized by George Gamow in One, Two, Three . . . Infinity. (Kragh 2014).

Let the n-th triangle number be $t(n) = 0 + 1 + \cdots + n = n(n+1)/2$. The function $t$ is monotonically increasing and there are infinitely many triangle numbers. Thus for every natural number $c$ there is a unique triangle number $t(n)$ that is maximal so that $c = t(n) + k$ for some $k \in \mathbb{N}$. Because $t(n+1) = t(n) + n + 1$, we see that $k < n + 1$, that is, $k \leq n$. Thus, to compute the diagonal number $d$ from the Cantor number $c$ of a pair, we have $(1/2)d(d+1) \leq c < (1/2)(d+1)(d+2)$. Applying the quadratic formula to the left half and right halves gives $(1/2)(-3 + \sqrt{1 + 8c}) < d \leq (1/2)(-1 + \sqrt{1 + 8c})$. Taking $(1/2)(-1 + \sqrt{1 + 8c})$ to be $\alpha$ gives that $c \in (\alpha - 1 .. \alpha]$ so that $d = \lfloor \alpha \rfloor$. (Scott 2012)

See https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it.

See (Wikipedia 2017d).

See (Wikipedia 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.” Michaelson was not a minor figure who was expressing a minority view. 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 is still well known today for the Michelson–Morley experiment that tried to find the presence of aether, the hypothesized medium through with 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 Michaelson’s statement. Often the significance of things takes time to be apparent

“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 simply 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.

potential 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.”

de the car is less complex than the robot This is an information theoretic version of the Second Law of Thermodynamics

quine Named for the philosopher Willard Van Orman Quine.

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 it to be about itself. Further, if we suppose the statement is true then it asserts that anything he says is false, so the statement is false. But if we suppose 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. See also https://www.smithsonianmag.com/science-nature/fresh-off-the-3d-printer-henry-segermans-mathematical-sculptures-2894574/?no-ist

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).

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)

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 (with the technicality also of taking the hypothesis of the Stationary Ramsey Property).
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 Legos 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 since 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 way of encoding character information in computers. The most common modern character encoding is UTF-8, which is a superset of ASCII.

*palindrome*  Sometimes people tease Psychology by labelling it 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 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 2017e)

*“the red big barn” sounds wrong.*  Experts vary on the exact rules but one source gives (article) + number + judgement/attitude + size, length, height + age + color + origin + material + purpose + (noun), so that “big red barn” is size + color + noun, as is Another example is “little green men.”  This is 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.

*grammars are the language of languages.*  Aphorism from Matt Swift, http://matt.might.net/articles/grammars-bnf-ebnf/.

This grammar  Taken from https://en.wikipedia.org/wiki/Formal_grammar.


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 also illustrates that point because it is, after all, not drawn.

a widely used 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 proposed the name Bacus Naur Form and 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.”

some extensions for grouping and replication The best current standard is https://www.w3.org/TR/xml/.

Time is a complicated engineering problem One example of a complication of time, among many, is leap seconds. The Earth is constantly undergoing a 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. 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 leap seconds cannot be predicted 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

what jobs can be done by a machine with bounded memory 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 a state $q_{\text{error}}$ 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 2017k).

how phone numbers used to be handled in North America See the description of the North America Numbering Plan (Wikipedia 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 2017b)

a wolf, a goat, and a bundle of cabbages This translation is from A Raymond, from the University of Washington.

amb$(S, R_0, R_1 ... R_{n-1})$ 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$(n) = \text{amb}(n - 1, \text{less}(n - 1))$.”

it does not fit our intuition of how computers work “That’s not how this works, that’s not how any of this works” a demon The term ‘demon’ comes from Maxwell’s demon. This is a 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 it out of the chamber, thereby raising the chamber’s temperature without applying 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 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).

This is the general version  This diagram is derived from (Hopcroft, Motwani, and Ullman 2001).

Goldbach’s conjecture  See https://en.wikipedia.org/wiki/Goldbach%27s_conjecture.

a structure is closed under an operation if 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 fact that we can describe these languages in so many completely different ways  (SE author David Richerby 2018).

be aware that another algorithm  See (Knuutila 2001).

\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 has admitted that the two slashes don’t have
a purpose (Firth 2009).

imore 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).

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.

Now they have two problems.  A classic example is trying to use regular expressions to parse significant parts of an
HTML document. See (bobnice 2009).

regex golf  See https://alf.nu/RegexGolf.
Complexity


clever algorithm  The idea is to let \( k = \lceil n/2 \rceil \) and write \( x = x_1 2^k + x_0 \) and \( y = y_1 2^k + y_0 \) (so \( 678 = 21 \cdot 2^5 + 6 \) and \( 42 = 1 \cdot 2^5 + 10 \)). Then we have \( xy = A \cdot 2^{2k} + B \cdot 2^k + C \) where \( A = x_1 y_1 \), and \( B = x_1 y_0 + x_0 y_1 \), and \( C = x_0 y_0 \) (here, \( 28476 = 21 \cdot 2^{10} + 216 \cdot 2^5 + 60 \)). The multiplications by \( 2^{2k} \) and \( 2^k \) are just bit-shifts to known locations independent of the values of \( x \) and \( y \), so they don't affect the time much. But the two multiplications for \( B \) seem to remove all the advantage and still give \( \Theta(n^2) \) time. However, what Karatsuba noted instead was that \( B = (x_0 + x_1) \cdot (y_0 + y_1) - A - C \). Boom: done. Just one multiplication.

This table is adapted from (Garey and Johnson 1979).

There are about \( 3.16 \times 10^7 \) 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 was granted a wish for having invented chess for the 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 that there is in the whole world, truly, for enough to cover the whole surface of the earth to the depth of the twentieth part of a cubit.”

Sissa has the right idea but his arithmetic is slightly off. A cubit is the length of a forearm, from the tip of the middle finger to the bottom of the elbow, so perhaps twenty inches. The geometric series formula gives \( 1 + 2 + 4 + \cdots + 2^{63} = 2^{64} - 1 = 18446744073709551615 \approx 1.84 \times 10^{19} \) 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 Edmunds, separately; see (Cobham 1965) and (Edmunds 1965).
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 \( O(n) \) 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 Edmonds, 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.”

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.

Assuming that the compiler does not optimize it out of the loop.

This discussion originated as (SE author babou and various others 2015).

For a rough idea of that these may be, here are some numbers that every programmer should know.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost in nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache reference</td>
<td>0.5–7</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10 000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20 000 000</td>
</tr>
<tr>
<td>Send packet CA to Netherlands to CA</td>
<td>150 000 000</td>
</tr>
</tbody>
</table>

A nanosecond is $10^{-9}$ seconds. For a fuller description see https://www.youtube.com/watch?v=JEpsKnWZrJ8&app=desktop.

periodically update the standard    Knuth had to update standards, from MIX to MMIX.

a concern involving the definition of Big $O$ and Big $Θ$    This discussion originated as (SE author templatetypedef 2013).


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 cheapest circuit that visits every city    Travelling Salesman was first posed by K Menger, in an article that appeared in the same journal issue as did Gödel's Incompleteness Theorem.

Counties of England and the derived planar graph    This is the map today. The map at the time had some counties that were non-contiguous.

An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities    See 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 by a formula in that form.

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 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, as far as we know today, 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 2017j).

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).)

There is no widely-accepted formal definition of ‘algorithm’ This discussion derives from (Pseudonym 2014).


input two numbers and output their midpoint See https://hal.archives-ouvertes.fr/file/index/docid/576641/filename/computing-midpoint.pdf.

where the number $n$ is represented with $n$-many 1’s An experienced programmer may have the reaction that unary is ridiculous because it is wasteful—if the input is one thousand then this representation would require the machine to take a thousand steps just to read the input, while in binary notation the input is 1111101000, which takes only ten steps to read. But unary is not completely useless; we have found that it suited our purpose in the past, when we simply wanted to illustrate Turing machines with a minimum of overhead detail.
makes the job trivial  Decimal is not much harder since a decimal number is divisible by four if and only if the final two digits are in the set \( \{ 00, 04, \ldots, 96 \} \).

everything of interest can be represented, reasonably efficiently, by bitstrings  See https://rjlipton.wordpress.com/2010/11/07/what-is-a-complexity-class/. Of course, a wag may well 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, 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.

detailed specifications for the programmer  (Grossman 2010)

capture the essential difficulty in the problem  Not every computational problem is easy to express as a decision problem for a language in a way that is natural. One example is the problem which, given an input string will return the string with the characters sorted into ascending order. We could recast it as the language of sorted strings \( \{ \sigma \in \Sigma^* \mid \sigma \text{ is sorted} \} \). But recognizing a sorted input \( \sigma \) does not seem to require that we find a good way to sort an unsorted input. Better is to consider the language of pairs \( \langle \sigma, p \rangle \) where \( p \) is a permutation of the numbers \( 0, \ldots, |\sigma| - 1 \) that brings the string into ascending order. Here also the recast formulation seems not to capture the sorting problem in that recognizing a correct permutation seems different than generating one from scratch. (Besides the informal feeling that they miss the mark, precise evidence that these do not get at the nut of the search problem is that we can decide both languages in time that is \( O(n) \) where \( n \) is the input size, whereas sorting takes \( O(n \lg n) \) time.)

RE  Recall that ‘recursively enumerable’ is an old-fashioned term for ‘computably enumerable’.

many ways to do this  At this writing there are 535 studied ways but the number changes frequently; see https://complexityzoo.uwaterloo.ca/Complexity_Zoo.

the problem of chess

Chess is known to be a solvable game. This is Zermelo’s Theorem (Wikipedia 2017)—there is a strategy for one of the two players that forces a win or a draw, no matter how the opponent plays

In practice authors usually omit it  This is like a working programmer saying “My program inputs a number,” rather than saying “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.”

We are going to put numbers into boxes.  From (Fortnow and Gasarch 2010).

a graph  This is the Petersen graph, a rich source of counterexamples for conjectures in Graph Theory


The next list  These are from the standard, classic, reference (Garey and Johnson 1979).

A large class  See (Richard M. Karp 1972).

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 in (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 is 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 surveyer 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 $P \neq NP$.

including Cook  See (S. Cook 2000).

Many observers  For example, (Viola 2018)

Their algorithm, called RSA  Originally the authors were listed in the standard alphabetic order: Adleman, Rivest, and Shamir. Adleman objected that he had not done enough work to be listed first and insisted on being listed last. He said later, “I remember thinking that this is probably the least interesting paper I will ever write.”

tremendous amount of interest and excitement  In his 1977 column, Martin Gardner posed a $100 challenge, to crack this message: 9686 9613 7546 2206 1477 1409 2225 4355 8829 0575 9991 1245 7431 9874 6951 2093 0816 2982 2514 5708 3569 3147 6622 8839 8962 8013 3919 9055 1829 9451 5781 5254

The ciphertext was generated by the MIT team from a plaintext (English) message using $e = 9007$ and this number $n$ (which is too long to fit on one line).

114, 381, 625, 757, 888, 867, 669, 235, 779, 976, 146, 612, 010, 218, 296, 721, 242, 362, 562, 561, 842, 935, 706, 935, 245, 733, 897, 830, 597, 123, 563, 958, 705, 058, 989, 075, 147, 599, 290, 026, 879, 543, 541

In 1994, a team of about 600 volunteers announced that they had factored $n$.

$p = 3, 490, 529, 510, 847, 650, 949, 147, 849, 619, 903, 898, 133, 417, 764,$

638, 493, 387, 843, 990, 820, 577

and

$q = 32, 769, 132, 993, 266, 709, 549, 961, 988, 190, 834, 461, 413, 177, 642, 967,$

992, 942, 539, 798, 288, 533

That enabled them to decrypt the message: the magic words are squeamish ossifage.

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 sure. 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 about $2.7 \times 10^{14}$ primes but only $246 \, 683 \approx 2.4 \times 10^5$-many Carmichael numbers.

the minimal pub crawl  See (W. Cook et al. 2017)

Appendix

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)
Bibliography


Euler, L (1766). “Solution d’une question curieuse que ne paroit soumise a aucune analyse (Solution of a curious question which does not seem to have been subjected to any analysis)”. In: *Mémoires de l’Academie Royale des Sciences et Belles Lettres, Année 1759* 15. [Online; accessed 2017-Sep-23, article 309], pp. 310–337. [URL: http://eulerarchive.maa.org/].


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.


Index

15 Game problem, 263
3 Dimensional Matching problem, 291
3-SAT, 259, 291
3-Satisfiability problem, 259, 291

accept an input, 172
acceptable numbering, 73
accepting state, 13, 168
  nondeterministic Finite State machine, 180
  Pushdown machine, 217
accepts, 181
Ackermann function, 30–33, 35, 49–52
Ackermann, W, 3
  picture, 32
action set, 8
action symbol, 8
addition, 6
adjacency matrix, 153
adjacent, 152
Adleman, L
  picture, 306
Agrawal, M
  picture, 263
AKS primality test, 263
algorithm, 268
  definition, 268
  reliance on model, 268
alphabet, 137, 314
  input, 168
  Pushdown machine, 217
tape, 8
amb function, 338
ambiguous grammar, 146
argument, to a function, 316
Aristotle's Paradox, 60, 62

Backus, J
  picture, 159
Backus-Naur form, BNF, 159
behavior, functions, 97
Berra, Y
  picture, 178
big $O$, 239–254
big $O$, $O$, 243

big theta, $\Theta$, 245
bijection, 318
bit string, 314
bitstring, 314
blank, 8
blank, B, 5
BNF, 159–164
body of a production, 142
Boole, G
  picture, 258
boolean, 258
  formula, 258
  function, 258
  variable, 258
bottom, $\bot$, 217
BPP, Bounded-Error Probabilistic Polynomial Time problem, 305
Broadcast problem, 261
Busy Beaver, 130–133
button
  start, 5
c.e. set, 101
caching, 71
Cantor's correspondence, 68–75
Cantor's Theorem, 77
Cantor, G
  and diagonalization, 330
  picture, 62
cardinality, 60–68
  less than or equal to, 77
Chromatic Number problem, 258
Church's Thesis, 14–21
  and uncountability, 79
  argument by, 19
  clarity, 17
  consistency, 16
  convergence, 15
  coverage, 15
  Extended, 275
Church, A
  picture, 14
circuit, 152
  Euler, 152
  Hamiltonian, 152
class, 138, 273
Clique problem, 260, 291
clique in a graph, 260
closed under an operation, 201
closed walk, 152
CNF, Conjunctive Normal Form, 258
Cobham’s Thesis, 249
Cobham, A
picture, 341
codomain, 316
codomain versus range, 317
coloring of a graph, 154
complete
for a class, 290
NP, 290
complexity class, 273
complexity function, 243
Complexity Zoo, 305
Composite problem, 263
composition, 319
computable
relative to a set, 106
set, 101
computable function, 11
computable relation, 11
computable set, 11
computably enumerable, 101–104
computably enumerable set, 101
collection of, RE, 273
computation
distributed, 268
Finite State machine, 172
parallel, 268
relative to an oracle, 105
step, 8
Turing machine, 9
concatenation of languages, 139
concatenation of strings, 314
configuration, 8, 171
halting, 172
initial, 8
conjunctive normal form, 42, 258
connected graph, 152
context free
grammar, 143
language, 223
control of Turing machine, 5
converge, 10
Conway, J
picture, 45
Cook, S
picture, 289
Cook-Levin theorem, 290
correspondence, 61, 318
Cantor’s, 70
countable, 63
countably infinite, 63
Course Scheduling problem, 266
CPU of Turing machine, 5
Crossword problem, 262
current symbol, 8
CW, 156
cycle, 152
dangling else, 147
De Morgan, A
picture, 257
decidable
language, 100
set, 101
decider, 11
decision problem, 3, 270
decrypter, 307
degree of a vertex, 155
degree sequence, 155
demon, or daemon, 180
derivation, 142–144
derivation tree, 143
description number, 73
determinism, 8, 16
diagonal enumeration, 70
diagonalization, 75–80, 110
effectivized, 90
digraph, 152
directed graph, 152
disjunctive normal form, DNF, 42
distinguishable states, 210
distributed computation, 268
diverge, 10
Divisor problem, 263
domain, 316, 317
Double-SAT problem, 284
doubler function, 3, 12
dovetailing, 101
Droste effect, 335
DSPACE, 304
DTIME, 303

edge, 152
edge weight, 152
Edmunds, J
picture, 341
effective, 3
effective function, 9
empty string, $\varepsilon$, 8, 314
encrypter, 307
Entscheidungsproblem, 3, 14, 270, 297
enumerate, 63
$\varepsilon$ moves, 182
$\varepsilon$ transitions, 182–184
equinumerous sets, 62
equivalent growth rates, 245
Euler Circuit problem, 257
Euler circuit, 152
Euler, L
picture, 256
eval, 83
EXP, 295
expansion of a production, 142
Extended Church’s Thesis, 275
extended regular expression, 225
extended transition function, 173
nondeterministic Finite State machine, 181

Factoring problem, 306
Fibonacci numbers, 29
final state, 168
finite set, 63
Finite State automata, 168
Finite State machine, 167–177
accept string, 172
accepting state, 168
alphabet, 168
computation, 172
configuration, 171
final state, 168
halting configuration, 172
initial configuration, 171
input string, 171
minimization, 209–216
next-state function, 168
nondeterministic, 168
reject string, 180
state, 168
step, 171
transition function, 168

Flauros
picture, 180
flow chart, 82
Four Color problem, 257
function, 316
91 (McCarthy), 29
Ackermann, 49
argument, 316
boolean, 258
codomain, 316
composition, 319
computable, 11
computed by a Turing machine, 9
converge, 10
correspondence, 61, 318
definition, 316
diverge, 10
domain, 316
doubler, 3, 12
effective, 3
enumeration, 63
exponential growth, 247
extended transition, 173
general recursive, 34
identity, 319
image under, 317
index, 316
injection, 318
inverse, 319
left inverse, 319
logarithmic growth, 247
$\mu$ recursive function (mu recursive), 34
next-state, 8, 168
one-to-one, 61
one-to-one, 318
onto, 61, 318
output, 316
pairing, 69, 70
partial, 10, 317
partial recursive, 34
polynomial growth, 247
predecessor, 6
projection, 24, 34
range, 317
recursive, 11, 34
restriction, 317
right inverse, 319
successor, 12, 21, 24
successor, 34
surjection, 318
total, 10, 317
transition, 8, 168
unpairing, 69, 70
value, 316
well-defined, 317
zero, 24, 34
function problem, 269
functions, 316–320
same behavior, 97

Galelio's Paradox, 62, 64
Galilei, G (Galileo)
picture, 60
Galileo, 60
Galileo's Paradox, 60
Game of Life, 45–48
gates, 42
general recursion, 30–36
general recursive function, 34
Gödel number, 73
Gödel, K, 14
letter to von Neumann, 298
picture, 15
picture with Einstein, 126
Gödel's theorem, 14
grammar, 141–151
ambiguous, 146
Backus-Naur form, BNF, 159
BNF, Backus-Naur form, 159
body of a production, 142
context free, 143
derivation, 142
expansion of a production, 142
head, 142
linear, 190
nonterminal, 142
production, 142, 143
rewrite rule, 142, 143
right linear, 190
start symbol, 143
syntactic category, 143
terminal, 142
graph, 151–159
adjacent edges, 152
circuit, 152
clique, 260
closed walk, 152
coloring, 154
connected, 152
cycle, 152
degree sequence, 155
digraph, 152
directed, 152
directed, 152
directed, 152
edge, 152
edge weight, 152
Euler circuit, 152
Hamiltonian circuit, 152
induced subgraph, 153
isomorphism, 154–155
loop, 152
matrix representation, 153
multigraph, 152
node, 152
open walk, 152
path, 152
representation, 153–154
simple, 152
spanning subgraph, 260
subgraph, 153
trail, 152
transition, 7
traversal, 152–153
tree, 157, 260
vertex, 152
vertex cover, 260
vertex degree, 155
walk, 152
walk length, 152
weighted, 152

Graph Colorability problem, 258
Grassman, H, 21
picture, 21
guessing by a machine, 180, 182
halt light, 5
halting configuration, 172
Halting problem, 90–91
discussion, 91–92
in wider culture, 124–127
significance, 93
unsolvability, 91
halting state, 12
Hamilton, W R
picture, 255
Hamiltonian Circuit problem, 255, 291
Hamiltonian circuit, 152
hard
for a class, 290
NP, 290
head
I/O, see read/write head
read/write, 4
head of a production, 142
Hilbert's Hotel, 116–117
Hilbert, D, 3
picture, 125
Hofstadter, D, 129
hyperoperation, 31
I/O head, see read/write head
identity
function, 319
Ignorabimus, 125
image under a function, 317
Incompleteness theorem, 14
Independent Set problem, 267
index number, 73
index set, 97
induced subgraph, 153
infinite set, 63
infinity, 60–68
initial configuration, 8, 171
injection, 318
input alphabet, 168
input string, 171
input symbol, 8
input, to a function, 316
instruction, 5, 8
inverse of a function, 319
left, 319
right, 319
two-sided, 319
isomorphic, 155
isomorphism, 155
K, the Halting problem set, 90
K, the halting problem set, 102
K₀, set of halting pairs, 107
Karatsuba, A, 240
Karp reducible, 286
Karp, R
picture, 291
Kayal, N
picture, 263
Kleene star, 63, 137, 139, 193, 314
Kleene's fixed point theorem, 111
Kleene's theorem, 194–198
Kleene, S, 34
picture, 191
Knapsack problem, 262
Knights Tour problem, 255
Knuth, D
picture, 250
Kolmogorov, A
picture, 240
Königsberg, 256
L'Hôpital's Rule, 246
lambda calculus, λ calculus, 14
language, 137–141
class, 138
concatenation, 139
context free, 223
decidable, 100
decided by a Turing machine, 13
decision problem, 270
derived from a grammar, 145
grammar, 143
Kleene star, 139
of a nondeterministic Finite State machine, 181
of a nondeterministic Turing machine, 279
operations on, 139
power, 139
regular, 200–204
reversal, 139
language decision problem, 270, 273
language recognition problem, 270
languages
  non-regular, 204–209
last in, first out (LIFO) stack, 217
left inverse, 319
leftmost derivation, 143
LEGO, 5
length, 152
length of a string, 314
Life, Game of, 45–48
LIFO stack, 217
light
  Halt, 5
Lipton's Thesis, 269
loop, 152
LOOP program, 52–57
machine
  state, 9
map, see function
matching, three dimensional, 261
McCarthy's 91 function, 29
memoization, 71
memory, 4
metacharacter, 142, 159, 192
Minimum Spanning Tree problem, 260
minimization of a Finite State machine, 209–216
minimization, unbounded, 33, 34
Minimum Spanning Tree problem, 260
modulus, 307
Morse code, 156
\(\mu\)-recursion (mu recursion), 33
\(\mu\) recursive function, 34
multigraph, 152
multiset, 262
Musical Chairs, 77

n-distinguishable states, 210
n-indistinguishable states, 210
Naur, P
  picture, 159
next state, 5, 8
next tape character, 5
next-state function, 8, 168
  nondeterministic Finite State machine, 180
NFSM, see nondeterministic Finite State machine
node, 152
nondeterminism, 177–191
  for Finite State machines, 180
  for Turing machines, 277
nondeterministic Finite State machine, 180
  accepts string, 181
  convert to a deterministic machine, 185
  language of, 181
nondeterministic Finite State machines
  \(\epsilon\) moves, 182
  \(\epsilon\) transitions, 182
nondeterministic Pushdown machine, 220–223
nondeterministic Turing machine
  language, 279
nonterminal, 142, 143
NP, 277–285
NP complete, 289–295
  basic problems, 291
NP hard, 290
NSPACE, 304
NTIME, 303
numbering, 73
  acceptable, 73
one-to-one function, 61, 318
onto function, 61, 318
open walk, 152
optimization problem, 270
oracle, 105–110
oracle Turing machine, 106
orders of growth, 248
ouroboros, 82
output, from a function, 316
P, 273–277
P versus NP, 280, 296–300
pairing function, 69, 70
Paley, W
  picture, 127
palindrome, 13, 138, 221, 315
paradox
  Aristotle's, 60
  Galileo's, 60
  Zeno's, 64
Parameter theorem, 85
parametrization, 85–87
parse tree, 143
partial function, 10, 317
partial recursive function, 34
Partition problem, 262, 291
path, 152
perfect number, 93
 Péter, R
picture, 49
pipe symbol, |, 142
pointer, in C, 113
polynomial time reducibility, 286
polytime, 274
power of a language, 139
power of a string, 315
predecessor function, 6
prefix of a string, 315
present state, 5, 8
present tape character, 5
Prime Factorization problem, 263
primitive recursion, 34
Primitive recursion, 21
primitive recursion, 23
arity, 23
primitive recursive functions, 24
private key, 307
problem, 268
decision, 270
function, 269
Halting, 90–91
language decision, 270
language recognition, 270
optimization, 270
search, 270
unsolvable, 91
problem miscellany, 255–267
problem reduction, 285–302
problems
tractable, 249
unsolvable, 90–100
production, 143
production in a grammar, 142
program, 268
projection function, 34
projection functions, 24
public key, 307
Pumping lemma, 204
pumping length, 204
Pushdown automata, 216–225
Pushdown machine, 216–225
halting, 218
input alphabet, 217
nondeterministic, 220–223
stack alphabet, 217
transition function, 217
Quantum Computing, 275
Quantum Supremacy, 275
quine, 128
Quine’s paradox, 335
r.e. set, 101
Radó, T
picture, 131
RAM, 249
Random Access machine, 249
range of a function, 317
RE, computably enumerable sets, 273
read/write head, 4
recursion, 21–36
Recursion theorem, 111
recursive function, 11, 34
recursive set, 11
recursively enumerable set, 101
reduces to, 106
reducibility
polynomial time, 286
Reflections on Trusting Trust, 129
regex, 225
regular expression, 191–200
extended, 225
operator precedence, 192
regex, 225
semantics, 193
syntax, 192
regular expressions
in practice, 225–232
regular language, 200–204
reject an input, 172
rejecting state, 13
relation, computable, 11
replication of a string, 315
representation, of a problem, 269
set of, 8
Stator Square, 342
STCON problem, see Vertex-to-Vertex Path problem
step, 8, 171
store, of a Turing machine, 4
string, 137, 314–315
  concatenation, 314
  decomposition, 315
  empty, 8, 314
  length, 314
  power, 315
  prefix, 315
  replication, 315
  reversal, 315
  substring, 315
  suffix, 315
subgraph, 153
  induced, 153
Subset Sum problem, 262
substring, 315
successor function, 12, 21, 24, 34
suffix of a string, 315
surjection, 318
symbol, 8, 137
  action, 8
  current, 8
  input, 8
  next, 8
syntactic category, 143

T equivalent, 107
T reducible, 106
table, transition, 7
tape, 4
tape alphabet, 8
tape character
  blank, 5
tape symbol, 8
terminal, 142, 143
tetration, 31
Thompson, K
  picture, 129
Three-dimensional Matching problem, 261
time taken by a machine, 249
token, 137
total function, 10, 317

Towers of Hanoi, 25
tractable, 247–249
trail, 152
transition function, 8, 168
  extended, 173
  graph of, 7
  Pushdown machine, 217
table of, 7
transition graph, 7
transition table, 7
Travelling Salesman problem, 178, 256
tree, 157, 260
Triangle problem, 277
triangular numbers, 25
truth table, 41, 258
Turing degree, 107
Turing equivalent, 107
Turing machine, 3–14
  accepting state, 13
  action set, 8
  action symbol, 8
  computation, 9
  configuration, 8
  control, 5
  CPU, 5
  current symbol, 8
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 decided, 13
multitape, 20
next character, 5
next state, 5, 8
next symbol, 8
next-state function, 8
nondeterminism, 277
numbering, 73
palindrome, 13
present character, 5
present state, 5, 8
rejecting state, 13
simulator, 36–40
tape alphabet, 8
transition function, 8
universal, 81–83, 89
with oracle, 106
Turing reducible, 106, 286
Turing, A
  picture, 3
two-sided inverse, 319
unbounded minimization, 33
unbounded search, 33
uncountable, 77
undecidable, 91
Unicode, 170, 338
uniformity, 84–85
Universal Turing machine, 81–83
universality, 81–89
unpairing function, 69, 70
unsolvability, 90–96, 100
unsolvable problem, 90–100
use-mention distinction, 113
value, of a function, 316
vertex, 152
**Vertex Cover** problem, 260, 267
**Vertex cover** problem, 291
vertex cover, 260
**Vertex-to-Vertex Path** problem, 259
von Neumann, J
  picture, 45
walk, 152
walk length, 152
weight, 152
weighted graph, 152
well-defined, 317
word, see string
working state, 12

Zeno’s Paradox, 64
zero function, 24, 34
Zoo, complexity, 305