# Logic and Computation

September 20, 2022

# Contents


**1 Propositional Logic** **9**

1.1 You already know what propositional logic is, even if you didn't know what it was called . . . 9

1.2 Propositional Formulas-Syntax . . . 10

1.2.1 Definition . . . 10

1.2.2 Examples . . . 11

1.2.3 Recursive Nature of the Definition. . . . 12

1.2.4 The Definition as a Grammar . . . 12

1.2.5 Avoiding parentheses . . . 12

1.3 Propositional Formulas-Semantics . . . 13

1.3.1 The value of a propositional formula. . . . 13

1.3.2 Example: Modeling ordinary language by propositional formulas. . . . 14

1.3.3 Truth tables . . . 15

1.3.4 Satisfiability and tautology . . . 16

1.4 New connectives: conditional, biconditional, exclusive or . . . 16

1.4.1 Conditional . . . 17

1.4.2 Biconditional . . . 17

1.4.3 Exclusive-or . . . 18

1.5 Equivalent formulas . . . 18

1.5.1 Identities . . . 19

1.5.2 DeMorgan's Laws and Duality . . . 20

1.5.3 Did we just prove something? . . . 23

1.5.4 Normal Forms . . . 24

1.6 Complete Sets of Connectives . . . 26

1.7 Historical Notes . . . 27

1.8 Exercises . . . 28

1.8.1 Syntax of propositional logic . . . 28

1.8.2 Semantics of propositional logic . . . 30

1.8.3 Equivalence . . . 34

1.8.4 Complete sets of connectives . . . 38

**2 Applications of Propositional Logic** **39**
2.1 Digital Logic . . . 39
2.1.1 Computation as Bit-manipulation . . . 39
2.1.2 Truth tables realized by networks of logic gates . . . 41
2.2 Liar Puzzles . . . 42
2.2.1 Some Knights-and-Knaves Puzzles . . . 43
2.2.2 A Vampire Puzzle . . . 44
2.2.3 Modeling the puzzles with propositional logic . . . 44
2.3 Satisfiability Problems . . . 47
2.3.1 A Scheduling Problem . . . 47
2.3.2 Sudoku: A Combinatorial Design Problem . . . 47
2.3.3 Modeling the examples as Satisfiability Problems . . . 48
2.3.4 Satisfiability Solvers . . . 50
2.3.5 *How do SAT solvers work? . . . 52
2.4 Historical and Bibliographic Notes . . . 53
2.5 Exercises . . . 54
2.5.1 Digital Logic . . . 54
2.5.2 Puzzles . . . 55
2.5.3 Satisfiability Solvers . . . 57

**3 Sets and Functions** **61**
3.1 Sets . . . 61
3.1.1 Notation for sets . . . 61
3.1.2 Subset . . . 62
3.1.3 New sets from old . . . 63
3.1.4 An application: defining complex structures . . . 64
3.1.5 Sets in Python . . . 65
3.1.6 Set Identities . . . 65
3.1.7 Venn Diagrams . . . 66
3.1.8 Extended Operations . . . 67
3.2 Functions . . . 70
3.2.1 Basic definitions . . . 70
3.2.2 One-to-one functions . . . 72
3.2.3 Onto functions . . . 72
3.2.4 Composition of functions . . . 74
3.2.5 Identity and inverse . . . 76
3.2.6 Extended function notation . . . 77
3.3 Counting . . . 77
3.3.1 Addition and multiplication principles . . . 78
3.3.2 Behavior of cardinality under functions . . . 78
3.3.3 Power set and sets of functions . . . 78
3.3.4 Permutations and the factorial function . . . 80
3.3.5 Binomial coefficients . . . 81

3.4 Infinite Sets . . . 84
3.4.1 Countable sets . . . 85
3.4.2 Uncountable sets . . . 88
3.4.3 Paradox! . . . 90
3.4.4 *Unsolvable problems . . . 90
3.5 Historical Notes . . . 93
3.6 Exercises . . . 93
3.6.1 Sets . . . 93
3.6.2 Functions . . . 96
3.6.3 Counting . . . 98
3.6.4 Infinite Sets . . . 99

**4 Predicate Logic and Relations** **101**
4.1 Predicate Logic . . . 101
4.1.1 What we talk about when we talk about love . . . 101
4.1.2 A Database Example: The Language of Kinship . . . 105
4.1.3 The syntax of predicate logic . . . 108
4.1.4 The semantics of predicate logic . . . 110
4.1.5 Logically valid and equivalent formulas . . . 111
4.1.6 Predicate logic with function symbols, and the language of arithmetic 113
4.2 Classification of binary relations . . . 115
4.2.1 Equivalence Relations . . . 115
4.2.2 Orders . . . 116
4.3 Historical Notes . . . 118
4.4 Exercises . . . 119
4.4.1 Translating between natural language and predicate logic . . . 119
4.4.2 A database language . . . 120
4.4.3 The language of arithmetic . . . 121
4.4.4 Predicate Logic in general . . . 122
4.4.5 Equivalence relations and orders . . . 125

**5 Proofs** **127**
5.1 Logic and Proofs . . . 127
5.2 Example: A property of even numbers . . . 130
5.3 *A closer look . . . 131
5.4 Example: An inequality, and working backwards . . . 133
5.5 Example: Another inequality, and proof by cases . . . 135
5.6 Counterexamples . . . 137
5.7 Example: Yet another inequality, and proofs by contradiction . . . 137
5.8 Example: More on even and odd integers, 'if and only if', and a famous proof. 138
5.9 Historical Notes . . . 141
5.10 Exercises . . . 141

**6 Mathematical Induction and Recursion** **145**
6.1 A Sampler of Induction Problems . . . 145
6.1.1 A summation identity . . . 145
6.1.2 Exponential versus polynomial growth . . . 148
6.1.3 Behind blue eyes . . . 148
6.1.4 An inductive proof of the summation identity . . . 148
6.1.5 An inductive solution to the inequality problem . . . 149
6.1.6 Solution to the puzzle of the blue-eyed villagers. . . . 150
6.2 Ordinary Induction . . . 151
6.2.1 Principle of Mathematical Induction . . . 151
6.2.2 Some proofs by mathematical induction. . . . 151
6.2.3 'Aren't You Assuming What You're Trying to Prove?' . . . 153
6.2.4 Cardinality of the Power Set of an $n$-element Set . . . 154
6.3 Exponential, Polynomial, and Logarithmic Growth . . . 155
6.3.1 Exponential versus polynomial growth . . . 155
6.3.2 Archimedean Principle . . . 157
6.3.3 Efficient algorithms versus inefficient algorithms . . . 157
6.3.4 Polynomial versus logarithmic growth . . . 158
6.4 Strong Induction . . . 161
6.4.1 The Unstacking Game . . . 161
6.4.2 The Principle of Strong Induction and the Least Integer Principle . . . 162
6.4.3 An application of strong induction: $\sqrt{2}$ revisited. . . . 163
6.5 Recursively Defined Sequences, Sets and Algorithms . . . 163
6.5.1 Fibonacci Numbers . . . 164
6.5.2 Recursive definitions of sets and structures, and proofs by structural induction . . . 165
6.5.3 Dual formulas . . . 166
6.5.4 Recursive Algorithms . . . 168
6.6 Historical Notes . . . 170
6.7 Exercises . . . 171
6.7.1 Basic identities . . . 171
6.7.2 Growth rate of functions . . . 173
6.7.3 Strong induction and the least integer principle . . . 175
6.7.4 Structural induction. . . . 176
6.7.5 Recursive algorithms . . . 177

**7 Basic Number Theory** **179**
7.1 Divisibility . . . 179
7.1.1 Quotient and Remainder . . . 179
7.1.2 The 'divides' relation . . . 181
7.2 Positional Number Systems . . . 182
7.2.1 Base conversion . . . 184
7.3 Common factors and Euclid's Algorithm . . . 186

7.3.1 A puzzle . . . 186
7.3.2 Euclid's Algorithm for Computing the GCD . . . 186
7.3.3 Extended Euclid's Algorithm . . . 188
7.3.4 Speed of Euclid's Algorithm . . . 189
7.4 Prime Numbers . . . 190
7.4.1 Prime factorization . . . 190
7.5 Congruence . . . 193
7.5.1 Definition and basic properties . . . 193
7.5.2 Fast Modular Exponentiation . . . 196
7.5.3 Fermat's Theorem and Primality Testing . . . 197
7.5.4 Card shuffling . . . 198
7.6 Public-Key Cryptography . . . 200
7.6.1 Symmetric and asymmetric cryptography . . . 200
7.6.2 Easy and hard problems . . . 202
7.6.3 The RSA algorithm . . . 202
7.7 Historical Notes . . . 205
7.8 Exercises . . . 206
7.8.1 Quotient and remainder . . . 206
7.8.2 Positional number systems . . . 207
7.8.3 Exotic number systems . . . 208
7.8.4 Euclid's algorithm . . . 210
7.8.5 Primes . . . 211
7.8.6 Congruences . . . 212
7.8.7 Public-key cryptography . . . 213

**8 Finite State Machines** **215**
8.1 Finite Automata and Regular Expressions . . . 215
8.1.1 Finding patterns in text . . . 215
8.1.2 Regular expressions defined . . . 218
8.1.3 Deterministic finite automata . . . 222
8.1.4 From regular expressions to automata . . . 225
8.1.5 From automata to regular expressions . . . 227
8.1.6 Closure under boolean operations . . . 233
8.1.7 A non-regular language . . . 234
8.2 Sequential circuits . . . 234
8.3 Exercises . . . 234
8.3.1 Regular expressions . . . 234
8.3.2 Deterministic finite automata . . . 235
8.3.3 Nondeterministic automata and operations on languages . . . 236
8.3.4 From automata to regular expressions . . . 237

**9 Turing Machines, Computability, and Undecidability** **239**
9.1 The Turing Machine . . . 239
9.1.1 An example . . . 240
9.1.2 Formal definition . . . 241
9.1.3 Operation and acceptance Behavior . . . 241
9.1.4 Variants of the Turing machine model . . . 243
9.2 Recursively enumerable and recursive languages . . . 245
9.2.1 The Church-Turing Thesis . . . 247
9.3 Universal Turing Machine . . . 248
9.3.1 Encoding Turing machines by strings . . . 248
9.3.2 Turing machines as inputs to Turing machines; a universal machine . 249
9.4 An Undecidable Problem . . . 250
9.5 The halting problem and other undecidable problems about computer programs 251
9.5.1 Reductions . . . 251
9.5.2 Halting problems . . . 251
9.6 *An Undecidable Problem from Logic . . . 252
9.7 Historical Notes . . . 255
9.8 Exercises . . . 255
9.8.1 Low-level problems on Turing machines . . . 255
9.8.2 High-level problems on Turing machines . . . 256
9.8.3 Undecidable problems . . . 256

# Chapter 1

# Propositional Logic

> *When you come to any passage you don't understand, read it again: if you still don't understand it, read it again: if you fail, even after three readings, very likely your brain is getting a little tired. In that case, put the book away, and take to other occupations, and next day, when you come to it fresh, you will very likely find that it is quite easy.*
>
> —Lewis Carroll, *Symbolic Logic,* 1896.

## 1.1 You already know what propositional logic is, even if you didn't know what it was called

Much of the power of computer programs stems from their ability to perform conditional execution—that is, to change the order in which statements are executed depending on the current values of program variables. For example, Figure 1.1 shows a Python function that returns `True` or `False` depending on whether or not its argument is a leap year. This is not so simple a matter as checking if the year is divisible by 4; the correct criterion is more complicated: Century years (like 1700 and 1900) are not leap years, unless they are divisible by 400 (like 1600 and 2000).

The code in Figure 1.2 performs the same calculation much more compactly, by returning the value of an expression of *boolean* type (called `bool` in Python). Python, like other programming languages, contains a kind of mini-language for creating complex boolean expressions by combining simpler expressions with the operators `and`, `or` and `not`. The expression building begins with 'atomic' boolean expressions like `year%400==0`, so-called because they cannot be broken down into smaller boolean expressions.

The same mini-language, often with different syntax rules, is present in some form in every programming language, in database query languages, and even within the formula language for Excel spreadsheets ( Figure 1.3).

This little language is called *propositional logic.*

```
def leapyear(y):
    if y%4==0:
        if y%100==0:
            if y%400==0:
                return True
            else:
                return False
        else:
                return True
    else:
        return False;
```

Figure 1.1: *A logical calculation in Python, determining if year* `y` *is a leap year*

```
def leapyear(y):
    return (y%4==0) and (not (y%100==0) or (y%400==0))
```

Figure 1.2: *The same calculation, using a complex boolean expression.*

## 1.2 Propositional Formulas-Syntax

We will shortly give a precise definition of propositional logic, but here, in a nutshell, is the idea: Ordinary algebraic expressions, like

$$4x^2y + 3x - 2$$

are built from variables (in this case $x$ and $y$), constants (2, 3 and 4), and operation symbols (for addition, subtraction and multiplication). When you substitute numbers for the variables (say 1 for $x$ and 2 for $y$) you get a value, which is itself a number (in this example, 3). The formulas of propositional logic are constructed and evaluated in much the same way, however the values are *logical* (**true** and **false**) rather than numerical, and the operation symbols are different. The ability to represent and manipulate logical processes using algebra is the essence of propositional logic.

As you read the definitions, keep these examples in mind, especially the one from Python: We use different symbols ($\wedge$, $\vee$ and $\neg$ in place of `and`, `or` and `not`), but the structure and meaning of formulas of propositional logic are pretty much identical to what you find in Python and other computer languages.

### 1.2.1 Definition

We first define the *syntax* of propositional logic, describing how formulas are built. The next section is devoted to *semantics*–what formulas *mean*.

A *propositional formula* is a string of symbols having one of the following forms:

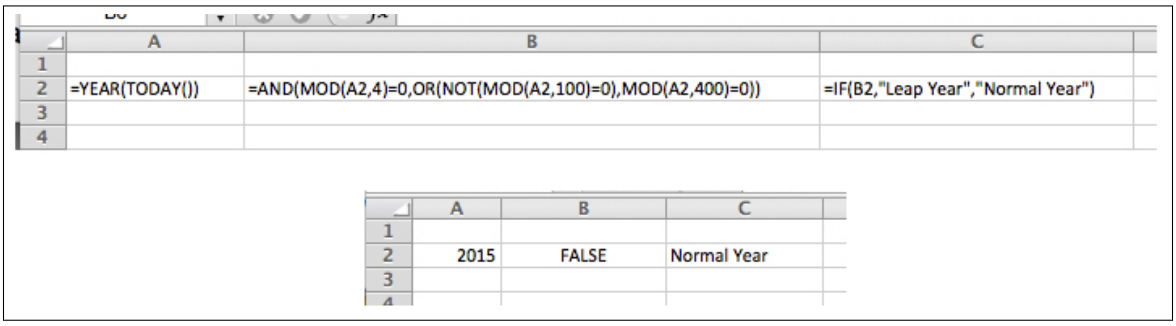



Figure 1.3: *The same calculation in an Excel spreadsheet. The top view displays the formulas, the bottom is the standard view*

- **T** or **F**
- $p, q, r, p_1, p_2, q_1$, *etc.* (These symbols are called *variables*.)
- $(\phi \wedge \psi)$ where $\phi$ and $\psi$ are propositional formulas.
- $(\phi \vee \psi)$ where $\phi$ and $\psi$ are propositional formulas.
- $\neg\phi$ where $\phi$ is a propositional formula.

### 1.2.2 Examples

- Here is a propositional formula

  $$\mathbf{T}$$

- And here is another:

  $$(\mathbf{T} \vee (\neg p \vee \neg(q \wedge p))).$$

  To see why, observe that the first two rules tell us that $\mathbf{T}, p$, and $q$ are formulas, the third rule then says that $(q \wedge p)$ is a formula, the fifth rule that $\neg(q \wedge p)$ and $\neg p$ are formulas, and so on.

- According to this definition, neither of the strings

  $$\neg p \vee (\neg q \wedge p), ((\neg p) \vee (\neg q \wedge p))$$

  is a formula, the first because it lacks a pair of parentheses bracketing the entire formula, and the second because it places a pair of parentheses around $\neg p$. But we will shortly relax our strict requirements and allow them both as acceptable alternatives for $(\neg p \vee (\neg q \wedge p))$. (In this connection, see 1.2.5 as well as the exercises at the end of the chapter, especially Exercises 3 and 7.)

### 1.2.3 Recursive Nature of the Definition.

The definition at first seems to be circular, since it uses 'propositional formula' in the definition itself. But this circularity is only apparent. The definition is actually *recursive,* much like the functions using recursion that you might have studied in an introductory programming course: It defines propositional formulas in terms of *smaller* propositional formulas, and the recursion bottoms out at the formulas that are not combinations of smaller formulas, *i.e.,* $\mathbf{T}$, $\mathbf{F}$, $p, q$, *etc.* These are the *atomic formulas.* The atomic formulas $\mathbf{T}$ and $\mathbf{F}$ are *constants,* and the others are *variables.* We will say more about recursive definitions in Chapter 4.

### 1.2.4 The Definition as a Grammar

Syntax rules like the ones we gave above are often written in a more schematic form:

$$\begin{aligned}
\text{formula} &\longrightarrow \mathbf{T}|\mathbf{F} \\
\text{formula} &\longrightarrow p|q|r\cdots \\
\text{formula} &\longrightarrow (\text{formula} \wedge \text{formula}) \\
\text{formula} &\longrightarrow (\text{formula} \vee \text{formula}) \\
\text{formula} &\longrightarrow \neg\text{formula}
\end{aligned}$$

A set of rules written in this way is called a *context-free grammar.* The grammar provides a compact scheme for showing the *derivation* of a formula—how it is built from the atoms. Here is a derivation of the formula $(r \vee (\neg p \vee \neg(q \wedge p)))$

$$\begin{aligned}
\text{formula} &\longrightarrow (\text{formula} \vee \text{formula}) \\
&\longrightarrow (r \vee \text{formula}) \\
&\longrightarrow (r \vee (\text{formula} \vee \text{formula})) \\
&\longrightarrow (r \vee (\neg\text{formula} \vee \text{formula})) \\
&\longrightarrow (r \vee (\neg p \vee \neg\text{formula})) \\
&\longrightarrow (r \vee (\neg p \vee \neg(\text{formula} \wedge \text{formula}))) \\
&\longrightarrow (r \vee (\neg p \vee \neg(q \wedge \text{formula}))) \\
&\longrightarrow (r \vee (\neg p \vee \neg(q \wedge p)))
\end{aligned}$$

See Exercise 2.

### 1.2.5 Avoiding parentheses

If you were to formulate similar rules for building algebraic expressions with operations for addition and multiplication, you might write something like, 'if $E_1$ and $E_2$ are expressions,

then $E_1 + E_2$ and $E_1 \times E_2$ are expressions'. The problem with this is that if you apply the rule twice in succession, you find that there are two different ways to generate expressions like

$$x + y \times z.$$

Does this mean add $x$ and $y$ and multiply by $z$? Or does it mean add $x$ to the product of $y$ and $z$? To avoid such ambiguity in our definition of propositional formulas, we require adding a pair of parentheses every time we apply one of the operators $\wedge$ and $\vee$. The result, though, is that formulas get cluttered up with lots of parentheses. We can clean things up a bit by observing that we never need to write the outermost pair of parentheses in a formula, so we can write our propositional formula above as

$$r \vee (\neg p \vee \neg(q \wedge p)).$$

Later we will see that the operator $\vee$, as well as $\wedge$, obeys an associative law, so that there is never any need to distinguish between $\phi \vee (\psi \vee \rho)$ and $(\phi \vee \psi) \vee \rho$. Thus we can write the formula above as

$$r \vee \neg p \vee \neg(q \wedge p).$$

What about 'formulas' like

$$p \wedge q \vee r \ ?$$

There is not universal agreement about how to interpret this. Some authors define a precedence rule, much like the rule that multiplication takes precedence over addition in algebraic formulas, and give $\wedge$ a higher precedence than $\vee$. We will take a more conservative approach and require parentheses. So we will allow both $(p \wedge q) \vee r$ and $p \wedge (q \vee r)$, but treat the unparenthesized formula as illegal. (See the Exercises beginning at Exercise 5.)

## 1.3 Propositional Formulas-Semantics

### 1.3.1 The value of a propositional formula.

In a boolean expression in Python, an atom, like `year%4 == 0` in our example, is a *proposition*: it asserts something that is either true or false, and the truth or falsehood varies, depending on the value of the integer variable `year`. The entire expression is also a proposition, whose truth depends in turn on the truth or falsehood of the atomic boolean formulas it contains. This is the situation in general in propositional formulas: The variables $p, q$, *etc.,* in the formula denote propositions that can be assigned the *values* **true** and **false** arbitrarily. Given such an assignment, the value of the entire formula is determined by the following recursive rules:

- **T** has the value **true**, and **F** has the value **false**.
- $\neg\phi$ has the value **true** if $\phi$ has the value **false,** and vice-versa.

- $(\phi \wedge \psi)$ has the value **true** if both $\phi$ and $\psi$ do; otherwise it has the value **false**.
- $(\phi \vee \psi)$ has the value **false** if both $\phi$ and $\psi$ do; otherwise it has the value **true**.

These are, of course, the same rules as for `and`, `or` and `not` in Python.
For example, consider again the formula

$$r \vee (\neg p \vee \neg(q \wedge p)).$$

(We'll now allow ourselves the luxury of leaving off the outermost pair of parentheses.)

What is the value of this formula if the variables $p, q, r$ are assigned, respectively, the values **false**, **false**, **true**? Since $p$ is **false**, $\neg p$ is **true**, Consequently both $\neg p \vee \neg(q \wedge p)$ and the complete formula $r \vee (\neg p \vee \neg(q \wedge p))$ have the value **true**.

What if the assignment is $p \mapsto$ **true**, $q \mapsto$ **true**, and $r \mapsto$ **false**? Then $q \wedge p$ is **true**, $\neg(q \wedge p)$ is **false**, $\neg p$ is **false**, and thus $\neg p \vee \neg(q \wedge p)$ is **false,** and consequently $r \vee (\neg p \vee \neg(q \wedge p))$ gets the value **false**.

Can you find another assignment of values to $p, q$ and $r$ that makes the formula **false**? (See Exercise 9.)

We will take another look at the semantics of propositional formulas in Chapter 3, when we discuss functions.

### 1.3.2 Example: Modeling ordinary language by propositional formulas.

Your eating habits at lunch are subject to several constraints of personal taste and the rules of the cafeteria:

- You can order either soup or salad.
- You can't order both soup and salad.
- While you are fond of soup, you only eat it on cold days, so it has to be less than 32 degrees outside for you to order it.

We will model these rules with a formula of propositional logic, much as we described the rule for determining if a year is a leap year with a boolean expression in Python. We proceed by identifying the underlying atomic propositions in the rule, and associate these with the variables $p, q, r$:

- $p$: You order the soup.
- $q$: You order the salad.
- $r$: It is less than 32 degrees outside.

The three constraints can then be encoded by the propositional formulas:

- $p \vee q$
- $\neg(p \wedge q)$
- $r \vee q$

In fact, there are many different ways to write these constraints as propositional formulas. Look especially at our encoding of the last constraint, which we are in essence rephrasing as 'Either it's freezing or you order the salad.' We could also have phrased it as, 'You can't eat soup on a warm day', which we would write as $\neg(p \wedge \neg r)$.

We can encode all three constraints by the single propositional formula

$$(p \vee q) \wedge \neg(p \wedge q) \wedge (r \vee q).$$

When this formula has the value **true**, the values assigned to the variables are consistent with the constraints above; when it's **false**, the constraints are not all satisfied.

### 1.3.3 Truth tables

A *truth table* shows how the values of a complex formula are computed from the values assigned to the variables. Below are truth tables for the formulas $\neg p$, $p \vee q$ and $p \wedge q$.

| $p$ | $\neg p$ |
|---|---|
| T | F |
| F | T |

| $p$ | $q$ | $p \wedge q$ | $p \vee q$ |
|---|---|---|---|
| T | T | T | T |
| T | F | F | T |
| F | T | F | T |
| F | F | F | F |

We can use such tables to compute the truth value of any formula, given assignments of truth values to the variables. Below is the truth table for our soup and salad rule. We begin in the left-hand columns by tabulating all the possible assignments of truth values to the variables. In subsequent columns we compute the values of the various subformulas entering into the formula. The last column displays the values of the formula itself.

The rows in which a **T** appears in the last column correspond to the three assignments of values to the variables $p, q, r$ that make the whole formula true: It's freezing and you order the soup but not the salad; it's freezing and you order the salad but not the soup; it's not freezing and you order the salad but not the soup.

| $p$ | $q$ | $r$ | $\phi_1 = p \vee q$ | $\phi_2 = \neg(p \wedge q)$ | $\phi_3 = r \vee q$ | $\phi_1 \wedge \phi_2 \wedge \phi_3$ |
|---|---|---|---|---|---|---|
| T | T | T | T | F | T | F |
| T | T | F | T | F | T | F |
| T | F | T | T | T | T | T |
| T | F | F | T | T | F | F |
| F | T | T | T | T | T | T |
| F | T | F | T | T | T | T |
| F | F | T | F | T | T | F |
| F | F | F | F | T | F | F |

Table 1.1: Truth table for the soup-or-salad rule

### 1.3.4 Satisfiability and tautology

A formula $\phi$ is *satisfiable* (also called *consistent*) if there is at least one assignment of truth values to the variables that makes the formula true. Another way to say this is that the last column of the truth table for $\phi$ contains at least one occurrence of **T**. For example, our soup-or-salad formula is satisfiable. In fact, there are several satisfying assignments, two in which $r$ is assigned the value **true** and one in which $r$ is assigned **false**—a good thing, since it means that you can always find something to eat, whatever the weather.

If *all* the entries in the final column are **T**, the formula is called a *tautology* (also *valid*). Such a formula has the value **true** regardless of the values of the variables. Another way to say this is that $\phi$ is a tautology if and only if $\neg\phi$ is not satisfiable. Every tautology is satisfiable, but not every satisfiable formula is a tautology, as our soup-and-salad example shows. A simple example of a tautology is the formula $p \vee \neg p$. An even simpler example is **T**.

A formula is a *contradiction* (also *inconsistent*) if it is not satisfiable, that is, all the entries in the last column of the truth table are **F**. Another way to say this is that $\phi$ is a contradiction if and only if $\neg\phi$ is a tautology. A simple example is $p \wedge \neg p$. An even simpler example is **F**.

## 1.4 New connectives: conditional, biconditional, exclusive or

Our logical operators $\vee$, $\wedge$, $\neg$ are often called *connectives*. Here we will describe three new connectives. In a sense, these connectives are not new at all: As we will explain at length, anything that we can say with these new symbols can also be expressed using the three original ones. What we gain by adding them is a more succinct, and in some respects, more natural way to write logical propositions.

### 1.4.1 Conditional

We could rephrase the last constraint in the soup-and-salad example by 'if you order the soup, then it has to be freezing outside'. (But *not* by, 'if it's freezing outside then you order the soup', which is false, since you might order salad on a cold day.) We introduce a new connective $\rightarrow$, and write $\phi \rightarrow \psi$, to mean 'if $\phi$ then $\psi$'. So, using the variable symbols from that example, we can write the last constraint in our example as $p \rightarrow r$. The official definition of the semantics of this connective is given by the truth table Table 1.2.

| $\phi$ | $\psi$ | $\phi \rightarrow \psi$ |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

Table 1.2: *Truth table defining the connective* $\rightarrow$

It seems strange to say that 'if $\phi$ then $\psi$' is true when both $\phi$ and $\psi$ are false! To make some sense of this, imagine that you go to the cafeteria every day: Some days it is freezing and you order the soup, some days it is freezing and you order the salad, and some days it is warmer and you order the salad (so that $p$ and $r$ are both false) . But it is *never* the case that you order the soup on a warmer day. The *only* way $p \rightarrow r$ can be false is for $p$ to be true and $r$ false. That is *all* that $\rightarrow$ means. [1]

### 1.4.2 Biconditional

We write $\phi \leftrightarrow \psi$ to mean 'if $\phi$ then $\psi$ and if $\psi$ then $\phi$', so you can think of $\phi \leftrightarrow \psi$ as an abbreviation for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. Another way to say this is '$\phi$ if and only if $\psi$'. For example: A different cafeteria customer in our soup-and-salad story, one with more rigid habits, might always order soup on freezing days and the salad on all other days. We can represent this constraint by $p \leftrightarrow r$.

Table 1.3 is the truth table for the biconditional.

[1] Equivalent formulations of $\phi \rightarrow \psi$ in English are '$\phi$ implies $\psi$', '$\psi$ is a necessary condition for $\phi$', '$\phi$ is a sufficient condition for $\psi$', and others. Implicit in some of these formulations is a notion of dependent and independent conditions. Saying 'cold weather is a necessary condition for ordering soup' is entirely reasonable, but 'ordering soup is a sufficient condition for cold weather' has a peculiar ring to it, as it suggests that your lunch choice influenced the weather. It is important to keep in mind that the formal definition of the connective $\rightarrow$ involves no such notion of dependence or causality.

| $\phi$ | $\psi$ | $\phi \leftrightarrow \psi$ |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | T |

Table 1.3: *Truth table defining the connective* $\leftrightarrow$.

### 1.4.3 Exclusive-or

The symbol $\vee$ does not mean 'or' in the exclusive sense it often has in English. When a restaurant waiter tells you, 'you can get the soup or the salad with your dinner', he certainly does not mean that you can order *both*, but the logical 'or' would allow this. We define a new connective $\phi \oplus \psi$ to mean just what the waiter intends here: '$\phi$ or $\psi$ but not both'. For instance, our cafeteria's soup-or-salad rule is precisely $p \oplus q$. The exact definition is given by the truth table Table 1.4.

| $\phi$ | $\psi$ | $\phi \oplus \psi$ |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

Table 1.4: *Truth table defining the connective* $\oplus$.

With these new symbols we can express the soup-and-salad example much more succinctly:

$$(p \oplus q) \wedge (p \rightarrow r).$$

## 1.5 Equivalent formulas

As we observed above, the only way for $p \rightarrow q$ to have the value **false** is for $p$ to be **true** and $q$ **false**. This means that the formulas $p \rightarrow q$ and $\neg(p \wedge \neg q)$ have the same truth value regardless of what values are assigned to the variables. We say that the two formulas are *equivalent*. Similarly, you may have noticed that for each assignment, $p \oplus q$ has the opposite value from $p \leftrightarrow q$, so that $p \oplus q$ is equivalent to $\neg(p \leftrightarrow q)$. Equivalent formulas say exactly the same thing, but with different notation.

In general, if $\phi$ and $\psi$ are formulas, we say that $\phi$ and $\psi$ are *equivalent,* and write $\phi \equiv \psi$, if for every assignment of truth values to the variables occurring in $\phi$ and $\psi$, the resulting

| $p$ | $q$ | $\neg p$ | $\neg q$ | $p \vee q$ | $\neg(p \vee q)$ | $\neg p \wedge \neg q$ |
|---|---|---|---|---|---|---|
| T | T | F | F | T | F | F |
| T | F | F | T | T | F | F |
| F | T | T | F | T | F | F |
| F | F | T | T | F | T | T |

Table 1.5: *Truth table establishing the identity* $\neg(p \vee q) \equiv \neg p \wedge \neg q$.

values of $\phi$ and $\psi$ are the same. Another way to say this is, $\phi \leftrightarrow \psi$ is a tautology. It's worth pointing out a little subtlety here: The symbol $\equiv$ is *not* another connective of propositional logic and never occurs inside formulas; it is a symbol we use when we talk *about* propositional formulas.

Here is another example. Table 1.5 displays the truth tables for the formulas $\neg(p \vee q)$ and $\neg p \wedge \neg q$. (We have combined the tables for these two formulas into a single array, giving the value of the first formula in the next-to-last column, and of the second formula in the last column.)

Since the last two columns are the same, we conclude that the two formulas are equivalent:

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

We really didn't need to go to all the trouble of constructing truth tables to show the equivalence. In cases like this it is just as easy to talk our way through the problem: the only way $\neg(p \vee q)$ can be true is for both $p$ and $q$ to be false, that is, for both $\neg p$ and $\neg q$ to be true.

### 1.5.1 Identities

Equivalences between propositional formulas work much like equations between algebraic expressions that hold regardless of the values of the variables they contain. An example of this is the distributive law for multiplication and addition of numbers.

$$x(y + z) = xy + xz.$$

Such equations are called *identities*. (In contrast, something like $xy = z$ is an equation, but not an identity, as there are values of $x, y$ and $z$ for which the two sides are unequal.) As with any equation, you can do the same thing to both sides of the identity—multiply them by the same value, add the same value to them, *etc.*—and the result will still be an identity. Moreover, if you replace each of the variables occurring in the identity by an arbitrary expression, the result is still an identity. For instance, if we replaced $x$ by $u + 2v$, $y$ by $w^2$ and $z$ by $5uv$, we would get the equation

$$(u + 2v)(w^2 + 5uv) = (u + 2v)w^2 + (u + 2v) \cdot 5uv,$$

which is still an identity, since it holds for all values of the variables $u, v, w$.

Equivalences between propositional formulas are also called identities, and work much the same way. Starting from the identity

$$\neg(p \vee q) \equiv \neg p \wedge \neg q,$$

we can apply the same operation to both sides of the identity to obtain a new identity:

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

And we can substitute any formulas for the variables in an identity and obtain further identities:

$$(s \wedge t) \wedge \neg\neg((s \vee t) \vee (s \wedge t)) \equiv (s \wedge t) \wedge \neg(\neg(s \vee t) \wedge \neg(s \wedge t)).$$

There are a number of standard propositional identities. These are analogous to (and in some instances closely resemble) algebraic identities like the distributive and associative laws. We have tabulated about half of the most basic ones in Table 1.6 below. Most of these are nearly obvious at a glance. Some, like the distributive law in the last row, require a little thought. But all can easily be proved, either by construction of truth tables, or by talking your way through, just as we proved the identity $\neg(p \vee q) \equiv \neg p \wedge \neg q$, which appears in the second-to-last line of the table.

| | | |
|---|---|---|
| $p \wedge \mathbf{T}$ | $\equiv$ | $p$ |
| $p \vee \mathbf{T}$ | $\equiv$ | $\mathbf{T}$ |
| $p \vee \neg p$ | $\equiv$ | $\mathbf{T}$ |
| $p \vee p$ | $\equiv$ | $p$ |
| $p \vee (p \wedge q)$ | $\equiv$ | $p$ |
| $\neg\neg p$ | $\equiv$ | $p$ |
| $p \vee q$ | $\equiv$ | $q \vee p$ |
| $p \vee (q \vee r)$ | $\equiv$ | $(p \vee q) \vee r$ |
| $\neg(p \vee q)$ | $\equiv$ | $\neg p \wedge \neg q$ |
| $p \vee (q \wedge r)$ | $\equiv$ | $(p \vee q) \wedge (p \vee r)$ |

Table 1.6: Some basic propositional identities

### 1.5.2 DeMorgan's Laws and Duality

The identity

$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

is called *DeMorgan's Law.* Let's play with it a bit: We can perform the same operations on both sides of an identity and the result will still be an identity, so let's apply $\neg$ to both sides. This gives us

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

We can also substitute arbitrary formulas for the variables in an identity and still have an identity, so let's substitute $\neg p$ for $p$ and $\neg q$ for $q$. Now we have

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

The identity in the sixth line of the table allows us to erase any occurrence of $\neg\neg$ (the two successive negations cancel one another), so we wind up with

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

This is *also* called DeMorgan's Law, and it is identical to the original version of DeMorgan's Law, except that we have changed the $\vee$ to $\wedge$ and the $\wedge$ to $\vee$.

Now we can use DeMorgan's Laws to show that this trick works for *every* identity. Let's take, for example, the distributive identity at the end of the table:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

We negate both sides,

$$\neg(p \vee (q \wedge r)) \equiv \neg((p \vee q) \wedge (p \vee r)),$$

apply DeMorgan's Law to each side,

$$\neg p \wedge \neg(q \wedge r) \equiv \neg(p \vee q) \vee \neg(p \vee r),$$

and apply DeMorgan's Law *again* on both sides,

$$\neg p \wedge (\neg q \vee \neg r) \equiv (\neg p \wedge \neg q) \vee (\neg p \wedge \neg r),$$

substitute $\neg p$ for $p$ and $\neg q$ for $q$,

$$\neg\neg p \wedge (\neg\neg q \vee \neg\neg r) \equiv (\neg\neg p \wedge \neg\neg q) \vee (\neg\neg p \wedge \neg\neg r),$$

and clean up the double negations:

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

This is *another* distributive law, this time with $\wedge$ distributing over $\vee$. If $\phi$ is a formula built from variables with the connectives $\vee$, $\wedge$, $\neg$ then its *dual* $\phi'$ is obtained by replacing every occurrence $\wedge$ by $\vee$ and every occurrence of $\vee$ by $\wedge$. The general principle is that if $\phi \equiv \psi$ is an identity, then so is $\phi' \equiv \psi'$.

If we rush ahead and try to apply this to the first identity in the table

$$p \wedge \mathbf{T} \equiv p,$$

we would get $p \vee \mathbf{T} \equiv p$. This is not an identity, as you can tell by assigning $p$ the value **false**. The problem is that we overreached in forming the dual of a formula that contains constants like $\mathbf{T}$ as well as variables. But the solution is simple: to form the dual of such a formula we change $\vee$ to $\wedge$, $\wedge$ to $\vee$, $\mathbf{T}$ to $\mathbf{F}$ and $\mathbf{F}$ to $\mathbf{T}$. Then everything works, and the new equation we get is $p \vee \mathbf{F} \equiv p$, which is indeed an identity.

Here is an 'official' statement of the general principle that we have just established:

**Theorem 1.5.1.** *(Duality Principle) Let $\phi$ and $\psi$ be propositional formulas consisting of variables, constants, and the connectives $\wedge$, $\vee$, $\neg$. Let $\phi'$ be the formula obtained from $\phi$ by changing all occurrences of $\vee$ to $\wedge$, and vice-versa, and all occurrences of $\mathbf{T}$ to $\mathbf{F}$, and vice-versa. Let $\psi'$ be obtained from $\psi$ in the same way. If $\phi \equiv \psi$, then $\phi' \equiv \psi'$.*

Thanks to Theorem 1.5.1, we can produce a new table of identities from Table 1.6:

| | | |
|---|---|---|
| $p \vee \mathbf{F}$ | $\equiv$ | $p$ |
| $p \wedge \mathbf{F}$ | $\equiv$ | $\mathbf{F}$ |
| $p \wedge \neg p$ | $\equiv$ | $\mathbf{F}$ |
| $p \wedge p$ | $\equiv$ | $p$ |
| $p \wedge (p \vee q)$ | $\equiv$ | $p$ |
| $p \wedge q$ | $\equiv$ | $q \wedge p$ |
| $p \wedge (q \wedge r)$ | $\equiv$ | $(p \wedge q) \wedge r$ |
| $\neg(p \wedge q)$ | $\equiv$ | $\neg p \vee \neg q$ |
| $p \wedge (q \vee r)$ | $\equiv$ | $(p \wedge q) \vee (p \wedge r)$ |

Table 1.7: Duals of the identities in Table 1.6

You might notice that we have left off the dual of one identity that appeared in the original table: this is because $\neg\neg p \equiv p$ is its own dual, so we chose not to write it twice.

The identities have a few obvious extensions, which should be familiar from the way we deal with standard algebraic expressions. Because of the associative law

$$p \vee (q \vee r) \equiv (p \vee q) \vee r,$$

we can just write these formulas as $p \vee q \vee r$; it doesn't matter how we insert parentheses into this expression, because the result will be the same either way.

By applying the associative law several times in succession, we get

$$\begin{aligned}(p_1 \vee (p_2 \vee p_3)) \vee p_4 &\equiv ((p_1 \vee p_2) \vee p_3) \vee p_4 \\ &\equiv (p_1 \vee p_2) \vee (p_3 \vee p_4) \\ &\equiv p_1 \vee (p_2 \vee (p_3 \vee p_4)) \\ &\equiv p_1 \vee ((p_2 \vee p_3) \vee p_4)\end{aligned}$$

so, again, no matter how we (legally) introduce parentheses into

$$p_1 \vee p_2 \vee p_3 \vee p_4,$$

the result is the same, in the sense that any two formulas we produce this way are equivalent. As you might expect, the same holds for any number of terms[2], so we can write

$$p_1 \vee p_2 \vee \cdots \vee p_k$$

[2]This might not be completely obvious (see the discussion below about 'hand-waving'). Later in the text, you will be asked to provide a careful proof of this fact.

without ambiguity.

We can similarly extend the distributive identity to any number of terms:

$$q \wedge (p_1 \vee \cdots \vee p_k) \equiv (q \wedge p_1) \vee \cdots \vee (q \wedge p_k).$$

Of course the same applies to the duals of all these formulas.

There is a more compact notation for the disjunction and conjunction of many terms. We write

$$p_1 \vee \cdots \vee p_k$$

as

$$\bigvee_{i=1}^{k} p_i$$

and

$$p_1 \wedge \cdots \wedge p_k$$

as

$$\bigwedge_{i=1}^{k} p_i.$$

With this notation the extended distributive law becomes

$$q \wedge \bigvee_{i=1}^{k} p_i \equiv \bigvee_{i=1}^{k} (q \wedge p_i).$$

### 1.5.3 Did we just prove something?

In the preceding paragraphs we made a number of claims about equivalent formulas and gave some arguments to support those claims. We even went so far as to call one of them a *theorem.* In mathematics, theorems have *proofs.* Did we actually prove anything?

We really did prove the first version of DeMorgan's Law, by carefully tabulating the values of the two sides of the identity on all four assignments of truth values to the variables. And we did show, step by step, how the dual version of the law could be derived from its original formulation. In each of these cases, we were dealing with only a single formula. But to justify the general duality principle, we gave just the example of just a single identity, and claimed that the same procedure would work in the same way for every identity. As if to underscore the problems with this approach, we immediately followed this with an identity for which the method *didn't* work, and so we patched up the definition of dual formula, said 'ok, *now* it works', and called the result a theorem.

This is a classic instance of the 'hand-waving' argument, in which the prover shoves aside a lot of difficulties that he is either too lazy or too uncertain to address, often punctuating his argument with 'obviously's and 'it is clear that's. Later in this text, we will take up the question of proofs, and give the need for careful proof the attention that it deserves. We will even give, as an example, a real proof of Theorem 1.5.1. But that will have to wait. In the next few sections we will continue with the cavalier approach, and establish a few more general principles of propositional logic, with a wave of the hand.

### 1.5.4 Normal Forms

*Disjunctive Normal Form.* Look again at the soup-or-salad example, and in particular at its truth table Table 1.1. When we first displayed this table, we noted that exactly three rows of the table gave the result **T** in the last column. The three rows correspond to the three assignments

$$\begin{array}{lll} p \mapsto \textbf{true} & q \mapsto \textbf{false} & r \mapsto \textbf{true} \\ p \mapsto \textbf{false} & q \mapsto \textbf{true} & r \mapsto \textbf{true} \\ p \mapsto \textbf{false} & q \mapsto \textbf{true} & r \mapsto \textbf{false} \end{array}$$

We can express the property that one of these assignments is made by the single formula:

$$(p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (\neg p \wedge q \wedge \neg r). \tag{1.5.1}$$

This formula has the value **true** for the three assignments above, and the value **false** otherwise. In other words, this formula is equivalent to the one we started from to make the truth table.

Formulas like (1.5.1) are said to be in *disjunctive normal form,* or DNF. In general, a formula is said to have this form if it is a disjunction

$$\phi_1 \vee \cdots \vee \phi_k,$$

where each subformula $\phi_i$ is a conjunction

$$L_{i1} \wedge L_{i2} \wedge \cdots \wedge L_{is_i},$$

and each $L_{ij}$ is a *literal*: either a variable itself or the negation of a variable.[3] The procedure we described above works in general: Starting from the truth table for a formula, you can extract an equivalent formula in DNF by looking at the rows in which the formula has the value **true**. Here is a formal statement of the result:

**Theorem 1.5.2.** *Every propositional formula is equivalent to a formula in disjunctive normal form.*

*Universality of propositional formulas.* There is another conclusion we can draw from the reasoning above. We didn't need to start from a formula; we could have begun with a table that gives all possible assignments to the variables, and *any* sequence of truth values in the last column. Following the procedure outlined above, we could have still produced a DNF formula that has the given table as its truth table. So propositional formulas are a kind of universal language: Any rule that we can describe with sufficient precision to be represented as a truth table can be described by a propositional formula. We will return to this very important point in the next chapter.

[3]The indices $k$ and $s_i$ in the above formulas can be equal to 1. That is, a formula in DNF could be just a single conjunction of literals, and a conjunction of literals can be just a single literal. Thus $p \vee \neg q$, $p \wedge q \wedge r$, and even $p$, are all in DNF.

*Conjunctive Normal Form.* Theorem 1.5.2 has a dual version. Let us begin with a formula $\phi$. It has a dual $\phi'$, and by Theorem 1.5.2, $\phi' \equiv \delta$, where $\delta$ is a DNF formula. Since $\phi = (\phi')'$, our Theorem 1.5.1 tells us that $\phi \equiv \delta'$, where $\delta'$ is the dual of $\delta$. What does the dual of a DNF formula look like? It is the *conjunction*

$$\phi_1 \wedge \cdots \wedge \phi_k,$$

where each $\phi_i$ is in turn a disjunction

$$L_{i1} \vee L_{i2} \vee \cdots \vee L_{is_i},$$

of literals. Such a formula is said, naturally enough, to be in *conjunctive normal form*, or CNF. We conclude:

**Theorem 1.5.3.** *Every propositional formula is equivalent to a formula in conjunctive normal form.*

**Example.** We will compute a CNF representation of the soup-and-salad condition. We begin again with the truth table, and this time select the five lines that contain **F** in the last column. Writing the corresponding assignments in DNF, and negating the whole thing, gives us an equivalent formula

$$\neg\big[(p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge \neg r)\big].$$

We now apply DeMorgan's Law: We move the negation inside the outermost brackets, changing the $\vee$'s to $\wedge$'s. Then we apply DeMorgan's Law again, bringing negation symbols inside the inner brackets, changing $\wedge$'s to $\vee$'s and negating each literal. The resulting formula is in conjunctive normal form, and is equivalent to (1.5.1):

$$(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee \neg r) \wedge (p \vee q \vee r).$$

Formulas do not have unique representations in these normal forms. In particular, it is possible to give a simpler CNF formula that is equivalent to the one we just derived. The first two conjuncts of the above formula can be combined with the help of some of our basic identities:

$$\begin{aligned}(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) &\equiv (\neg p \vee \neg q) \vee (\neg r \wedge r) \\ &\equiv (\neg p \vee \neg q) \vee \mathbf{F} \\ &\equiv \neg p \vee \neg q.\end{aligned}$$

Similarly, the last two conjuncts are equivalent to $p \vee q$, so we get the simplified CNF formula

$$(\neg p \vee \neg q) \wedge (\neg p \vee q \vee r) \wedge (p \vee q).$$

See Exercise 26.

## 1.6 Complete Sets of Connectives

As we noted above, any truth table can be realized by a propositional formula, and we even have some control over the form of the formula (*e.g.,* it can be in DNF or CNF). We have already mentioned that the 'new' connectives $\rightarrow$, $\leftrightarrow$ and $\oplus$ introduced in Section 1.4 are not, strictly speaking, necessary: They can all be expressed in terms of $\vee$, $\wedge$ and $\neg$. For example, $p \rightarrow q$ is equivalent to $\neg p \vee q$, and thus for any formulas $\phi$ and $\psi$, $\phi \rightarrow \psi$ is equivalent to $\neg\phi \vee \psi$.

Are all three of the original connectives necessary? By DeMorgan's Law (negating both sides),

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

This means that we don't need $\vee$ either: you can use this identity to replace every occurrence of $\vee$ by combinations of $\neg$ and $\wedge$. Thus every propositional formula is equivalent to one using only the two connectives $\neg$ and $\wedge$: These two form a *complete* set of connectives.

In precisely the same fashion, we can use the dual version of DeMorgan's Law to write $\phi \wedge \psi$ in terms of $\neg$ and $\vee$, and thus these two as well form a complete set of connectives.

Can we do better? Can we get things down to just a single connective? Using $\wedge$ by itself would not suffice, because then every formula could be put in the form.

$$p_1 \wedge \cdots \wedge p_k.$$

Such a formula could not be equivalent to $\neg p$. For the same reason, $\vee$ by itself does not suffice. And if all you had was $\neg$, you could not write a formula that involved more than one variable. So we seem to have come to the end of the line as far as reducing the number of connectives necessary in our logic.

Or have we? Let's define a new connective, called NAND (for 'not-and'). Its definition is

$$p \text{ NAND } q \equiv \neg(p \wedge q).$$

We then have

$$\neg p \equiv p \text{ NAND } p,$$

and

$$\begin{aligned} p \wedge q &\equiv \neg(p \text{ NAND } q) \\ &\equiv (p \text{ NAND } q) \text{ NAND } (p \text{ NAND } q). \end{aligned}$$

We already know that we can replace every propositional formula by an equivalent one that uses only the connectives $\wedge$ and $\neg$, so the foregoing equations show that we need only the single connective NAND.

**Example.** To illustrate this, we will show how to express the connective $\oplus$ using only NAND. It will be convenient to have a more compact notation for the NAND operator, so we will write $\phi|\psi$ in place of $\phi$ NAND $\psi$.

As we saw earlier,

$$p \oplus q \equiv (p \vee q) \wedge \neg(p \wedge q).$$

We provisionally write $p \vee q$ as $\phi$, so that the formula above is equivalent to

$$\begin{aligned} \phi \wedge (p|q) &\equiv \neg\neg(\phi \wedge (p|q)) \\ &\equiv \neg(\phi|(p|q)) \\ &\equiv (\phi|(p|q))|(\phi|(p|q)). \end{aligned}$$

We can, in turn, rewrite $\phi = p \vee q$ as

$$\begin{aligned} p \vee q &\equiv \neg(\neg p \wedge \neg q) \\ &\equiv (\neg p|\neg q) \\ &\equiv (p|p)|(q|q) \end{aligned}$$

Substituting this for $\phi$ in the previous formula gives

$$p \oplus q \equiv (((p|p)|(q|q))|(p|q))|(((p|p)|(q|q))|(p|q)).$$

Obviously, there is a big tradeoff here: reducing the number of connectives makes the resulting formulas quite long and difficult to read. Maybe expressing everything with just a single connective is not such a good idea after all!

Nonetheless, this is more than a mere mathematical curiosity, and even has some practical importance, as we'll see in the next chapter.

## 1.7 Historical Notes

The systematic study of logic was initiated by the Greek philosopher Aristotle (*circa* 350 BC), who undertook a classification of syllogistic reasoning, the kind contained in arguments like 'No dog is green. Rover is green. Therefore Rover is not a dog.' Aristotle used letters $A, B, \Gamma$, to denote the properties in the premises and conclusions of the syllogisms, and thus would write them as 'no B is Γ', and the like. This practice, which continued down through the centuries, embodied the crucial insight that logic deals with the *form* of propositions and how they are combined, and not their *content.*

It was widely believed up until at least the 19th century that Aristotle's system of logic was complete, and that it was not possible to make any further discoveries in the subject.[4] However, this view was challenged by a number of writers, beginning with Leibniz (late 17th century), who tried to work out the details of 'logical calculus', in which logical reasoning could be carried out by performing algebra-like computations.

[4]Kant (late 18th c.) for example, wrote, 'There are but few sciences that can come into a permanent state, which admits of no further alteration. To these belong Logic and Metaphysics. Aristotle has omitted no essential point of the understanding; we have only become more accurate, methodical, and orderly.' (Kant's 'Introduction to Logic').

The watershed development was the work of the British mathematician George Boole in the first half of the nineteenth century, who presented a detailed algebraic system of logic. Boole called his work *An Investigation of the Laws of Thought* and described its purpose as

> to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of Logic and construct its method...

and, even more ambitiously,

> to collect from the various elements of truth brought to view in the course of these inquiries some probable intimations concerning the nature and constitution of the human mind.

It is from his name, of course, that we get 'boolean'.

Boole's methods were expanded upon and refined by the American C. S. Peirce, whose system ('The Algebra of Logic', 1880) begins to resemble the propositional logic described in this text. Peirce also was the first to observe that every propositional formula is equivalent to one using the single connective NAND.

We have said very little about modeling *deduction*. Much of the research in logic, beginning with Frege in 1879, was aimed at giving a precise foundation for mathematical reasoning, and extended far beyond propositional logic. *Logicomix: An Epic Search for Truth* (2009) is entertaining account of this quest, centered around the figure of Bertrand Russell, in the form of, of all things, a graphic novel. When restricted to propositional logic, the system of Frege gives a collection of axioms and rules of deduction from which all the tautologies can be deduced. The result of Exercise 31, which outlines how to derive every propositional identity from a small number of basic identities, is in this spirit.

Much of this history closely parallels that of the developing interest in automatic computation: Leibniz also invented a calculating machine. Boole and DeMorgan, who also wrote extensively on symbolic logic, were contemporaries and acquaintances of Charles Babbage, who designed a programmable general-purpose computer in the 1840's. William Stanley Jevons, beginning in the 1860's, both wrote on symbolic logic and designed a 'logic piano', a mechanical device for carrying out logical calculations. C. S. Peirce, in an 1886 letter to a former student, described how a logic machine could be built using electrical circuits.

We will see this theme reappear in the last chapter, when the history of logic and computation converge in the work of Turing.

## 1.8 Exercises

### 1.8.1 Syntax of propositional logic

1. Which of the following is a propositional formula? For purposes of this problem, you should *strictly* follow the rules given in the definition in 1.2.1, rather than the relaxed

rules discussed in Exercise 3 below and in 1.2.5. The relaxed rules allow you to leave off the outermost pair of parentheses in a formula, and to put an extra pair of parentheses around any subformula. How would these new rules change your answer?

*(a)* $p_1 \wedge \neg q$

*(b)* $((p_1 \wedge \neg q))$

*(c)* $(\neg(p_1 \wedge \neg q))$

*(d)* $(p_1 \wedge (\neg q \vee r))$

*(e)* $(p_1 \wedge \neg(q \vee r))$

*(f)* $(p_1 \vee \neg q \wedge r)$

*(g)* $(p_1(\vee)((\neg)q \wedge r)$

*(h)* $(p_1 \wedge ((\neg q) \vee r)$

2. For those parts of Exercise 1 that are legal formulas, show a derivation of the formula using the grammar in 1.2.4.

3. Our definition of propositional formula does not allow $(\neg p)$ or $(((p \wedge q)))$ as formulas. While the extra parentheses in these strings are unnecessary, one really ought to be allowed to place parentheses around any subformula occurring within a formula. On the other hand, we should not allow things like $(p(\wedge)q)$.

   *(a)* Add another rule to the grammar so as to allow these extra parentheses.

   *(b)* With this new rule, several of the illegal formulas in Exercise 1 become legal. Show the derivations of these formulas using the new grammar.

4. Another way to exhibit the syntactic structure of a formula is by use of a *parse tree*. In such a tree, the leaves are labeled by atomic formulas, internal nodes with one child by $\neg$, and internal nodes with two children by $\vee$ or $\wedge$.

   *(a)* You can probably figure out how the tree encodes a formula. Write down the formula represented by the parse tree in Figure 4. Include whatever parentheses are required by the definition.

   *(b)* Draw parse trees for the parts of Exercise 1 that represent legal formulas.

5. The following rules are part of a grammar for generating algebraic expressions.

$$\begin{aligned} E &\longrightarrow x|y|z \\ E &\longrightarrow E + E \\ E &\longrightarrow E \times E \end{aligned}$$

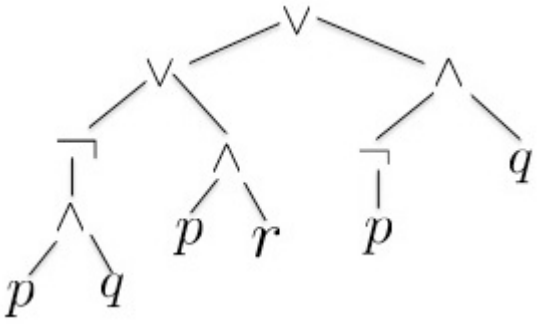



Figure 1.4: *Parse tree for Problem 4*

Find two different derivations of the expression $x + y \times z$ using this grammar. Which of these corresponds to the interpretation 'multiply first, then add', and which to the interpretation 'add first, then multiply'?

6. Even with our more relaxed rules, we still disallow ambiguous formulas like $p \vee q \wedge r$. Does Python allow boolean formulas of the form `p or q and r`? If so, how does it resolve the ambiguity? You can search the online documentation for the answer, but it's not easy to find. (See Exercise 7.) Alternatively, you can perform a simple experiment.

7.* How can we revise the grammar for propositional formulas so that it allows us to leave off the outermost pair of parentheses, or to add a new pair of parentheses around any formula? Thus it should be possible to derive $p \wedge (q \vee r)$ with this new grammar, but not $p \wedge q \vee r$ or, for that matter, $p \vee q \vee r$. (HINT: Our present grammar has just the single symbol `formula` to represent a grammatical category. Consider adding a new symbol representing a special kind of formula, those to which connectives can be applied...)

8.* The on-line Python documentation includes within it a grammar for boolean expressions. (In fact, it includes a grammar for all of Python! ) Find where this grammar is hidden, and explain how it works. Show the derivation of `x and not y or z` in this grammar.

### 1.8.2 Semantics of propositional logic

9. In 1.3, we saw that the assignment $p \mapsto$ **true**, $q \mapsto$ **true**, and $r \mapsto$ **false** results in the formula $r \vee (\neg p \vee \neg(q \wedge p))$ having the value **false**. Is there a different assignment of truth values to the variables that gives the same result?

The next several problems ask you to translate a variety of rules, expressed in some other manner, into the language of propositional logic. These can all be done using only the

standard connectives $\wedge, \vee$ and $\neg$, but you might find it helpful to use an occasional $\rightarrow$ in your formulas.

10. (Can you get in?) The following is a verbatim extract from academic regulations posted a few years back on the University of Missouri website. (The posting has since been modified, removing the interesting ambiguity in the version quoted here.)

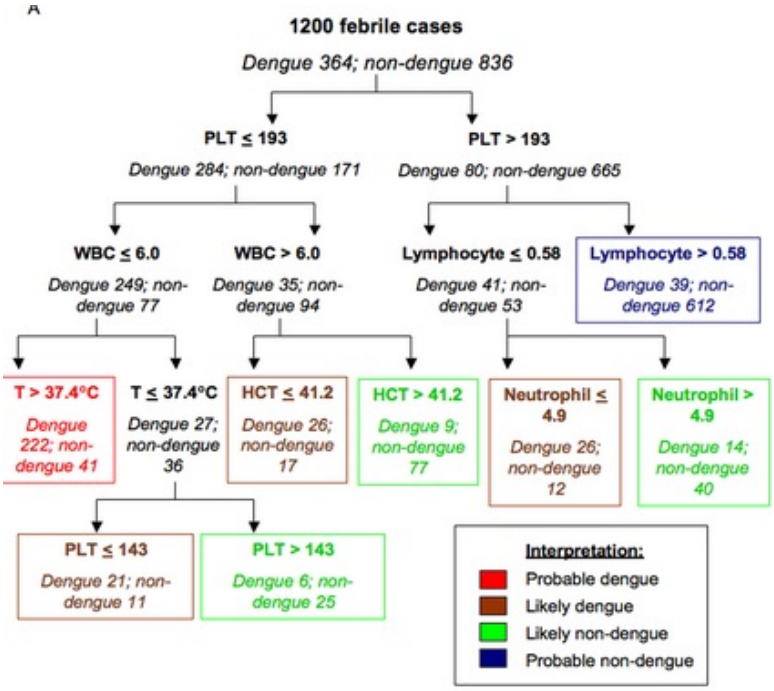
    > Incoming freshmen are eligible for automatic admission to the Honors College upon submission of an application, if they have 29 or higher on the ACT or 1280 on the SAT and are in the top 10 percent of their high school graduating class. Students from high schools that do not rank will be automatically eligible if their core GPA is greater than 3.70.

    The most natural interpretation of the language of the rule is that a student with a GPA higher than 3.70, graduating from a high school that does not rank its graduates, is automatically eligible, regardless of their standardized test scores. That is probably not what was intended, because why should test scores matter only in high schools that compute a class rank?

    Select variables to associate with the atomic propositions in these criteria. (One of the variables should denote 'the student's high school ranks its graduates', and another should be 'the student's ACT score is at least 29'. There should be five variables in all.) Then write two propositional formulas using these variables that are **true** if the student is eligible under these criteria and **false** otherwise. One of the formulas should capture the 'natural interpretation' described above, and the other the intended interpretation. [5]

11. (Do you have Dengue Fever?) The figure below is reproduced from a research article 'Decision tree algorithms predict the diagnosis and outcome of Dengue Fever in the early stage of illness'.

[5] Students to whom I assigned this problem in the past have pointed out other ambiguities to me: For example, you have to have '29 or higher on the ACT' or '1280 on the SAT'. The rule does not say '1280 or higher', which allows for the interpretation that you have to hit 1280 right on the nose. Should the first condition '29 or higher on the ACT or 1280 on the SAT and are in the top 10 percent of their high school graduating class' be read as $(p \vee q) \wedge r$ or as $p \vee (q \wedge r)$? One of these makes much more sense than the other. For these additional ambiguities, assume the most plausible interpretation.




A positive diagnosis ('Probable dengue' or 'Likely dengue' in the legend) is made based on the platelet count (PLT), hematocrit (CT), white blood cell count (WBC) and body temperature (T). Choose appropriate interpretations for the propositional variables (one of them, for example, should be '$WBC \leq 6.0$') and write a propositional formula using these variables that is **true** if and only if the diagnosis is positive.

12. (Are you kosher?) The following is an extract from the Bible (Leviticus 11):

> The Lord spoke again to Moses and to Aaron, saying to them, "Speak to the sons of Israel, saying, 'These are the creatures which you may eat from all the animals that are on the earth. Whatever divides a hoof, thus making split hoofs, and chews the cud, among the animals, that you may eat. Nevertheless, you are not to eat of these, among those which chew the cud, or among those which divide the hoof: the camel, for though it chews cud, it does not divide the hoof, it is unclean to you. Likewise...the pig, for though it divides the hoof, thus making a split hoof, it does not chew cud, it is unclean to you.
>
> 'These you may eat, whatever is in the water: all that have fins and scales, those in the water, in the seas or in the rivers, you may eat. But whatever is in the seas and in the rivers that does not have fins and scales among all the teeming life of the water, and among all the living creatures that are in the water, they are detestable things to you, and they shall be abhorrent to you; you may not eat of their flesh, and their carcasses you shall detest. Whatever in the water does not have fins and scales is abhorrent to you.' "

Introduce propositional variables encoding 'this animal lives on land', '...lives in the water', '..has scales', '...is okay to eat', *etc.* There should be seven variables in all. Write a propositional formula using these variables that encodes this commandment. Observe that God has helpfully provided Moses and Aaron with some examples illustrating the rules, but your answer should not refer to camels or pigs. By the way, this is not an exhaustive list of what is and isn't okay to eat; the actual biblical commandment goes on to discuss birds, bugs, and reptiles. As a result, your answer will admit some assignments that aren't consistent with the complete rule.

13. (Are you being served?) Figure 1.5 shows another sort of commandment.


Figure 1.5: No shoes

Once again, identify the atoms in the proposition to encode by variables, and write a propositional formula equivalent to the rule. Be careful—it doesn't say that if you ARE wearing shoes and a shirt, then you get service. You might not be wearing pants.

14. Construct the truth table for the formula $r \vee (\neg p \vee \neg(q \wedge p))$ that we considered earlier. Is this formula satisfiable? Is it a tautology?

15. How many rows does the truth table you constructed in the last problem have? The examples in Exercises 10 and 11 each have five variables. Without actually constructing the truth table, find how many rows the truth tables for these formulas have. How did you figure this out? What is the general rule?

16. Consider again the college admission problem in Exercise 10. Without constructing the truth table, tell how many satisfying assignments there are—that is, how many rows of the truth table have **T** in the final column? There are two different answers for this question, depending on which of the two interpretations of the rule you use.

17. In each part of the problem below, tell whether the formula is satisfiable, or a tautology, or a contradiction. You can always solve these problems by explicit construction of a

truth table, but it is often more efficient to talk your way through it, reasoning from the formula itself ('...what would it take to make this formula true?') to constraints on the values of the variables.

*(a)* $(p \wedge q) \wedge (\neg p \vee \neg q)$

*(b)* $r \vee (p \vee \neg(q \wedge p))$

*(c)* $(p \wedge q) \vee (\neg p \vee r)$

*(d)* $(p \wedge q) \rightarrow (\neg p \vee r)$

*(e)* $(p \wedge q) \oplus (\neg p \vee r)$

*(f)* $(p \wedge q) \rightarrow (p \vee q)$

*(g)* $(p \vee q) \rightarrow (p \wedge q)$

18. Consider the two formulas

$$p \rightarrow ((p \rightarrow q) \rightarrow q), (p \rightarrow (p \rightarrow q)) \rightarrow q$$

*(a)* One of these is a tautology, and the other is not. Determine which is the tautology. Is the formula that is not a tautology satisfiable? If so, give a satisfying assignment; if not, explain why not.

*(b)* Write a formula that is equivalent to the one that is not a tautology, and that is as simple as possible.

### 1.8.3 Equivalence

19. Prove the last identity (the distributive law) in Table 1.6, using either a truth table or a 'talk it through' argument.

20. Prove the following identities involving the connective $\rightarrow$ .

*(a)* $p \vee q \equiv \neg p \rightarrow q$.

*(b)* $p \wedge q \equiv \neg(p \rightarrow \neg q)$.

*(c)* $(p \rightarrow q) \wedge (p \rightarrow r) \equiv p \rightarrow (q \wedge r)$

*(d)* $(p \rightarrow r) \vee (q \rightarrow r) \equiv (p \wedge q) \rightarrow r$.

21. We saw that the connectives $\vee$ and $\wedge$ obey the associative and commutative laws. What about the connectives $\rightarrow$, $\leftrightarrow$ and $\oplus$? (There are six different questions here: for each of the three connectives, you have to determine whether it is associative, and whether it is commutative.)

22. Let $\phi, \rho$ be formulas, and let $p$ be a variable appearing in $\phi$. We denote by $\phi[p \mapsto \rho]$ the formulas that results when every occurrence of $p$ in $\phi$ is replaced by $\rho$. As we remarked in the text, if

$$\phi \equiv \psi$$

then

$$\phi[p \mapsto \rho] \equiv \psi[p \mapsto \rho].$$

*(a)* Let $\phi$ be a tautology. Is $\phi[p \mapsto \rho]$ necessarily a tautology?

*(b)* Same question, with 'tautology' replaced by 'satisfiable formula'.

*(c)* Same question, with 'tautology' replaced by 'contradiction'.

23. Find the duals of the formulas in parts (a)-(c) of Exercise 17

24. Suppose you are given the truth table for a formula $\phi$. Describe general procedure for obtaining the truth table for the dual $\phi'$. It's possible to figure out such an algorithm by looking at the way we 'proved' the duality theorem Theorem 1.5.1. If you don't see it, try a few particular cases, constructing the truth tables for formulas together with their duals, and seeing if you can discern the rule.

25. Find DNF formulas equivalent to those of Exercise 17. Then find equivalent CNF formulas.

26. Find a CNF formula for the soup-and-salad example that is even simpler than the one given in the example in the text.

27. Can a formula be simultaneously a DNF and a CNF formula? The answer is yes! Give at least three different examples.

28.* Describe an algorithm for determining if a formula in DNF is satisfiable. First describe how to determine if a conjunction of literals is satisfiable. Then show how to use this to determine if the disjunction of such conjunctions is satisfiable. The algorithm that you produce should be *fast* in the sense that it only requires a single scan of the formula, so that the time required to determine satisfiability of a DNF formula with $n$ literals is proportional to $n$.

29.* The preceding problem implies that there is an algorithm for determining if *any* formula is satisfiable: Convert it to an equivalent DNF formula, and apply the algorithm for satisfiability of DNFs. How fast is this algorithm?

30.* Our algorithm form converting a formula to an equivalent formula in CNF proceeded by using truth tables, first extracting a DNF formula for the either the dual or the negation of the original formula. The purpose of this problem is to outline another method for putting a formula in CNF that proceeds by directly transforming the formula through basic identities.

*(a)* Show how, by use of the distributivity of $\vee$ over $\wedge$, a formula in DNF can be converted to CNF. (First practice on $(p_1 \wedge \neg p_2) \vee (q_1 \wedge q_2)$).)

*(b)* Use part *(a)* to show how, given, a formula $\phi$ in CNF, we can find a CNF representation of $\neg\phi$.

*(c)* Given formulas $\phi$ and $\psi$ in CNF, show how to find a CNF representation of $\phi \wedge \psi$. Why is this almost a silly problem?

*(d)* Given formulas $\phi$ and $\psi$ in CNF, show how to find a CNF representation of $\phi \vee \psi$. (Use *(b)*, *(c)*, and DeMorgan's Laws for this.)

*(e)* Now, put it all together and describe an algorithm for converting a formula into an equivalent CNF formula. Demonstrate your algorithm with the formula $\neg((p \wedge q) \vee \neg(p \wedge r)) \vee (q \wedge r)$.

31.* The purpose of this exercise is to show that the identities in Tables 1.6 and 1.7 are *complete,* in the sense that every propositional identity can be derived from them. That is, there are no 'missing' identities.

To set the stage, we have to be a little bit more precise about what it means to derive an identity from other identities. Propositional identities follow some basic familiar rules of algebra:

- If $\phi \equiv \psi$ is an identity, then so is $\psi \equiv \phi$.
- If $\phi \equiv \psi$ and $\psi \equiv \rho$ are identities, then so is $\phi \equiv \rho$.
- If $\phi \equiv \psi$ is an identity, and $\rho$ is any formula, $\phi \vee \rho \equiv \psi \vee \rho$, $\phi \wedge \rho \equiv \psi \wedge \rho$, and $\neg\phi \equiv \neg\psi$ are identities.
- Suppose $\phi \equiv \psi$ is an identity. Let $p$ be a variable, $\rho$ a formula, and $\phi[p \mapsto \rho]$, $\psi[p \mapsto \rho]$ the formulas that result when every occurrence of $p$ in $\phi$ and $\psi$ is replaced by $\rho$. Then
$$\phi[p \mapsto \rho] \equiv \psi[p \mapsto \rho]$$
is an identity. (See Exercise 22).

*(a)* Show that a formula $\phi$ in CNF is a tautology if and only if every clause of literals
$$L_1 \vee \cdots \vee L_k$$
that it contains has $L_i = p$ and $L_j = \neg p$ for some variable $p$ and indices $i, j$.

*(b)* Now use the results of the preceding problem along with part *(a)* to show that if $\phi$ is a tautology, then $\phi \equiv \mathbf{T}$ can be derived from the identities in Tables 1.6 and 1.7, along with the above rules.

*(c)* Now suppose $\phi \equiv \psi$ is an identity. Then $\phi \leftrightarrow \psi$ is a tautology (why?). From part *(b)* above, we can derive the identity
$$(\phi \leftrightarrow \psi) \equiv \mathbf{T}.$$

Show that from this we can derive the identity

$$\phi \wedge \psi \equiv \phi.$$

(HINT: AND both sides of the identity with $\phi$.)

*(d)* Show similarly that we can derive

$$\phi \wedge \psi \equiv \psi$$

and conclude that we can derive $\phi \equiv \psi$.

32. The if-then-else statement in the Python code in Figure 1.1 at the beginning of the chapter can be represented as a *binary decision tree* (Figure 1.6). Each interior node of such a tree is labeled by a boolean variable (or, as in this example, an atomic boolean formula) and branches to another node, depending on whether the variable has the value **true** or **false**. The leaves of the tree are labeled True or False. The tree in Exercise 11 provides another example, although one with more than two possible outcomes at the leaves.

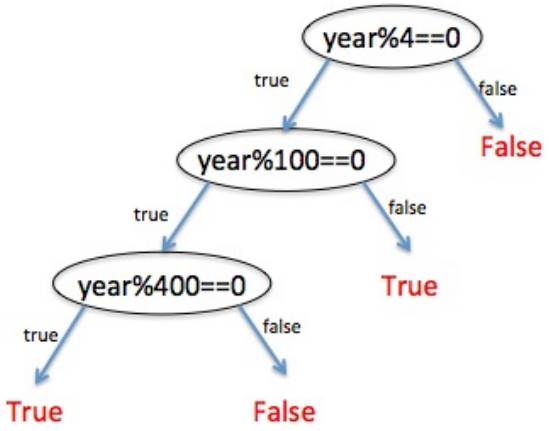



Figure 1.6: A binary decision tree derived from the Python code in Figure 1.1.

*(a)* The essence of the example at the start of the chapter is to show that the binary decision tree represented by this Python code can be replaced by a single propositional formula. Describe how to turn *any* binary decision tree into an equivalent propositional formula. Illustrate your method with the tree shown in Figure 1.7

*(b)* Conversely, describe how to convert any propositional formula into an equivalent binary decision tree. Illustrate your method with any of the formulas from Exercise 17.

*(c)* Describe how to quickly determine from a binary decision tree whether it represents a tautology, or a satisfiable formula.

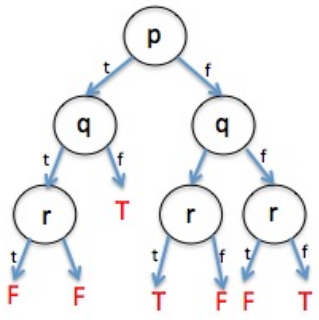



Figure 1.7: Another binary decision tree.

### 1.8.4 Complete sets of connectives

33. Find formulas, using only $\neg$ and $\wedge$, equivalent to the formulas in Exercise 17. Find equivalent formulas using only $\neg$ and $\vee$.

34. Find formulas, using only NAND, equivalent to $p \to q$, $p \oplus q$, and $p \leftrightarrow q$.

35. Argue that the two connectives $\neg$ and $\to$ form a complete set of connectives. Write formulas using these two connectives equivalent to those in Exercise 17.

36. Suppose we define

$$p \text{ NOR } q \equiv \neg(p \vee q).$$

Argue that every propositional formula can be expressed using NOR alone. Then find formulas, using only NOR, equivalent to $p \to q$, $p \oplus q$, and $p \leftrightarrow q$.

37.* Give a convincing argument that neither the pair $\vee, \wedge$, nor the pair $\neg, \oplus$ is a complete set of connectives. This is a hard problem to solve at this stage of the course, but it is worth thinking about. Somehow you will have to identify properties shared by all formulas built using only $\vee$ and $\wedge$ that are not true of every propositional formula, and similarly for the pair $\neg, \oplus$. For starters, think about what happens to the value of such formulas when we change the value assigned to one variable from **F** to **T**, and vice-versa.

See Exercise 16 of Chapter 6 for a real proof of these claims.

# Chapter 2

# Applications of Propositional Logic

## 2.1 Digital Logic

### 2.1.1 Computation as Bit-manipulation

> *Attorney: So, let's get you prepared for your testimony tomorrow. Do you know what color my dress is?*
>
> *Client: It's green.*
>
> *Attorney: Wrong! My dress* **is** *green, but the right answer is 'yes'.*

The smallest amount of information you can give, and still give *some* information, is the answer to a single yes-or-no question. It is a common convention to denote the **yes** answer by 1 and the **no** answer by 0, and to call each such unit of information a *bit,* both because it is a tiny little bit of information, and also because it is a **b**inary dig**it**.

A critical insight of Computer Science is that *all* information can be encoded as sequences of bits.

An example (which long predates computers) is the Braille alphabet for the blind, which represents each text character as a grid of 6 dots, some of which are raised from the surface of the page. (See Figure 2.1.) If we think of the raised dots as 1s and the unraised dots as 0s, and read from left to right starting at the top row, then the letter 'a' is encoded by the bit sequence 100000, the letter 'z' by 100111. The ubiquitous seven-segment displays that you see on every sort of electronic device (Figure 2.2) encode each of the ten decimal digits by a sequence of seven bits: If we denote a lighted segment by 1 and an unlighted segment by 0, and read the segments from left to right starting at the top, then, for example, '1' is encoded by 0010010 and '4' by 0111010.

But even very complex information that you do not ordinarily think of as textual or numeric can be encoded by sequences of bits. A video (without the sound) consists of a sequence of frames, each frame is a grid of pixels, each pixel a collection of three numbers giving the relative quantities of red, green and blue in the pixel, and each number in turn encoded by a sequence of bits.

Since all the information a computer manipulates is represented by sequences of bits, you can view any computation as a process that takes as its input a sequence $a_1 a_2 \cdots a_m$ of bits, and produces as its output another sequence $b_1 \cdots b_n$ of bits. Let us look at a very simple example, called a *full adder*: here we take a sequence $a_1 a_2 a_3$ of bits as an input, and compute the sum

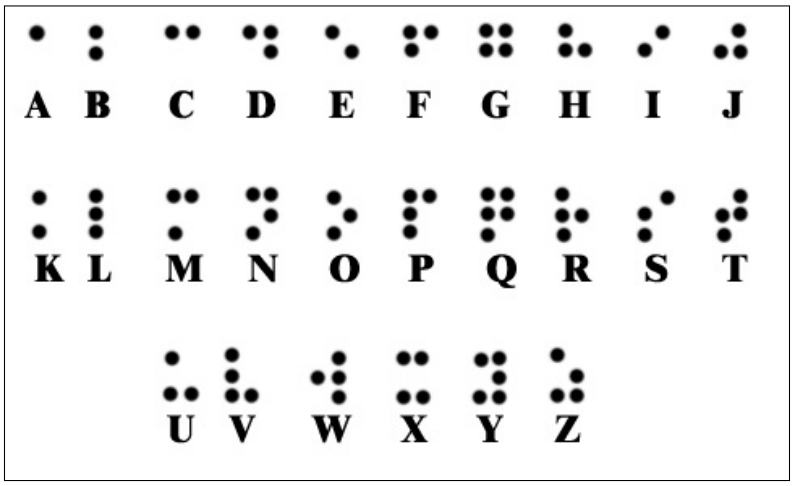



Figure 2.1: *The Braille Alphabet for the blind. Each letter is encoded as a sequence of six bits.*

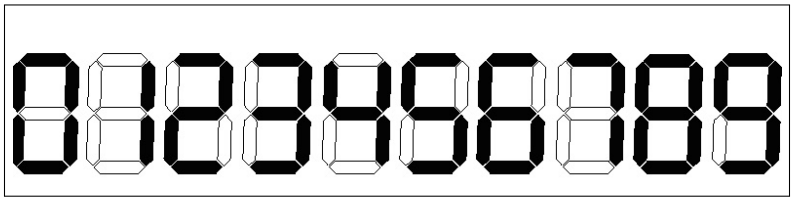


Figure 2.2: *Decimal digits on a seven-segment display. Each digit is encoded as a sequence of seven bits.*

$a_1 + a_2 + a_3$. The possible values for the sum are 0,1,2,3. We will encode each of these possibilities as a sequence of two bits:[1]

$$\begin{array}{ccc} 0 & \leftrightarrow & 00 \\ 1 & \leftrightarrow & 01 \\ 2 & \leftrightarrow & 10 \\ 3 & \leftrightarrow & 11 \end{array}$$

We can represent the process by a table giving the complete input-output relation. (Table 2.1.1.) The table itself depends on how we choose to encode the inputs and outputs by bits; here we have used the standard encoding of numbers by bits.

This, of course, is a truth table—just think of the 1s as **T**s and the 0s as **F**s. As we noted in the previous chapter, the output columns in such a table can be represented by propositional formulas whose variables represent the inputs, and we can read off these formulas in disjunctive normal form.

[1]This is the standard binary encoding of integers by bits, which we will discuss in detail in Chapter 7: The right-hand digit is the 1s position, the left-hand digit is the 2s position; if we had another digit further to the left it would be the 4s position, *etc.*

| input 1 | input 2 | input 3 | output 1 | output 2 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Table 2.1: Input-output table for the full adder

In this case, we'll represent the three inputs by $p, q, r$ respectively. The output formulas are:

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

$$(\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge \neg r) \vee (p \wedge q \wedge r).$$

These formulas can be simplified: you can show that the first of these formulas is equivalent to

$$(p \wedge q) \vee (p \wedge r) \vee (q \wedge r)$$

and the second to

$$p \oplus q \oplus r.$$

(In connection with this last formula, see Exercise 21 of Chapter 1.)

### 2.1.2 Truth tables realized by networks of logic gates

We saw above that any computation can be realized by a truth table, and thus by a collection of propositional formulas. Each propositional formula can in turn be realized in hardware. The idea is that each logical connective is simulated by a simple device, called a *logic gate.* The gates, in turn, are connected together in a network whose inputs are the values of the variables and whose output is the resulting value of the formula. The exact implementation of the gates in hardware is irrelevant. In principle, the gate could be a mechanical spring-driven device, like an old-fashioned clock, or run on steam, but typically the gates are small electrical circuits, and the logical values **true** and **false** are represented by high and low voltage levels. Thus, for example, the connective NAND is implemented by a circuit with two inputs and a single output. Setting the voltage at both of the inputs high switches the voltage at the output low, while setting either input voltage low results switches the output voltage high. Standard graphical symbols for logic gates are shown in Figure 2.3.

A full adder circuit is thus constructed by building the two formulas from these basic gates. (Figure 2.4)

The existence of small, functionally complete, sets of connectives means that we can construct all such circuits using only a few basic gate types—in fact, using only a single gate type. As we

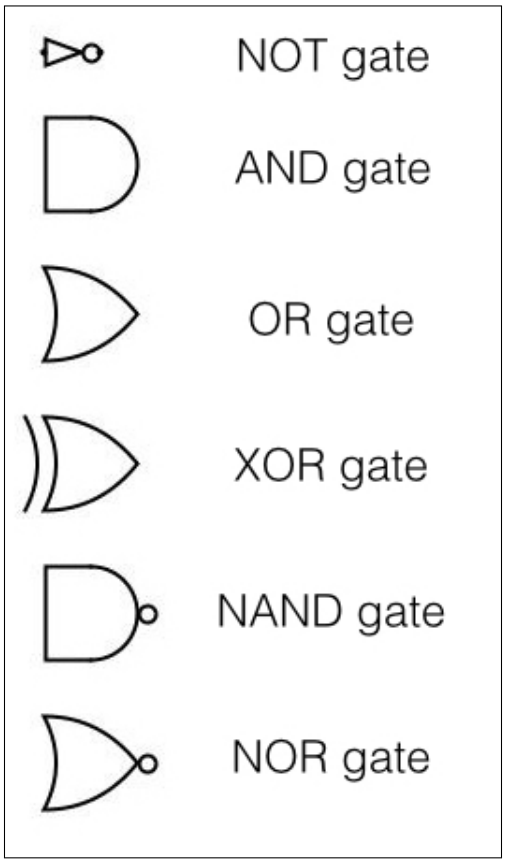



Figure 2.3: Standard Gate Symbols

saw before, the NAND connective suffices all by itself, and in fact a NAND gate is a particularly simple hardware component to construct. Figure 2.5 shows the construction of an exclusive-or gate from NAND gates, using the formula we computed in 1.6.

In the same manner *any* input-output table can be realized by a network consisting of many copies of this one type of gate. So in a very real sense, any computation at all can be carried out by repeatedly performing this single simple computation. We will return to this point in the exercises for this chapter, and later in the book.

## 2.2 Liar Puzzles

There is a large class of puzzles about an island in which there are two groups of inhabitants: Members of one group always lie, members of the other always tell the truth. The rules of this world are such that the liars are just as reliable as the truth-tellers: They are not trying to deceive you; they are not trying to do *anything*. They just can't help themselves; every statement they make is false.

The puzzles below ask you to determine which of the two groups the characters belong to, based on the statements they make, or to devise a question that will extract accurate information

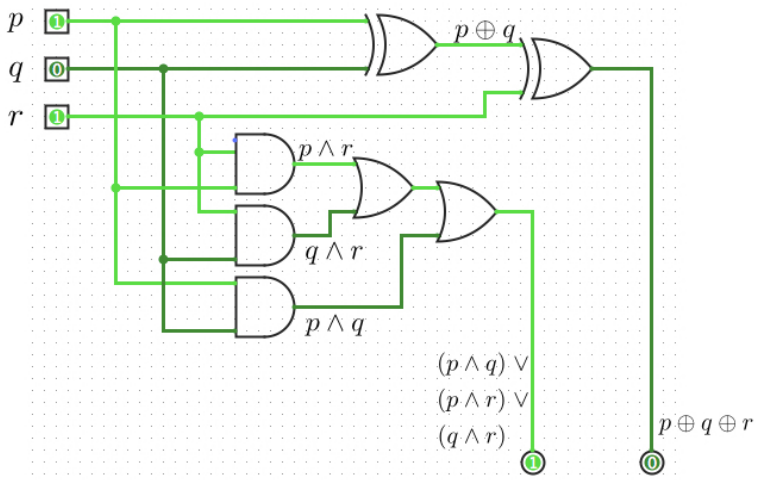



Figure 2.4: The Full Adder Circuit

from an inhabitant, whether he is a truth-teller or a liar. The puzzles themselves, along with the Knights-and -Knaves/Humans-and-Vampires back story, are all due to the mathematician Raymond Smullyan.

This is not really a practical application; but questions about statements that assert their own falsehood play a serious role in logic and Computer Science, as we will see later.

### 2.2.1 Some Knights-and-Knaves Puzzles

Knights only make true statements. Knaves only make false statements.

#### Puzzle 1.

$A$ says, '$B$ and I are both knights'. $B$ says '$A$ is a knave'. Determine, if possible, which groups $A$ and $B$ belong to.

#### Puzzle 2.

You meet an islander. Ask her a single question to determine whether she is a knight or a knave. (Observe that 'Are you a knight?' doesn't work!)

#### Puzzle 3.

You come to a fork in the road. An inhabitant of the island stands at the fork. You know that one of the roads leads to the village, where there is a decent restaurant and a comfortable hotel, and that the other road leads to a crocodile-infested swamp. You have to ask *one* question whose answer will tell you which road leads to the village. What do you ask?

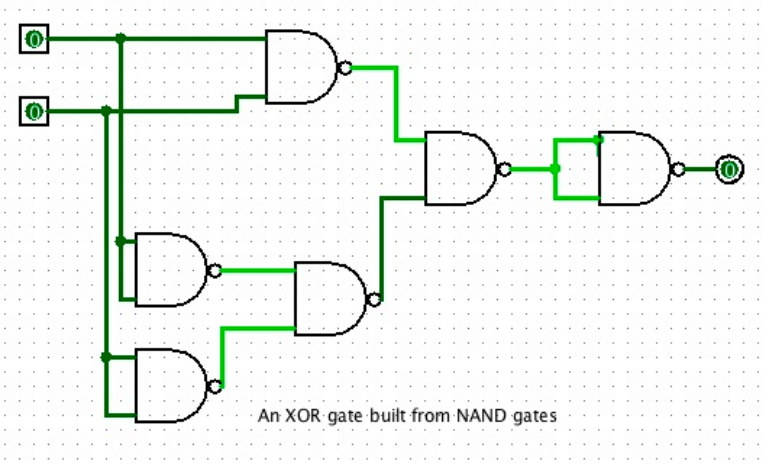



Figure 2.5: An exclusive-or gate built from NAND gates

### 2.2.2 A Vampire Puzzle

In Transylvania, everyone is a human or a vampire, and either sane or insane. Humans always say what they believe to be true, and vampires always say what they believe to be false. Sane beings only believe what is true, and insane beings only believe what is false. As a result, sane humans and insane vampires always tell the truth, and sane vampires and insane humans always lie.

Exactly one of the sisters Lucy and Minna is a vampire. They have the following exchange with Inspector Craig of Scotland Yard:
*Lucy:* We are both insane.
*Craig:* Is that true?
*Minna:* Of course not!

Which one of the sisters is the vampire?

### 2.2.3 Modeling the puzzles with propositional logic

Here we will use the methods of the first chapter to solve these puzzles. We begin with Knights-and-Knaves puzzles. For each individual $A, B, C, \ldots$ in the problem, introduce a propositional variable $a, b, c, \ldots$. The variable $a$ is true if $A$ is a knight and false if $A$ is a knave; similarly $b, c, \ldots$.

If $A$ asserts that a proposition $x$ is true (or, what is the same thing, answers 'yes' to the question 'Is $x$ true?') then the proposition $a \leftrightarrow x$ is true. This is because if $A$ is a knight, and $A$ asserts $x$, then $x$ is true, while if $A$ is a knave and $A$ asserts $x$, then $x$ is false. Similarly, if $A$ denies $x$ (answers 'no' to 'Is $x$ true') then $a \leftrightarrow x$ is false. So for any proposition $x$, $a \leftrightarrow x$ is equivalent to '$A$ asserts $x$'.

How does this help us solve the puzzles? Look at Puzzle 1 : The data in the problem tells us that both

$$a \leftrightarrow (a \wedge b), b \leftrightarrow \neg a$$

are true. The problem is to determine what values of $a$ and $b$ lead to both these propositions being true. We can solve this with truth tables.

| $a$ | $b$ | $a \wedge b$ | $a \leftrightarrow (a \wedge b)$ | $b \leftrightarrow \neg a$ |
|---|---|---|---|---|
| T | T | T | T | F |
| T | F | F | F | T |
| F | T | F | T | T |
| F | F | F | T | F |

Table 2.2: Truth table used to solve Puzzle 1

| $a$ | $p$ | $a \leftrightarrow x$ | $x$ |
|---|---|---|---|
| T | T | T | T |
| T | F | F | F |
| F | T | T | F |
| F | F | F | T |

Table 2.3: Truth table used to solve Puzzle 3.

There is only one row of Table 2.2 where both of the last columns have the entry **T**. This corresponds to $A$ knave and $B$ knight, so that is the solution.

Let's do Puzzle 2. The problem is to find a proposition $x$ so that $a \leftrightarrow x$ has the same truth value as $a$. This makes the answer to 'Is $x$ true?' the same as the *correct* answer to 'Are you a knight?' We just need a proposition $x$ that is always true regardless of the value of $a$ So we can ask

"Is two plus two equal to four?"

For Puzzle 3,we'll again calculate with truth tables. Let $p$ be the proposition 'the left road leads to the village'. We need to find $x$ so that $a \leftrightarrow x$ has the same value as $p$. We can then ask 'Is $x$ true?' In Table 2.3 we set the value of $a \leftrightarrow x$ to be identical to $p$, and deduce what $x$ has to be as a result.

From the table we find that proposition $x$ we need for our question must have the same value as $a \leftrightarrow p$, which is equivalent to '$A$ asserts $p$'. So we can ask:

"Would you say that the left road leads to the village?"

In case you don't quite see it: If the islander is a Knave, and the road really leads to the village, she would say that the left road does not lead to the village (since she always lies). Thus she must lie about what she would say, and thus answer the question above "Yes". Note that in our rigid logical world, this is a different question from "Does the left road lead to the village?"

Now let's solve the Vampire puzzle. For each individual $A, B, C \ldots$, we introduce two variables $a, a', b, b', c, c', \ldots$, where $a$ means '$A$ is human', and $a'$ means '$A$ is sane', and similarly for the other pairs of variables. Observe that '$A$ always tells the truth' is equivalent to $A$ being both sane and human or neither sane nor human, that is, to $a \leftrightarrow a'$. Thus '$A$ asserts $x$' is equivalent to

$$(a \leftrightarrow a') \leftrightarrow x.$$

| $l$ | $l'$ | $m$ | $m'$ | $l \leftrightarrow l'$ | $m \leftrightarrow m'$ | $l' \vee m'$ | $(l \leftrightarrow l') \leftrightarrow \neg(l' \vee m')$ | $(m \leftrightarrow m') \leftrightarrow (l' \vee m')$ |
|---|---|---|---|---|---|---|---|---|
| T | T | T | T | T | T | T | F | T |
| T | T | T | F | T | F | T | F | F |
| T | T | F | T | T | F | T | F | F |
| T | T | F | F | T | T | T | F | T |
| **T** | **F** | **T** | **T** | **F** | **T** | **T** | **T** | **T** |
| T | F | T | F | F | F | F | F | T |
| T | F | F | T | F | F | T | T | F |
| T | F | F | F | F | T | F | F | F |
| **F** | **T** | **T** | **T** | **F** | **T** | **T** | **T** | **T** |
| F | T | T | F | F | F | T | T | F |
| F | T | F | T | F | F | T | T | F |
| **F** | **T** | **F** | **F** | **F** | **T** | **T** | **T** | **T** |
| F | F | T | T | T | T | T | F | T |
| **F** | **F** | **T** | **F** | **T** | **F** | **F** | **T** | **T** |
| F | F | F | T | T | F | T | F | F |
| F | F | F | F | T | T | F | T | F |

Table 2.4: Truth table for the Vampire puzzle.

In our problem, there are four variables $l, l', m, m'$. The sisters' statements are

$$(l \leftrightarrow l') \leftrightarrow \neg(l' \vee m')$$

$$(m \leftrightarrow m') \leftrightarrow (l' \vee m').$$

In other words, Minna is asserting that at least one of them is sane, and Lucy the negation of this—that they are both insane. Let's look at the truth table for these two propositions:

There are four rows in which the last two columns are true. The first such row corresponds to an assignment in which both Lucy and Minna are human, which we can rule out, since we know that one of them is a vampire. Another of these rows has them both being vampires, which we similarly rule out. The remaining two solutions have Lucy being a vampire and Minna a human, so that is the solution. We cannot tell from the information given whether or not they are sane, but we know that they are either both sane or both insane.

If you find yourself thinking that constructing a truth table with over one hundred entries is overkill, you are quite right. We could have deduced the correct solution by thinking it through: If Minna is a vampire, she is either sane or insane. If she is insane, then what she asserts — namely that at least one of them is sane – is true. That means that Lucy is a sane vampire, contradicting our assumption that exactly one of them is a vampire. If Minna is a sane vampire, then what she asserts is false, and thus she would have to be insane—a contradiction. So we cannot have a solution in which Minna is a vampire and Lucy is human.

But the question for us, as computer scientists, is whether we can automate this reasoning so that a computer can solve the puzzles, and this is something that the truth tables provide. While

the truth tables are easy to program, they quickly become impractically large as the number of variables increases. We will return to this point in the next section.

## 2.3 Satisfiability Problems

Satisfiability problems have the following form: You're given a propositional formula $\phi$, and you have to find an assignment of truth values to the variables that makes $\phi$ true, or to determine that no such assignment exists. In other words, you have to determine whether $\phi$ is satisfiable, and if it is, to find a satisfying assignment.

Below we give examples of several such problems. Each example is a toy problem, but each also belongs to a class of problems that arise in practice. We'll discuss how to translate them into the form described, and say something about how to go about solving them.

### 2.3.1 A Scheduling Problem

An overly eager student wants to sign up for seven courses: English, Math, Chinese, History, Computer Science, Philosophy and Music. Each class meets just once a week, in either a long morning session, or a long afternoon session, Monday through Friday. The student is trying to make a schedule that meets the following constraints:

- The student does not want to schedule any classes on Wednesday, a day he prefers to stay in bed.
- The English professor allows students to drink beer in class, so our student would like to schedule English for Friday afternoon.
- Philosophy is only taught on Monday.
- History is only taught in morning sessions.
- The student would like to schedule Chinese and Music on the same day.

The problem is to find a schedule that meets these constraints.

### 2.3.2 Sudoku: A Combinatorial Design Problem

You have probably Sudoku puzzles. An example puzzle, together with its solution, is shown in Figure 2.3.2. [2]

The problem is to fill in the boxes in the grid with the integers 1 through 9 in such a manner that no number appears twice in the same row, nor twice in the same column, nor twice in any of the outlined 3x3 subgrids.

[2]Source for these images: "Sudoku-by-L2G-20050714 solution" by en:User:Cburnett - Added red numbers based on en:Image:Sudoku-by-L2G-20050714.svg. Licensed under CC BY-SA 3.0 via Commons -https://commons.wikimedia.org/wiki/File:Sudoku-by-L2G-20050714_solution.svg#/media/File:Sudoku-by-L2G-20050714_solution.svg.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

Figure 2.6: A Sudoku puzzle and its solution

### 2.3.3 Modeling the examples as Satisfiability Problems

The input to a satisfiability problem is a boolean formula $\phi$, and the solution is either a satisfying assignment to the variables in the formula, or a determination that no such assignment exists.

Both of the problems given above can be modeled as satisfiability problems. In each case, we can express the required condition as a collection of separate constraints that must simultaneously be satisfied. This suggests that the condition ought to be represented in conjunctive normal form.

Let's start with the scheduling problem. There are seven courses and eight time slots. Our formula will have 56 different variables, one for each combination of course and time slot. We'll denote the variables in this manner:

$$\mathrm{Mu}_{m,Th}, \mathrm{Ph}_{a,F},$$

etc., where the first of these is true if Music is scheduled for Thursday morning, and the second is true if Philosophy is scheduled for Friday afternoon. We construct the clauses of $\phi$ as follows:

- Each of the seven subjects must be scheduled at one of the available times. This gives a clause

  $$\mathrm{Mu}_{m,M} \vee \mathrm{Mu}_{a,M} \vee \mathrm{Mu}_{m,T} \vee \mathrm{Mu}_{a,T} \vee \mathrm{Mu}_{m,Th} \vee \mathrm{Mu}_{a,Th} \vee \mathrm{Mu}_{m,F} \vee \mathrm{Mu}_{a,F},$$

  for Music, and six others just like it for the remaining subjects.

- Different classes cannot be scheduled in the same time slot. This gives, for instance, a clause

  $$\neg\mathrm{Mu}_{a,T} \vee \neg\mathrm{Ch}_{a,T},$$

  which says that Music and Chinese cannot both be scheduled for Tuesday afternoon. Altogether, there are $21 \times 8 = 168$ such clauses, since there are 21 different pairs of subjects and 8 time slots.

- A single class cannot be scheduled at two different time slots. This gives rise to clauses like

$$\neg \mathrm{Mu}_{a,T} \vee \neg \mathrm{Mu}_{m,F}.$$

  There are 196 such clauses in all.

- English has to be taken Friday afternoon:

$$\mathrm{En}_{a,F}$$

- Philosophy has to be taken Monday:

$$\mathrm{Ph}_{m,M} \vee \mathrm{Ph}_{a,M}$$

- The constraint about Chinese and Music can be expressed by the conjunction of eight formulas, the first two of which are

$$\mathrm{Ch}_{m,M} \to \mathrm{Mu}_{a,M}, \mathrm{Ch}_{a,M} \to \mathrm{Mu}_{m,M},$$

  and the remaining six of which use the three other available days. Furthermore, an implication $a \to b$ can be rewritten as $\neg a \vee b$.

So the CNF for our formula has 56 variables and 382 clauses, the majority of which just specify that there is a unique spot in the schedule for each class. The large size of the CNF, compared to the relatively succinct original specification of the problem, seems to have made things harder, rather than easier. In fact, as we'll see below, it is the key to automating the solution.

For Sudoku, we have $9 \times 9 \times 9 = 729$ boolean variables, which we'll denote $p_{i,j,k}$ where $i, j, k$ are integers between 1 and 9 inclusive. $p_{i,j,k}$ is true if the cell in row $i$ and column $j$ contains the number $k$. The constraints can be formulated as CNF clauses in the following way:

- Every cell contains a label. This gives 81 clauses

$$p_{i,j,1} \vee p_{i,j,2} \vee \cdots \vee p_{i,j,9}$$

  one for each choice of $i, j$. We can denote each of these with the more compact notation

$$\bigvee_{k=1}^{9} p_{i,j,k}.$$

- No cell contains two labels. This gives a clause

$$\neg p_{i,j,k} \vee \neg p_{i,j,k'}$$

  for all $i, j, k, k'$ with $k \neq k'$. There $81 \times 36 = 2916$ such clauses.

- Every row contains every value. For example, to say row 3 contains 7, we have the clause

$$\bigvee_{j=1}^{9} p_{3,j,7}.$$

  There are 81 such clauses, one for each row and each value .

- Similarly we get 81 clauses saying that each column contains every value. For instance

$$\bigvee_{i=1}^{9} p_{i,5,4}$$

says column 5 contains a 4.

- And, similarly, there are 81 clauses saying that each subgrid contains every value. The following clause says that the central subgrid contains 2:

$$\bigvee_{i=4}^{7}\bigvee_{j=4}^{7} p_{i,j,2}.$$

- Finally, there is a small number of additional clauses that give the initial configuration of the puzzle. For instance, $p_{3,3,3}$ says that the cell in the third row of the third column contains three.

### 2.3.4 Satisfiability Solvers

A *satisfiability solver,* also called a *SAT solver,* is a program for solving satisfiability problems. The input formula is usually given in conjunctive normal form, and the output is either a satisfying assignment, or a message that the formula is unsatisfiable.

Most SAT solvers in use today share a standard format for encoding CNF formulas as text files, called the DIMACS format. The file starts with optional comment lines, each of which begins with the letter `c`. The comment lines are followed by a line of the form

`p cnf` *num1 num2*

where *num1* is the number of variables and *num2* is the number of *clauses,* or conjuncts. The remaining lines of the file contain the conjuncts themselves. Let us denote the variables $p_1, p_2, \ldots$. If the literal $p_i$ appears in the clause, then the number $i$ appears in the line. If the literal $\neg p_i$ appears in the clause, then $-i$ appears in the line. Each such line is terminated by 0.

For example, recall that our soup-and-salad format in CNF is

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (q \vee r).$$

We'll treat $p$ as $p_1$, $q$ as $p_2$, and $r$ as $p_3$.

There are three variables and three clauses. The complete text file is given by:

```
c CNF file for soup and salad
c variables: order soup, order salad, it's freezing
p cnf 3 3
1 2 0
-1 -2 0
2 3 0
```

This example was given as input to a SAT solver called `sat4j`. The program produced dozens of lines of output reporting on what the program was doing, but the real result is contained in the last three lines:

```
s SATISFIABLE
v -1 2 -3 0
c Total wall clock time (in seconds) : 0.0070
```

This tells us that the formula is satisfiable, and that a satisfying assignment is $p \mapsto$ **false** (don't order the soup), $q \mapsto$ **true** (order the salad), and $r \mapsto$ **false** (it's not freezing).

If we wanted to find a different satisfying assignment, we would have to add a clause saying that the satisfying assignment found in this first run is *not* made. We do this by adding the following line to our specification, and running the program on the new specification.

```
1 -2 3 0
```

The result is

```
s SATISFIABLE
v -1 2 3 0
c Total wall clock time (in seconds) : 0.0090
```

This corresponds to the solution where we don't order the soup, order the salad, and it's freezing outside. If we want to see yet another satisfying assignment, we repeat the process, adding to our specification the negation of the assignment we just found:

```
1 -2 -3 0
```

and get the result

```
s SATISFIABLE
v 1 -2 3 0
c Total wall clock time (in seconds) : 0.0090
```

which corresponds to ordering soup, not ordering the salad, and it being freezing outside.

What if we keep going? We will add one more line to our specification, the negation of this last satisfying assignment:

```
-1 2 -3 0
```

The result is, just as you might expect:

```
s UNSATISFIABLE
c Total wall clock time (in seconds) : 0.01
```

Even in our toy problems of the student schedule and Sudoku, the number of variables and clauses is so large that it is not practical to prepare the text file that encodes the CNF formula by hand. However, it is not hard in either case to generate the CNF encoding with a program, using the original succinct problem descriptions as a guide. We will say more about formalizing such succinct descriptions in Chapter 4.

### 2.3.5 *How do SAT solvers work?

We already know how to test if a formula is satisfiable: construct the truth table for the formula, and see if there is a **T** in the rightmost column. Consider what this method implies for the Sudoku problem: There are 789 variables, and thus the truth table would have $2^{789} \approx 10^{237}$ rows. There is not enough space in the universe to store such a table.

Of course, we do not have to store the entire table—we can just compute one row at a time, and reuse the same storage for each row. But there are still $2^{789}$ rows to compute, and if the result is that the puzzle has no solution, we would have to compute *all* of them. The sun will burn out before this calculation can be completed.

One trick that can be used to speed things up is to tentatively assign truth values to variables one at a time. After each assignment is made, we can write a simplified formula involving fewer variables. If, as a result of this process, we wind up with an unsatisfiable formula, we backtrack to the last assignment we made, and make the opposite assignment. Let us illustrate how this works with CNF formula

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (q \vee r) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r).$$

This is the soup-and-salad problem with additional constraints. Let's begin by choosing a variable and assigning it a value. Say we begin with $q$ and give it the value **T**. Our formula now becomes

$$(p \vee \mathbf{T}) \wedge (\neg p \vee \mathbf{F}) \wedge (\mathbf{T} \vee r) \wedge (p \vee \mathbf{F} \vee r) \wedge (p \vee \mathbf{F} \vee \neg r).$$

Now a clause of the form $\mathbf{T} \vee \psi$ always has the value **true**, so we can just eliminate this clause. A clause of the form $\mathbf{F} \vee \psi$ is equivalent to $\psi$, so we can just erase the **F** from the clause. The resulting simplified formula is

$$\neg p \wedge (p \vee r) \wedge (p \vee \neg r).$$

The literal $\neg p$ all by itself in a clause means that we are forced to assign **F** to $p$ in order to get a satisfying assignment. This gives

$$\mathbf{T} \wedge (\mathbf{F} \vee r) \wedge (\mathbf{F} \vee \neg r),$$

which simplifies to

$$r \wedge \neg r.$$

Of course, this contradiction cannot be satisfied. So we backtrack to the only place where we had a choice to make, when we assigned **T** to $q$, and now change this to **F**. The result is

$$(p \vee \mathbf{F}) \wedge (\neg p \vee \mathbf{T}) \wedge (\mathbf{F} \vee r) \wedge (p \vee \mathbf{T} \vee r) \wedge (p \vee \mathbf{T} \vee \neg r).$$

This simplifies to

$$p \wedge r \wedge (p \vee r) \wedge (p \vee \neg r).$$

The variables $p$ and $r$ now appear as separate clauses, and this forces the assignment of **T** to both $p$ and $r$, so we have found our satisfying assignment of **T** to both $p$ and $r$, and **F** to $q$.

Even though we started with a formula that has only a single satisfying assignment, we did not have to try out too many possibilities to find it. To speed things up, we used the trick of simplifying the formula at each step, eliminating both clauses and variables. This leads to clauses that have

only a single literal, in which the assignment is forced, with the result that there are many possible assignments that we never have to test.

This method, called the DPLL algorithm (for its creators Davis, Putnam, Logemann and Loveland) dates back to the early 1960s, and is at the core of most SAT solvers in wide use today. Much of the speedup that has been achieved in such programs is due to improved strategies for choosing the next unassigned variable to work with, and improved data structures for representing the formulas.

SAT solvers show an interesting gap between theory and practice. In principle, a backtracking solver like this might be forced into testing a large proportion of the $2^n$ possible assignments to a formula with $n$ variables. For formulas with a few hundred or a few thousand variables, this would throw us right back into the 'not enough time in the universe' dilemma. There is no algorithm that is known to solve *every* possible satisfiability problem more efficiently than this, and there is some theoretical support for the belief that no such algorithm exists. Indeed, one of the most important unsolved problems in computer science and mathematics—the *P vs. NP Conjecture*—is whether one can prove that, in a sense that can be made precise, there are no efficient algorithms that can solve every instance of the satisfiability problem. If, as most experts believe, this conjecture is true, it would seem to rule out the possibility of creating practical SAT solvers that run on problems with thousands of clauses and variables.

Nonetheless, practical SAT solvers that give fast answers to enormous problems really do exist—the hard cases seem to rarely arise in practice. We will have more to say about the question of what computers can and cannot do, and what they can do efficiently, at the end of the book.

## 2.4 Historical and Bibliographic Notes

Claude E. Shannon, in his 1940 MIT masters thesis, described how to use propositional logic in the analysis and design of electrical switching circuits, essentially inventing the idea of logic gates described in the text. Shannon's gates were implemented as electromechanical relays, but when electronic computers began to appear, vacuum tubes and, later, transistors, replaced them. At around the same time, V. I. Shestakov in the Soviet Union made essentially the same discovery, also in a graduate dissertation. Independently of both Shannon and Shestakov, in 1937, George R. Stibitz, a scientist at Bell Laboratories, built a two-bit binary adder from relays, a 'play project' that was at the origin of a series of successively more sophisticated relay-based computers.

As mentioned in the text, you don't need electricity to implement logic gates. The Web harbors many zany and earnest projects displaying mechanical implementations of logic circuits, of which this two-bit adder built from Legos is a fairly typical example.

The word 'bit' (in the computer science sense!) as well as the notion of a bit as the smallest unit of information, appeared in print for the first time in 1948 in Claude Shannon's most famous work, laying the foundations of Information Theory. [3]

The logic puzzles of Raymond Smullyan used in the text and in the exercises in Section 2.5.2 are from three collections: *What is the Name of This Book?*, *Forever Undecided*, and *The Lady or the Tiger? And Other Logic Puzzles*. The original story of 'The Lady, or the Tiger?', by Frank Stockton, was published in 1882. Notwithstanding the cheesy over-the-top language of the story,

[3] Shannon attributed this brilliant coinage to John Tukey, who must have had a gift for this sort of thing: He also gave us *software.*

the themes of paradox and unsolvable problem that it evokes will be concerns of ours throughout this book.

The DPLL algorithm was originally described by Davis and Putnam, and then improved by Davis, Logemann and Loveland. See Gomes, *et. al.,* for a modern survey on satisfiability solvers. There is an annual conference at which a competition to find the fastest SAT Solver takes place.

## 2.5 Exercises

### 2.5.1 Digital Logic

The exercises in this section ask you to design some circuits by wiring together logic gates, along the lines of those displayed in Figures 2.4 and 2.5. If you wish, you can simply draw these by hand, but it is more instructive, and a good deal more fun, to use a *logic simulator* program that allows you to create the circuit diagrams on the computer, test their behavior on various inputs, and save them as modules that can be incorporated into larger designs.

1. Design a circuit with four inputs and a single output that has the value 1 if exactly two of the inputs are 1, and that has the value 0 otherwise. You can use any gate types that you wish.

2. Design a circuit with four inputs and two outputs. The first output should have the value 1 if at least two of the inputs are 1, and the second output should have the value 1 if at least three of the inputs are 1. (Note that there are only three possible output configurations: 00, 10, and 11.)

3. A three-input AND gate gives the output 1 if all inputs are 1 and the output 0 otherwise. It is a simple matter to construct a three-input AND gate from the standard two-input AND gates, because $AND(x, y, z) = AND(AND(x, y), z)$. A three input NAND gate computes the negation of a three-input AND gate: that is, it gives output 0 if all the inputs are 1, and the output 1 otherwise.

   *(a)* Show that you cannot build a 3-input NAND gate from a pair of 2-input NAND gates in this manner. That is, show that $NAND(x, y, z)$ is not the same as $NAND(NAND(x, y), z)$.

   *(b)* Build a 3-input NAND gate using only 2-input NAND gates. How many of the 2-input gates did you need?

4. Design a circuit that adds two 2-bit integers and gives the 3-bit sum. Recall from the discussion in the text that the 3-bit sequence $b_2b_1b_0$ represents the integer $4b_2 + 2b_1 + b_0$. So for example, if the four input bits are, in order, $1, 0, 1, 1$, then this will be treated as adding 10 (two) and 11 (three) to give 101 (five). A second version of this problem is given below. Here you are asked to try to solve this problem by creating a propositional formula corresponding to each of the outputs. You can use any gate types that you wish.

5. Another approach to building the 2-bit adder of the preceding problem is to think of it as two one-bit adders chained together: Connect one of the outputs of the first stage to one of the inputs of the second. If you see how this problem works, you should be able to assemble

a $k$-bit adder for any value of $k$. (This problem works best if you use a logic simulator that lets you save circuits as modules that can be incorporated into other circuits.)

6. The circuits we have constructed in the text and the preceding exercises contain no loops—that is, there is never a path from the output of a gate back to its input. As a result, the value at the output of any gate in the circuit can be computed from the inputs. Here you will show that this property may fail to hold if we allow such loops, but that other interesting properties may emerge.

   The circuit shown in Figure 2.7 is called a *latch.* In the illustration, the output is 1, but this fact cannot be determined just by looking at the inputs.

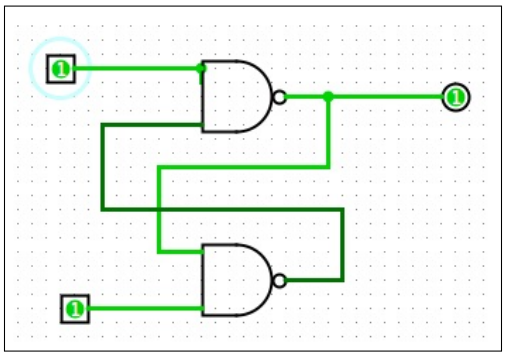


Figure 2.7: A circuit that remembers

   *(a)* What happens to the output when we switch the *bottom* input from 1 to 0? What happens if we then switch this input back to 1?

   *(b)* And now, what happens to the output when we switch the *top* input from 1 to 0, and then back again?

   (The latch with both inputs 1 functions as a 1-bit *memory,* recording which of the two inputs was most recently changed.)

### 2.5.2 Puzzles

The puzzles in this section, like those in Section 2.2, are all taken from the collections published by Raymond Smullyan. All of these can be approached as illustrated in the examples in the text, by first translating the puzzle into propositional logic.

7. *(More Knights and Knaves)*

   *(a)* You approach three inhabitants– A, B and C–of the island of Knights and Knaves. A and B tell you:

   *A:* B and C are both Knights.

   *B:* A is a Knave and C is a Knight.

Determine, if possible, which group each of the three islanders belongs to.

*(b)* The setup here is the same as in part *(a)*. The statements are:

*A:* B is a Knave.

*B:* A and C are of different types.

*(c)* Surprisingly, Knights and Knaves sometimes intermarry, so you cannot always assume, when you encounter a married couple, that both husband and wife are of the same type.You meet a couple, and the husband says, 'If I am a Knight, then so is my wife.' Can you determine what types this man and his wife belong to?

*(d)* This is a continuation of part *(c)*: If an islander says 'If I am a Knight, then $p$', where $p$ is some proposition, what (if anything) can you conclude about the type of the speaker and the truth value of $p$?

8. *(Lady and Tiger Puzzles)* Generations of American schoolchildren were made to read a 19th century short story called 'The Lady, or the Tiger?'. In it, a rather sadistic ancient ruler administers justice by putting the prisoner in an arena and asking him to choose one of two doors. Behind one door is a woman (whom he is supposed to marry on the spot); behind the other is a hungry tiger, who will devour him instantly.

   Smullyan used the story as the basis for a series of logic puzzles. In the puzzles, each door has either a lady or a tiger behind it, but there is not necessarily one of each: there could be two tigers or two ladies. In addition, there are signs on each door, but the messages on the signs are not necessarily true. Your task in these puzzles is to determine, based on the information given, which door the prisoner should open. (Assume that the prisoner prefers instant marriage to a stranger to being eaten by a tiger!)

   To solve these puzzles, model each sign as a propositional formula in two variables: the first variable is interpreted to mean, 'there is a lady behind the first door', and the second variable, 'there is a lady behind the second door'. Then find an expression for the additional condition in the problem about the truth of the signs, and find a satisfying assignment for the variables.

   *(a)* The prisoner knows that both signs are true or both are false. First sign: THERE IS A LADY BEHIND AT LEAST ONE OF THESE DOORS. Second sign: THERE IS A TIGER BEHIND THE FIRST DOOR.

   *(b)* This time the prisoner knows that one of the signs is true, and the other false. First sign: THERE IS A LADY BEHIND THE SECOND DOOR. Second sign: THERE IS A LADY BEHIND ONE OF THESE DOORS AND A TIGER BEHIND THE OTHER.

9* *The Island of Zombies.* The humans on this island always tell the truth, but the zombies always lie. The words for 'Yes' and 'No' in the language of the islanders are 'Bal' and 'Da', but you do not know which one means 'yes' and which 'no'. In each of these problems you only get to ask one question—you are not, for example, allowed to use the solution of part *(a)* to determine whether 'Bal' means 'yes', to solve the other parts!

   HINT: To get started, let $a$ denote the proposition, 'the islander is a human', let $b$ denote the proposition 'Bal' means 'yes', and let $p$ be an arbitrary proposition. Try to find a formula

using these three variables that is equivalent to 'The islander answers "Bal" when asked if $p$ is True'.

*(a)* You are allowed to ask an islander a single question, which will tell you whether 'Bal' means 'yes' or 'no'. What do you ask?

*(b)* You and a friend approach an islander. Your friend dares you: I will bet that you cannot make him say 'Bal'. Prove your friend wrong by asking the islander a single question, to which she must answer 'Bal'.

*(c)* You heard a rumor that there is gold in the hills on the island. You approach an islander and ask him a single question, whose answer will tell you whether or not there is gold in the hills.

### 2.5.3 Satisfiability Solvers

All the problems in this section, except for the first, are programming problems. To solve them, you should install one of numerous freely available SAT solvers that read input in the DIMACS format (for example `minisat` or `sat4j`).

10. In the example of the DPLL algorithm presented in Section 2.3.5, we began by selecting the variable $q$ and initially setting it to **T**. The time this algorithm takes to find a satisfying assignment is highly dependent on the order in which we assign truth values to variables. What happens in this example if we choose to work with $p$ first and assign it the value **F**? What if we choose $r$ first and assign it **T**?

11. *(a)* Produce a specification file for the CNF formula that encodes the student scheduling problem. There are, of course, too many clauses to create this file by hand, but there are only six types of clauses, and it is not difficult to write a computer program to generate the file.

    To get you started, use the integers 1 through 56 to denote the variables, where variables 1-8 correspond to the 8 successive possible time slots for English, 9-16 for the Math time slots, *etc.* The clause that English has to be scheduled in at least one of the 8 time slots is then

    ```
    1  2  3  4  5  6  7  8  0
    ```

    The 28 clauses that say that English cannot be scheduled in two different time slots begin

    ```
    -1 -2 0
    -1 -3 0
    -1 -4 0
    ```

    and so on up through

    ```
    -7 -8 0
    ```

*(b)* Now, run the SAT solver on the specification you produced in part *(a)*. What is the schedule that the result corresponds to? How would you use the SAT solver to find a different schedule that satisfies the specifications?

*(c)* The output produced by the SAT solver for this problem is a bit hard to read, since it gives us the satisfying assignment for *every* variable, where we are really only interested in knowing the variables that are set to **T**. (That is, if `2` appears in the output, then we know that English is scheduled for Monday afternoon, and the output values `-1,-3,-4,...,-8` are simply distracting.) Design a smart back end for this problem that reads the output of the SAT solver and prints the student's schedule in human-readable form.

12. Repeat the preceding problem for Sudoku puzzles. Here you will have to use the integers 1-789 to encode the underlying variables of the problem. The first 3240 clauses of the CNF specification, as described in the text, encode the rules of the puzzle, and the last few lines give the particular puzzle configuration. Write a smart front end that allows you to enter the specification of the puzzle position in a simple intuitive form, and generates the CNF specification file, as well as a smart back end that displays the solution as a square array of integers.

13.* One application of SAT solvers in Artificial Intelligence is the solution of planning problems. We'll illustrate this with classic river-crossing puzzles. Your problem is to model the puzzles as satisfiability problems, and then use the resulting specification as input to a SAT solver. (It is a bit time-consuming to produce the specifications, and the first puzzle is very easy to solve without any special tools. So this should be considered a kind of 'proof-of-concept' problem.)

> A farmer has a fox, a goose, and a sack of grain. He wants to cross a river with these items. His boat is only big enough to hold one of the items along with the farmer. He cannot leave the fox alone with the goose (the fox will eat the goose), and he cannot leave the goose alone with the grain (same reason). Describe how he can get all the items across the river in no more than seven crossings.
>
> The jealous husbands problem: Three married couples want to cross a river, but their boat only holds two people. Each husband is an insanely jealous man, and will not allow his wife to be on one of the river banks with another man unless he himself is present as well. Describe how to get all six people across the river with no more than twelve crossings.

HINT: How can we model these problems as satisfiability problems? The key here is the bound on the number of time steps. Consider the first problem. Number the time steps 0,1,...,7. Introduce variables to encode the propositions:

- the boat is on the left bank at time $t$
- the fox is on the left bank at time $t$
- the goose is on the left bank at time $t$
- the grain is on the left bank at time $t$

- the fox is placed in the boat at time $t$
- the goose is placed in the boat at time $t$
- the grain is placed in the boat at time $t$

We do not require variables to say that an item or the boat is on the right bank, since these propositions can be encoded by negations of ones described above.

The clauses in the CNF then express the rules of the puzzle: For example, if the fox and the goose are both on the left bank at time $t$, then the boat must be there at time $t$; if the goose boards the boat on the left bank at time $t$ then both the boat and the goose will be on the right bank at time $t + 1$, *etc.* These will also include the initial condition (everything on the left bank at time 0) and the goal condition (everything on the right bank at time 7). This can be done with about 200 clauses in all. Like our other puzzles, there are only a few basic kinds of clauses, each repeated a number of times for each time step and each bank of the river.

Here is another take on the first problem.

# Chapter 3

# Sets and Functions

## 3.1 Sets

A *set* is a collection of things, which are called the *elements* of the set. For instance, the collection consisting of the integers 3, 4 and 7, is a set. Let's name this set $A$. So we write

$$A = \{3, 4, 7\}.$$

We write $\in$ for 'is an element of', and $\notin$ for 'is not an element of'. So with $A$ as above we have

$$3 \in A$$

$$6 \notin A.$$

A set is completely determined by the elements it contains, so $\{4, 7, 3\}$ is the same set as $\{3, 4, 7\}$. So, too, is $\{4, 7, 3, 7\}$, thus

$$\{3, 4, 7\} = \{4, 3, 7\} = \{4, 7, 3, 7\}.$$

In practice we would never really write anything like$\{4, 7, 3, 7\}$, but the example is meant to illustrate the basic principle: Two sets are equal if they contain exactly the same elements: order and multiplicity of elements are irrelevant.

### 3.1.1 Notation for sets

If a set has only a few elements, we can write it by simply listing the elements, as we did above, and enclose the list in curly braces: $\{3, 4, 7\}$

One variant of this notation is to just list some of the elements and depend on the reader's ability to fill in what is missing. For instance,

$$\{1, 2, \ldots, 100\}$$

denotes the set consisting of the first one hundred positive integers, and

$$\{1, 3, 5, 7, \ldots\}$$

denotes the *infinite* set consisting of all the odd positive integers.

We can give any name to any set we like; above we gave the name $A$ to the set $\{3, 4, 7\}$. Some sets of numbers are referred to so often that they have special names.

$\mathbf{N} = \{0, 1, 2, \ldots\}$ = the set of natural numbers (nonnegative integers)

$\mathbf{Z} = \{0, \pm 1, \pm 2, \pm 3, \ldots\}$ = the set of integers

$\mathbf{Z}^+ = \{1, 2, 3, \ldots\}$ = the set of positive integers

$\mathbf{Q}$ = the set of rational numbers (that is numbers of the form $\frac{a}{b}$, where $a, b \in \mathbf{Z}$ and $b \neq 0$).

$\mathbf{R}$ = the set of real numbers

So, for instance $\sqrt{2} \in \mathbf{R}$, but $\sqrt{2} \notin \mathbf{Q}$, which is something that we will prove later.

We can define a set using *set comprehension notation,* by declaring our set to consist of all the elements of some larger set that satisfy a particular property. As an example,

$$\{x \in \mathbf{Z}^+ : x \leq 7\}$$

is another way to write $\{1, 2, 3, 4, 5, 6, 7\}$. The same notation, with $\mathbf{Z}$ replaced by $\mathbf{R}$, defines the infinite set that consists of all real numbers less than 7. Many authors use a vertical bar | in place of the colon : in set comprehension notation.

There is a set with no elements, called the *empty set*, and also the *null set*. It is denoted

$$\emptyset$$

Like zero, and like the empty string and the empty list that you met in your programming courses, the empty set is one of those nothings that turns out to be something you cannot live without.

### 3.1.2 Subset

We write

$$A \subseteq B$$

and say '$A$ is a subset of $B$,' to mean that every element of $A$ is an element of $B$.

Some examples:

$$\{3, 4\} \subseteq \{3, 4, 7\},$$

$$\{3\} \subseteq \{3, 4, 7\},$$

but NOT $3 \subseteq \{3, 4, 7\}$. 3 is not the same thing as $\{3\}$. This kind of distinction should be familiar from programming languages: for instance, in Python, the list `[3]` is not the same thing as the `int` object `3`. [1]

For any set $A$, $\emptyset \subseteq A$. This may seem odd at first, but consider it in light of what you have learned about propositional logic: $A \subseteq B$ is equivalent to saying that for any element $x$ at all, $(x \in A) \rightarrow (x \in B)$ is true. Since $x \in \emptyset$ is false for any $x$, $(x \in \emptyset) \rightarrow (x \in A)$ holds for any $x$ and any $A$.

[1] We often say informally '$A$ is contained in $B$' to mean $A \subseteq B$, and it is tempting to say '$A$ is contained in $B$' to mean $A \in B$ as well, but doing so blurs this important distinction. If we write 'is contained in' at all in this book, it will be with the first meaning, not the second.

For any set $A$, $A \subseteq A$. That is not quite so odd, but it's nice to have a special notation for *proper subsets:*

$$A \subsetneq B$$

means $A$ is a subset of $B$ and $A \neq B$, so that there is some element of $B$ that is not in $A$.

For example

$$\mathbf{Z}^+ \subsetneq \mathbf{N} \subsetneq \mathbf{Z} \subsetneq \mathbf{Q} \subsetneq \mathbf{R}.$$

The last proper inclusion in this chain is true because of the existence of irrational numbers like $\sqrt{2}$.

### 3.1.3 New sets from old

#### Power set

We define

$$\mathcal{P}(A) = \{B : B \subseteq A\}.$$

That is, $\mathcal{P}(A)$ is the set of all subsets of $A$. (Some authors write $2^A$ instead of $\mathcal{P}(A)$.) We call $\mathcal{P}(A)$ the *power set* of $A$.

For example,

$$\mathcal{P}(\{3,4,7\}) = \{\emptyset, \{3\}, \{4\}, \{7\}, \{3,4\}, \{3,7\}, \{4,7\}, \{3,4,7\}\}.$$

The power set of the empty set has a single element, which is the empty set:

$$\mathcal{P}(\emptyset) = \{\emptyset\}.$$

#### Cartesian Product

An ordered *n-tuple* is just a sequence of $n$ elements.

$$(a_1, a_2, \ldots, a_n).$$

Order and multiplicity count here! $(3,4,3)$ is different from both $(3,4,4)$ and $(3,3,4)$. ('Tuple' is one of those peculiar words that seems to be used only in mathematics and computer science. For small $n$ we say 'pair', 'triple', 'quadruple'.)

$$A \times B = \{(a,b) : a \in A, b \in B\},$$

*i.e.,* the set of all ordered pairs with first component in $A$ and second component in $B$. Likewise, we write

$$A_1 \times A_2 \times A_3$$

for sets of ordered triples, *etc.*

For example

$$\{3,4\} \times \{1,2,3\} = \{(3,1), (4,1), (3,2), (4,2), (3,3), (4,3)\}.$$

$A \times B$ is called the *Cartesian Product* of $A$ and $B$. This is a reference to Ren Descartes, who invented the scheme of representing points in the plane by ordered pairs of real numbers.

### Union, Intersection and Complement

The *union* and *intersection* of two sets $A$ and $B$, and the *relative complement* of $B$ in $A$, are defined, respectively, as follows.

$$A \cup B = \{x : x \in A \vee x \in B\}.$$

$$A \cap B = \{x : x \in A \wedge x \in B\}.$$

$$A \backslash B = A - B = \{x : x \in A \wedge x \notin B\}.$$

For example, with $A = \{3, 4, 7\}$, $B = \{4, 6, 11\}$ we have

$$A \cup B = \{3, 4, 6, 7, 11\}$$
$$A \cap B = \{4\}$$
$$A - B = \{3, 7\}$$
$$B - A = \{6, 11\}.$$

Also, for any set $A$,

$$A \cup \emptyset = A,$$
$$A \cup A = A \cap A = A - \emptyset = A.$$
$$A \cap \emptyset = \emptyset$$

We say that two sets $A$ and $B$ are *disjoint* if $A \cap B = \emptyset$. For example $\{3, 4, 5\}$ and $\{2, 6, 7\}$ are disjoint.

You will often see reference to an *absolute complement*

$$\bar{A} = \{x : x \notin A\}$$

This is problematic if we take it literally, since it talks about *everything* that is not $A$. In practice, it is a shorthand for the relative complement of $A$ in some implicitly understood 'universe' $U$ to which everything under discussion is assumed to belong. For instance, $\overline{\{3, 4, 7\}}$ would represent different sets depending on whether we are talking about sets of positive integers, or sets of integers, or sets of real numbers. In spite of this ambiguity, when it is reasonably clear from the context what our universe is, we will use this notation.

## 3.1.4 An application: defining complex structures

The language of sets allows us to make precise definitions of structures that we might ordinarily define informally by giving a few examples with diagrams and drawings. The drawings are usually easier to understand, but the formal definition is useful when we need to prove properties of these objects, and when we design data structures to represent them in a computer program.

For example, Figure 3.1 depicts a *directed graph*, or *digraph*. Such an object consists of a collection of *vertices,* or *nodes,* together with a collection of *arrows,* each of which is determined by specifying the node at which the arrow begins, and the node at which it ends. So we can specify the

graph as an ordered pair $(V, E)$, where $V$ is the set of nodes, and where $E$ is the set of arrows. Each element of $E$ is in turn an ordered pair of nodes, and is thus an element of $V \times V$. Our definition of digraph is thus: an ordered pair $(V, E)$, where $V$ is a set and $E \subseteq V \times V$. For example, with this definition, the digraph in the figure is:

$$(\{1, 2, 3, 4\}, \{(1, 2), (2, 1), (1, 3), (3, 2), (2, 4), (4, 4)\}).$$

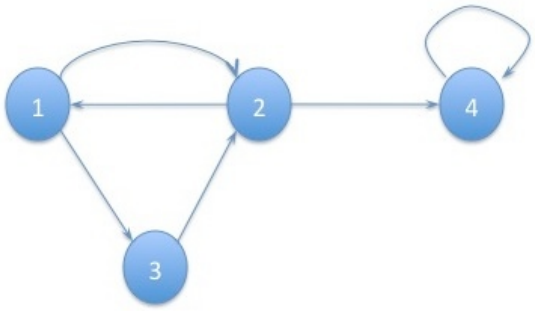



Figure 3.1: Directed graph

### 3.1.5 Sets in Python

The Python programming language has a built-in class called `set`. It allows the use of bracket notation and a version of set comprehension notation, as illustrated in the code fragment in Figure 3.2.

Observe that the ordering of the elements you see when you ask to display the set is not what you might have guessed: this is a quirk of how Python represents sets internally. In mathematical notation we would have defined this last set as

$$\{x^2 : x \in \mathbf{Z}, 100 < x^2 < 200\}.$$

Of course, a programming language will not allow us to create an infinite set, so we require the qualifier `for x in range (100)` in the Python definition.

This class supports the set operations of union, intersection, and relative complement (Figure 3.3).

### 3.1.6 Set Identities

There are some basic identities for sets—these are equations involving sets and set operations that hold no matter how sets are substituted for the symbols in the equation. Here, for example, is one of De Morgan's Laws: For all sets $A$ and $B$,

$$\overline{(A \cup B)} = \overline{A} \cap \overline{B}.$$

Look familiar? This exactly mirrors DeMorgan's Law from propositional logic, with $\vee$ replaced by $\cup$, $\wedge$ by $\cap$, and $\neg$ by complement. In the chain of equalities below we derive this law by applying

```
>>> s={3,4,7}
>>> type(s)
<type 'set'>
>>> s
set([3, 4, 7])
>>> 3 in s
True
>>> 2 in s
False
>>> t={4,7,3,7}
>>> s==t
True
>>> a={x*x for x in range(100) if 100<x*x<200}
>>> a
set([144, 121, 196, 169])
```

Figure 3.2: Defining sets in Python

the logical version of the law in the third equation. The rest is just translation from the definitions of union, intersection, and complement:

$$\begin{aligned}
\overline{(A \cup B)} &= \{x : x \notin A \cup B\} \\
&= \{x : \neg(x \in A \vee x \in B)\} \\
&= \{x : \neg(x \in A) \wedge \neg(x \in B)\} \\
&= \{x : \neg(x \in A)\} \cap \{x : \neg(x \in B)\} \\
&= \{x : x \notin A\} \cap \{x : x \notin B\} \\
&= \overline{A} \cap \overline{B}.
\end{aligned}$$

All the identities from propositional logic in Tables 1.6 and 1.7 translate in precisely the same manner into set identities.

### 3.1.7 Venn Diagrams

A *Venn diagram* is a pictorial representation of set operations. The most basic Venn diagrams, depicting the operations on a pair of sets, are shown in Figure 3.4: The interiors of the two circles represent the sets $A$ and $B$, and the shaded region represents the set given by the expression in question.

We can use Venn diagrams to illustrate the point we made concerning the correspondence between basic set identities and the identities of propositional logic. Let's start with the propositional identity

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

```
>>> s={2,3,4}
>>> t={1,3,5,8}
>>> s.union(t)
set([1, 2, 3, 4, 5, 8])
>>> s.intersection(t)
set([3])
>>> s-t
set([2, 4])
>>> t-s
set([8, 1, 5])
```

Figure 3.3: Set operations in Python

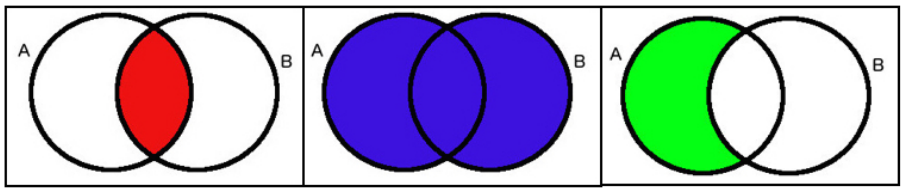



Figure 3.4: Venn diagrams illustrating the basic operations on two sets

If we replace the variable $p$ by the proposition $x \in A$, $q$ by $x \in B$ and $r$ by $x \notin C$, we arrive at the set identity

$$(A - C) \cup (B - C) = (A \cup B) - C.$$

Figure 3.5 is a kind of pictorial proof of this identity, using Venn diagrams. Each frame of the diagram contains three overlapping circles. The interiors of the circles are supposed to represent sets $A, B$ and $C$. The 'proof' in this case consisted of shading in the regions of the diagram corresponding to the two sides of the identity, and checking that they are equal.

Why does this work? For an identity involving three sets, the circles in the diagram partition the plane into eight separate regions, corresponding to the eight different assignments of truth values to the propositions $x \in A$, $x \in B$, $x \in C$. So the Venn diagram functions as a kind of pictorial truth table. It gets a little tricky with more than three sets.

### 3.1.8 Extended Operations

As we have seen, identities from propositional logic can be translated into set identities. In particular, the set operations of union and intersection are both associative and commutative, just like their counterparts $\bigvee$ and $\bigwedge$ in logic.

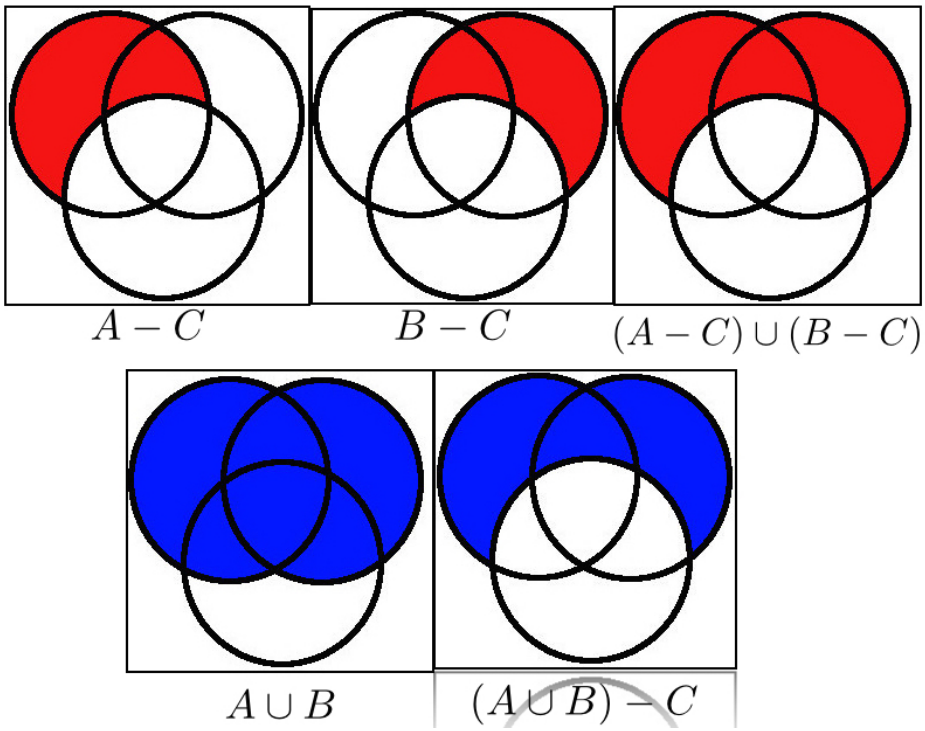



Figure 3.5: 'Proof' of $(A - C) \cup (B - C) = (A \cup B) - C$

This means that we can write

$$A_1 \cup A_2 \cup \cdots \cup A_n, \quad A_1 \cap A_2 \cap \cdots \cap A_n$$

without ambiguity. We will use an abbreviated notation for these expressions, much like the one we saw earlier.

$$\bigcup_{i=1}^{n} A_i, \quad \bigcap_{i=1}^{n} A_i.$$

What if you have an *infinite* family of sets, like

$$A_1, A_2, \ldots ?$$

Can we form the union, or the intersection, of all the sets in the family? For concreteness, let's use the sets

$$\begin{aligned} A_1 &= \{x^2 : x \in \mathbf{Z}^+\} \\ A_2 &= \{x^3 : x \in \mathbf{Z}^+\}, \end{aligned}$$

*etc.* In other words, $A_1$ is the set $\{1, 4, 9, 16, \ldots\}$ of squares of positive integers, $A_2$ the set of cubes, and in general, $A_k$ is the set of $(k+1)^{th}$ powers of positive integers.

We write

$$\bigcap_{k=1}^{\infty} A_k$$

and also

$$\bigcap_{k \in \mathbf{Z}^+} A_k$$

to denote the intersection of this sequence of sets. This means the set of all things that belong to *every one* of the $A_i$. The only integer that belongs to every one of these sets is 1, so

$$\bigcap_{k=1}^{\infty} A_k = \{1\}.$$

What would be the result if we had used $\mathbf{N}$ in the definitions of the sets $A_i$ instead of $\mathbf{Z}^+$ ?

We similarly define the union of the family of sets to be the set of all elements that belong to *at least one* of the $A_i$. In this example, the union consists of all the squares, cubes, fourth and fifth powers, *etc.* of the positive integers. It does not contain every positive integer as an element—for instance, 2 is not in any of the sets. So we have

$$\bigcup_{k \in \mathbf{Z}^+} A_k \subsetneq \mathbf{Z}^+.$$

## 3.2 Functions

### 3.2.1 Basic definitions

If $X$ and $Y$ are sets, then a *function* $f : X \to Y$ assigns a single element $f(a) \in Y$ to each $a \in X$. We say that $f$ is a function *from* $X$ *to* $Y$.

Domain: The *domain* of $f$ is $X$.

Range: The *range* of $f$ is $\{f(a) : a \in X\}$

Codomain: The *codomain* of $f$ is $Y$.

It is tempting to think of the word *function* as designating a function defined by some formula, like $f(x) = x^2 \cos x$, since this is the form in which you first encounter functions in a pre-calculus or calculus class. But we would like to discourage this narrow point of view, and have you think of functions more along the lines of what you see in Figure 3.6. Here the function is depicted as a *bipartite directed graph* where every arrow goes from an element $x$ of the domain to the element $f(x)$ of the codomain. For such a picture to really define a function, we require that there be exactly one arrow originating at each domain element. Thus the diagram in Figure 3.6 defines a function, but the diagram in Figure 3.7 does not.

We can formally define a function $f$ from $X$ to $Y$ to be the set of ordered pairs $\{(a, f(a)) : a \in X\} \subseteq X \times Y$. Not every subset $Z$ of $X \times Y$ defines a function in this way: We require that no two elements of $Z$ have the same left component, and that every element of $X$ appear as the left component of some element of $Z$. For example, if $X = \{1, 2, 3\}$ and $Y = \{x, y\}$, then the set

$$Z_1 = \{(1, y), (2, y), (3, x)\}$$

defines a function $f_1 : X \to Y$ with $f_1(1) = f_1(2) = y$ and $f_1(3) = x$. However,

$$Z_2 = \{(1, y), (1, x), (2, y), (3, x)\}$$

does not define a function, because it gives two different values for the function at $1 \in X$. The set

$$Z_3 = \{(1, x), (3, y)\}$$

does not define a function $f : X \to Y$, because it gives no value for $f(2)$. However, it does define a function with domain $\{1, 3\}$. It is customary in this case to call $f$ a *partial function* from $X$ to $Y$.

You may have noticed that the notion of codomain is a bit fuzzy: A function $f : \{1, 2, 3\} \to \{x, y\}$ is also a function from $f : \{1, 2, 3\} \to \{x, y, z\}$. Like the absolute complement of a set, you cannot determine the codomain of a function from the underlying set of ordered pairs; the codomain is an implicit 'universe' from which we draw the function values.

The set of *all* functions from $X$ to $Y$ is denoted $Y^X$.

Here are some more examples:

**Example.** Let's say we have a bunch of employees in a company, and assign each of them an office. This defines a function $f : E \to O$ where $E$ is the set of employees and $O$ is the set of offices. It would fail to be a function from $E$ to $O$ if some employee were assigned two offices, or if some employee were not assigned an office. (In the latter case it might still be a partial function from $E$ to $O$.)

**Example.** We did not mean to suggest above that the functions you study in calculus are *not* functions in this sense, just that the term denotes a much broader category of things. For instance,

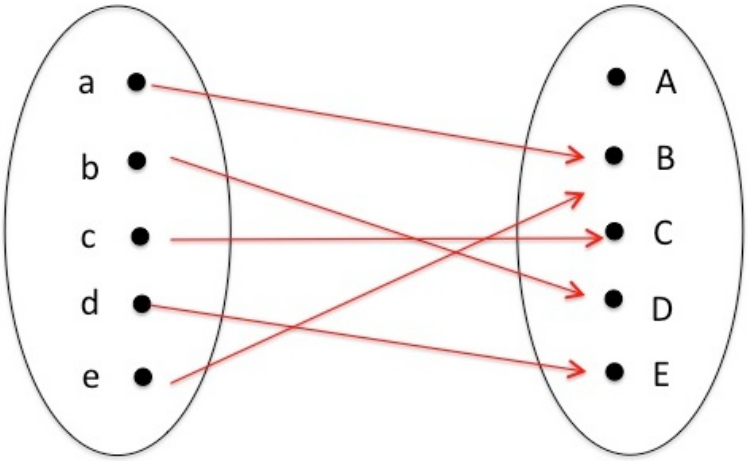



Figure 3.6: Bipartite graph representation of a function. $\text{dom}(f) = \{a, b, c, d, e\}$, $\text{codomain}(f) = \{A, B, C, D, E\}$, $\text{range}(f) = \{B, C, D, E\}$.

$f : \mathbf{R} \to \mathbf{R}$ defined by $f(x) = x^2$ is indeed a function. The formula $f(x) = \pm\sqrt{x^2 + 1}$ assigns two values to each $x \in \mathbf{R}$ and so is not a function. The formula $f(x) = \sqrt{1 - x^2}$ does not define a function from $\mathbf{R}$ to $\mathbf{R}$, because it does not give a value for $|x| > 1$. However it does define a function whose domain is the interval $\{x \in \mathbf{R} : -1 \leq x \leq 1\}$.

**Example.** We now revisit our definition of the semantics of propositional formulas from Section 1.3, and formulate it in our language of sets and functions. The idea was that given an assignment of truth values to the variables occurring in a formula $\phi$, we obtain a truth value for $\phi$ itself. An assignment of values to the variables is a function

$$\alpha : V \to \{\textbf{true}, \textbf{false}\},$$

where $V$ is a set of variables that contains all the variables in the formula. Let us denote the truth value of the formula $\phi$ on the assignment $\alpha$ as $[\phi](\alpha)$. We can thus view $\phi$ as a function

$$[\phi] : \{\textbf{true}, \textbf{false}\}^V \to \{\textbf{true}, \textbf{false}\}.$$

With this point of view, the recursive definition of the semantics is as follows:

$$[\mathbf{T}](\alpha) = \textbf{true}, \text{for any assignment } \alpha$$

$$[\mathbf{F}](\alpha) = \textbf{false}, \text{for any assignment } \alpha$$

$$[p](\alpha) = \alpha(p), \text{for any variable } p \text{ in the domain of } \alpha$$

$$[\phi \wedge \psi](\alpha) = \textbf{true}, \text{if } [\phi](\alpha) = [\psi](\alpha) = \textbf{true}, \text{ and } \textbf{false} \text{ otherwise}$$

$$[\phi \vee \psi](\alpha) = \textbf{false}, \text{if } [\phi](\alpha) = [\psi](\alpha) = \textbf{false}, \text{ and } \textbf{true} \text{ otherwise}$$

$$[\neg\phi](\alpha) = \textbf{false}, \text{if } [\phi](\alpha) = \textbf{true}, \text{ and } \textbf{false} \text{ otherwise.}$$

It might look as though we have taken something relatively simple and clear, and made it difficult! But giving this kind of precise definition of the semantics really is helpful when we set about to prove something about formulas, as we will see in Section 6.5.2.

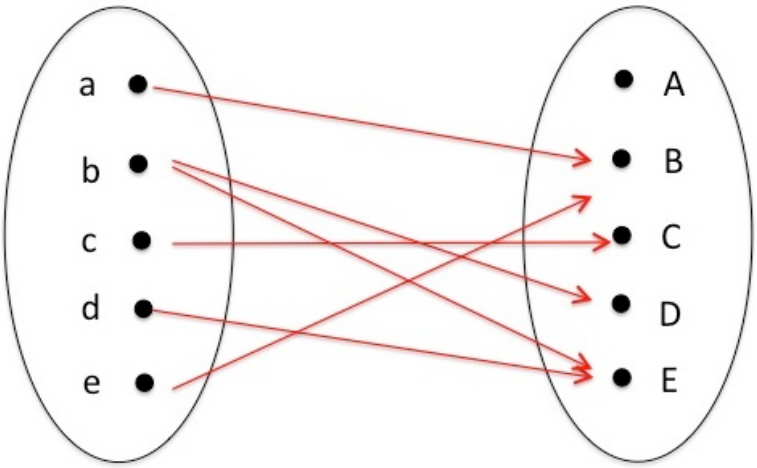



Figure 3.7: The diagram does not illustrate a function, because two different elements are assigned to $b$.

### 3.2.2 One-to-one functions

A function $f : X \to Y$ is *one-to-one* (also called *injective*) if for all $x, y \in X$, $f(x) = f(y)$ implies $x = y$. In other words, we never have two different elements of the domain mapped to the same element of the range. The function illustrated in Figure 3.2.2 is one-to-one. In contrast, the function in Figure 3.6 is not one-to-one, since $f(a) = f(e) = B$.

Let's return to some of our previous examples: The office function is one-to-one if everyone gets a private office (no sharing of offices).

$f : \mathbf{R} \to \mathbf{R}$ defined by $f(x) = x^2 + 1$ is not one-to-one, because $f(1) = f(-1)$. However $f(x) = x^3 + 1$ defines a one-to-one function.

### 3.2.3 Onto functions

A function $f : X \to Y$ is *onto* (also called *surjective*) if the range of $f$ is $Y$. A function that is both one-to-one and onto is also called a *bijective function,* also a *bijection* or a *one-to-one correspondence*. Figure 3.9 illustrates an onto function, and Figure 3.10 a bijective function.

In our examples, the office function is onto if there are no empty offices, and bijective if every office gets a single occupant.

The function $f : \mathbf{R} \to \mathbf{R}$ defined by $f(x) = x^2 + 1$ is not onto, since it only has numbers greater than or equal to 1 as values. The function defined by $f(x) = x^3 + 1$ is however onto $\mathbf{R}$, since given any real number $a$, $a = f(\sqrt[3]{a - 1})$. This function is also a bijection from $\mathbf{R}$ to $\mathbf{R}$, since as we observed earlier, it is also one-to-one. The function defined by $f(x) = x^3 - x$ is onto, but not one-to-one. (See Figure 3.11.)

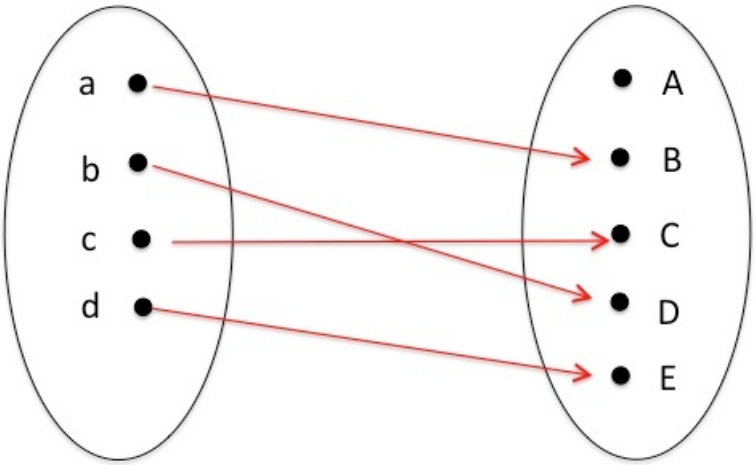



Figure 3.8: A one-to-one function. No pair of arrows leads to the same element of the domain.

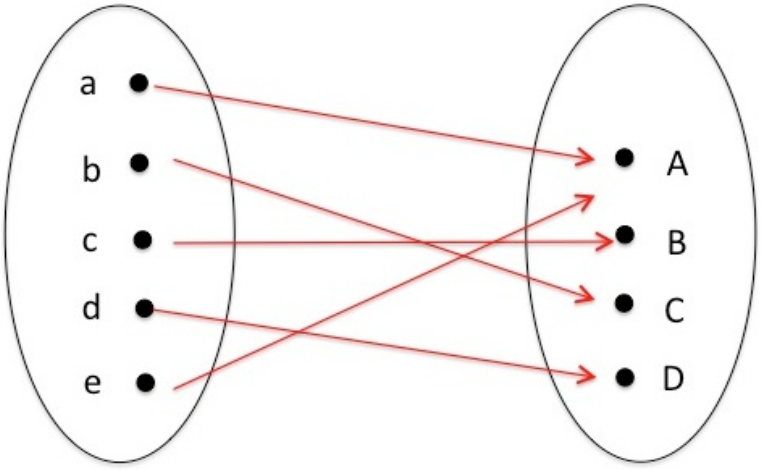



Figure 3.9: An onto function. There is an arrow to every codomain element.

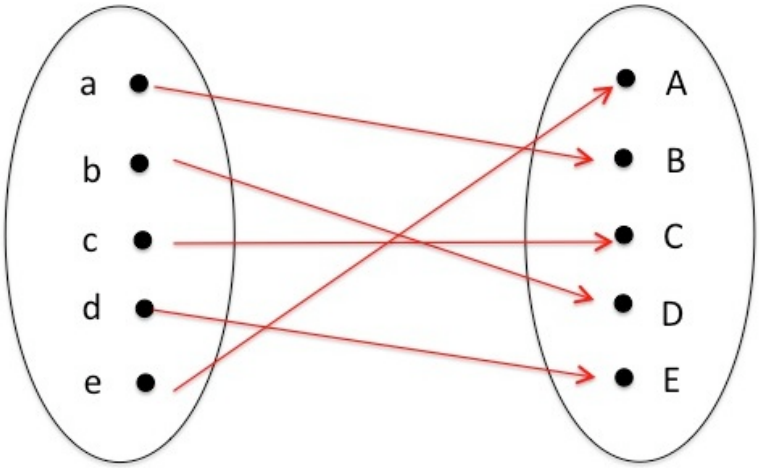



Figure 3.10: A bijective function

### 3.2.4 Composition of functions

If $f : X \to Y$ and $g : Y \to Z$ are functions, then the *composition*

$$g \circ f : X \to Z$$

is defined by setting

$$g \circ f(a) = g(f(a))$$

for all $a \in X$.

This operation on functions is not commutative—in fact, if $X$ and $Z$ are different sets, then $f \circ g$ doesn't even make sense. Even if $X = Y = Z$ so that $f \circ g$ and $g \circ f$ are both defined, usually $f \circ g$ and $g \circ f$ are different functions.

**Example.** If $f$ is the function from the set of employees of a company to the set of offices, and if $g$ is the function from the set of offices to $\mathbf{R}$ giving the number of square feet of an office, then

$$g \circ f : \texttt{Employees} \to \mathbf{R}$$

assigns to each employee the number of square feet in his or her office.

**Example.** Let $f : \mathbf{R} \to \mathbf{R}$ and $g : \mathbf{R} \to \mathbf{R}$ be defined by the formulas

$$f(x) = x^2 + 3, g(x) = \sqrt{1 + |x|}$$

for all $x \in \mathbf{R}$. Then for all $x \in \mathbf{R}$,

$$f \circ g(x) = (\sqrt{1 + |x|})^2 + 3 = 4 + |x|,$$

$$g \circ f(x) = \sqrt{1 + |x^2 + 3|} = \sqrt{x^2 + 4}.$$

These are different functions, as can be seen, for example, by evaluating them both at $x = 0$.

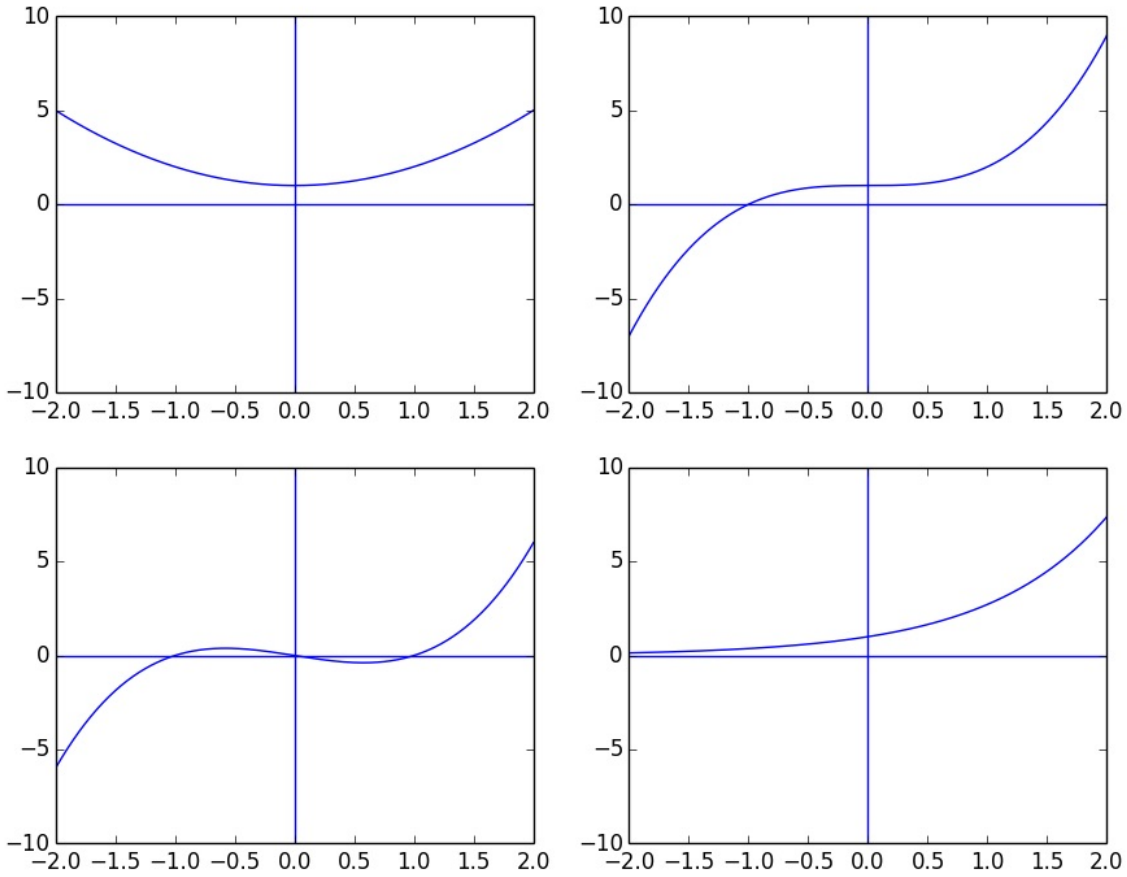



Figure 3.11: The graphs of four functions $f : \mathbf{R} \to \mathbf{R}$. In the upper left, $f(x) = x^2 + 1$ is neither one-to-one nor onto. In the upper right, $f(x) = x^3 + 1$ is both one-to-one and onto. In the lower left, $f(x) = x^3 - x$ is onto, but not one-to-one, and in the lower right, $f(x) = e^x$ is one-to-one but not onto.

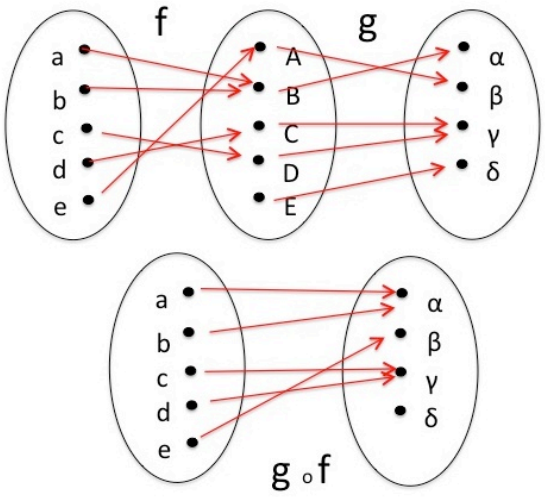



Figure 3.12: Two functions $f$ and $g$, together with their composition $g \circ f$.

### 3.2.5 Identity and inverse

If $X$ is a set, then the *identity function* from $X$ into $X$, which we denote $\mathbf{id}_X$, is defined by $\mathbf{id}_X(x) = x$ for all $x \in X$. If $f : X \to Y$ is a bijection, then for every $b \in Y$ there is *exactly* one element $a \in X$ such that $f(a) = b$. We then define a new function from $Y$ to $X$ whose value at $b$ is this unique $a$. The new function is denoted

$$f^{-1} : Y \to X$$

and is called the *inverse function* of $f$. Observe that as we have defined it,

$$f^{-1} \circ f(a) = a$$

for every $a \in X$, and

$$f \circ f^{-1}(b) = b$$

for every $b \in Y$. We can write these facts as equations between functions:

$$f \circ f^{-1} = \mathbf{id}_Y, f^{-1} \circ f = \mathbf{id}_X.$$

Conversely, if $f : X \to Y$ and $g : Y \to X$ are functions such that $f \circ g = \mathbf{id}_Y$ and $g \circ f = \mathbf{id}_X$, then $f$ and $g$ are bijections, and $g = f^{-1}$, $f = g^{-1}$.

**Example.** In the bipartite graph representation of a bijective function, the inverse function is obtained just by reversing the directions of the arrows. Doing this for an arbitrary function from $X$ to $Y$ will in general not give you a function from $Y$ to $X$, either because you will wind up with two or more arrows originating from the same element of $Y$, or an element of $Y$ from which no arrow originates. Bijective functions are precisely the ones for which this procedure yields a function.

**Example.** If our office function is bijective, then every office is assigned to exactly one employee, and no employee gets more than one office. In this case the inverse function assigns to each *office* the employee who occupies that office.

**Example.** As we saw earlier, the function $f : \mathbf{R} \to \mathbf{R}$ defined by $f(x) = x^2$ for every $x$ is neither one-to-one nor onto. But the function $f : \mathbf{R}^+ \to \mathbf{R}^+$ defined by the same formula is both. ($\mathbf{R}^+$ denotes the set of positive real numbers.) In this case, the inverse function is given by $f^{-1}(x) = \sqrt{x}$ for every $x \in \mathbf{R}^+$.

### 3.2.6 Extended function notation

Look again at the function in Figure 3.6. The elements of the subset $\{a, b, c\}$ of the domain are mapped to the elements $B, C$ and $D$ of the image. It is tempting to write

$$f(\{a, b, c\}) = \{B, C, D\}.$$

Now, strictly speaking, this is not right. Our original function $f$ has domain $X = \{a, b, c, d, e\}$ and codomain $Y = \{A, B, C, D, E\}$, but here we are treating it as a function from $\mathcal{P}(X)$ to $\mathcal{P}(Y)$. To be careful about the distinction, we might define

$$\hat{f} : \mathcal{P}(X) \to \mathcal{P}(Y)$$

by setting

$$\hat{f}(Z) = \{f(z) : z \in Z\} \subseteq Y$$

for all $Z \subseteq X$. We can then write our equation above as $\hat{f}(\{a, b, c\}) = \{B, D\}$. In fact, the common practice is to be careless about the distinction, and use the symbol $f$ to denote both the original function with domain $X$ and the extended function with domain $\mathcal{P}(X)$. [2]

We similarly write things like

$$f^{-1}(\{A, B, C\}) = \{a, c, e\}.$$

Here we are using '$f^{-1}$' to denote the function from $\mathcal{P}(Y)$ to $\mathcal{P}(X)$ defined by

$$f^{-1}(Z) = \{x \in X : f(x) \in Z\}$$

for all $Z \subseteq Y$. Observe that the original function $f : X \to Y$ is neither one-to-one nor onto, and so does not have an inverse in the sense defined earlier.

## 3.3 Counting

If $A$ is a set, then we denote by $|A|$ the number of elements in $A$. The fancy way to say the number of elements in $A$ is the *cardinality* of $A$. For now, we will only talk about the cardinality of $A$ if $A$ is a finite set—we will take up the question of what the cardinalities of $\mathbf{Z}$, $\mathbf{Q}$, and $\mathbf{R}$ ought to be in a subsequent section.

For example, $|\{3, 4, 7\}| = 3$, $|\{x \in \mathbf{Z} : -3 \leq x \leq 3\}| = 7$, $|\emptyset| = 0$.

[2] Mathematicians often refer to this practice of loading the same symbol with several different meanings as 'abuse of notation'. The reason we do it is to avoid creating a confusing array of new notations, and because in situations like this, the intended meaning of the symbol is usually clear from the context.

### 3.3.1 Addition and multiplication principles

Here are a few basic facts about how the cardinality of sets behaves under various set operations. They are all pretty obvious once you understand what they are saying:

For all sets $A, B$:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

(This is because we can count the elements in the union by counting all the elements of $A$, and then adding to this the count of all the elements of $B$. But in doing so, we have counted the elements of $A \cap B$ twice.)

In particular, if $A$ and $B$ are disjoint, then $A \cap B = \emptyset$, which gives us:

$$|A \cup B| = |A| + |B|$$

in this special case.

The Cartesian product $A \times B$ contains $|B|$ ordered pairs with first component $a$ for every $a \in A$. Thus,

$$|A \times B| = |A||B|.$$

More generally,

$$|A_1 \times A_2 \times \cdots A_n| = |A_1||A_2| \cdots |A_n|.$$

### 3.3.2 Behavior of cardinality under functions

The following principles are also obvious. Think about the pictures depicting one-to-one, onto, and bijective functions and you will see why they are true:

If $f : A \to B$ is one-to-one, then $|A| \leq |B|$.

If $f : A \to B$ is onto, then $|A| \geq |B|$.

Put the two together, and you get:

If $f : A \to B$ is bijective, then $|A| = |B|$.

This last statement gives us an important method for proving that two functions $A$ and $B$ have the same number of elements: Find a bijective function between the two sets.

### 3.3.3 Power set and sets of functions

How many elements are in $\mathcal{P}(A)$? The examples we gave earlier showed that

$$|\mathcal{P}(\{3, 4, 7\})| = 8 = 2^3.$$

$$|\mathcal{P}(\emptyset)| = 1 = 2^0.$$

If you try out a few more examples with small sets $A$, you'll see that you always have

$$|\mathcal{P}(A)| = 2^{|A|}.$$

We'll state this as Theorem, and give it an official-looking proof.

**Theorem 3.3.1.** *If $A$ is a finite set, then*

$$|\mathcal{P}(A)| = 2^{|A|}.$$

*Proof.* Suppose $|A| = n$. We will show that there is a bijective function

$$F : \mathcal{P}(A) \to \underbrace{\{0,1\} \times \{0,1\} \times \cdots \times \{0,1\}}_{n \quad \text{times}}.$$

By the multiplication principle, the codomain has $2^n$ elements, so this will show us that $\mathcal{P}(A)$ also has $2^n$ elements.

Before we write down the details formally, let's first give the simple idea of the proof. Suppose for example that $n = 5$. The correspondence between 5-tuples of bits and subsets of $\{a_1, a_2, a_3, a_4, a_5\}$ works like this:

$$\begin{array}{ccc} (0,1,1,0,1) & \leftrightarrow & \{a_2, a_3, a_5\} \\ (1,1,0,0,0) & \leftrightarrow & \{a_1, a_2\} \\ (0,0,0,0,0) & \leftrightarrow & \emptyset \end{array}$$

If you see the pattern then you should be able to produce the 5-tuple corresponding to any subset and vice-versa. In fact, you should be able to do this for any number of elements, not just 5.

Now for the formal details. Let $A = \{a_1, \ldots, a_n\}$, and let $X \in \mathcal{P}(A)$. That is, $X \subseteq A$. We define

$$F(X) = (b_1, \cdots, b_n),$$

where $b_i = 1$ if $a_i \in X$, and $b_i = 0$ if $a_i \notin X$. $F$ is a function with the required domain and codomain, because it assigns an $n$-tuple of bits to every subset of $A$. We now have to show that it is bijective.

Why is $F$ one-to-one? Suppose $F(X_1) = F(X_2)$, and suppose that $a_i \in X_1$. Then the $i^{th}$ component of $F(X_1)$ is 1, and thus the $i^{th}$ component of $F(X_2)$ is 1, which means $a_i \in X_2$. Thus every element of $X_1$ is an element of $X_2$, so $X_1 \subseteq X_2$. The same argument shows $X_2 \subseteq X_1$. So $X_1 = X_2$, and thus $F$ is one-to-one.

Why is $F$ onto? We have to show that every $n$-tuple $(b_1, \ldots, b_n)$ of $n$ bits is $F(X)$ for some subset $X$ of $A$. But we get this if we set

$$X = \{a_i : b_i = 1\}.$$

So $F$ is onto as well, and is thus a bijection, so the two sets have the same number of elements. □

We will give a different proof of this theorem in Chapter 6.

Essentially the same argument shows that if $A$ and $B$ are sets, then the set of all functions $f : A \to B$ has cardinality $|B|^{|A|}$. The idea is that if $A = \{a_1, \ldots, a_m\}$, then we assign to the function $f : A \to B$ the $m$-tuple $(f(a_1), \ldots, f(a_m))$ of elements of $B$. This gives a bijection from the set of all functions to the Cartesian product $B \times \cdots \times B$ of $m$ copies of $B$.

This explains the notation $B^A$ for the set of functions from $A$ to $B$. For the same reason, the power set of $A$ is sometimes written $2^A$.

### 3.3.4 Permutations and the factorial function

If $f : A \to B$ is one-to-one, then as we stated above, $|A| \leq |B|$. How many such injective functions are there? Set $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$. As we argued above when we counted the total number of functions, we can associate to each function $f$ the $m$-tuple $(f(a_1), \ldots, f(a_n))$ of elements of $B$. However, the requirement that $f$ be one-to-one restricts which $m$-tuples can arise. To be precise, we can have any one of the $n$ elements of $B$ for $f(a_1)$, but $f(a_2)$ has to be a different value, so there are only $n-1$ choices once we have fixed $f(a_1)$. Similarly, $f(a_3)$ has to be different from the first two components, so there are $n-2$ possibilities, *etc.* All in all, there are

$$n \times (n-1) \times \cdots \times (n-m+1)$$

ways to choose the $m$-tuple, and hence to define the one-to-one function $f$.

Let's illustrate this with $A = \{1, 2\}$ and $B = \{1, 2, 3, 4\}$. The one-to-one functions are represented by ordered pairs of elements from $B$ in which the two components are different: We have 4 choices for the first component, and 3 for the second, so there are $4 \times 3 = 12$ such functions altogether.

If $A = B = \{1, \ldots, n\}$ then every one-to-one function is also onto, and hence a bijection. A bijection from a set to itself is called a *permutation* of the set. As in our arguments above, we can identify a permutation of $A$ with a sequence $(a_1, \ldots, a_n)$, in which every element of $A$ appears exactly once. Our formula for the number of one-to-one functions in this case gives:

$$n \times (n-1) \times \cdots \times 2 \times 1.$$

This number, the product of the first $n$ positive integers, is called '$n$ factorial' and is denoted $n!$.

Note that if $|A| = m < n = |B|$, then we can write our formula above for the number of one-to-one functions from an $m$-element set to an $n$-elements set as

$$n \times (n-1) \times \cdots \times (n-m+1) = \frac{n!}{(n-m)!}.$$

The same formula works for the case where $|A| = |B| = n$, provided we set $(n-n)! = 0! = 1$. It seems strange at first to define $0! = 1$, but it actually makes good sense from several different standpoints, and not just because it's the choice that makes our formula give the right answer.

**Example.** Suppose you have to make a list of 4 people from a collection of 15 different people whom you know. How many possible such lists are there? Here the order matters, so 'Matt, Carla, Alex, Jane' is a different list from 'Alex, Carla, Jane, Matt'. In this case we are counting the number of injective functions from $\{1, 2, 3, 4\}$ to your set of 15 friends. The number is

$$\frac{15!}{11!} = 15 \times 14 \times 13 \times 12 = 32760.$$

**Example.** The value of $n!$ grows very quickly as $n$ increases. For instance, the number of different ways in which we can arrange the 52 cards in a standard deck of playing cards is

$$52! \approx 8.066 \times 10^{67}.$$

It is hard to get a feel for the size of this number, which is vastly larger than what any computer, or even all the computers on earth working together, can count to, even if they were to operate for trillions of years. If the deck of cards is randomly shuffled, then the resulting arrangement has never before been seen in the history of humankind, and, if the deck is then thoroughly reshuffled, will never be seen again.

### 3.3.5 Binomial coefficients

Look again at the example in the preceding section where we counted the number of 4-element sequences of distinct elements we could make from a set of 15 elements. You were asked to treat the list 'Matt, Carla, Alex, Jane' as different from 'Alex, Carla, Jane, Matt'. But what if the task was simply to choose a collection of four friends, where the order is immaterial? Now the problem is to find the number of 4-element *subsets* of a 15-element set. There are 4! ways in which we can arrange the elements of the set $\{\texttt{Alex}, \texttt{Carla}, \texttt{Jane}, \texttt{Matt}\}$, and likewise for every other subset. So the number of sequences is exactly 4! times the number of subsets. The number of subset is consequently

$$\frac{15!}{11!4!} = \frac{15 \cdot 14 \cdot 13 \cdot 12}{4 \cdot 3 \cdot 2 \cdot 1} = 1365.$$

Precisely the same reasoning shows that if $m \leq n$, then the number of $m$-element subsets of a $n$-element set is

$$\frac{n!}{m!(n-m)!}.$$

This number is called a *binomial coefficient* (we'll see why in a moment), and is denoted

$$\binom{n}{m},$$

and usually pronounced '$n$ choose $m$'.

**Example.** Here's an example: How many 5-card poker hands can be made from a 52-card deck? We are asking for the number of 5-element subsets of the 52 cards, since we're not concerned about the ordering of the cards. However, we treat

$$5\clubsuit, 6\diamondsuit, 8\diamondsuit, J\spadesuit, Q\heartsuit,$$

as a different hand from

$$5\heartsuit, 6\spadesuit, 8\spadesuit, J\diamondsuit, Q\clubsuit,$$

since it involves different cards, even though the two hands are equivalent from the standpoint of playing poker. The number of such hands is consequently

$$\binom{52}{5} = \frac{52!}{5!47!}.$$

This number is large, but not astronomically so. Observe that we can write $\frac{52!}{47!}$ as $52 \times 51 \times 50 \times 49 \times 48$, so that the binomial coefficient $\binom{52}{5}$ is

$$\frac{52 \times 51 \times 50 \times 49 \times 48}{5 \times 4 \times 3 \times 2 \times 1}.$$

Now we can simplify this expression with a lot of cancellation: divide 50 by 5, 51 by 3, 52 by 4, and 48 by 2. The result is

$$13 \times 17 \times 10 \times 49 \times 24 = 2598960.$$

**Example.** We give an application to the analysis of algorithms. Figure 3.13 shows a Python function for sorting a list, using the simple (and slow) Selection Sort algorithm. The algorithm

takes as input a list with indices $0, \ldots, n-1$. The `if` statement in the innermost loop is executed once for each pair $(j, k)$ such that $0 \leq j < k < n$. That is, it is executed once for each 2-element subset of $\{0, 1, \ldots, n-1\}$, so the total number of times this comparison is made is

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

The statements within the outer loop are each executed $n$ times, and the assignment at the beginning only once. So the total time it takes to execute this function is

$$c \cdot \frac{n^2}{2} - c \cdot \frac{n}{2} + d \cdot n + e = \{\frac{c}{2} + (d - \frac{c}{2})/n + e/n^2\} \cdot n^2.$$

where $c, d$, and $e$ are constants depending on the programming environment and the units of time.[3] We don't need to know the relative sizes of these constants: If $n$ is very large, then the terms $(d - \frac{c}{2})/n$ and $e/n^2$ are very small, and thus the overall running time is roughly $\frac{c}{2} \cdot n^2$. This allows us to make accurate predictions about how the time to execute the algorithm will grow as $n$ gets larger: for example, if the size of the list doubles, we should expect the running time to increase by a factor of 4.

```
def selectsort(thelist):
    n = len(thelist)
    for j in range(n-1):
        minval=thelist[j]
        minindex=j
        for k in range(j+1,n):
            if thelist[k]<minval:
                minval=thelist[k]
                minindex=k
        thelist[minindex]=thelist[j]
        thelist[j]=minval
```

Figure 3.13: The Selection Sort Algorithm. The if statement in the inner loop is executed $\binom{n}{2}$ times.

The binomial coefficients have several important properties, which we list below. Each of these properties can be proved in (at least) two ways: by applying the formula for the binomial coefficients and doing a bit of algebra, or by resorting to the combinatorial definition of binomial coefficients as counting subsets.

[3] We are fudging just a little bit here, because the assignment statements in the body of the `if` statement may or may not be executed, depending upon the original order of the list. This means that $c$ is not a constant, but varies between two different positive constants. This does not affect the conclusion of the analysis.

**Theorem 3.3.2.** *For $n \geq 0$,*

$$\binom{n}{n} = \binom{n}{0} = 1.$$

*For $0 \leq k \leq n$,*

$$\binom{n}{k} = \binom{n}{n-k}.$$

*For $0 \leq k < n$,*

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}.$$

A combinatorial proof of the first property is the observation that an $n$-element set $A$ has exactly one subset ($A$ itself) with $n$ elements, and exactly one subset (the empty set) with 0 elements. A formula-based proof of the third property is given in the sequence of equations below:

$$\begin{aligned}
\binom{n}{k} + \binom{n}{k+1} &= \frac{n!}{k!(n-k)!} + \frac{n!}{(k+1)!(n-k-1)!} \\
&= \frac{(k+1)\cdot n!}{(k+1)!(n-k)!} + \frac{(n-k)\cdot n!}{(k+1)!(n-k)!} \\
&= \frac{(k+1)\cdot n! + (n-k)\cdot n!}{(k+1)!(n-k)!} \\
&= \frac{(n+1)\cdot n!}{(k+1)!(n-k)!} \\
&= \frac{(n+1)!}{(k+1)!((n+1)-(k+1))!} \\
&= \binom{n+1}{k+1}
\end{aligned}$$

The formulas above give a method for constructing a table of the binomial coefficients: Both the rows and columns of the table are indexed by the integers $0, 1, 2, \ldots$... Using the first formula, we write the value 1 in column 0 of every row, for as many rows as we want to extend the table, and 1 in column $n$ of row $n$. We compute the values in row $n+1$ of the table from the values in row $n$ by using the third formula: each entry is the sum of the entry immediately above it, with the entry one row above and one column to the left. The first few rows of the resulting table are illustrated in Figure 3.14.

This table is called *Pascal's Triangle,* after Blaise Pascal, who described it in the 17th century, although it was discovered independently, centuries earlier, by mathematicians in India, China, and the Arab world.

The name *binomial coefficients* arises from their appearance in formulas for the powers of the 'binomial' $x + y$. Here we compute the first few such powers:

$$\begin{array}{lcccl}
(x+y)^2 & = & xx + xy + yx + yy & = & x^2 + 2xy + y^2 \\
(x+y)^3 & = & xxx + xxy + xyx + xyy + yxx + yxy + yyx + yyy & = & x^3 + 3x^2y + 3xy^2 + y^3 \\
(x+y)^4 & = & xxxx + xxxy + xxyx + \cdots + yyyx + yyyy & = & x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4
\end{array}$$

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | | | | | | |
| 1 | 1 | | | | | |
| 1 | 2 | 1 | | | | |
| 1 | 3 | 3 | 1 | | | |
| 1 | 4 | 6 | 4 | 1 | | |
| 1 | 5 | 10 | 10 | 5 | 1 | |
| 1 | 6 | 15 | 20 | 15 | 6 | 1 |

Figure 3.14: 'Pascal's Triangle': Table of binomial coefficients $\binom{n}{k}$ for $0 \leq k \leq 6$.

When we expand $(x + y)^n$ completely using the distributive law, we get the sum of all the different possible sequences of $n$ $x$'s and $y$'s. Each such term, when we group all the $x$'s together and all the $y$'s together, has the form $x^m y^{n-m}$, where $0 \leq m \leq n$. The coefficient of $x^m y^{n-m}$ in the simplified expression is the number of sequences that contain exactly $m$ $x$'s. How many such sequences are there? We need to choose $m$ out of the $n$ available positions to write $x$ in, and then we write $y$ in the remaining $n - m$ positions, so there are $\binom{n}{m}$ such sequences in all. You can see in the example above that the coefficients in $(x + y)^n$ for $n \leq 4$ are identical to the corresponding rows of Pascal's Triangle, and the argument we just gave shows that this holds for all $n$. We thus get the following formula:

**Theorem 3.3.3.** *For all $x, y \in \mathbf{R}$, and for all $n \in \mathbf{N}$.*

$$(x + y)^n = \sum_{m=0}^{n} \binom{n}{m} x^m y^{n-m}.$$

This fact is called the *Binomial Theorem.* (For more on the $\sum$ notation for sums, see Chapter 6.)

## 3.4 Infinite Sets

What does it mean for two sets to have the same number of elements? What does it mean for one set to have more elements than another? For finite sets, this is clear: Two sets $A$ and $B$ have the same number of elements if and only if there is a bijective function

$$f : A \to B,$$

that is, if $A$ and $B$ can be put in one-to-one correspondence. We also say that $B$ has more elements than $A$ if $A \subsetneq B$, that is if $A$ is a proper subset of $B$, or if $A$ can be put in one-to-one correspondence with a proper subset of $B$.

There is a problem when we try to extend these notions to infinite sets. Consider the sets

$$\mathbf{N} = \{0, 1, 2, \ldots\}$$

and

$$\mathbf{N}_{\text{even}} = \{2x : x \in \mathbf{N}\} = \{0, 2, 4, 6, \ldots\}.$$

On one hand, $\mathbf{N}_{\text{even}}$ is a proper subset of $\mathbf{N}$, so we would say that $\mathbf{N}$ has strictly more elements. On the other hand, the function

$$f : \mathbf{N} \to \mathbf{N}_{\text{even}}$$

given by $f(n) = 2n$ for $n \in \mathbf{N}$ is bijective, so we would say that the two sets have the same number of elements. In fact, if we instead defined $f(n) = 4n$, then we would have a one-to-one correspondence between $\mathbf{N}$ and a proper subset of $\mathbf{N}_{\text{even}}$, which would incline us to say that $\mathbf{N}_{\text{even}}$ has *more* elements than $\mathbf{N}$! Evidently, something has to give—we cannot transport the notion of size of a set from finite to infinite sets and expect to preserve all the familiar properties.

### 3.4.1 Countable sets

We will adopt the point of view that two sets have the same number of elements if they can be put in one-to-one correspondence. We will *not* say that $B$ has a larger number of elements than $A$ if $A$ is a proper subset of $B$, so as not to fall into the kind of paradox described above.

We say a set $A$ is *countable* if there is a bijective function

$$f : \mathbf{N} \to A.$$

So our example above showed that $\mathbf{N}_{\text{even}}$ is countable. You can think of 'countable' as meaning 'listable', since if we have such a bijective function, then

$$f(0), f(1), f(2), \ldots$$

is a list that contains every element of $A$ exactly once.

We're going to show in a moment that a whole bunch of infinite sets are countable. In doing so, we will also describe algorithms, written as Python functions, that output the resulting list of elements.

The set $\mathbf{Z}$ of all integers is countable. Why? The natural order on $\mathbf{Z}$ is

$$\ldots, -3, -2, -1, 0, 1, 2, \ldots,$$

which looks like a list that is infinite in both directions. However, we can rearrange things and create a one-way list as follows:

$$0, -1, 1, 2, -2, \ldots$$

It is possible, if you really wanted to do it, to produce a formula for the bijection $f : \mathbf{N} \to \mathbf{Z}$ that underlies this list, but this is not really necessary. The code fragment in Figure 3.15 lists the elements of $\mathbf{Z}$.

This function, along with the other examples in this section, enumerates an infinite set, so it contains a loop controlled by `while True`. In other words, it 'runs forever', so if you try it out, be prepared to hit CTRL-C, or whatever it is you need to do to interrupt execution!

The set $\mathbf{N} \times \mathbf{N}$ of all ordered pairs of natural numbers is also countable. We first list those pairs $(m, n)$ such that $m + n = 0$ (there is only one), then those for which the sum of the components is 1, those with sum 2, *etc.* This works because there are only finitely many pairs for each component sum. The resulting list begins

$$(0,0), (0,1), (1,0), (0,2), (1,1), (0,2), (0,3), (1,2), \ldots$$

```
def enumerate_z():
    count = 0
    while True:
        if count==0:
            print 0,
        elif count%2==0:
            print -count/2,
        else:
            print (count+1)/2,
        count += 1
```

Figure 3.15: Python function for enumerating $\mathbf{Z}$

```
def enumerate_n_cross_n():
    componentsum=0
    while True:
        for j in range(componentsum+1):
            print(j,componentsum-j)
        componentsum+=1
```

Figure 3.16: Python function for enumerating $\mathbf{N} \times \mathbf{N}$

```
def enumerate_strings():
    prevlist=['']
    print ''
    while True:
        newlist=[]
        for s in prevlist:
            for letter in 'abc':
                newlist.append(s+letter)
        prevlist = newlist
        for s in prevlist:
            print s
```

Figure 3.17: Python function for enumerating all the strings over the alphabet $\{a, b, c\}$.

and is the output of the function in Figure 3.16.

We can treat each pair $(m, n)$ in the above list as a fraction $m/n$, provided $n \neq 0$. If we only list those pairs for which the fraction $m/n$ is in lowest terms, we get a list that includes every positive rational number exactly once:

$$0, 1, 1/2, 2, 1/3, 3, \ldots.$$

This result has the surprising interpretation that the set $\mathbf{Q}^+$ of positive rational numbers has the same number of elements as the set $\mathbf{N}$ of natural numbers. This is in spite of the fact that if you draw the two sets on the number line, the rational numbers seem vastly more numerous, since between any two rational numbers you can find a third.

The set of all strings over a finite alphabet is countable: For instance, if the alphabet is $\{a, b, c\}$ we can list first the empty string, then the strings of length 1, then those of length 2, *etc.* Once again, there are only finitely many strings of each fixed length, so this procedure enumerates all the strings. The list, which begins

```
`','`a','`b','`c','`aa','`ab','`ac','`ba','`bb','`bc','`ca','`cb','`cc','`aaa'
```

is the output of the function in Figure 3.17.

Let us summarize what we have found.

**Theorem 3.4.1.** *The following sets are countable:*

$\mathbf{N}$

$\mathbf{Z}$

$\mathbf{N} \times \mathbf{N}$

$\mathbf{Q}^+$, *the set of positive rational numbers.*

*The set of all strings over a finite alphabet.*

We can say even more. The set $\mathbf{Q}$ of all rational numbers (not just the positive ones) is also countable, which should not be a surprise. The set of all strings over an *infinite* alphabet $\{a_1, a_2, a_3, \ldots, \}$ is countable, which is more of a surprise. See the end-of-chapter problems, beginning at Exercise 24. It is beginning to look as though every set can be represented by a list that includes all its elements, and that therefore all infinite sets have the same number of elements. We'll see below that this is not the case.

### 3.4.2 Uncountable sets

The main work of this section is to establish the following fact:

**Theorem 3.4.2.** *Let $A$ be any set. There is no bijective function*

$$f : A \to \mathcal{P}(A).$$

If we let $A$ be the set $\mathbf{N}$ of natural numbers, this tells us that the set of *sets* of natural numbers cannot be put into one-to-one correspondence with $\mathbf{N}$. In fact, we will give two proofs of Theorem 3.4.2—or rather, two different views of the same proof— one very abstract and generic, applicable to every set $A$, and the other a fresh look at how the argument works, specialized to the case where $A = \mathbf{N}$. When we interpret this argument in terms of computer programs, we get a very important result on what programs can and cannot do.

Here is our first argument. We are actually going to show that there is no *onto* function from $A$ to $\mathcal{P}(A)$. This tells us there can't be a bijective function, since bijective functions are onto. What that means is that for any function $f : A \to \mathcal{P}(A)$, there must be some $B \subseteq A$ that is not in the range of $f$.

So suppose we have a function $f : A \to \mathcal{P}(A)$. It assigns to every element $a \in A$ a subset $f(a) \subseteq A$. Now $f$ might send $a$ to a subset that $a$ itself belongs to, and it might send some other element $b$ to a subset that $b$ doesn't belong to. In any case, we can distinguish between the elements $x \in A$ such that $x \in f(x)$, and those such that $x \notin f(x)$. Let us look at the set of all the elements $x$ in the latter category:

$$B = \{x \in A : x \notin f(x)\}.$$

Then $B \in \mathcal{P}(A)$. We will complete the argument by showing that $B$ cannot be in the range of $f$. If it were, then there would be some $b \in A$ such that $f(b) = B$. But then we can ask the question, is $b \in B$?

If $b \in B$, then by the definition of $B$, $b \notin f(b) = B$. That is, if $b$ is an element of $B$ then it's *not* an element of $B$.

On the other hand, if $b \notin B$, then by the definition of $B$, $b \in B$. In other words, if $b$ is not in $B$ then it is.

This absurd conclusion resulted from supposing that there *is* an element $b$ of $A$ with $f(b) = B$. So the real conclusion is that no such $b$ can exist. So no matter what the function $f$ is, it cannot be onto, which proves our theorem.

Let's take the argument apart a little and look at what it is saying in the case where $A = \mathbf{N}$. If there were a bijection, or even an onto function $f : \mathbf{N} \to \mathcal{P}(N)$, then there would be a list

$$f(0), f(1), f(2), \ldots$$

that includes all the subsets of $\mathbf{N}$. Now let us adapt the encoding of subsets by sequences of bits that we used to prove Theorem 3.3.1. So a subset $B$ of $\mathbf{N}$ is encoded by an infinite sequence $(b_0, b_1, \ldots)$ of bits such that $b_i = 1$ if $i \in B$, and $b_i = 0$ otherwise. For instance, the set of even natural numbers is encoded by $(1, 0, 1, 0, 1, 0, \ldots)$, the empty set by $(0, 0, 0, \ldots)$. Our function $f$ thus gives us a list that contains all the infinite sequences of bits. It might begin

$$\begin{aligned} f(0) &\leftrightarrow 1, 0, 1, 0, 1, 0, \ldots \\ f(1) &\leftrightarrow 1, 1, 1, 0, 0, 0, \ldots \\ f(2) &\leftrightarrow 1, 1, 0, 0, 1, 1, \ldots \\ f(3) &\leftrightarrow 1, 0, 0, 1, 0, 0, \ldots \\ &\vdots \end{aligned}$$

What, in the context of this view of the problem, is the set $B = \{x \in \mathbf{N} : x \notin f(x)\}$? It's the set of all integers $j$ such that bit $j$ in the bit representation of $f(j)$ is 0. We obtain the bit representation of $B$ by taking the elements on the diagonal of the above array and 'flipping' them—changing 1s ot 0s and vice-versa:

$$\begin{aligned} &\mathbf{0}, 0, 1, 0, 1, 0, \ldots \\ &1, \mathbf{0}, 1, 0, 0, 0, \ldots \\ &1, 1, \mathbf{1}, 0, 1, 1, \ldots \\ &1, 0, 0, \mathbf{0}, 0, 0, \ldots \\ &\vdots \end{aligned}$$

so that

$$B \leftrightarrow 0, 0, 1, 0, \ldots.$$

But this sequence corresponding to $B$ cannot be on our original list of sequences, because it differs from $f(0)$ in bit 0, from $f(1)$ in bit 1, *etc.*—that is, it is different from every sequence on our list.

Because of this interpretation of the proof of Theorem 3.4.2, it is often referred to as the *diagonal argument.*

One important consequence of Theorem 3.4.2 is that the set of real numbers is uncountable.

**Theorem 3.4.3.** $\mathbf{R}$ *is uncountable.*

*Proof.* The proof uses the fact that every infinite sequence of decimal digits, *e.g.,*

$$0.123456789101112\ldots$$

$$0.333333333333333\ldots$$

represents a real number between 0 and 1, and that this representation is almost unique: the only numbers that have two different representations are those whose decimal expansion ends in an infinite sequence of 0s or of 9s; for instance

$$0.249999999999\ldots$$

$$0.250000000000\ldots$$

represent the same number.

Suppose we had a list

$$r_1, r_2, \ldots$$

that contains every real number. Let's ignore those elements of the list that are not between 0 and 1, as well as those elements whose decimal expansion contains a digit other than 0 or 1. So we have a list of real numbers that looks like

$$0.110011001\ldots$$
$$0.001001101\ldots$$
$$.$$
$$.$$

As we proved above, there must be a sequence of 0s and 1s that is not on that list, and this sequence is the decimal expansion of a real number that is not on the list (since a real number cannot have two different expansions consisting only of 0s and 1s). This shows that no such list can exist. □

### 3.4.3 Paradox!

The diagonal argument given in the preceding section was discovered by Georg Cantor in 1891. In 1901, Bertrand Russell imagined what would happen if we applied a variant of this argument to the set of all sets. A set $X$, Russell reasoned, is either an element of itself, or it is not an element of itself. Let us form the set of all $X$ in the latter category:

$$B = \{X : X \notin X\}.$$

We then ask the question, is $B \in B$? Once again, we get the contradiction that if it is, it isn't, and if it isn't it is.

Russell explained the paradox this way: Imagine a group of men in a village, one of whom is a barber. Some of the men shave themselves; the village barber shaves all the men who do not shave themselves, and no one else. Then who shaves the barber? If he shaves himself, then by the rule we just gave, he doesn't shave himself. And if he doesn't, he does.

The resolution of the barber paradox is that no village as just described could possibly exist. But there is not such an easy exit from Russell's Paradox. It tells us that we cannot just write any old formula, such as the one defining $B$ above, and assume that it defines a set. But prior to Russell's discovery, logicians who were working to provide a solid foundation for mathematics proposed systems in which one was in fact allowed to do exactly that. The result was that the foundation crumbled, and had to be rethought from the ground up.

### 3.4.4 *Unsolvable problems

The diagonal argument gives another apparent paradox if we consider it in the light of computer programs that generate sequences of bits. The resolution of the paradox reveals a very important fact about what computer programs can and cannot do. Let's return to the functions we used in

```
def function_name():
    #stuff, but no output
    while True:
        #more stuff, without output
        if flag:
            print 0
        else:
            print 1,
        #and still more stuff without output
```

Figure 3.18: A standard form for a function generating an infinite sequence of bits.

Section 3.4.1 to enumerate infinite sequences. If we were only interested in generating sequences of 0s and 1s, we could put these functions in the form shown in Figure 3.18.

Such a function has only two output statements, embedded in a single `if-else` statement at the bottom of an infinite `while` loop. The other stuff in the program can be just about anything, as long as it contains no output.

We will use a couple of facts about Python programs: First, you can check the syntax of Python programs with a Python program. That is, one can write a Python function `syntax_check(s)` that returns `True` if the string `s` is a legal Python function, and returns `False` otherwise. With just a slight tweak, we can create a function `standard_form_syntax_check(s)` that returns `True` if `s` is a function in the standard form shown in Figure 3.18.

We already saw above that we can write a Python function that enumerates all the strings over a finite alphabet. We can thus combine this with `standard_form_syntax_check,` enumerating each string in turn, and checking if it is in standard form. We can thus create a Python function:

```
def standard_form(n)
```

that on input `n` returns the $n^{th}$ string in the enumeration that is a valid Python function in the standard form. Observe that this function prints no output.

Here is the second fact that we need: You can write a Python *interpreter* in Python: This is a function

```
def interpret(s)
```

that takes a string `s` as input, and, if `s` is a valid Python function with no arguments, runs `s`. We can specialize `interpret` to obtain a function

```
def get_output(s,n)
```

with the following behavior: if `s` is a valid Python function in normal form, then `get_output` returns 0 or 1, the $n^{th}$ element of the sequence generated by the function. If `s` does not have the required form, then the behavior of `get_output` is not specified. We can now put these elements together in a new function:

```
def diagonal():
    n=1
    while(True):
        s=standard_form(n)
        val=get_output(s,n)
        flag=(val!=0)
        if flag:
            print 0
        else:
            print 1
        n=n+1
```

We do not actually call the functions `normal_form` and `get_output`, but rather include the code for these functions as part of `diagonal`.

The function `diagonal` is a function in standard form. Thus the text of this function is equal to the string returned by `standard_form(n)` for some value of $n$. But the $n^{th}$ bit printed by `diagonal` is the opposite of the $n^{th}$ bit printed by `standard_form(n)`. The contradiction is the same as before—the sequence generated by `diagonal` is designed to be different from all the sequences generated by programs in standard form, and yet `diagonal` is in normal form.

So the function `diagonal` cannot exist. But what was wrong with our argument constructing this function? It is not the assumption that a Python syntax-checker and a Python interpreter can be written in Python. Rather, it is the behavior of `get_output(s,n)`. What if the function `s` contains an infinite loop inside the hidden stuff, so that we never get to the $n^{th}$ output? We have described our standard form syntactically, but not every Python function in standard form generates an infinite sequence: some of them get stuck inside an infinite loop within the main loop and stop printing output.

Perhaps we can modify our `standard_form` function so that in addition to checking for correct syntax, it also filters out functions that go into infinite loops (apart from the main infinite `while True` loop). Maybe there is a function:

```
def better_standard_form(n):
```

that returns the $n^{th}$ function in normal form that does not get stuck and actually generates an infinite sequence. If we replaced the invocation of `standard_form` in `diagonal` by this improved version, we would get back our paradoxical conclusion—that `diagonal` is a function in standard form that behaves differently from all the functions in standard form.

And this is precisely what the diagonal argument proves: The function `better_standard_form` *does not exist*: there is no Python program that can read another Python program as input, and tell whether it gets stuck in an infinite loop.

Of course, there is nothing special about Python here. No C program can tell if a C program gets stuck in a loop, and likewise Java, or MATLAB. The only property we require is that the programming languages are sufficiently powerful to check syntax and interpret code of strings in the language. For that matter, no C program can tell if a Python program gets stuck in a loop, or vice-versa, because we can syntax-check and interpret Python programs in C and C programs in Python. The diagonal argument has uncovered a fundamental limitation in the power of computer programs.

We will return to this in the last chapter of the book, when we discuss Turing machines, and use the same kind of argument to reach a bolder conclusion: Determining if a program contains an infinite loop is an *undecidable* problem—there is no algorithm, period, that can solve it.

## 3.5 Historical Notes

The practice of treating a set (more often referred to as a 'class' in earlier writing on the subject) as a single entity–along with the basic properties of inclusion, intersection, and the like–figure importantly in the works of early writers on symbolic logic, dating back to Boole. Boole, like others writing on the subject, usually identified a proposition with the class of all things for which the proposition is true.

Venn diagrams were discussed at length in 1880 by John Venn, in 'On the Diagrammatic and Mechanical Representation of Proposition and Reasonings'. Venn even describes a mechanical device, incorporating these diagrams, for performing logical computations. The idea of using such circle diagrams in logic long predates Venn, who attributed the practice to Euler, in the 18th century.

Many of the modern notations associated with sets (*e.g.,* the symbols for union, intersection, complement, element) and functions were introduced by Giuseppe Peano in the 1890's.

Blaise Pascal wrote a treatise describing the triangular table of binomial coefficients and many of its properties in 1653. The paper was published in 1665, several years after his death, and the diagram soon became known as 'Pascal's Triangle' in the mathematical literature. In fact, there are descriptions of the triangle in much older mathematical writings from China, India, Persia and Europe.

'Set theory' as commonly understood in mathematics is concerned with the treatment of infinite sets as we outlined in Section 3.4. The difficulties inherent in trying to reason about the size of infinite sets were noted as far back as the 17th century by Galileo, who concluded that the relations of equality, less than, and greater than, could not be meaningfully assigned to infinite collections. This was the prevailing view until the work of Georg Cantor in the late 19th century, who introduced the modern notion of equal cardinality and established the distinction between countable and uncountable sets.

Russell's paradox was discovered by Bertrand Russell in 1902.

The application of Cantor's diagonal argument to prove the undecidability of computational problems is due to Turing, and we will have much more to say about it in the last chapter of the book. The treatment we gave in this chapter, based on Python programs printing sequences of bits, while not the traditional one, is essentially the same idea.

## 3.6 Exercises

### 3.6.1 Sets

1. Let $A = \{2, 3, 4\}$. Which of the following is true?

   (a) $2 \in A$

   (b) $2 \subseteq A$

(c) $\{2\} \in A$

(d) $\{2\} \subseteq A$

(e) $\{2\} \notin A$

(f) $\emptyset \in A$

(g) $\{4, 2\} \subseteq A$.

2. Let $A = \{a, b, c\}$, $B = \{b, c, d, e\}$. Which of the following is true?

   (a) $(a, b) \in A \times B$.

   (b) $(b, a) \in A \times B$.

   (c) $\{a, b\} \in A \times B$.

   (d) $\{a, b\} \subseteq A \times B$.

   (e) $\{(a, b)\} \subseteq A \times B$.

   (f) $B \subseteq A \times B$.

   (g) $\{a, b\} \in \mathcal{P}(A)$.

   (h) $|A \times B| = |\mathcal{P}(B)|$.

3. Draw the directed graph

$$G = (\{1, 2, 3, 4\}, \{(1, 2), (2, 4), (3, 4), (4, 3), (2, 2)\}).$$

4. An *undirected graph* is an ordered pair $(V, E)$ where $V$ is a set and $E \subseteq P(V)$, where $|e| = 2$ for every $e \in E$. The idea is that an edge of an undirected graph does not have an arrow, and so is simply described as a set of two vertices, rather than an ordered pair.

   Write the (undirected) graph depicted in Figure 3.19 as an ordered pair of sets. (Use the style of the expression for $G$ in the preceding problem, but remember that the edges are just sets, not ordered pairs.)

5. Let $A = \{1, 2, 3, 4, 5\}$, $B = \{0, 2, 4, 6\}$. Find

   (a) $A \cup B$

   (b) $A \cap B$

   (c) $A - B$

   (d) $B - A$

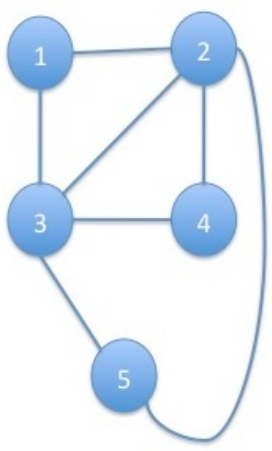



Figure 3.19: An undirected graph

6. Here is another operation on sets, called *symmetric difference.* It is defined as follows

$$A \Delta B = (A - B) \cup (B - A).$$

   (a) Illustrate this operation with a Venn diagram.

   (b) As we've noted, $\cup$ corresponds to the propositional operation $\vee$ and $\cap$ to $\wedge$ because

$$x \in A \cup B \quad \text{if} \quad \text{and} \quad \text{only} \quad \text{if} \quad (x \in A) \vee (x \in B)$$

$$x \in A \cap B \quad \text{if} \quad \text{and} \quad \text{only} \quad \text{if} \quad (x \in A) \wedge (x \in B).$$

   In a similar manner, $A \Delta B$ corresponds to another propositional connective. What is it?

7. Demonstrate the following set identities using Venn diagrams.

   (a) $(A - B) - C \subseteq A - C$.

   (b) $(A - C) \cap (C - B) = \emptyset$

   (c) $(B - A) \cup (C - A) = (B \cup C) - A$

8. Can one conclude that $A = B$ if $A, B, C$ are sets such that the following condition holds? In each case, either give a counterexample, or proof that $A = B$.

   (a) $A \cup C = B \cup C$.

   (b) $A \cap C = B \cap C$.

   (c) $A \cap C = B \cap C$ and $A \cup C = B \cup C$.

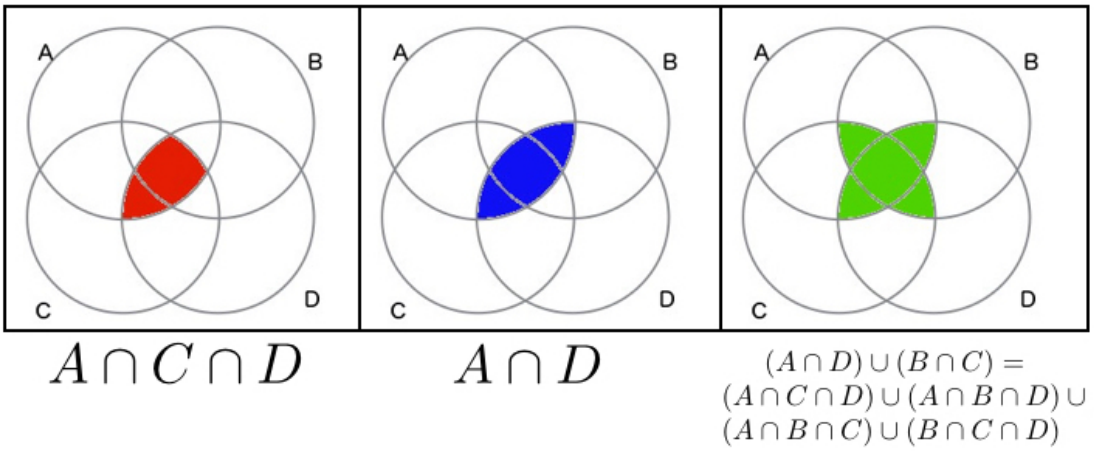



Figure 3.20: A Venn diagram 'proof' of a false set identity

9. Figure 3.20 shows a succession of three Venn diagrams. The shaded region in the leftmost panel is $A \cap C \cap D$, and the union of the four regions with the same shape is equal to the shaded region in the right panel. The shaded region in the center panel is $A \cap D$, and the union of the two regions with the same shape is *also* equal to the shaded region in the right panel. Thus this diagram 'proves' the set identity:

$$(A \cap C \cap D) \cup (A \cap B \cap C) \cup (A \cap B \cap D) \cup (B \cap C \cap D) = (A \cap D) \cup (B \cap C).$$

*(a)* Show that this identity is in general false by giving an example of sets $A, B, C, D$ for which the equation does not hold. (HINT: It is possible to construct a counterexample with very small sets.)

*(b)* So what is wrong with the reasoning in this proof? What exactly makes the Venn diagram method fail here, while it works correctly for the examples we gave in the text?

*(*c)* What is the *right* way to draw Venn diagrams for four sets?

### 3.6.2 Functions

10. Each of the descriptions below gives a candidate definition of a function $f : A \to B$, where $A$ and $B$ are specified sets. In each case, you are to tell whether the formula really defines a function, and if it does, whether the function is one-to-one, onto, both, or neither. Don't just give the answer; explain why it is true.

(a) $f : \mathbf{Z} \to \mathbf{R}$, $f(n) = \pm\sqrt{n}$.

(b) $f : \mathbf{Z} \to \mathbf{R}$, $f(n) = \sqrt{|n|}$.

(c) $f : \mathbf{R} \to \mathbf{R}$, $f(x) = \sqrt[3]{x}$.

(d) $f : A \times B \to B$, $f((x, y)) = y$, where $A = \{2, 3, 4\}$, $B = \{2, 3\}$.

(e) Let $A, B$ be the same sets as in part (d). $f : B \to A$, $f(x) = x + 1$.

(f) Let $A$ be the same set as in part (d). $f : A \to \mathcal{P}(A)$, $f(x) = \{x\} \cup \{3\}$.

(g) $f : \mathbf{R} \to \mathbf{Z}$, $f(x)$ is largest integer $m$ such that $m \leq x$. (For example $f(3) = f(3.2) = 3$. $f(-2.7) = -3$. This is called the *floor* function and is usually denoted by $\lfloor x \rfloor$.

11. Find formulas for $f \circ g$ and $g \circ f$, where $f, g : \mathbf{R} \to \mathbf{R}$ are defined by the formulas

$$f(x) = x^2 + 1, g(x) = 3x - 2.$$

Is either $f \circ g$ or $g \circ f$ one-to-one or onto?

12. Let $f : A \to B$, $g : B \to C$ be functions. Which of the following statements is true in general (*i.e,* regardless of what functions and sets are involved)? Which are not true in general? To figure out the answer, you might try drawing a few pictures. If you believe that the statement is true in general, you must give a proof, but if you believe that it is not true in general, a single counterexample (a picture will do) is sufficient.

About the proofs: To prove that a function $h : X \to Y$ is one-to-one, you have to show that whenever $h(x_1) = h(x_2)$ then $x_1 = x_2$. To prove that $h$ is onto, you have to show that for every $y \in Y$ there exists an $x \in X$ such that $f(x) = y$.

(a) If $f$ and $g$ are one-to-one, then $g \circ f$ is one-to-one.

(b)If $f$ and $g$ are onto, then $g \circ f$ is onto.

(c) If $g$ and $g \circ f$ are one-to-one, then $f$ is one-to-one.

(d) If $g$ and $g \circ f$ are onto, then $f$ is onto.

13. This exercise concerns properties of the extended function notation described in Section 3.2.6. Let $f : X \to Y$ be a function, and let $W, W_1, W_2 \subseteq X$; $Z, Z_1, Z_2 \subseteq Y$. Which of the following properties are true in general? If the stated property is true, try to prove it. If false, give a counterexample. (In these instances, counterexamples in which $X$ and $Y$ are very small sets are easy to find.)

*(a)* $f(W_1 \cup W_2) = f(W_1) \cup f(W_2)$.

*(b)* $f(W_1 \cap W_2) = f(W_1) \cap f(W_2)$.

*(c)* $f(X - W) = f(X) - f(W)$.

*(d)* $f^{-1}(Z_1 \cup Z_2) = f^{-1}(Z_1) \cup f^{-1}(Z_2)$.

*(e)* $f^{-1}(Z_1 \cap Z_2) = f^{-1}(Z_1) \cap f^{-1}(Z_2)$.

*(f)* $f^{-1}(Y - Z) = X - f^{-1}(Z)$.

### 3.6.3 Counting

14. In the text we saw the equation

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Find an analogous formula for

$$|A \cup B \cup C|.$$

The expression can use the operations of addition, subtraction, and intersection, but should not employ union. (HINT: Apply the original equation for the union of two sets twice. De Morgan's Law helps here.) What about the union of four sets? Of $n$ sets?

15. Older Massachusetts license plates consisted of a sequence of three digits followed by a sequence of three letters. You can think of this as an element of the Cartesian product of three copies of the set $\{0, 1, \ldots, 9\}$ with three copies of the set $\{A, B, \ldots, Z\}$. How many different such license plate numbers are possible? Is this sufficient to give a different license plate number to every registered vehicle in the state? (This is a math problem combined with a factual research problem—you will have to hunt for the information on the number of registered vehicles.) Would the same scheme be sufficient to give a different license plate number to every registered vehicle in California? You might want to also look up the standard format for California license plates.

16. How many different three-person committees can be formed from a group of ten people? How many such committees can be formed if part of the committee description is the person designated as the committee chairman? (For example, a committee consisting of Joe, Jack and Joan with Joe as chairman is different from the committee with the same three members and Joan as chairman.)

17. How many four-person committees can be formed from a group of ten people if two of the members are designated as co-chairs?

18. Give a combinatorial proof of the third identity for binomial coefficients:

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

by showing that there is a bijection between the sets

$$\{A \subseteq \{1, 2, \ldots, n+1\} : |A| = k+1\}$$

and

$$\{B \subseteq \{1, 2, \ldots, n\} : |B| = k\} \cup \{\{B \subseteq \{1, 2, \ldots, n\} : |B| = k+1\}.$$

19. Suppose you expand the expression $(a^2 + 2b)^5$. What is the coefficient of $a^6b^2$? Express the answer in terms of binomial coefficients.

20. How many ways are there to arrange the letters of the word MISSISSIPPI? Give an expression for this in terms of binomial coefficients. To get you started, how many ways are there to choose positions for the four S's in the word?

21. What is the coefficient of $x^2y^2z^2$ in $(x + y + z)^3$? Give an answer in terms of binomial coefficients. (This is closely related to the preceding problem.)

22. Write a Python function `powerset(a)` that takes a set `a` as an argument and returns a list of all the subsets of `a`. (The reason we do not ask for the *set* of all subsets of `a` is that in Python a set object is not hashable, and thus cannot be an element of another set object. An alternative is to use the Python type `frozenset`, which can be an element of a set, but we prefer this simpler formulation of the problem using lists.) Here is a sample run of this function:

```
>>> powerset({1,2,3})
[set([]),set([1]),set([2]),set([1, 2]),set([3]),
set([1, 3]),set([2, 3]),set([1, 2, 3])]
```

It is tricky to get this right!

### 3.6.4 Infinite Sets

23. Show that if $A$ is a countable set, and $B \subseteq A$ is infinite, then $B$ is countable. (How would we produce an enumeration of $B$ from an enumeration of $A$?)

24. Prove the claim made in the text that the set of all rational numbers is countable. You can use the fact, demonstrated above, that the set $\mathbf{Q}^+$ is countable.

25* Show that the set of all *finite sequences* of positive integers is countable. Such a sequence looks like

$$43, 1, 978624, 591.$$

This is equivalent to the claim made in the text that the set of finite strings over an infinite alphabet $\{a_1, a_2, \ldots\}$ is countable, since we can identify the sequence above with the string

$$a_{43}a_1a_{978624}a_{591}.$$

HINT: There are a couple of ways to approach this problem: You can either describe the enumeration directly (and even write a Python function that prints out the enumeration), or you can encode each such sequence by a string over the alphabet $\{a, b\}$ and thus find a one-to-one correspondence between the set of sequences of integers and a subset of the set of strings over $\{a, b\}$. Then use the fact that this set is countable and the result of Exercise 23.

26. Write a Python function that enumerates all the rational numbers, positive, negative and zero. The numbers should be printed in the form `-4/17, 355/113,` *etc.* The enumeration should print each rational number exactly once. (HINT: The tricky part is making sure that the same rational number is not printed more than once, so if your program prints `3/2,` say, it should not subsequently print `6/4.` There are several ways to ensure this, either by maintaining a list of the numbers that have already been printed, or by doing some arithmetic to check that every fraction printed is in lowest terms. We'll learn how to do this arithmetic in Chapter 7.

27. *The Hilbert Hotel.* This was a story used by David Hilbert to illustrate the counterintuitive properties of infinite sets. Imagine that there is a hotel with infinitely many rooms, numbered 1, 2, 3, 4, 5, *etc.* Suppose further that every room of the hotel is occupied.

    *(a)* A guest arrives and asks for a room. Explain how the hotel manager can accommodate the new guest, by moving the current guests to other rooms.

    *(b)* An infinite bus, with passengers in seats numbered 1,2,3, *etc.*, now pulls up to the hotel. Explain how the manager can accommodate *all* of these new passengers by sending the current guests to new rooms.

    *(c)* Now an infinite sequence $B_1, B_2, \ldots$, of infinite buses, each filled with passengers in seats $1, 2, 3, \ldots$, arrives at the hotel. Explain how the hotel manager can accommodate all of the passengers.

28. Is the set of *irrational* numbers countable? Explain.

# Chapter 4

# Predicate Logic and Relations

*A European friend of mine related the following story: I married a widow who had a grown-up daughter. My father visited our house very often, fell in love with my stepdaughter, and married her. So my father became my son-in law, and my stepdaughter my mother, because she was my father's wife. Some time afterwards, my wife got a son—he was my father's brother-in-law, and my uncle, for he was the brother of my stepmother. My father's wife, i.e., my stepmother, had also a son, he was of course my brother, and in the meantime my grandchild, for he was the son of my daughter. My wife was my grandmother, for she was my mother's mother. I was my wife's husband and grandchild at the same time. And as the husband of a person's grandmother is his grandfather, I was my own grandfather.*

–from the *Public ledger,* Memphis, Tennessee, October 18, 1866.[1]

## 4.1 Predicate Logic

### 4.1.1 What we talk about when we talk about love

Alex loves Brett, but Brett does not love Alex. Dana is not a girl. Everybody loves Raymond. Nobody loves Chris. Everybody loves somebody. Not all girls love themselves. All the boys love all the girls, but some girl loves only one boy.

Each of these sentences asserts something. Some of them can be partly decomposed into simpler sentences, using the connectives of propositional logic. In Chapter 2, we saw that *any* input-output behavior that we can represent in a truth table can also be represented by a formula of propositional logic. In this sense, propositional logic is very powerful, even all-powerful. But the propositional logic viewpoint misses most of the fine structure of these assertions, which involve properties of individuals, and relations between individuals. For this we need the expressive power of the richer language called predicate logic.

*First-order predicate logic* is the language of mathematical propositions. In Computer Science, it is, among other things, the language underlying database queries, as well as the language used by automatic verification tools for the specification of hardware and software systems.

[1]The identical 'news item' appeared in numerous American newspapers in the late 19th and early 20th centuries, sometimes with other embellishments like actual places and names of the participants in the drama. It even got worked into the lyrics of a 1940's novelty song, 'I'm my Own Grandpa', by Dwight Latham and Moe Jaffe.


Figure 4.1: Two models of the sentence $L(\texttt{Alex}, \texttt{Brett})$.

In this section, we informally introduce this new language, using the sentences above as examples. This should give you a good feel for the language, even if you trip over the somewhat dense definitions of the syntax, and, especially, the semantics, which we present in the subsequent sections of the chapter.

We will write our first example sentence

Alex loves Brett.

as

$$L(\texttt{Alex}, \texttt{Brett}).$$

$L$ is called a *binary relation symbol*–'binary', because it has two arguments–and `Alex` and `Brett` are *constants*. The entire expression is a *sentence,* also called a *closed formula,* and it asserts something that is either true or false.

Whether it is true or false depends, of course, on the world in which we interpret these symbols. We can draw a picture of this world as a directed graph, in which individuals are represented as vertices, and the relation '$x$ loves $y$' as an arrow from $x$ to $y$. The digraphs in Figure 4.1 depict two different worlds in which the sentence is true; these are *models* of the sentence.

We translate the sentence

Alex loves Brett, but Brett does not love Alex.

as

$$L(\texttt{Alex}, \texttt{Brett}) \wedge \neg L(\texttt{Brett}, \texttt{Alex}).$$

Our new language contains the usual propositional connectives . (You may have already noticed that at the level of truth tables, 'and' and 'but' mean the same thing—'but' is a kind of rhetorical flourish we use to call attention to a contrast.) In Figure 4.1, the right-hand diagram is a model of this new sentence, but the left-hand diagram is not.

The translation of

Dana is not a girl.


Figure 4.2: Two proposed models for $\forall x L(x, \texttt{Raymond})$. The one on the left is not correct

is

$$\neg G(\texttt{Dana}).$$

Here $G$ is another relation symbol, this time a *unary* relation symbol, or a *predicate* symbol. We could indicate such unary relations in our diagrams by using different colors for the vertices.

Here is a different sort of formula:

$$L(x, \texttt{Raymond}).$$

The symbol $x$ is a *variable*. The formula is an *open formula*. It does not assert anything by itself, but if we substitute a constant for the variable $x$, we obtain a sentence. We say that the occurrence of $x$ is a *free variable* of the formula. .

There's another thing that we can do with these free variables, apart from replacing them by constants. We write

$$\forall x L(x, \texttt{Raymond})$$

to mean 'For every $x$, $x$ loves Raymond', or, more simply, 'Everybody loves Raymond.' The symbol $\forall$ is called a *universal quantifier*. The variable $x$ is *bound* by this quantifier; the resulting formula is now once again a sentence. The universal quantifier functions much like the conjunction of the open formulas $L(x, Raymond)$ over all the individuals $x$ in the universe. Figure 4.2 shows two candidate models of this sentence, but only the one on the right is correct: the universal quantifier ranges over *all* the individuals in the model, and so must include the assertion that Raymond loves himself.

In informal speech, 'Everybody loves Raymond' refers to everyone other than Raymond, and is silent on the question of whether Raymond loves himself. To capture this in predicate logic, we write

$$\forall x(\neg(x = \texttt{Raymond}) \rightarrow L(x, \texttt{Raymond})).$$

It is a common practice to suppose that languages used for predicate logic all include a symbol for equality. Now both diagrams in the figure are models for the sentence.

What about 'Nobody loves Chris'? This looks like it should be translated by adding a negation to the sentence above, but where do put the negation symbol? Is it

$$\neg\forall x L(x, \texttt{Chris}),$$

or

$$\forall x \neg L(x, \texttt{Chris})?$$

The original English sentence is a universal statement: it asserts a property of everybody in the universe, namely that they don't love Chris. So it is the second of the two predicate logic formulations above that is correct. The first one has a somewhat awkward formulation in English 'It is not the case that everybody loves Chris', but this is the same as saying, 'Someone does not love Chris'. In other words, this is an *existential* statement, asserting the existence of an individual with a particular property. We can also write this as

$$\exists x \neg L(x, \texttt{Chris}).$$

The symbol $\exists$ is an *existential* quantifier. We read $\exists x \phi$ as 'there exists an $x$ such that $\phi$', where typically $x$ is a free variable in $\phi$.

Just as $\forall$ functions like a conjunction, $\exists$ functions like a disjunction. We even have a version of DeMorgan's Laws: For any formula $\phi$, $\neg \forall x \phi$ means the same thing as $\exists x \neg \phi$, as the above example shows.

Things get more interesting when the two kinds of quantifiers are used together. Consider now the English sentence

Everybody loves somebody.

This asserts some property of all the individuals in our universe, so it will be translated as a universally quantified statement. We get

$$\forall x \exists y L(x, y).$$

What happens if we switch the order of the quantifiers, or the order of the arguments to $L$? We get three new sentences

$$\exists x \forall y L(x, y)$$

$$\forall x \exists y L(y, x)$$

$$\exists x \forall y L(y, x).$$

The first of these three alternatives says 'someone loves everyone', the second that 'everyone is loved by someone', and the third that 'someone is loved by everyone'. All four sentences, then, say different things, and one can prove this by constructing, for each pair of sentences, a model universe in which one sentence of the pair is true and the other false. However, there are some relations among these sentences: for example, if someone is loved by everyone, then everybody loves somebody, that is, any model of $\exists x \forall y L(y, x)$ is also a model of $\forall x \exists y L(x, y)$.

Matters are not nearly so complicated when two quantifiers of the same kind are nested: for example,

$$\forall x \forall y L(x, y)$$

and

$$\forall y \forall x L(x, y)$$

say, respectively, 'everyone loves everyone', and 'everyone is loved by everyone'. They sound different, but a little reflection shows that these two assertions say exactly the same thing.

One last example, before we leave the Land of Love:

Every boy loves a girl who loves no one.

We can approach this as follows, initially writing

$$\forall x(B(x) \to \phi(x)),$$

where $\phi(x)$ is a formula with one free variable that says '$x$ loves a girl who loves no one'. We represent $\phi(x)$ as

$$\exists y(L(x,y) \wedge G(y) \wedge \psi(y)),$$

where $\psi(y)$ says that $y$ loves no one. Putting it all together, we get

$$\forall x(B(x) \to \exists y(L(x,y) \wedge G(y) \wedge \forall z \neg L(y,z))).$$

Statements like these, with three alternating levels of quantification, are common in mathematics. As the number of levels of quantification increases, it becomes more and more difficult to grasp what the formula says.

### 4.1.2 A Database Example: The Language of Kinship

Imagine a database in which each record contains the name of a person, the person's sex, and the names of the person's mother and father. The parents themselves will have records in the database, which contain the names of *their* parents. But our database contains only a finite number of records, so this cannot be extended indefinitely; thus there will be some records that have empty parent fields. A tiny example is shown in Table 4.1, containing recent generations of the British royal family.

From this table we can extract some propositions about properties of the individuals and the relations between them. For example, Elizabeth II is female and Henry Windsor is not, facts that we express by writing:

$$\texttt{Female}(\texttt{ElizabethII}), \neg\texttt{Female}(\texttt{HenryWindsor}).$$

Elizabeth II is Charles Windsor's mother, so we write:

$$\texttt{Mother}(\texttt{ElizabethII}, \texttt{CharlesWindsor})$$

and similarly:

$$\texttt{Father}(\texttt{GeorgeV}, \texttt{GeorgeVI}) \wedge \texttt{Father}(\texttt{GeorgeV}, \texttt{EdwardVIII}).$$

These are formulas of predicate logic in which `Female`, `Mother`, and `Father` are relation symbols in our language, the first a unary relation and the others binary.

An open formula with a free variable, like

$$\texttt{Mother}(\texttt{ElizabethII}, x).$$

becomes a sentence when we substitute individuals in the database for the free variable. We can think of such a formula as a *query* to the database: *Find all individuals $x$ that make the formula true.* The response to this query is the two individuals Charles Windsor and Anne Windsor. We can

| Name | Mother | Father | Sex |
|---|---|---|---|
| George V | | | Male |
| Mary of Teck | | | Female |
| Edward VIII | Mary of Teck | George V | Male |
| George VI | Mary of Teck | George V | Male |
| Elizabeth Bowes-Lyon | | | Female |
| Margaret Windsor | Elizabeth Bowes-Lyon | George VI | Female |
| Anthony Armstrong-Jones | | | Male |
| David Armstrong-Jones | Margaret Windsor | Anthony Armstrong-Jones | Male |
| Elizabeth II | Elizabeth Bowes-Lyon | George VI | Female |
| Philip Mountbatten | | | Male |
| Anne Windsor | Elizabeth II | Philip Mountbatten | Female |
| Mark Phillips | | | Male |
| Peter Phillips | Anne Windsor | Mark Phillips | Male |
| Zara Phillips | Anne Windsor | Mark Phillips | Female |
| Charles Windsor | Elizabeth II | Philip Mountbatten | Male |
| Diana Spencer | | | Female |
| Henry Windsor | Diana Spencer | Charles Windsor | Male |
| William Windsor | Diana Spencer | Charles Windsor | Male |
| Catherine Middleton | | | Female |
| George Windsor | Catherine Middleton | William Windsor | Male |
| Charlotte Windsor | Chaterine Middleton | William Windsor | Female |

Table 4.1: A database of recent generations of the British Royal Family

combine formulas with the usual propositional connectives to define new properties and relations. For example,

$$\texttt{Mother}(x, y) \vee \texttt{Father}(x, y)$$

defines the relation '$x$ is a parent of $y$'. We could, as a shorthand, introduce a new relation symbol `Parent` as an abbreviation for this. The response to this query is the collection of all pairs $(a, b)$ of individuals such that $a$ is a parent of $b$. There are 26 such pairs in our example database.

What if you want to say '$x$ is $y$'s grandfather'? This is the same as asserting that there is some person $z$ such that:

$$\texttt{Father}(x, z)$$

and (using the abbreviation `Parent`)

$$\texttt{Parent}(z, y).$$

We can thus use the existential quantifier to write

$$\exists z(\texttt{Father}(x, z) \wedge (\texttt{Parent}(z, y)).$$

What if we wanted to say '$x$ is $y$'s brother'? This means that $x$ is male, $x$ and $y$ have the same father, and $x$ and $y$ have the same mother. We can do also do this with existential quantifiers:

$$\neg\texttt{Female}(x) \wedge \exists z(\texttt{Mother}(z,x) \wedge \texttt{Mother}(z,y)) \wedge \exists w(\texttt{Father}(w,x) \wedge \texttt{Father}(w,y)).$$

Notice that although we used different bound variables $z$ and $w$, we don't really have to. We could just as well write

$$\neg\texttt{Female}(x) \wedge \exists z(\texttt{Mother}(z,x) \wedge \texttt{Mother}(z,y)) \wedge \exists z(\texttt{Father}(z,x) \wedge \texttt{Father}(z,y)).$$

Every occurrence of $z$ in the first clause is bound by the first existential quantifier, and every occurrence of $z$ in the second is bound by the second existential quantifier.

But there is actually a problem with this formula, in that it does not capture precisely what we mean by 'brother': If we choose $x$ and $y$ to be the *same* individual, then the formula becomes true. Since we usually do not consider someone to be his own brother, we will need to use the relation '='. We can then write our formula as

$$\neg(x = y) \wedge \neg\texttt{Female}(x) \wedge \exists z(\texttt{Mother}(z,x) \wedge \texttt{Mother}(z,y)) \wedge \exists z(\texttt{Father}(z,x) \wedge \texttt{Father}(z,y)).$$

We could have translated the above sentence instead as

$$\neg(x = y) \wedge \neg\texttt{Female}(x) \wedge \exists z \exists w(\texttt{Mother}(z,x) \wedge \texttt{Mother}(z,y) \wedge \texttt{Father}(w,x) \wedge \texttt{Father}(w,y)).$$

You can read this as 'there exists a person $z$ such that there exists a person $w$ such that...'. This is the same thing as saying 'there exist persons $z$ and $w$ such that...'. Here it is essential that the two bound variables $z$ and $w$ be different. (From here on, we will also replace $\neg(x = y)$ by the abbreviation $x \neq y$.)

Suppose we wanted to say that all of Charles Windsor's children are male. We can do this using the existential quantifier, but is more direct to use the universal quantifier.

$$\forall x(\texttt{Father}(\texttt{CharlesWindsor}, x) \rightarrow \neg\texttt{Female}(x)).$$

Sentences such as this one can also be thought of as queries to the database–these are *boolean queries*, in which the desired response is not a set of individuals but 'True' or 'False'. In this case the query will give the value 'True', because all of Charles's children are male. But what about these sentences:

$$\neg\exists x(\texttt{Father}(\texttt{HenryWindsor}, x) \wedge \neg\texttt{Female}(x)).$$

$$\forall x(\texttt{Father}(\texttt{HenryWindsor}, x) \rightarrow \texttt{Female}(x)).$$

The first sentence seems to say 'Henry Windsor has no son'. That certainly is true, since Henry Windsor has no children. The second seems to say, 'every child of Henry Windsor is female'. Is that true? It is, because no matter what individual we substitute for $x$, $\texttt{Father}(\texttt{CharlesWindsor}, x)$ is false, and thus the implication $\texttt{Father}(\texttt{CharlesWindsor}, x) \rightarrow \neg\texttt{Female}(x)$ is true.

Database query languages in wide use are based on a formalism called *relational algebra*. What we have presented here is an alternative formulation, called the *domain relational calculus*. A basic theorem of database theory is that the two formalisms can express exactly the same queries. However, not everything of interest can be expressed in this language; for example, there is no sentence in this predicate logic that says '$x$ is an ancestor of $y$', As a result, useful query languages are often supplemented with additional power so that this sort of query can be posed.

### 4.1.3 The syntax of predicate logic

We now turn to formal definitions of the syntax and semantics of first-order predicate logic. We begin with the syntax. As with propositional logic, our definition of the syntax uses a context-free grammar. The precise formulation of the grammar depends on the choice of constants and relation symbols. Below is a grammar for our first example, first-order predicate logic with relation symbols for 'loves', and 'boy', and 'girl'.

$$
\begin{array}{rcl}
\textbf{variable} & \to & x|y|x_1, x_2, \ldots \\
\textbf{constant} & \to & \texttt{Alex}, |\texttt{Brett}|\texttt{Chris}|\texttt{Dana} \\
\textbf{term} & \to & \textbf{var}|\textbf{constant} \\
\textbf{unary\_relation} & \to & B|G \\
\textbf{binary\_relation} & \to & L \\
\textbf{atomic\_formula} & \to & \mathbf{T}|\mathbf{F}|\textbf{unary\_relation}(\textbf{term})| \\
 & & \textbf{binary\_relation}(\textbf{term}, \textbf{term})| \\
 & & \textbf{term} = \textbf{term} \\
\textbf{formula} & \to & \textbf{atomic\_formula}|(\textbf{formula} \vee \textbf{formula})| \\
 & & (\textbf{formula} \wedge \textbf{formula})|\neg\textbf{formula}| \\
 & & \forall\ \textbf{variable formula}|\exists\ \textbf{variable formula}
\end{array}
$$

Different languages of first-order predicate logic will have a different assortment of constant and relation symbols, and may contain relation symbols with larger numbers of arguments. The tree in Figure 4.3 depicts the derivation in this grammar of the sentence

$$\exists x(L(\texttt{Chris}, x) \wedge \forall y(L(\texttt{Chris}, y) \to x = y).$$

That is, 'Chris loves exactly one person'. (We treat $\alpha \to \beta$ as an abbreviation for $\neg\alpha \vee \beta$.).

In a nutshell, the grammar says that formulas of predicate logic are built from atomic formulas using the usual propositional connectives as well as the quantifier symbols. The atomic formulas, in turn, assert properties of, and relations between, *terms,* which denote individual elements. A term can be either a constant or a variable.

To each formula $\phi$ we associate the set of *free variables* $FV(\phi)$ of the formula, which we define recursively as follows: If $\phi$ is an atomic formula, then $FV(\phi)$ is the set of variables occurring in $\phi$. We then set

$$
\begin{array}{lllll}
FV(\neg\phi) & = & FV(\phi) & & \\
FV((\phi \vee \psi)) & = & FV((\phi \wedge \psi)) & = & FV(\phi) \cup FV(\psi) \\
FV(\forall x\phi) & = & FV(\exists x\phi) & = & FV(\phi)\backslash\{x\}.
\end{array}
$$

So, for example, the set of free variables in $x = y$ is $\{x, y\}$, and in $L(\texttt{Chris}, y)$ is $\{x\}$. The set of free variables in

$$L(\texttt{Chris}, y) \to x = y$$

is also $\{x, y\}$, and in

$$(L(\texttt{Chris}, x) \wedge \forall y(L(\texttt{Chris}, y) \to x = y))$$

it is $\{x\}$. The set of free variables in the original example is accordingly $\{x\}\backslash\{x\} = \emptyset$, which makes the formula a sentence.

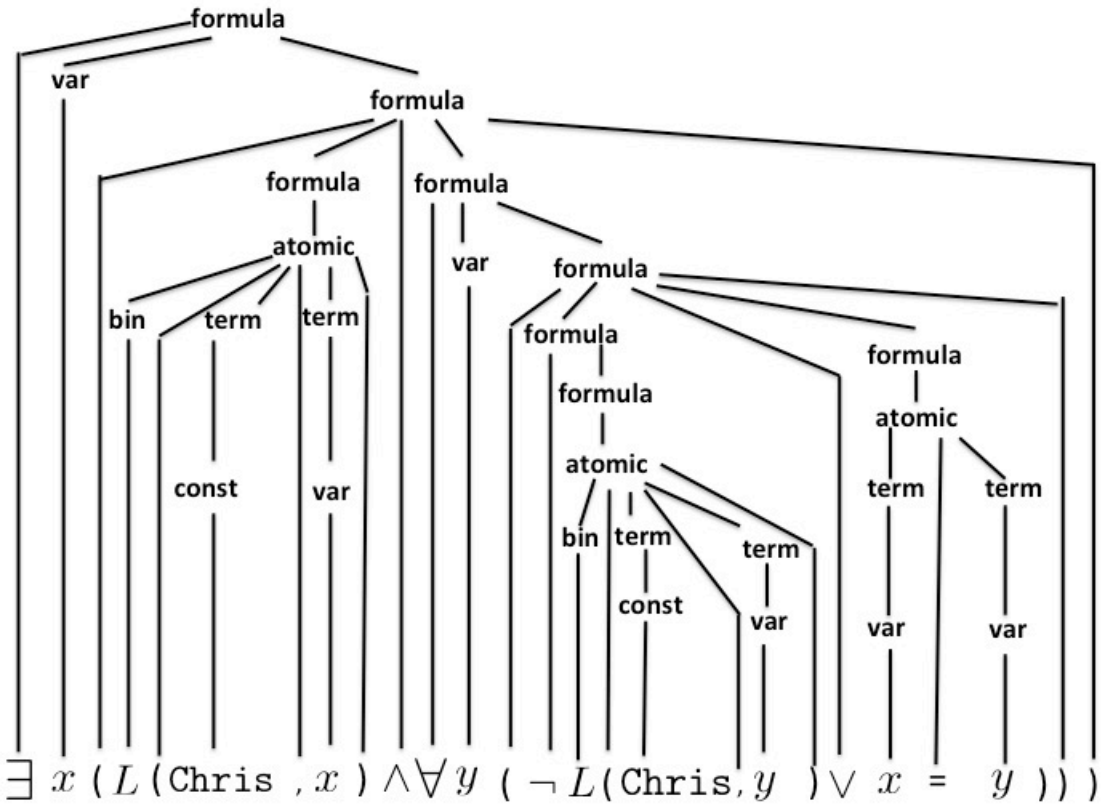



Figure 4.3: Derivation of $\exists x(L(\texttt{Chris}, x) \wedge \forall y(L(\texttt{Chris}, y) \rightarrow x = y))$.


Figure 4.4: An interpretation of the predicate logic of the Land of Love

### 4.1.4 The semantics of predicate logic

In propositional logic, the truth or falsehood of a formula depends on truth values that we assign to the variables in the formula. In predicate logic, truth or falsehood depends on the world in which we interpret the constant and relation symbols of the formula, and the values (which now are *not* truth values) that we assign to the variables of the formula. We've already seen how this works, but let us give a precise description.

An *interpretation* $\mathcal{I}$ of a first-order predicate logic consists of a nonempty set $D_\mathcal{I}$ called the *domain* of $\mathcal{I}$, along with an element $c_\mathcal{I}$ of $D_\mathcal{I}$ for each constant symbol $c$ of the logic, and a relation $R_\mathcal{I}$ on $D_\mathcal{I}$ for each relation symbol of the logic. The relation $R_\mathcal{I}$ is a unary relation (that is, a subset of $D_\mathcal{I}$) if $R$ is a unary relation symbol, a binary relation (that is, a subset of $D_\mathcal{I} \times D_\mathcal{I}$) if $R$ is a binary relation symbol, *etc.*

Figure 4.4 is a graphical depiction of one of infinitely many possible interpretations of our Land of Love logic, in the case where the only constant symbols are `Alex` and `Chris`. The elements of the domain are drawn as vertices. We illustrate the interpretation $L_\mathcal{I}$ of the binary relation symbol $L$ by arrows between the relevant vertices. For the unary relation symbols $B$ and $G$, we draw the vertex as a small square if it represents a domain element for which the property $B_\mathcal{I}$ is true, and as a circle if not. We draw the vertex red if $G_\mathcal{I}$ is true and black otherwise. If we had more than one binary relation symbol, we could use different colors for the arrows. (Once we get past binary relations, graphical depictions become more difficult, and it would probably be clearer to use a table.)

Let us call the three elements of the domain 1,2 and 3, reading from left to right. So in this interpretation,

$$D_\mathcal{I} = \{1, 2, 3\},$$

$$\texttt{Alex}_\mathcal{I} = 1, \texttt{Chris}_\mathcal{I} = 2,$$

$$L_\mathcal{I} = \{(1, 2), (2, 3)\}, B_\mathcal{I} = \{1, 2\}, G_\mathcal{I} = \{1\}.$$

Note that the interpretation of the symbols can be quite arbitrary: There is no requirement in the definition that every domain element $d$ satisfy exactly one of $B_\mathcal{I}(d)$ or $G_\mathcal{I}(d)$. In the pictured example, 1 is both a 'boy' and a 'girl', while 3 is neither. Nor is there a requirement that every element of the domain be represented by a constant symbol of the language—in our example the element 3 is not associated with any constant.

Let $V$ be the set of variables that contains $FV(\phi)$, and let $\alpha : V \to D_\mathcal{I}$ be a function—this is how we assign values to the free variables of a formula. We write

$$(\mathcal{I}, \alpha) \models \phi$$

to mean that $\phi$ is true in the interpretation $\mathcal{I}$ with the assignment $\alpha$. We give a few examples to show how this works.

Let $\mathcal{I}$ be the interpretation depicted in Figure 4.4. Then

$$(\mathcal{I}, \alpha) \models L(\texttt{Chris}, y)$$

if and only if $\alpha(y) = 3$. While $\alpha$ might be defined on a larger domain than just $\{y\}$, only the value of $\alpha$ on $y$ is relevant here.

We have

$$(\mathcal{I}, \alpha) \models (L(\texttt{Chris}, y) \to x = y)$$

if $\alpha(y) = \alpha(x) = 3$, or if $\alpha(y) \neq 3$.

The key to the semantics of predicate logic is the meaning of quantified formulas. Let $\alpha : V \to D_{\mathcal{I}}$ be an assignment, and let $x$ be a variable, which may or may not be in $V$. We can define a new assignment

$$\alpha' : V \cup \{x\} \to D_I$$

that is identical to $\alpha$ on all variables with the possible exception of $x$. That is, if $x \in V$, we can set $\alpha'(x)$ to be either equal to or different from $\alpha(x)$, and if $x \notin V$, we set $\alpha'(x)$ to be an element of $D_I$, but for all other variables $y \in V$, we have $\alpha'(y) = \alpha(y)$. We then say $(\mathcal{I}, \alpha) \models \forall x \phi$ if for *every* such extension $\alpha'$ of $\alpha$, $(\mathcal{I}, \alpha') \models \phi$. We say that $(\mathcal{I}, \alpha) \models \exists x \phi$ if there is *some* such extension $\alpha'$ of $\alpha$ with $(\mathcal{I}, \alpha') \models \phi$.

Let's apply this to our example above. Whether

$$(\mathcal{I}, \alpha) \models \forall y (L(\texttt{Chris}, y) \to x = y)$$

depends only on the value of $\alpha(x)$. We require that no matter how we extend $\alpha$ to assign a value to $y$, the resulting assignment will make $(L(\texttt{Chris}, y) \to x = y)$ true in this interpretation. If $\alpha(x)$ is 1 or 2, then we could make $\alpha'(y) = 3$ and falsify the formula, so we must have $\alpha(x) = 3$ to have the quantified formula be true. Since we now know that there is an assignment of a value to $x$ that makes the formula true, we have

$$(\mathcal{I}, \alpha) \models \exists x (L(\texttt{Chris}, x) \wedge \forall y (L(\texttt{Chris}, y) \to x = y)$$

for any assignment $\alpha$. Since the formula is a sentence, the actual values of $\alpha$ are irrelevant, so we just write

$$\mathcal{I} \models \exists x (L(\texttt{Chris}, x) \wedge \forall y (L(\texttt{Chris}, y) \to x = y),$$

and say that the sentence is true in the interpretation $\mathcal{I}$, or equivalently, that $\mathcal{I}$ is a model of $\phi$.

Of course, you can see that 'Chris loves exactly one person' is true in this interpretation just by glancing at the diagram!

### 4.1.5 Logically valid and equivalent formulas

We observed earlier that any model of

$$\exists x \forall y L(y, x)$$

is also a model of

$$\forall x \exists y L(x, y).$$

Thus the sentence

$$\exists x \forall y L(y, x) \rightarrow \forall x \exists y L(x, y)$$

is true in *every* interpretation. This has nothing to do with some special property of people, or of love. In this sense, the sentence is analogous to the tautologies of propositional logic that are true regardless of the truth values assigned to variables. Such sentences are called *logically valid* sentences.

More generally, we say that two formulas of predicate logic are *equivalent* if they have the same truth value for every interpretation and every assignment of values to free variables. Thus a sentence $\phi$ is logically valid if it is equivalent to the sentence $\mathbf{T}$.

The sentences

$$\forall x P(x) \wedge \forall y Q(y)$$

and

$$\forall x (P(x) \wedge Q(x))$$

are equivalent. One way to see this is to think of the universal quantifier as conjunction. If the domain elements are $a_1, a_2, \ldots$ then the first formula says

$$(P(a_1) \wedge P(a_2) \wedge \cdots) \wedge (Q(a_1) \wedge Q(a_2) \wedge \cdots)$$

while the second says

$$(P(a_1) \wedge Q(a_1)) \wedge (P(a_2) \wedge Q(a_2)) \wedge \cdots$$

The equivalence follows from the associativity and commutativity of $\wedge$. These quantified formulas are not *exactly* the same things as abbreviations for propositional formulas, since the domains can be infinite, but the basic reasoning still applies.

In a similar manner, we can treat existential quantification like disjunction.

You might be tempted to then think we can distribute universal quantification over disjunction, and conclude that

$$\forall x (P(x) \vee Q(x))$$

and

$$\forall x P(x) \vee \forall x Q(x)$$

are equivalent. To see that they are not, imagine a domain with just two elements, one colored red and the other colored blue. We interpret the unary relation symbol $P$ by the property of being red, and $Q$ by the property of being blue. For this interpretation, the first sentence is true, because every element of the domain is either red or blue. But the second is false, since it asserts that either every element is red or every element is blue. Remember that if formulas are equivalent then they have the same truth value under any interpretation of the symbols, so this single counterexample is enough to show that they are not equivalent.

On the other hand, as we observed earlier the analogues of DeMorgan's Laws do work: $\neg \exists x \phi$ is equivalent to $\forall x \neg \phi$, and $\neg \forall x \phi$ to $\exists x \neg \phi$ for any formula $\phi$. This fact is quite useful. In mathematical reasoning we typically do *not* work with symbolic logical formulas like these, but we do work with more informal statements that are equivalent to rather complicated formulas with several nested levels of quantifiers, and we often need to be able to form the negations of such statements.

For example, consider an interpretation of formulas in which the domain elements are Major League baseball players and Major League baseball teams. We interpret $N(x)$ to mean that $x$ is

a National League team. and $L(x, y)$ to mean that $x$ is a player who has played on team $y$. The statement 'Some player has played on every National League team' is written

$$\exists x \forall y (N(y) \to L(x, y)).$$

We form the negation of this working from the outside in:

$$\begin{aligned} \neg\exists x \forall y (N(y) \to L(x, y)) &\equiv \\ \forall x \neg\forall y (N(y) \to L(x, y)) &\equiv \\ \forall x \exists y \neg (N(y) \to L(x, y)) &\equiv \\ \forall x \exists y (N(y) \wedge \neg L(x, y)). & \end{aligned}$$

Under the interpretation we specified, we can read this as 'for every player there is some National League team for which he has not played'.

### 4.1.6 Predicate logic with function symbols, and the language of arithmetic

Mathematical statements about the set $\mathbf{N}$ of natural numbers are informally expressed in English by sentences like:

*The sum of two even numbers is even.*

Now this is really a universally quantified statement, since it means that for any two natural numbers $x$ and $y$, the sum $x + y$ is even:

$$\forall x \forall y ((\texttt{even}(x) \wedge \texttt{even}(y)) \to \texttt{even}(x + y)).$$

The unary predicate `even` can in turn be expressed in terms of quantifiers and addition: $\texttt{even}(t)$ is equivalent to

$$\exists z (t = z + z).$$

Let's give a precise description of this logical language of arithmetic. As with our preceding examples, we have variables and quantifiers, and we will also include constants 0 and 1. We could introduce relation symbols to express the operations of arithmetic; for instance we could write something like $\texttt{Plus}(x, y, z)$ to mean $z = x + y$. However, it is more convenient to treat $+$ and $\times$ as what are called *function symbols*.

The grammar for this language contains additional rules

$$\begin{aligned} \textbf{binary\_function} &\to \texttt{Plus}|\texttt{Times} \\ \textbf{term} &\to \textbf{variable}|\textbf{constant}|\textbf{binary\_function}(\textbf{term}, \textbf{term}) \end{aligned}$$

We will use the standard notations $x + y$ and $x \times y$ instead of writing $\texttt{Plus}(x, y)$ and $\texttt{Times}(x, y)$. So, for example,

$$(x + 1) \times ((y + 1) + 0)$$

is a term, and

$$(x+1)\times((y+1)+0)=(z+1)+1$$

is an atomic formula. Terms are algebraic expressions that denote numbers, once numbers are substituted for the free variables, and atomic formulas are equations between terms. Our language for arithmetic contains no relation symbols, but a logic could contain both relation symbols and function symbols.

What if you want to use this language to say that $x<y$? You might want to introduce a new relation symbol for this, but we can define this relation in terms of the core language described above: This is the same as saying that $y=x+z$ for some natural number $z$ that is different from 0. This is expressed by the formula with two free variables

$$\exists z(y=x+z\wedge z\neq 0).$$

(Keep in mind that formally, $z\neq 0$ is an abbreviation for the formula $\neg(z=0)$.) We can use $x<y$ in other formulas, as an abbreviation for this longer formula. We can even write $t_1<t_2$, where $t_1$ and $t_2$ are terms, as an abbreviation for the same formula with $t_1$ and $t_2$ in place of $x$ and $y$.

What if we want to say that $y$ is a multiple of $x$? It's the same idea: We write

$$\exists z(y=x\times z)$$

Again, this is a formula with two free variables $x$ and $y$. We can abbreviate this as $x|y$ (read '$x$ divides $y$'), and use it with any terms in place of $x$ and $y$, as a new relation symbol.

We can use the language we've built up so far to say that a number is *prime*: A natural number $x$ is prime if it is different from 1, and has no divisors other than itself and 1. We can write this as

$$x\neq 1\wedge\forall y(y|x\rightarrow(y=1\vee y=x)).$$

Again, this has one free variable $x$, and we can abbreviate it as $\texttt{prime}(x)$, where `prime` is treated as a relation symbol.

There is a famous theorem of arithmetic, which we will study in Chapter 7, that says that the set of primes is infinite. We can rephrase this as, 'for any prime, there is a larger prime'. We write this as

$$\forall x(\texttt{prime}(x)\rightarrow\exists y(x<y\wedge\texttt{prime}(y))).$$

This has no free variables, so it is a sentence: It asserts something that is true or false as it stands (and in fact is true). It is possible to expand all the abbreviations and come up with a sentence that uses only the core symbols of our language for arithmetic, but the result will be a formula that is very long and difficult to decipher.

The result is that first-order predicate logic equipped with only the constants 0, 1 and the operations $+$ and $\times$, is capable of expressing very complex assertions about arithmetic. (In Exercise 7 you are asked to find some of these expressions.)[2]

Observe that we can give a different interpretation of this logic—the very same symbols have natural interpretations in the set of integers, in the set of rational numbers, and in the set of real numbers. As always, the truth of a sentence depends on the interpretation. For example,

$$\forall x\forall y\exists z(x+z=y)$$

is false if we interpret the sentence in $\mathbf{N}$, but true in $\mathbf{Z}$.

[2]The expressive power of this language is surprising: There is even a formula that expresses exponentiation: $x=y^z$, although this is far from obvious, and the proof that such a formula exists is quite difficult.

## 4.2 Classification of binary relations

Formally, a *relation* is a subset $R$ of

$$A_1 \times \cdots \times A_k,$$

where $k \geq 1$ is an integer and $A_1, \ldots A_k$ are sets. We say that $R$ is a $k$-ary relation, and when $k = 1, 2, 3$ we have the special words *unary, binary, ternary.*

For example, if $A$ is the set of students at this university, and $B$ is the set of all courses offered here, then

$$\texttt{HasTaken} = \{(a, b) : a \in A, b \in B, a \texttt{ has taken } b\}$$

is a binary relation. Typically we will write

$$\texttt{HasTaken}(a, b)$$

in preference to

$$(a, b) \in \texttt{HasTaken}.$$

For binary relations it is also common to use *infix* notation

$$a \texttt{ HasTaken } b.$$

If $A_1 = A_2 = \cdots = A_k = A$, then we say $R$ is a *relation on* $A$.

There is special terminology and classification for binary relations on sets, which we will explore in the sections that follow.

### 4.2.1 Equivalence Relations

We begin with an example. Let $A$ be the set of words

$$A = \{\text{‘apple’,‘peach’,‘pear’,‘plum’,‘orange’,‘lemon’}\}.$$

We will define the binary relation $R$ on this set by $R(w, w')$ if and only if $w$ and $w'$ have the same number of letters. Thus $R(\text{‘apple’, ‘lemon’})$ is true, but $R(\text{‘apple’, ‘orange’})$ is false.

This relation $R$ has some particular properties, which we list below.

- Every element of $A$ is related to itself. For instance, $R(\text{‘apple’, ‘apple’})$. We can express this property in first-order logic as
$$\forall x R(x, x).$$
Relations with this property are called *reflexive.*

- Order doesn't matter: $R(\text{‘apple’, ‘peach’})$ and $R(\text{‘peach’, ‘apple’})$ are both true. We can express this property as
$$\forall x \forall y (R(x, y) \to R(y, x)).$$
Such relations are called *symmetric.*

- If $w$ is related to $w'$, and $w'$ to $w''$, then $w$ is related to $w''$. We can write this as
$$\forall x \forall y \forall z (R(x, y) \wedge R(y, z) \to R(x, z)).$$
Such relations are called *transitive.*

A relation that is reflexive, symmetric, and transitive is called an *equivalence relation.* If $R$ is an equivalence relation on $A$ and $a \in A$, then we define the *equivalence class* of $a$ to be the set

$$[a]_R = \{b \in A : R(a, b)\}.$$

For instance, in our example above, the equivalence class of 'apple' is {'apple','peach','lemon'}.

Equivalence relations have an important property:

**Theorem 4.2.1.** *Let $R$ be an equivalence relation on a set $A$. If $a, b \in A$, then either*

$$[a]_R = [b]_R,$$

*or*

$$[a]_R \cap [b]_R = \emptyset.$$

This means that the equivalence relation partitions the set $A$ into blocks that don't overlap. For example, the full set of equivalence classes in our example is

$$\{\{\text{'apple','peach','lemon'}\}, \{\text{'pear,'plum'}\}, \{\text{'orange'}\}\}.$$

To see why Theorem 4.2.1 is true, suppose that two equivalence classes $[a]_R$ and $[b]_R$ have a nonempty intersection, so that there is some element that is in both sets:

$$c \in [a]_R \cap [b]_R.$$

Now let $d \in [a]_R$. So $R(a, d)$. We are also supposing $R(a, c)$ and $R(b, c)$. By the symmetric property $R(c, b)$, so by transitivity, $R(a, b)$. By symmetry $R(b, a)$, and thus by transitivity again, $R(b, d)$. But that means $d \in [b]_R$, so what we've proved is that every element of $[a]_R$ is an element of $[b]_R$. By precisely the same argument, every element of $[b]_R$ is in $[a]_R$, so $[a]_R = [b]_R$. That is, if the two classes have a nonempty intersection, they are identical, which is what the theorem says.

In mathematics, equivalence relations are often written with a symbol that is some variant on an 'equals' sign. For instance in high school geometry you may have seen the expression $T_1 \cong T_2$ used to mean that two triangles are congruent. This is an equivalence relation on the set of all triangles in the plane. In Chapter 7 you will see another class of important equivalence relations, the congruence relations on the set of integers.

### 4.2.2 Orders

Now let $A$ be the set of all words in English—we'll define a few new binary relations on $A$. We will use infix notation for these relations, and the symbols we use to denote the relations are all variants on the symbol $<$ .

#### Preorder

First let's define

$$w_1 \preceq w_2$$

if every letter of $w_1$ appears in $w_2$. For example

$$\text{'apple'} \preceq \text{'leaped'}.$$

This relation is reflexive and transitive, but not symmetric (for example, it is *not* the case that 'leaped' $\preceq$ 'apple'.) A relation that is both reflexive and transitive is called a *preorder.*

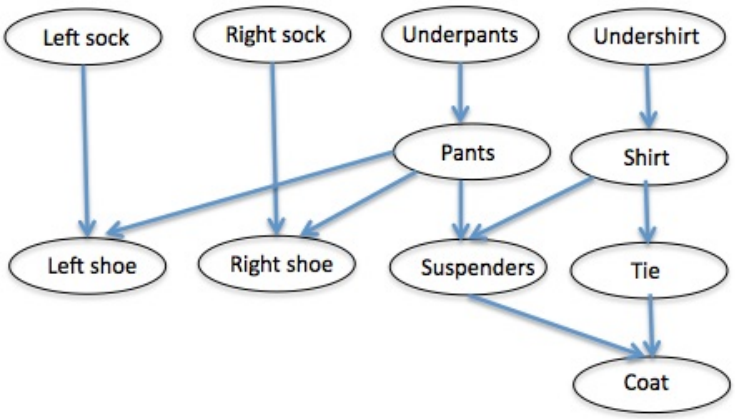



Figure 4.5: A partial order–How to get dressed in the morning: $a \preceq b$ if item $b$ cannot be put on before item $a$. In the graphical representation, this means that there is a path from $a$ to $b$.

### Partial order

Now we define

$$w_1 \subset w_2$$

if $w_1$ is a subsequence of $w_2$–that is, $w_1$ can be obtained from $w_2$ by erasing zero or more letters. For example

$$\text{'apple'} \subset \text{'appliance'}.$$

This relation is both transitive and reflexive, so it is a preorder. But it has an additional property: in our first example of a preorder, we could have distinct words that are related to each other in both directions:

$$\text{'apple'} \preceq \text{'leap'} \preceq \text{'apple'}.$$

But for our new relation, the only way to have both $w_1 \subset w_2$ and $w_2 \subset w_1$ is if $w_1 = w_2$. Relations with this property are said to be *antisymmetric*. A relation that is reflexive, transitive, and antisymmetric is called a *partial order,* and sometimes just an *order*.

Here is another example: A businessman getting dressed for work in the morning has to put his socks on before his shoes, his underpants before his pants, his shirt before either his coat or his tie. He probably wants to get his pants on before either of his shoes. On the other hand, he could put on his left sock and left shoe before he puts on his right sock, and there are many other options in the order in which he gets dressed. The relation 'has to put $a$ on before $b$' is a partial order on the set of items of clothing. Figure 4.5 is a graphical representation of this order: $a \preceq b$ if and only if there is a path in the digraph from $a$ to $b$.

### Total (*aka* linear) order

Now consider the following binary relation on the set of English words: Take the longest common prefix $u$ of $w_1$ and $w_2$. For instance, the longest common prefix of 'apple' and 'application; is 'appl'. We can write $w_1 = uv_1$, $w_2 = uv_2$. (Typically, $u$ will be the empty string, for instance if $w_1$ is 'apple'

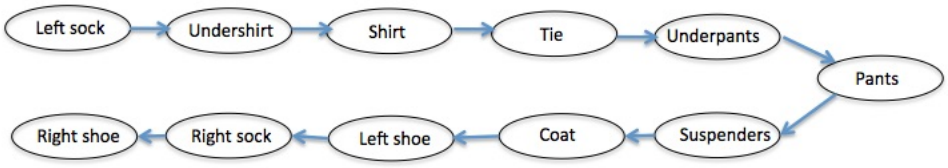



Figure 4.6: A total order–This somewhat unorthodox approach to getting dressed is nonetheless consistent with the partial order, and thus is a topological sort.

and $w_2$ is 'banana'.) If $v_1$ is the empty string, then we say $w_1 \leq_{lex} w_2$. If $v_2$ is the empty string, then $w_2 \leq_{lex} w_1$. Otherwise, compare the first letters of $v_1$ and $v_2$ in the usual alphabetic order. These letters must be different, otherwise $u$ would not be the *longest* common prefix of the two words. If the first letter of $v_1$ precedes the first letter of $v_2$ in the alphabet, then $w_1 \leq_{lex} w_2$, otherwise $w_2 \leq_{lex} w_1$. This is the usual dictionary order of words, and it is called the *lexicographic* order, for this reason. $\leq_{lex}$ is reflexive (since every word is a prefix of itself), transitive and antisymmetric, so it is a partial order. In addition, *every pair of distinct elements is comparable*: that is, for any two words $w_1, w_2$, either $w_1 \leq_{lex} w_2$ or $w_2 \leq_{lex} w_1$. Partial orders with this additional property are called *total orders,* and also *linear orders.*

In computer science, these ideas arise in the problem of scheduling: Consider again our businessman getting dressed. He has to choose some sequence in which to put his clothes on, but this sequence has to be consistent with the constraint that some items must be put on before some other items. That is, he starts from a partial order $\preceq$ on the set of items of clothing, and must arrive at a total order $\leq$ with the property that if $x \preceq y$, then $x \leq y$. This problem of extending a partial order to a total order is called *topological sorting.* Figure 4.6 shows a topological sort (one of many) of this partial order.

## 4.3 Historical Notes

Quantificational logic as a more expressive language than propositional logic was developed independently by Gottlob Frege in 1878, and by Charles Sanders Peirce in 1885. Frege used a rather complicated diagrammatic language, while Peirce's notation is closer to what we use today; he used the symbols $\Sigma$ and $\Pi$ for the existential and universal quantifiers, respectively.

The effort to develop the foundations of mathematics, and, in particular the arithmetic of the natural numbers, began with Giuseppe Peano in 1889.

The relational database model was developed by E. F. Codd, beginning in 1969. Codd showed in particular that a database query language based on first-order logic (*the relational calculus*), more or less along the lines of our example in Section 4.1.2 has the same expressive power as a procedural query language (*the relational algebra*).

Condorcet's Paradox, the subject of Exercise 23, was discovered by Condorcet in the 18th century. It was the first development in a long line of research on the mathematics of elections.

You might wonder if there is an automatic way to check if a formula of predicate logic is true in every interpretation. As we noted above, this is analogous to the problem of determining if a formula of propositional logic is a tautology. And we do have such an automatic procedure for

tautologies: Construct the truth table and check if the formula is true for all assignments. One of the great ideas of computer science, proved by Alan Turing and Alonso Church in the 1930's, is that no such algorithm exists—determining if a given sentence of predicate logic is logically valid is an undecidable problem. We will have much more to say about this in the last chapter.

## 4.4 Exercises

### 4.4.1 Translating between natural language and predicate logic

1. This exercise and the next concern predicate logic with a unary relation symbol $M$, and binary relation symbols $S$ and $T$, as well as constants `Joe`, `Ellen`, `Matt`, `Alex`, `Elizabeth`, *etc.* We interpret $M(x)$ to mean that $x$ is a man. $S(x, y)$ to mean that $x$ has seen $y$, and $T(x, y)$ to mean that $x$ has talked to $y$.

   Write sentences of predicate logic that express the same thing as the following sentences of English. (For purposes of this problem, 'is a woman' means the same thing as 'is not a man'.)

   *(a)* Alex is a woman.

   *(b)* Joe has seen Matt but has not talked to him.

   *(c)* Ellen has talked to everybody.

   *(d)* Matt has talked to no one but himself.

   *(e)* Everyone who has seen Elizabeth has talked to her.

   *(f)* No woman has seen every man.

   *(g)* Some man has talked to every woman, but no woman has ever talked to him.

2. Now solve the problem in the other direction: Take the following sentences of predicate logic, and translate them into sentences of English, using the same interpretation. (Insofar as possible, the sentences should be in precise but natural-sounding English, not in math-ese. So avoid expressions like 'for all', 'there exists', and 'if then'.

   *(a)* $S(\texttt{Matt}, \texttt{Joe}) \wedge \neg T(\texttt{Joe}, \texttt{Matt})$.

   *(b)* $\forall x(\neg M(x) \rightarrow \neg T(\texttt{Steve}, x))$.

   *(c)* $\exists x(M(x) \wedge S(\texttt{Elizabeth}, x))$.

   *(d)* $\forall x((\neg M(x) \wedge S(x, \texttt{Joe})) \rightarrow T(\texttt{Joe}, x))$.

   *(e)* $\exists x(M(x) \wedge S(\texttt{Elizabeth}, x) \wedge \forall y((M(y) \wedge S(\texttt{Elizabeth}, y)) \rightarrow (x = y)))$

   *(f)* $\forall x \exists y S(x, y)$.

   *(g)* $\exists x \exists y(S(x, y) \wedge S(y, x) \wedge T(x, y) \wedge T(y, x))$.

3. Quantifiers are called quantifiers because they express something about the number of elements that satisfy a certain property. Show how to express the following 'quantified' statements in predicate logic with a unary relation symbol $R$. You don't have to introduce any

new quantifiers to do this, although you will need to keep in mind that the atomic formula $x = y$ is always considered to be part of the language.

*(a)* Exactly one $x$ satisfies $R(x)$.

*(b)* At least one $x$ satisfies $R(x)$.

*(c)* No more than three $x$ satisfy $R(x)$.

*(d)* All but one $x$ satisfy $R(x)$.

### 4.4.2 A database language

4. In this problem we consider a toy version of a student registration database. The database consists of three tables. One is the *Student* table, which contains one record for each student. A record in this table has the form:

| **Student ID** | **Name** |
|---|---|
| 12345679 | Igor Eagle |

The second is the *Course* table, which contains a record for each course section offered in the university. Here is what two such records look like:

| **Section Number** | **Course** | **Instructor** |
|---|---|---|
| CSCI224301 | Logic and Computation | Howard Straubing |
| CSCI224302 | Logic and Computation | Sergio Alvarez |

Of course, in a realistic implementation we would expect the records in these tables to contain many more fields such as the student's address and year in school, the time and number of credits for each course, *etc.*.

Finally, there is a the *Registration* table, which contains entries like this one

| **Student ID** | **Section Number** |
|---|---|
| 12345679 | CSCI224301 |

telling us that student 123456789 is registered for section CSCI224301.

The predicate logic underlying this database has variables and constants that run over objects of different kinds: Student ID numbers, course names, instructor names, *etc.* The constants are strings like 'Igor Eagle' and 'CSCI224302' that occur as entries in the table. The relation symbols are binary relations $S$ and $R$ for the student and registration databases, and a ternary relation symbol $C$ for the course database. A sentence asserts a *property* of the database, for example

$$\exists x(S(x, \text{Igor Eagle}) \wedge R(x, \text{CSCI224301})),$$

which says that a student named 'Igor Eagle' is registered for the course CSCI224301. A formula with free variables can be viewed as a *query* to the database, whose answer is the set

of assignments to the free variables that make the formula true. For example, if you wanted to list all the section numbers and course names for courses taught by Sergio Alvarez, you could write

$$C(x, y, \text{Sergio Alvarez}).$$

This has two free variables, and the answer to the query contains the pair

$$(\text{CSCI224302}, \text{Logic and Computation}).$$

(It may contain other pairs as well; in our example records we may not have given the complete list of courses Sergio Alvarez teaches.)

In this exercise you are to write formulas expressing the following queries:

*(a)* Find the ID of all students registered in CSCI224301.

*(b)* Find the name of all students registered in CSCI224301.

*(c)* Find the names and ID numbers of all students who are registered in a section that Igor Eagle is enrolled in.

*(d)* Find the names and ID numbers of all students who are registered in a course (not necessarily the same section) that Igor Eagle is taking.

*(e)* Find the names and ID number of all students who are registered in a course that Howard Straubing is teaching.

*(f)* Find the names and ID numbers of all students who are taking only one course.

*(g)* Find the names of all students who have the same name as a different student. For example if students 111111111 and 303030303 are both named John Smith, then 'John Smith' should appear as one of the answers to the query.

### 4.4.3 The language of arithmetic

5. Give translations into English of the following three sentences. Then tell whether these sentences are true if we interpret them in the domain *(i)* $\mathbf{N}$ (the natural numbers), *(ii)* $\mathbf{Z}$ (the integers), and *(iii)* $\mathbf{R}$ (the real numbers).

   *(a)* $\exists x \forall y \exists z (x + z = y)$

   *(b)* $\forall x \forall y \exists z (x + z = y)$

   *(c)* $\forall x \exists y (y + y = x)$.

6. We saw in the text that we could define the relation $x < y$ in the first-order language of arithmetic in $\mathbf{N}$ by the sentence

$$\exists z (z \neq 0 \wedge x + z = y).$$

   *(a)* Show that if we interpret formulas of our language in the set $\mathbf{R}$ of real numbers, giving the symbols $+$, $\times$, 0, and 1 there usual meanings, then this formula is not equivalent to $x < y$.

*(*b)* Show, nonetheless, that there is a formula equivalent to $x < y$ in this interpretation. (HINT: Try to express '$x$ is positive'.)

7. Express the following statements about the natural numbers in the predicate logic of arithmetic.

   *(a)* If $m \in \mathbf{N}$ there is no number strictly between $m$ and $m+1$. (In this and the following parts of the problem, you can use the symbol $<$. In the text we saw how to define this relation in terms of the core language.)

   *(b)* If $m, n \in \mathbf{N}$, $m^2 + n^2 \geq 2mn$. Be sure to write this carefully using *nothing* but the core language, together with the symbol $<$ .

   *(c)* Every even natural number is the sum of two primes. (For this problem you may use the unary predicate symbol `prime`. The statement itself, known as *Goldbach's conjecture*, is an open problem—it has never been proved or disproved.)

   *(d)* $2 \times 3 = 4 + 2$. Note that '2', '3', and '4' are *not* constant symbols in our language, so we need, for example, to define 3 as the term ((1+1)+1).

   *(e)* $x$ is a power of 2. The answer to this will be a formula in which $x$ is a free variable. (HINT: Use the unary predicate symbol `prime`. How can you express the fact that $x$ is a power of 2 in terms of the prime divisors of $x$?)

### 4.4.4 Predicate Logic in general

8. In the following formulas, identify all free occurrences of variables, all bound occurrences of variables, all function symbols, and all relation symbols. (You may assume that the letters $x, y, z$ denote variables, not constants.)

   *(a)* $\forall x(P(x) \vee z = Q(x, y))$.

   *(b)* $\forall x P(x) \vee Q(R(x, y), z)$.

   *(c)* $\forall x((\exists y \exists z R(x, z, z')) \wedge T(y, z))$.

   *(d)* The sentence

   $$\forall x \exists y \exists z (x = z + z)$$

   looks a bit peculiar because of the quantification of the variable $y$, which does not appear within the scope of the quantifier. (The same peculiarity appears in part *(c)* of this problem.) Nonetheless, the sentence is perfectly legal. If we interpret it in the natural numbers, is it true? How would you write an equivalent formula without using the quantifier $y$?

9. As we saw in the text, a good way to depict an interpretation of predicate logic, at least those in which there are only unary and binary relation symbols, is to draw a little directed graph, where the vertices represent domain elements, and the arrows represent relations holding between these elements. We can even use different colors for vertices and arrows if we want to interpret more than one unary or binary relation symbol. We illustrate this in Figure 4.7.In this interpretation the domain is a two-element set; we interpret a binary relation symbol $R$

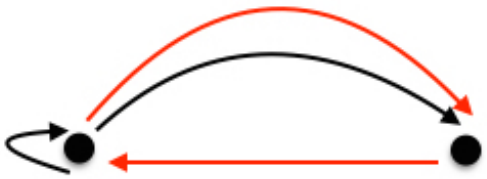


Figure 4.7: A small model for sentences of predicate logic: the red and black arrows interpret binary relation symbols $R$ and $B$.

by the red arrows and a binary relation symbol $B$ by the black arrows. In the diagram the sentence

$$\forall x \exists y B(x, y)$$

is false, because there is no black arrow originating at the right-hand vertex, while

$$\forall x \exists y R(x, y)$$

is true, because there are red arrows originating at both vertices.

Which of the following sentences is true in the pictured interpretation?

*(a)* $\forall x \exists y B(y, x)$.

*(b)* $\exists x B(x, x)$

*(c)* $\exists x \exists y (B(x, y) \wedge R(x, y))$.

*(d)* $\exists x \forall y (B(x, y) \wedge R(x, y))$

*(e)* $\forall x \forall y (B(x, y) \rightarrow \exists z R(x, z))$.

*(f)* $\exists x \forall y R(x, y)$.

*(g)* $\exists x \forall y B(x, y)$.

10. Which of the following is a logically valid sentence? To show that something is *not* a valid sentence, it is sufficient to produce an interpretation in which the sentence fails to be true. The method described in the preceding problem is very helpful here—draw a directed graph representing a binary relation $R$ in which the sentence is false. To show something *is* a valid sentence, try to give a cogent argument in words why it must be true in every interpretation. Incidentally, it is usual to require in the definition of the semantics that the domain be *nonempty*, which means, for example, that $\forall x P(x) \rightarrow \exists x P(x)$ is valid. (It would not be if we allowed empty domains.)

    *(a)* $\forall x \forall y R(x, y) \rightarrow \forall x R(x, x)$.

    *(b)* $\forall x \exists y R(x, y) \rightarrow \exists x R(x, x)$.

    *(c)* $\exists y \forall x R(x, y) \rightarrow \forall x \exists y R(x, y)$.

    *(d)* $\forall x \exists y R(x, y) \rightarrow \exists x \forall y R(x, y)$.

11. The tall tale in the quotation at the beginning of the chapter depends on interpreting 'parent' to mean what is usually intended by 'step-parent'. Imagine we have a collection of individuals with binary relations `Mother`, `Father`, and `Spouse`. We require `Spouse` to be symmetric:

$$\forall x \forall y(\texttt{Spouse}(x, y) \to \texttt{Spouse}(y, x)).$$

We will redefine $\texttt{Parent}(x, y)$ to be equivalent to

$$\texttt{Mother}(x, y) \vee \texttt{Father}(x, y) \vee \exists z(\texttt{Spouse}(x, z) \wedge (\texttt{Mother}(z, y) \vee \texttt{Father}(z, y))).$$

(That is, a 'parent' of $y$ is either $y$'s mother or father, or is married to $y$'s mother or father.) Finally, we will define $\texttt{Grandparent}(x, y)$ to be equivalent to

$$\exists z(\texttt{Parent}(x, z) \wedge \texttt{Parent}(z, y)).$$

Draw a directed graph, with different colored arrows for the relations `Mother`, `Father`, `Spouse`, `Parent`, `Grandparent`, that satisfies the above conditions, and also satisfies

$$\neg\exists x \texttt{Spouse}(x, x) \text{ (no one is their own spouse)}$$

$\neg\exists x \exists y(\texttt{Spouse}(x, y) \wedge (\texttt{Mother}(x, y) \vee \texttt{Father}(x, y)))$ (no one is the spouse of their mother or father)

along with

$$\exists x \texttt{Grandfather}(x, x).$$

Try to do this with as few vertices as possible. The story does this with six individuals, but that was probably done just to make it sound even more tangled; there is an example with fewer people.

12*. This problem shows a limitation of the 'small graphs' approach described in Exercise 9. Consider the following three sentences:

$$\phi_1 : \exists x \forall y \neg R(y, x).$$

$$\phi_2 : \forall x \forall y \forall z((R(x, z) \wedge R(y, z)) \to x = y).$$

$$\phi_3 : \exists x \forall y \neg R(x, y).$$

*(a)* Now imagine that you have a directed graph representing $R$. Describe in English (and with diagrams!) the graph properties expressed by the sentences $\phi_1, \phi_2$ and $\phi_3$.

*(b)* Show that the sentence

$$(\phi_1 \wedge \phi_2) \to \phi_3$$

is true for every *finite* directed graph.

*(c)* Show nonetheless that $(\phi_1 \wedge \phi_2) \to \phi_3$ is *not* a valid sentence by giving an example of an interpretation in an *infinite* domain in which the sentence is false.

13. *(a)* For each part of Exercise 1, write a sentence of predicate logic that is equivalent to the negation of the given English sentence. You should be able to do this automatically, and the only negation symbols in your answer should be in front of atomic formulas. So for instance, it is ok to have $\neg T(x, y)$ somewhere in the answer, but not $\neg\forall y \cdots$.

    *(b)* Then take these answers and translate them back into English. Try, insofar as possible, to forget the original English sentence, and do this relying only on the answers you produced in *(a)*.

14. An infinite sequence

$$x_1, x_2, \ldots$$

    of real numbers converges to a limit $L$ if you can get as close to $L$ as desired by going sufficiently far along in the sequence. Formally, the sequence is convergent if the following sentence, interpreted in the real numbers, is true:

$$\exists L \forall \epsilon (\epsilon > 0 \to \exists N \forall n ((n \in \mathbf{Z}^+ \wedge n > N) \to |x_n - L| < \epsilon)).$$

    In other words, for any measure $\epsilon$ of 'closeness', no matter how small, there exists a place $N$ in the sequence, such that all subsequent terms are within $\epsilon$ of $L$. Note that there are four nested levels of quantification in this sentence.

    *(a)* The negation of this sentence says that $\{x_n\}_{n>0}$ is a divergent sequence. Write the formal definition of divergent sequence, by using the rules for negating quantifier formulas.

    *(b)* Now reformulate the sentence you produced in *(a)* in more natural language.

### 4.4.5 Equivalence relations and orders

15. *(a)* Define the following binary relation on the set of all people: $x \approx y$ if $x$ and $y$ have the same mother. Is $\approx$ an equivalence relation?

    *(b)* Now define $x \sim y$ if $x$ and $y$ have a parent, either mother or father, in common. Is $\sim$ an equivalence relation?

16. Define a relation on the set of points in the coordinate plane by $(x, y) \equiv (x', y')$ if

$$|x| + |y| = |x'| + |y'|.$$

    Show that $\equiv$ is an equivalence relation. What do the equivalence classes look like when you plot them in the coordinate plane?

17. Define a relation on the set of points in the coordinate plane by setting $(x, y) \leq_1 (x', y')$ if $x \leq x'$. Is $\leq_1$ a preorder? a partial order? a total order?

18. Repeat the preceding question for the relation defined by $(x, y) \leq_2 (x', y')$ if $x \leq x'$ and $y \leq y'$.

19. Repeat the preceding question for the relation defined by $(x, y) \leq_3 (x', y')$ if $x \leq x'$ or $y \leq y'$.

20. Repeat the preceding question for the relation defined by $(x, y) \leq_4 (x', y')$ if $x < x'$, or $x = x'$ and $y \leq y'$. (For example, $(3, 4) \leq_4 (3, 5)$, and $(4, 5) \leq_4 (5, 3)$.)

21. Let $A$ be any set. Is the relation $\subseteq$ on $\mathcal{P}(A)$ a preorder? partial order? total order?

22. Give formulas of predicate logic with a binary relation symbol $R$ that define the properties '$R$ is antisymmetric' and '$R$ is a total order'.

23. We are conducting an election with three candidates $A$, $B$, $C$. If $A$ receives 40% of the vote and $B$ and $C$ each receive 30%, then under a plurality voting system, $A$ would win the election. Let us imagine, however, that $B$ and $C$ have very similar views. Perhaps $B$ and $C$ are liberals and $A$ is a conservative: in an election between either $B$ and $A$ or between $C$ and $A$, $A$ would lose. Thus the outcome of the plurality vote might be considered unfair.

    To remedy this, we might try the following voting system: We ask each voter to rank the three candidates in order of preference. For each pair of candidates $X$ and $Y$ we say $X \prec Y$ if more than half the voters prefer $Y$ to $X$. The idea is that $\prec$ should provide a sort of global preference ranking for the candidates, which will enable us to avoid the unfairness of electing a candidate who would lose in a one-on-one contest against every opponent.

    Show that the vote can turn out in such a manner that $\prec$ is not transitive—in other words, we could have $A \prec B$, $B \prec C$, and $C \prec A$. It is thus impossible to choose a winner by this method. (This is called *Condorcet's paradox*.)

# Chapter 5

# Proofs

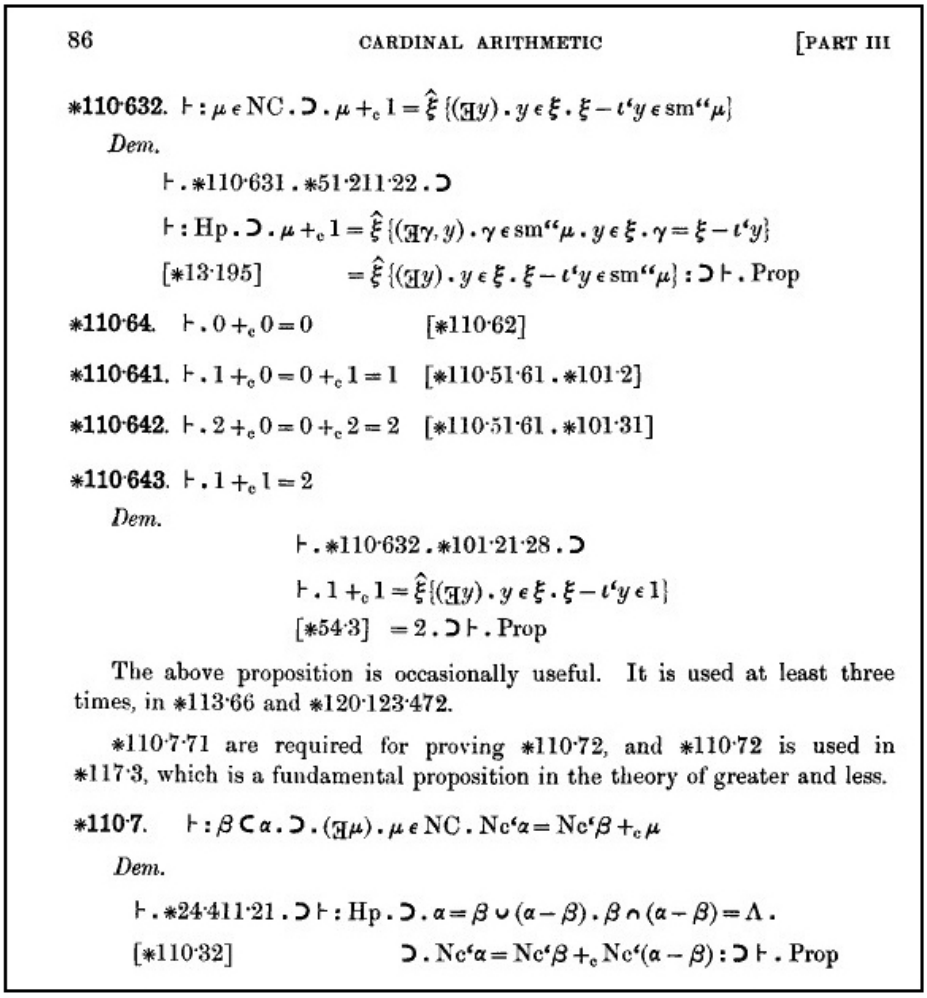
Up until this point, we've occasionally paused to give a 'proof' of something, and even went so far as to call what we were proving a 'theorem', but we have said little about what a proof is. In fact, one of the goals of this text is to develop your skills at following, dissecting, modifying, and creating mathematical proofs. This takes a lot of practice. There is no sequence of steps you can follow that will lead without fail to the proof or disproof of a statement (if there were, there would be no unsolved problems in mathematics), so we can't do much more than provide a lot of examples and some useful tips.

## 5.1 Logic and Proofs

Obviously proofs have something to do with logic! Much of the text up until now has been devoted to formal logic, and we have treated logic in a computer science-y sort of way: as a language for talking about the behavior of programs and computing devices, for specifying the desired properties of a problem solution, for formulating queries to a database. But this language was originally developed to model reasoning, and was part of a sustained research effort to provide formal rigorous foundations for mathematics. Figure 5.1, part of a page from Russell and Whitehead's *Principia Mathematica*, shows what that effort looked like: In the middle of the page is the proof of the 'occasionally useful' proposition that $1 + 1 = 2$. The text itself consists of little besides a succession of formulas (which are in fact formulas of predicate logic, with somewhat different notation from what we have used–you may recognize the symbol for the existential quantifier) together with numbers cross-referencing the formulas and theorems, and only the briefest of remarks written in ordinary language.

Yet people who do research in mathematics, even in pure mathematics, never write formal proofs like this. Figure 5.2, extracted from an article by Maryam Mirzakhani (one of the recipients of the 2014 Fields Medal, the highest honor in mathematics) is typical. The technical vocabulary makes the paper incomprehensible to the uninitiated, but the text is presented in complete English sentences and most of it is not nearly so dense with formulas as you might imagine. Some of the reasoning is illustrated by appeal to diagrams, a practice seen even in papers that are not about geometry, as this one is.

Such arguments can in principle be rendered in formal logical language, although no one would undertake the gargantuan effort involved in doing this. If one examines the proofs closely, one finds



86 CARDINAL ARITHMETIC [PART III

***110·632.** $\vdash : \mu \in \mathrm{NC} . \supset . \mu +_c 1 = \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in \mathrm{sm}``\mu\}$

*Dem.*

$\vdash . *110·631 . *51·211·22 . \supset$

$\vdash : \mathrm{Hp} . \supset . \mu +_c 1 = \hat{\xi}\{(\exists \gamma, y) . \gamma \in \mathrm{sm}``\mu . y \in \xi . \gamma = \xi - \iota`y\}$

[*13·195] $= \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in \mathrm{sm}``\mu\} : \supset \vdash . \mathrm{Prop}$

***110·64.** $\vdash . 0 +_c 0 = 0$ [*110·62]

***110·641.** $\vdash . 1 +_c 0 = 0 +_c 1 = 1$ [*110·51·61 . *101·2]

***110·642.** $\vdash . 2 +_c 0 = 0 +_c 2 = 2$ [*110·51·61 . *101·31]

***110·643.** $\vdash . 1 +_c 1 = 2$

*Dem.*

$\vdash . *110·632 . *101·21·28 . \supset$

$\vdash . 1 +_c 1 = \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in 1\}$

[*54·3] $= 2 . \supset \vdash . \mathrm{Prop}$

The above proposition is occasionally useful. It is used at least three times, in *113·66 and *120·123·472.

*110·7·71 are required for proving *110·72, and *110·72 is used in *117·3, which is a fundamental proposition in the theory of greater and less.

***110·7.** $\vdash : \beta \subset \alpha . \supset . (\exists \mu) . \mu \in \mathrm{NC} . \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mu$

*Dem.*

$\vdash . *24·411·21 . \supset \vdash : \mathrm{Hp} . \supset . \alpha = \beta \cup (\alpha - \beta) . \beta \cap (\alpha - \beta) = \Lambda .$

[*110·32] $\supset . \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mathrm{Nc}`(\alpha - \beta) : \supset \vdash . \mathrm{Prop}$

Figure 5.1: A formal proof: A page from Russell and Whitehead's *Principia Mathematica* proving $1 + 1 = 2$

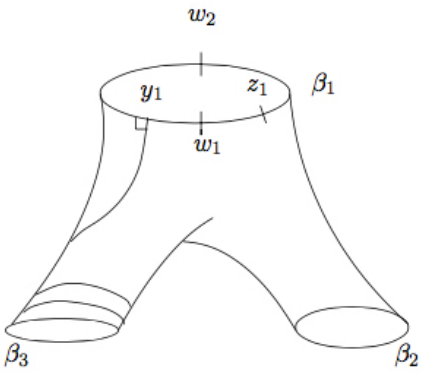



Figure 2. complete geodesics in a pair of pants

Each boundary component of $\mathcal{C}$ has two canonical points, the end points of the length minimizing geodesics connecting it to the other two boundary components.

On the other hand, we can obtain $\mathcal{C}(x_1, x_2, x_3)$ by pasting two copies of the (unique) right angled geodesic hexagons with pairwise non-adjacent sides of length $x_1/2$, $x_2/2$ and $x_3/2$ along the remaining three sides. Thus $\mathcal{C}(x_1, x_2, x_3)$ admits a reflection involution $\sigma$ which interchanges the two hexagons.

**Complete geodesics on hyperbolic a pair of pants.** A hyperbolic pair of pants contains 5 complete geodesics disjoint from $\beta_2, \beta_3$ and orthogonal to $\beta_1$. More precisely, 2 of these geodesics meet $\beta_1$ at $y_1$ and $y_2$ and spiral around $\beta_3$, the other 2 meet $\beta_1$ at $z_1$ and $z_2$ and spiral around $\beta_2$. There

Figure 5.2: Part of a proof in a contemporary mathematical research paper, by Maryam Mirzakhani.

that the methods of reasoning can all be expressed in terms of some basic laws of predicate logic. Below we will discuss what these laws are, illustrate how they are realized in practice in informal, natural language, and give a few tricks of the trade to help you along when you have to find and write proofs of your own.

## 5.2 Example: A property of even numbers

We'll begin by proving something very simple (and, frankly, kind of boring) and closely analyze both the process of discovering the proof and the formal reasoning that underlies the argument.

We already saw this example when we introduced the predicate logic of arithmetic. If you start adding randomly chosen integers, you'll notice that the sum of two even integers always turns out to be even (*e.g.,* $4 + 6 = 10$, $2 + (-8) = -6$, *etc.*). We might write the general principle as:

*The sum of two even integers is even.*

This informal language is typical of the way mathematical statements are often formulated. In truth, it is a bit sloppy, because it leaves unstated the fact that this is a universally quantified statement. It would be more accurate to write this as

*The sum of any two even integers is even.*

or

*Let $x, y \in \mathbf{Z}$ be even. Then $x + y$ is even.*

As we saw earlier, in the formal language of predicate logic, interpreted in the integers, this statement is the sentence

$$\forall x \forall y((\texttt{even}(x) \wedge \texttt{even}(y)) \rightarrow \texttt{even}(x + y)).$$

We treat $\texttt{even}(t)$ as an abbreviation for

$$\exists z(t = 2z)$$

where $z$ is a variable that does not appear free in the term $t$.

How do we go about finding a proof of such a statement? Sentences of the form $\forall x(P(x) \rightarrow Q(x))$ (or, in this case, because the universal quantification is over a pair of numbers rather than a single number, $\forall x \forall y(P(x, y) \rightarrow Q(x, y))$) are quite common. A good way to approach them is to say, 'OK, we have $x$ and $y$. What do we know about them and what are we trying to prove about them?' Here we know is that $x$ and $y$ are even, and that means that there are integers $u$ and $v$ such that

$$x = 2u, y = 2v.$$

What we are trying to prove about them is that $x + y$ is even, in other words that there is an integer $w$ such that $x + y = 2w$. How do we get there? What we know about $x$ and $y$, coupled with the distributive law for addition, gives

$$x + y = 2u + 2v = 2(u + v),$$

so that $w = u + v$ is the integer we are looking for.

Let us assemble these musings into a proof written in standard math-ese.

**Theorem 5.2.1.** *If $x, y \in \mathbf{Z}$ are even, then $x + y$ is even.*

*Proof.* Since $x$ and $y$ are even, $x = 2u$ for some integer $u$, and $y = 2v$ for some integer $v$. Thus $x + y = 2u + 2v = 2(u + v)$. Since $x + y$ is twice the integer $w = u + v$, $x + y$ is even, as required. □

## 5.3 *A closer look

The proof we just wrote is about as short and simple as mathematical proofs get, and it seems to be based on nothing more than natural intuitive reasoning. In fact, a surprisingly large number of principles of deduction are at work in this argument. Here we'll take a closer look at what those logical principles are.

A theorem is a sentence of predicate logic (in this case, the predicate logic of the integers) that we prove, usually by appealing to theorems that we have already proved. But we can't conjure these sentences out of thin air without already knowing something about arithmetic. So we also need some 'starter theorems' that we can treat as known. Such starter theorems are called *axioms.*

*'Basic properties' of numbers are axioms.* Any basic property of the algebra and ordering of numbers will be treated as an axiom. We will not be too fussy about providing an official list of these properties, or about avoiding treating something as an axiom if it can be proved from simpler properties. What we have in mind are basic algebraic identities like the associative and commutative laws and the distributive law

$$\forall x \forall y \forall z (x \times (y + z) = x \times y + x \times z),$$

which we used in our proof of Theorem 5.2.1. We also will use the fact that adding the same number to both sides of an inequality preserves the inequality, and that multiplying both sides of an inequality by a positive number preserves the inequality. This last property, for example, is the sentence:

$$\forall x \forall y \forall z ((x < y \wedge z > 0) \rightarrow x \times z < y \times z).$$

*Tautologies are axioms.* We also treat any propositional tautology as an axiom. Usually these appear so unassumingly in proofs that you scarcely notice they are being used: For example, our proof silently applied

$$(\mathtt{even}(x) \wedge \mathtt{even}(y)) \rightarrow \mathtt{even}(x),$$

which is an instance of the tautology $\phi \wedge \psi \rightarrow \phi$, early in the argument.

*Modus ponens.* The way we used the tautology just cited was to couple it with the hypothesis $\mathtt{even}(x) \wedge \mathtt{even}(y)$ in order to deduce $\mathtt{even}(x)$. The rule that one can conclude $\phi$ from $\psi$ and $\psi \rightarrow \phi$ is called *modus ponens.* Axioms alone are not enough to get our proof machine to run. We also require rules of deduction, of which *modus ponens* is one example, to tell us exactly how we infer new sentences from those that have come before.

*Deduction principle.* There is a kind of flip side to *modus ponens* that we used in the proof. We began by *assuming* $\mathtt{even}(x) \wedge \mathtt{even}(y)$, and derived $\mathtt{even}(x + y)$ as a consequence. As a result we conclude

$$(\mathtt{even}(x) \wedge \mathtt{even}(y)) \rightarrow \mathtt{even}(x + y).$$

The idea is that if you can prove $\psi$ by assuming $\phi$, then you can prove $\phi \rightarrow \psi$. (In mathematical logic, this principle is usually called the 'Deduction Theorem'.)

*Properties of equality.* We also used basic properties of equality. If we know $t_1 = t_2$, where $t_1$ and $t_2$ are terms, and if we know that some property $P(t_1)$ holds, then we can conclude that $P(t_2)$ holds. Let us be a bit more precise as to how this plays out in terms of predicate logic: The 'property' in question will be some formula $\phi$, with a free variable $x$. Suppose we have been able to deduce somehow $\phi[t_1/x]$, where by this we mean the formula that results when we substitute the term $t_1$ for all free occurrences of $x$ in $\phi$. Suppose further that we have deduced $t_1 = t_2$. Then we can conclude $\phi[t_2/x]$.

We used this principle in our proof: We consider $x + y = x + y$ as an axiom. We can treat this formula as $\phi[x/x']$, where $\phi$ is the formula $x + y = x' + y$. The formula $x = 2u$, which we deduced, allows us to conclude $\phi[x/x'] = \phi[2u/x']$, that is, $x + y = 2u + y$, and another application of the same reasoning allows us to conclude $x + y = 2u + 2v$. (But see below about the careful practice of substitution.)

*Universal generalization.* The rules for manipulating quantifiers are more subtle. What we actually proved, thanks to the deduction principle, is

$$(\mathtt{even}(x) \wedge \mathtt{even}(y)) \rightarrow \mathtt{even}(x + y).$$

But then we slapped some universal quantifiers on it and said that we proved:

$$\forall x \forall y ((\mathtt{even}(x) \wedge \mathtt{even}(y)) \rightarrow \mathtt{even}(x + y)).$$

This process of starting from 'arbitrary $x$ and $y$', proving something about these arbitrary values, and then concluding that what you proved holds for all $x$ and $y$, is sometimes called *universal generalization*.

*Universal instantiation.* If we know some property holds for *all* $x$, then we should be able to conclude that the property holds for $t$, where $t$ is any term at all. In other words, from $\forall x\phi$, we can deduce $\phi[t/x]$. We used this in our proof when we started from the distributive law and replaced, in turn, each of the quantified variables by the terms $2, u$ and $v$, respectively, to arrive at

$$2(u+v) = 2u + 2v.$$

(We *instantiate* the bound variables, hence the name of this principle.)

When you do the substitution $\phi[t/x]$, you need to be just a little careful about how you name variables. The following sentence is a fact about the integers:

$$\forall x \exists y (y = x + 3).$$

So let's apply universal instantiation and instantiate the variable $x$ by the term...$y$. We get the absurd conclusion

$$\exists y (y = y + 3)$$

which is most certainly false for integers. We just can't be silly about this and perform the substitution $\phi[t/x]$ if it causes a free variable in $t$ to be bound by a quantifier in $\phi$.

*Existential generalization.* The punch line of our proof is: now that we have found a term $u + v$ such that $x + y = 2(u + v)$ holds, we can conclude

$$\exists z (x + y = 2z).$$

In other words, $\texttt{even}(x + y)$. We can always pass from $\phi[t/x]$ to $\exists x\phi$ like this. In the example here, $\phi$ is the formula $x + y = 2z$, the term $t$ is $u + v$, and the quantified variable for which this term was substituted is $z$. The same caveat about when you can perform the substitution applies: we can't bind a free variable in $t$ by a quantifier in $\phi$.

*Existential instantiation.* At a certain point in the argument we said, in effect, $x$ is even, so let $u$ be a number such that $x = 2u$. That is, we started from $\exists z(x = 2z)$, introduced a new variable $u$ to instantiate (or, as is often said, to 'witness') this existential quantification, and proceeded to argue using $u$, yielding a result that does not involve the witness.

This is a very common mode of argument, but there are many potential pitfalls in its use, and it is actually quite difficult to formulate it correctly. It is important that the witness $u$ appear neither in the final result, nor in any preceding step. Look, for example, at this misuse of existential instantiation:

> $x$ is even, so $x = 2u$ for some $u$. $y$ is even, so $y = 2u$ for some $u$. Thus $x + y = 2u + 2u = 4u$. We have proved that the sum of two even integers is always divisible by 4.

The result here is, of course, false, and the error occurred because of how we chose to name the witnesses. Furthermore, existential instantiation does not play well together with universal generalization, which must not be applied to a formula after we instantiate the quantified variable. Consider this twisted argument:

> Let $x$ be an arbitrary integer. There is an integer $y$ such that $y = x + 7$, that is $\exists y(y = x + 7)$. Let $u$ be such an integer. So $u = x + 7$. Since $x$ is arbitrary, we can apply universal generalization to conclude $\forall x(u = x + 7)$. We can now apply existential generalization, with the result $\exists y \forall x(y = x + 7)$. Thus there is a magic integer that is exactly 7 more than every integer (including itself).

It is a little hard to put your finger on exactly what went wrong here. The choice of the witness $u$ in effect fixes the possible values of $x$, so that it is no longer arbitrary and we cannot apply universal quantification.

That is about as deep as we care to dig for now, but before we leave this section, let's assemble our arguments into something resembling a real formal proof of Theorem 5.2.1, the only such proof in this book:

| | | |
|---|---|---|
| (1) | $\texttt{even}(x) \wedge \texttt{even}(y)$ | Hypothesis |
| (2) | $\texttt{even}(x) \wedge \texttt{even}(y) \rightarrow \texttt{even}(x)$ | Tautology |
| (3) | $\texttt{even}(x)$ | 1,2, and *modus ponens* |
| (4) | $\exists z(x = 2z)$ | 3, definition of `even` |
| (5) | $x = 2u$ | 4, existential instantiation |
| (6) | $\texttt{even}(x) \wedge \texttt{even}(y) \rightarrow \texttt{even}(y)$ | Tautology |
| (7) | $\texttt{even}(y)$ | 1, 6, and *modus ponens* |
| (8) | $\exists z(y = 2z)$ | definition of `even` |
| (9) | $y = 2v$ | 8, existential instantiation |
| (10) | $\forall x \forall y \forall z(x \times (y + z) = x \times y + x \times z)$ | distributive axiom |
| (11) | $2u + 2v = 2(u + v)$ | 10, universal instantiation, applied three times |
| (12) | $\forall z(z = z)$ | we'll call this an axiom |
| (13) | $x + y = x + y$ | 12, universal instantiation |
| (14) | $x + y = 2u + 2v$ | 13, equality property applied twice |
| (15) | $x + y = 2(u + v)$ | 11,14, equality property |
| (16) | $\exists z(x + y = 2z)$ | 15, existential generalization |
| (17) | $\texttt{even}(x + y)$ | 16, definition of `even` |
| (18) | $(\texttt{even}(x) \wedge \texttt{even}(y)) \rightarrow \texttt{even}(x + y)$ | 1,17,deduction theorem |
| (19) | $\forall x \forall y((\texttt{even}(x) \wedge \texttt{even}(y)) \rightarrow \texttt{even}(x + y))$ | universal generalization, applied twice |

You can see why real mathematicians don't write formal proofs!

It was not our intention here to present in detail a formal deductive system, but merely to give a broad outline of what such a system looks like. We wish to underscore is that the customary informal arguments found in mathematical proofs can be precisely formalized, and that proofs can, at least in principle, be reduced to mechanical symbol-manipulation operating according to a few fixed rules. We will return to this very important point in Chapter 9.

## 5.4 Example: An inequality, and working backwards

The average of two numbers, of course, is found by adding them and dividing by 2. The value $(x + y)/2$ is called the *arithmetic mean* of $x$ and $y$ (or often just the *mean*, but we will really need the adjective). Simply put, it's the number halfway between $x$ and $y$.

But sometimes that's not the right way to compute an average. Suppose you have an investment account that initially has \$1000. The account grows by 7% the first year, and 3% the following year. What is the average rate of growth? If you take the arithmetic mean (5%) of the two growth rates, the suggestion is that the account behaves as though it grew by 5% in each of the two years. But in our scenario, the account balance at the end of two years is \$1102.10, while 5% annual growth both years would give us slightly more: \$1102.50. As another example, suppose the growth rate is 7% the first year and -7% the next. The arithmetic mean of 0% suggests that the account balance should be unchanged at the end of two years, but in fact you lose about \$4.90 in this scenario.

Since we are measuring growth by the ratio between the account balances at the beginning and end of each year, the correct value 'halfway' between $x$ and $y$ is the number $r$ such that

$$r/x = y/r,$$

in other words,

$$r = \sqrt{xy}.$$

This is called the *geometric mean* of $x$ and $y$. We usually define it only if $x$ and $y$ are both positive. In our first investment account example, we are looking for the geometric mean of 1.07 and 1.03, which is 1.0498, and in the second, it is the geometric mean of 1.07 and 0.93, which is about 0.9975, representing an annual loss of close to a quarter of one per cent.

In both of these examples, *the geometric mean turned out to be less than the arithmetic mean*, something you will find again and again if you test a few examples. (The exception is if you choose $x = y$, in which case you will get the same value for both means.)

Let's try to prove that this is always the case. Where do we start? What do we know at the outset? Lots of things, certainly, but unlike our previous example, we don't have a strong hypothesis like '$x$ and $y$ are even' to work from. All we have are $x$ and $y$, and all we know about them is that they are positive, so it's hard to know just where to begin.

But we do know where to end: We are trying to prove that

$$\sqrt{xy} \leq \frac{x+y}{2}.$$

To figure out how we get there, we might try to work backwards from this goal. If we square both sides of the inequality above, we get

$$xy \leq \left[\frac{x+y}{2}\right]^2.$$

Expanding the right-hand-side gives

$$xy \leq \frac{x^2 + 2xy + y^2}{4},$$

and multiplying both sides by 4 gives

$$4xy \leq x^2 + 2xy + y^2.$$

Have we gotten anywhere? There's one more thing to try: Subtract $4xy$ from both sides.

$$0 \leq x^2 - 2xy + y^2.$$

Is this true? When can you conclude that the expression on the right-hand side is never negative? You can, if you know that it's a square, or a sum of squares. As it turns out, the right-hand side is $(x - y)^2$, so this last inequality is true. It all seems to work out.

Is that a proof? Even in the informal sense, this does not constitute an acceptable proof, because we have worked backwards, reasoning from our desired conclusion back to something we know to be true, which is not at all the same thing as working in the right direction. On the other hand, this is a great way to *find* a proof, and we can turn it into a real proof as long as we can verify that all the steps are reversible.

And indeed they are, as we will see below when we present the finished product. In fact, the logical structure of this proof is simpler than the one we saw in Theorem 5.2.1. While we have more basic algebra and inequality-juggling to do, we don't have to cope with manipulating existential quantifiers. [1]

Here is our official statement and proof.

**Theorem 5.4.1.** *Let $x, y \in \mathbf{R}$ be positive. Then*

$$\sqrt{xy} \leq \frac{x + y}{2}.$$

*Proof.* Since the square of a real number is never negative

$$0 \leq (x - y)^2 = x^2 - 2xy + y^2.$$

We can add the same number to both sides of an inequality, so add $4xy$ to both sides to obtain

$$4xy \leq x^2 + 2xy + y^2 = (x + y)^2.$$

Further, we can multiply both sides of an inequality by a positive number, and keep the inequality. Multiply both sides by $\frac{1}{4}$ to get:

$$xy \leq \frac{x^2 + 2xy + y^2}{4} = \left[\frac{x + y}{2}\right]^2.$$

[1] In more detailed terms, we are proving the following sentence:

$$\forall x \forall y((x > 0 \wedge y > 0) \rightarrow \sqrt{xy} \leq \frac{x + y}{2})$$

interpreted in the real numbers. We only have the universal quantification at the beginning, so we only do universal instantiation applied to the axiom $\forall z(z^2 \geq 0)$, and universal generalization, just as we did in the previous argument.

Whenever we have an inequality $a < b$ where $a$ and $b$ are positive, we can take the square root of both sides and preserve the inequality. Applying this to the previous line we get our desired result:

$$\sqrt{xy} \leq \frac{x+y}{2}.$$

□

You may feel that our observation about taking the square root of both sides of an inequality is a little bit more than a 'basic property' and ought to be given more of a proof. We will do that later in this chapter, when we write about proofs by contradiction.

There is one point we made when we first discussed this result that we did not include in the final theorem. That is, that the only way we can have the arithmetic and geometric means equal is if $x = y$. We can prove this with our original working-backwards argument: If $\sqrt{xy} = (x+y)/2$, then we can follow all the steps in that argument to conclude $(x-y)^2 = 0$. This implies $x - y = 0$, so $x = y$.

## 5.5 Example: Another inequality, and proof by cases

When we need to prove some universally quantified statement, for instance some property that we claim holds for all real numbers, we might find that there is one argument that works for some numbers, say the positive ones, but that we require an entirely different argument for the negative numbers, and perhaps still another argument for 0. The proof in essence branches off into three subproofs, one for each of the three cases.

Many proofs contain this kind of case-by-case analysis. We will illustrate this with the proof of a fundamental inequality. The absolute value of a real number $x$, as you know, is given by

$$|x| = \begin{cases} x & \text{if } x \geq 0, \\ -x & \text{if } x < 0 \end{cases}$$

As a consequence, $x \leq |x|$ for all real numbers $x$.

Here we are going to prove the very useful *triangle inequality,* that $|x+y| \leq |x| + |y|$ for all real numbers $x$ and $y$. We'll try to get a feeling for why it is true by examining a number of possibilities, and see if we can assemble our observations into a coherent argument.

If $x$ and $y$ are both positive, then there's really nothing to prove, since there is no difference between the absolute values of the numbers involved and the numbers themselves. The situation is similar if $x$ and $y$ are both negative. In this case $|x+y| = -(x+y) = -x + -y = |x| + |y|$. Notice that in both these cases we have equality.

The more interesting cases occur when the signs of the numbers are opposite. If we play around a little with the possibilities, we see that when the positive summand has the larger absolute value, say 4 and -3, then the sum is a positive number whose value is less than the larger summand:

$$|4 + (-3)| = 4 + (-3) < 4 + 3 = |4| + |-3|.$$

If the negative summand has a larger absolute value, then the sum is a negative number whose absolute value is less than that of the summand with a larger absolute value:

$$|-4+3| = -(-4+3) = 4 + (-3) < 4 + 3 = |-4| + |3|.$$

That seems to cover all the cases (although for 'positive' in our scratch work, we should really say 'nonnegative'). Let's put the pieces together.

**Theorem 5.5.1.** *If* $x, y \in \mathbf{R}$, *then* $|x + y| \leq |x| + |y|$.

*Proof.*
*Case 1.* $x, y \geq 0$. Then $x + y \geq 0$, so $|x + y| = x + y$. Further, $|x| + |y| = x + y$, so $|x + y| = |x| + |y|$. The conclusion holds in this case, with equality.

*Case 2.* $x, y < 0$. Then $x + y < 0$, so $|x + y| = -(x + y)$. Further, $|x| = -x$, $|y| = -y$, so $|x| + |y| = -x + (-y) = -(x + y) = |x + y|$. Once again, the conclusion holds, with equality.

*Case 3.* $x \geq 0 > y$ *and* $x + y \geq 0$. Then $|x + y| = x + y$ and $y < -y = |y|$, so $|x + y| = x + y = |x| + y < |x| + |y|$. So the conclusion holds with strict inequality.

*Case 4.* $x \geq 0 > y$ *and* $x + y < 0$. Then $|x + y| = -(x + y) = -x + -y = -x + |y| \leq |x| + |y|$. Again the conclusion holds. (Strict inequality holds unless $x = 0$.)

*Case 5.* $y \geq 0 > x$ *and* $x + y \geq 0$. In this instance, Case 3 tells us $|y + x| < |y| + |x|$, which is equivalent to what we are trying to prove.

*Case 6.* $x \geq 0 > y$ *and* $x + y < 0$. Here Case 4 tells us $|y + x| \leq |y| + |x|$, which is equivalent to the desired conclusion. □

Observe that Cases 5 and 6 are really identical to Cases 3 and 4, because of the symmetry between $x$ and $y$ in the statement of the theorem. We can reduce the propagation of extra cases by observing at the outset that if $x$ and $y$ have different signs, then one of them has to be nonnegative, and because of the symmetry in the problem, we might as well suppose that $x$ is the nonnegative one. The typical way that this is handled is to write, before starting Case 3, 'Suppose $x$ and $y$ have opposite signs. We can assume without loss of generality that $x$ is the nonnegative value', and then we only need to show Cases 3 and 4 in detail. The phrase 'without loss of generality' is so common that people giving mathematics lectures write 'WLOG' on the board and in their slides, and can safely assume that the audience knows what this means.

We arrived at this proof by a rather painstaking analysis of all the different combinations possible. The result is a correct proof, but even with the six cases reduced to four, it is still a bit clunky. One's first proof of a result is usually not the *best* proof, and it often happens that you later discover a simpler, more streamlined and elegant argument. The following proof has just two cases:

*Another proof.* If $x + y \geq 0$ then $|x + y| = x + y \leq |x| + |y|$, because $x \leq |x|$ and $y \leq |y|$. If $x + y < 0$, then $|x + y| = -(x + y) = (-x) + (-y) \leq |-x| + |-y| = |x| + |y|$. So the inequality holds in both possible cases. □

A proof by cases does not involve some new principle of reasoning beyond those discussed in Section 5.3. Let's look at the simplest instance where we have two cases. This means we have a proof of some sentence $\psi$ assuming $\phi$ (the first case), and a proof of $\psi$ assuming $\neg\phi$. We thus, by the deduction principle, have proofs of $\phi \to \psi$ and $\neg\phi \to \psi$. Now

$$\phi \to \psi \to ((\neg\phi \to \psi) \to \psi)$$

is a tautology, as you can verify. Thus by *modus ponens*, we have a proof of

$$(\neg\phi \to \psi) \to \psi,$$

and by another application of *modus ponens,* we get a proof of $\psi$.

## 5.6 Counterexamples

If you want to *disprove* a universally quantified statement, it is sufficient to give a single instance for which the statement is false. Exhibiting a counterexample may be the most widely-used form of proof.

As a very simple example of a counterexample, the integers 3,5, and 7 are prime. We might be hastily conjecture that every odd integer greater than 1 is prime, but the conjecture is proved false by the very next value we test:

$$9 = 3 \times 3.$$

An instance of this with more meat on its bones, but still just as simple in concept, is the problem of *Fermat primes.* In 1650, the mathematician Pierre de Fermat observed that the numbers

$$2^{2^0} + 1 = 3, 2^{2^1} + 1 = 5, 2^{2^2} + 1 = 17, 2^{2^3} + 1 = 257, 2^{2^4} + 1 = 65537$$

are all prime numbers, and conjectured that every integer of the form $2^{2^n} + 1$ is prime. Eighty-two years later, Leonhard Euler disproved this conjecture with the single calculation[2]

$$2^{2^5} + 1 = 4294967297 = 641 \times 6700417.$$

## 5.7 Example: Yet another inequality, and proofs by contradiction

Back in Section 5.3 we said that we would permit ourselves to use, without proof, 'basic properties' of inequalities, but were a little bit vague about what those basic properties are. So let's be more explicit: You can use the fact that the $\leq$ relation on real numbers is a linear order. You can add any number to both sides of an inequality and preserve the inequality. You can multiply both sides of an inequality by a *positive* number and preserve the inequality, and multiply both sides of an inequality by a negative number and *reverse* the inequality. You can check that nothing more than these basic properties was used in our proof of the triangle inequality.

When we discussed the relation between the arithmetic and geometric means, we assumed a little more: In working backwards to find a proof, we squared both sides of an inequality and said that this preserves the inequality. When we assembled the final proof, we said that we could take the square roots of both sides of an inequality. Rather than treat these as new 'basic properties', let's try to prove them from the other principles. Doing so will reveal something important about the logistics of proofs.

First let's do the squaring:

**Theorem 5.7.1.** *If* $0 < x \leq y$, *then* $x^2 \leq y^2$.

[2]While a single multiplication really does disprove the conjecture, this begs the question of how Euler *found* the counterexample. He did not repeatedly divide prime after prime into 4294967297, but instead first proved that any divisor of this number has to have a particular form, and so was able to narrow the search considerably.

Notice that the hypothesis requires $x$ and $y$ to be positive, otherwise the conclusion is false. (*E.g.,* $-3 < -2$, but $4 < 9$.)

*Proof.* Since $x \leq y$, we can multiply both sides of this inequality by the positive number $x$ to get $x^2 \leq xy$.

Alternatively, we can multiply both sides by the positive number $y$ to get $xy \leq y^2$.

And now we apply the transitivity of $\leq$ to conclude $x^2 \leq y^2$. □

Now let's tackle the opposite operation:

**Theorem 5.7.2.** *If* $0 < x \leq y$ *then* $\sqrt{x} \leq \sqrt{y}$.

Keep in mind that we can only take square roots of nonnegative numbers, and that the symbol $\sqrt{x}$ when $x$ is positive always refers to the positive square root. (For example, we *never* write $\sqrt{4} = -2$.)

We're going to use a different kind of strategy for proving this theorem. Instead of arguing from the hypothesis to the conclusion, we are going to begin by supposing that the conclusion is *false*. We will then argue that in this case the hypothesis must be false as well. That is, we are trying to prove a statement of the form $p \to q$, and we do this by proving the *contrapositive* $\neg q \to \neg p$, which is equivalent to $p \to q$. Proofs in which we begin by assuming that the conclusion is false are called *proofs by contradiction*, or *indirect proofs,* or, if you are fond of throwing around Latin, proofs by *reductio ad absurdum*.

*Proof of Theorem 5.7.2.* Let us suppose $x$ and $y$ are positive numbers and that $\sqrt{x} \leq \sqrt{y}$ is *false*. Then we have $\sqrt{x} > \sqrt{y}$. (This is where we use the fact that $\leq$ is a linear ordering.) We now apply Theorem 5.7.1 to square both sides, and find $x > y$, contrary to the hypothesis. This completes the proof. □

In another version of indirect proof, which we will see later, instead of arguing from the negation of the conclusion to the negation of the hypothesis, we assume both the hypothesis *and* the negation of the conclusion, and deduce something we know to be false. In this case we are proving $(p \wedge \neg q) \to \mathbf{F}$, which is also equivalent to $p \to q$. This is truly a 'reduction to the absurd'.

## 5.8 Example: More on even and odd integers, 'if and only if', and a famous proof.

We'll continue with the kind of statement we proved in Section 5.2. The *product* of two even integers is also even, and the product of two odd integers is odd. The proofs of both these statements follow the same pattern as our argument about the sum of even integers. Let's just do the second one, and leave the first as an exercise.

We will take '$n$ is odd' to mean $\exists z(n = 2z+1)$, where we interpret this sentence in the integers. There is actually a very subtle issue in defining 'odd' in this way, as we will see shortly.

**Theorem 5.8.1.** *Let* $x, y$ *be odd integers. Then* $xy$ *is odd.*

*Proof.* Since $x$ and $y$ are odd, there exist integers $u, v$ with $x = 2u + 1$ and $y = 2v + 1$. Then

$$\begin{aligned} xy &= (2u+1)(2v+1) \\ &= 4uv + 2u + 2v + 1 \\ &= 2(2uv + u + v) + 1. \end{aligned}$$

We have found an integer $k = 2uv + u + v$ such that $xy = 2k + 1$, so $xy$ is odd. □

If we put our observations about products together we get the following property of the squares of integers.

**Theorem 5.8.2.** *Let $x$ be an integer. Then $x^2$ is even if and only if $x$ is even.*

The fact that is stated here should come as no surprise—even integers have even squares 0,4,16,36, *etc.*, and odd integers have odd squares 1, 9, 25, *etc.* Before we proceed to the proof, let's note something important about the statement. There's an 'if and only if' in it, which means that we need to prove a statement of the form $p \leftrightarrow q$. Such statements then require us to prove *two* different things, both $p \to q$ and its *converse* $q \to p$. It often happens that the two parts of such a proof are very different.

*Proof.* First, suppose $x$ is even. As we stated above, and as you should prove as an exercise, the product of two even integers is even, so $x^2$ is even. Now suppose $x^2$ is even. We have to prove $x$ is even. We will do this by contradiction: if $x$ isn't even, then it's odd. By Theorem 5.8.2, $x^2$ is odd, contradicting our hypothesis that $x^2$ is even. So $x$ must be even. □

There is a hidden assumption in the above argument: We *defined* odd integers to be those of the form $2m+1$ (and even integers to be those of the form $2m$), but we supposed in our proof that *odd* is the same thing as *not even*. Now that is true, but it is actually a special property of the integers that is more subtle than algebraic identities like the distributive and associative laws. We will have more to say about the even-odd distinction at the start of Chapter 7.

Theorem 5.8.2 has as a consequence one of the most famous theorems of mathematics, but we have one more step before we get there. The argument requires yet another special property of the integers.

We're going to prove that you can never have a square integer that is exactly twice another square. $49 = 7^2$ and $25 = 5^2$ are close. $9801 = 99^2$ and $4900 = 70^2$ are even closer, at least in terms of their ratios, but you can't hit it on the nose.

**Theorem 5.8.3.** *There are no positive integers $m$ and $n$ such that $2m^2 = n^2$.*

We are asked to prove a statement of the form 'there are no', that is $\neg\exists x\exists y(x^2 = 2y^2)$. As we saw in Chapter 4, this is equivalent to a universally quantified statement $\forall x\forall y(x^2 \neq 2y^2)$. But in proving such a 'nonexistence' theorem, it is almost always better to try to find a proof by contradiction, and begin by supposing that the things in question really do exist.

So suppose $2m^2 = n^2$. That makes $n^2$ even, and by Theorem 5.8.2, $n$ is even as well. So we can write $n = 2k$, for some integer $k$, and now we have

$$2m^2 = n^2 = (2k)^2 = 4k^2,$$

so

$$m^2 = 2k^2.$$

What we've proved is that if we have a square $n^2$ that is twice another square $m^2$, then $m^2$ is itself twice another square $k^2$. We could apply the argument again and find that $k^2$ is in turn twice another square $q^2$, and so on. In other words, there would be an *infinite descending chain* $n > m > k > q > \cdots$ of positive integers.

The special property of integers we require is that there is no such infinite descending chain of positive integers. Another way to say it is that if there is positive integer $n$ whose square is exactly twice another square, then *there must be a smallest one.* We will have much more to say about this *least integer principle* in Chapter 6.

The official proof is this:

*Proof.* Suppose, contrary to what we are trying to prove, that there are positive integers $m$ and $n$ such that $n^2 = 2m^2$. Suppose $n$ is the *smallest* such positive integer. Since $n^2$ is even, Theorem 5.8.2 implies $n$ is even, and thus $n = 2k$ for some positive integer $k$. This gives

$$2m^2 = n^2 = (2k)^2 = 4k^2,$$

so

$$m^2 = 2k^2.$$

We now have a positive integer $m$ whose square is exactly twice another square, and that is smaller than $n$. This contradicts the assumption that $n$ was the smallest such integer. We conclude that there is no such integer. □

Why is this such a big deal? Theorem 5.8.3 has an important corollary:

**Theorem 5.8.4.** *$\sqrt{2}$ is an irrational number.*

*Proof.* Again, we proceed by contradiction. If $\sqrt{2}$ were a rational number, there would be positive integers $m$ and $n$ such that

$$\sqrt{2} = \frac{n}{m}$$

so that

$$2 = \left(\frac{n}{m}\right)^2 = \frac{n^2}{m^2},$$

and thus

$$2m^2 = n^2.$$

But we just saw in Theorem 5.8.3 that this is impossible, a contradiction. □

And to see why *that* is such a big deal, see the Notes below.

## 5.9 Historical Notes

The *Elements* of Euclid (4th century BC) contains the first known systematic presentation of mathematics derived from a small list of primitive assumptions. Euclid began his work with a list of definitions, 'common notions' (axioms concerning basic properties of equality and inequality), and 'postulates' (basic principles of geometry). Euclid's arguments are in fact filled with hidden assumptions not addressed in the axioms, so by modern standards, his system is seriously flawed. But the scope of the work, and its subsequent impact, are monumental: everything of importance in mathematics discovered up to his time was carefully stated and proved, in a sequence of 13 books containing over 400 theorems.

There are few extant works of ancient Greek mathematics predating Euclid; what we know about them is largely through the writing of later commentators. The proof of the irrationality of the square root of 2 is widely attributed to the Pythagoreans, perhaps as early as the 6th century BC, and is alluded to in works of both Plato and Aristotle. This discovery had an enormous impact on the early development of geometry. For example, a fundamental theorem of geometry is that two triangles with corresponding angles equal are *similar*. That is, the lengths of their respective sides are in the same proportion: If one side of triangle 1 is twice the length of a side of triangle 2, then the lengths of the other sides of triangle 1 will be twice the lengths of the corresponding sides of triangle 2. (The two triangles have the same *shape*.) This theorem has a simple interpretation and a rather easy proof if one assumes that the proportions in question are always the ratio of two integers. However, the 'Pythagorean Theorem' (known long before the ancient Greeks) implies that the proportion of the diagonal of a square to the side of the square is $\sqrt{2}$. Thus the whole notion of proportion, which had appeared to be an elementary concept, required a sophisticated reworking. This was supplied in the work of Eudoxus (4th century BC) and presented in Euclid's *Elements*. Eudoxus had to bring in ideas very close to the modern notions of limits and continuity, that would not be treated with the same rigor in post-Renaissance mathematics until the 19th century.

The deductive system for first-order predicate logic that we described in Section 5.3 is very loosely based on a number of such systems developed initially by Frege in the late 19th century, and continuing with work of Hilbert in the 1920's and Gentzen in the 1930's. As we stated at the end of the section, our treatment is intended as only the roughest of outlines, and not as a complete presentation of such a system.

Mathematics as practiced by contemporary research mathematicians does not present things at the level of formality present in such a formal system. However, since the beginnings of modern digital computers, there has been great interest in automatic theorem proving. The Coq Proof Assistant, which has been in continuous development since the 1980's, is a software tool for checking and developing formal proofs. Coq has been used in practice mostly to help find proofs of correctness of computer programs, but it has recently begun to attract the attention of mathematicians outside of computer science.

## 5.10 Exercises

1. *(a)* Prove that the sum of an even integer and an odd integer is always odd.

   *(b)* Prove that the sum of two odd integers is always even.

2. *(a)* Prove that if $x$ is an integer and $y$ is an even integer, then $xy$ is even.

*(b)* It follows from part *(a)* that the product of two even integers is even. But you can derive a *stronger* conclusion about the product of two even integers–what is it?

3. The geometric mean of $-4$ and $-9$ is $\sqrt{(-4)(-9)} = \sqrt{36} = 6$, and the arithmetic mean is -6.5. Of course, this does not contradict Theorem 5.4.1, because this theorem's hypothesis requires $x$ and $y$ to be positive. But why doesn't our proof of this theorem *prove* that $6 \leq 6.5$? Find *exactly* where in the proof of the theorem this goes wrong.

4. You work six days a week. On Monday, Wednesday, and Friday, you drive your vintage Hummer H1 the 20 miles back and forth to work. On Tuesday, Thursday and Saturday, you drive your Prius. On Sunday, you go for a bike ride. The Hummer goes only 9 miles on a gallon of gasoline, the Prius gets 51 miles per gallon.

   *(a)* What is your average gas mileage for the week? (It's not 30 mpg.)

   *(b)* Find a formula involving $x$ and $y$ that gives the average gas mileage in this scenario for a car that gets $x$ miles per gallon, and one that gets $y$ miles per gallon. This expression is called the *harmonic mean* of $x$ and $y$.

   *(c)* Let $x, y$ be positive real numbers. What is the order relation between the harmonic mean, geometric mean, and arithmetic mean? We already know, from Theorem 5.4.1 that the geometric mean is less than the arithmetic mean, when $x \neq y$. So the question is where the harmonic mean fits in this scheme. Try out a few values to guess the answer, and then prove your result.

5. Prove that if $x, y$ are real numbers, then

$$|xy| = |x||y|.$$

   A simple proof by cases will work.

6. Prove that if $x, y$ are real numbers, then

$$|x - y| \geq ||x| - |y||.$$

   A proof by cases is perfectly all right, but there is a slick argument that derives this result from the original triangle inequality.

7. Let $\mathbf{u} = (x, y)$, where $x$ and $y$ are real numbers. The *norm* of $\mathbf{u}$, denoted $||\mathbf{u}||$, is defined by

$$||\mathbf{u}|| = \sqrt{x^2 + y^2}.$$

   If you interpret $(x, y)$ as a point in the coordinate plane, then $||\mathbf{u}||$ is the distance of this point from the origin. We often visualize $\mathbf{u}$ as a vector whose tail is at the origin, and whose head is at $(x, y)$. In this view, $||\mathbf{u}||$ is the length of the vector.

   If $\mathbf{u} = (x, y)$ and $\mathbf{v} = (x', y')$, then we define their sum

$$\mathbf{u} + \mathbf{v} = (x + x', y + y').$$

   You may already be familiar with what this vector sum looks like.

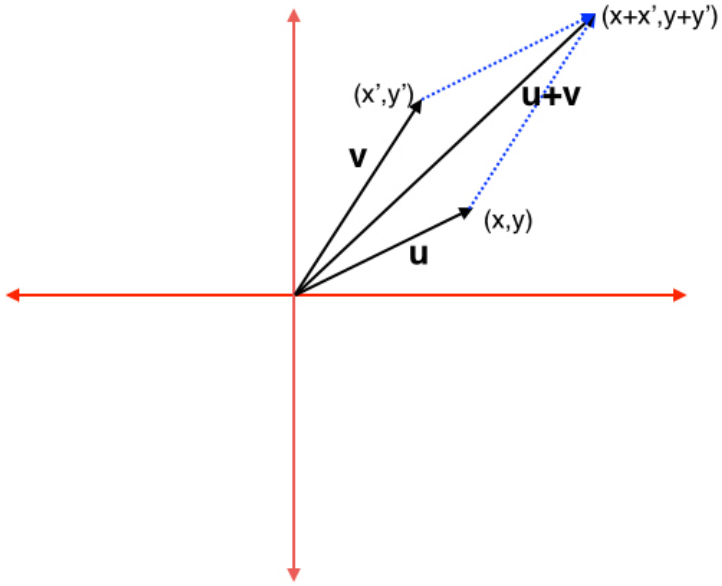



Figure 5.3: Vector sum in two dimensions: The sum is the diagonal of the parallelogram two of whose sides are **u** and **v**.

*(a)* Prove

$$||\mathbf{u}+\mathbf{v}|| \leq ||\mathbf{u}|| + ||\mathbf{v}||.$$

(Work backwards, and be patient. There is some messy algebra involved.)

*(b)* Why is the triangle inequality called the triangle inequality?

8. We define $\max(x, y) = x$ if $x \geq y$ and $\max(x, y) = y$, otherwise. Prove that for all real numbers $x, y$

$$\max(x, y) = \frac{x + y + |x - y|}{2}.$$

Derive, and prove, a similar formula for the minimum of two numbers.

9. Prove that the sum of two rational numbers is a rational number.

10. Prove that the sum of a rational and an irrational number is an irrational number.

11. What about the sum of two irrational numbers? Explain your answer carefully.

12. Prove that $\sqrt[3]{2}$ is an irrational number. You should imitate the proof for the square root.

13. You take 25 days off from work each year. Prove that at least three of the days fall in the same month. (Consider an indirect proof: what would it mean for this statement to be *false*?)

# Chapter 6

# Mathematical Induction and Recursion

*Great fleas have little fleas,*
*Upon their backs to bite 'em,*
*And little fleas have lesser fleas,*
*and so, ad infinitum.*
*And the great fleas, themselves, in turn*
*Have greater fleas to go on;*
*While these again have greater still,*
*And greater still, and so on.*
–Augustus De Morgan, *A Budget of Paradoxes*[1]

The preceding chapter dealt with general strategies for discovering and writing proofs. The present one is largely devoted to a more specialized, but very widely-used proof technique called *mathematical induction.* We will also study the closely-related notion of *recursion,* which is a method for defining sequences, structures and algorithms. These subjects are not completely new—both recursion and (in thin disguise) proof by induction, have already made appearances in this book.

In addition, in Section 6.3 we will explore a side alley and learn something about the relative rates of growth of sequences and functions, a topic of central importance in the analysis of algorithms.

## 6.1 A Sampler of Induction Problems

### 6.1.1 A summation identity

If you calculate the sums of successive odd positive integers, a clear pattern emerges:

[1] De Morgan does not attribute these lines, but there are similar verses (complete with fleas and the rhyme of 'bite 'em' and '*ad infinitum*') by Jonathan Swift.

$$\begin{array}{lcccl} 1 & = & 1 & = & 1^2 \\ 1+3 & = & 4 & = & 2^2 \\ 1+3+5 & = & 9 & = & 3^2 \\ 1+3+5+7 & = & 16 & = & 4^2 \end{array}$$

It's hard to avoid making the conjecture:

*The sum of the first $n$ odd positive integers is $n^2$.*

We can write the sum of the odd positive integers using the $\Sigma$-notation, noting that the $j^{th}$ odd positive integer is $2j-1$:

$$\sum_{j=1}^{n}(2j-1).$$

This notation for sums works just like those for iterated $\wedge, \vee, \cup$, and $\cap$ that we saw earlier. $\sum$ is the upper-case Greek letter Sigma, an 'S' in Greek. The symbol $\prod$, the upper-case letter Pi, a Greek 'P', is used in the same manner for products. The notation looks a bit cryptic at first, but once you get accustomed to it, you'll want to use it. A good way to understand summation notation is to think of it in terms of programming language syntax (shown below in Python) for computing an iterated sum:

```
s=0
for j in range(1,n+1):
       s=s+2*j-1
```

In fact, the 'list comprehension' feature of Python lets you write this as

```
s=sum([2*j-1 for j in range(1,n+1)])
```

which looks even *more* like our $\sum$ notation. (The exact correspondence is spoiled by the quirk in Python that `range(1,n)` returns the list of integers in the range 1 to $n-1$.)

In this new notation we can express the conjecture as:

*Conjecture* 6.1.1. For all positive integers $n$,

$$\sum_{j=1}^{n}(2j-1) = n^2.$$

We want to prove that this holds for *all* positive integers, so it is not enough to tabulate the sum on the left-hand side of the equation for just *some* values, no matter how extensive a calculation we perform. We will discuss below how to go about producing just such a proof.

| n | $1.5^n$ | $n^2$ |
|---|---|---|
| 1 | 1.500 | 1 |
| 2 | 2.250 | 4 |
| 3 | 3.375 | 9 |
| 4 | 5.063 | 16 |
| 5 | 7.594 | 25 |

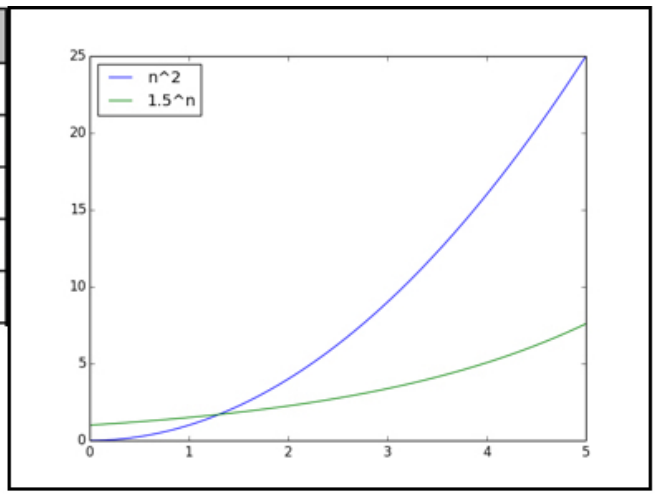



Figure 6.1: Values of $1.5^n$ and $n^2$ for $n \leq 5$. $n^2$ overtakes $1.5^n$ between $n = 1$ and $n = 2$.

| n | $1.5^n$ | $n^2$ |
|---|---|---|
| 1 | 1.500 | 1 |
| 2 | 2.250 | 4 |
| 3 | 3.375 | 9 |
| 4 | 5.063 | 16 |
| 5 | 7.594 | 25 |
| 6 | 11.391 | 36 |
| 7 | 17.086 | 49 |
| 8 | 25.629 | 64 |
| 9 | 38.443 | 81 |
| 10 | 57.665 | 100 |
| 11 | 86.498 | 121 |
| 12 | 129.746 | 144 |
| 13 | 194.620 | 169 |
| 14 | 291.929 | 196 |
| 15 | 437.894 | 225 |

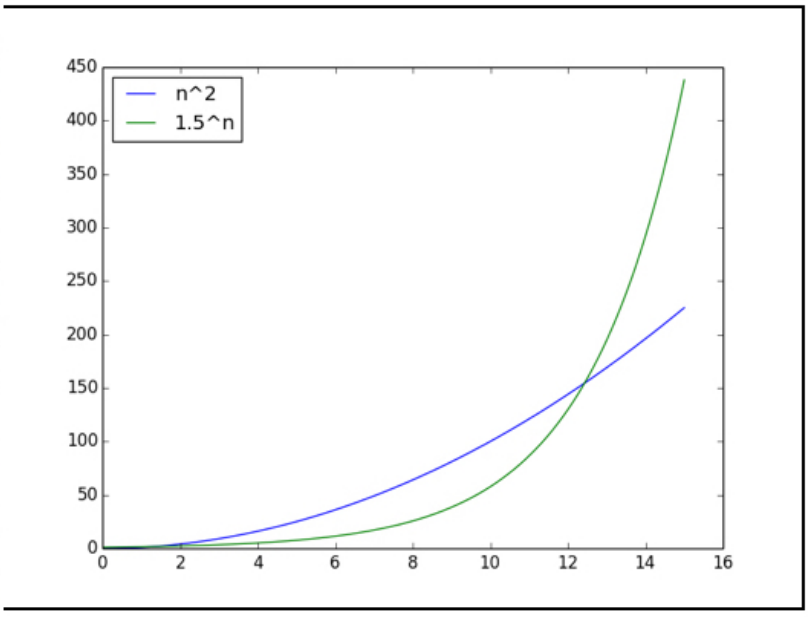



Figure 6.2: Values of $1.5^n$ and $n^2$ for $n \leq 15$. $1.5^n$ crosses back above $n^2$ between $n = 12$ and $n = 13$. Will they cross again?

### 6.1.2 Exponential versus polynomial growth

Which is larger, $1.5^n$ or $n^2$? Figure 6.1 displays both a table and a plot of the values of the two sequences for positive integers up to $n = 5$. (We only tabulated at integer values of $n$; the plot was produced by drawing a smooth curve through the discrete points.)

The plot of $n^2$ starts out below that of $1.5^n$, but crosses above it between $n = 1$ and $n = 2$. If you visually extrapolate, it looks like $n^2$ will continue to grow faster than $1.5^n$, and remain larger forever after. But look at Figure 6.2, which displays the values up through $n = 15$.

So much for visual extrapolation! It looks now as though $1.5^n$ wins the race. But how can we be sure? We will tentatively advance this as a conjecture:

*Conjecture* 6.1.2.

$$1.5^n > n^2$$

for all integers $n \geq 13$.

### 6.1.3 Behind blue eyes

The following logic puzzle is variously called the Muddy Children Puzzle, the Unfaithful Wives Puzzle, the Unfaithful Husbands Puzzle, *etc.* It is often cited by computer scientists interested in multi-agent systems in which each agent must reason about the state of knowledge of the other agents.

> Ten people live in the village. Blue-eyed people are banned from the village, but this rule is enforced on the honor system. If you discover that you have blue eyes, you must leave by sundown of the day you find this out. No one talks about who has, or doesn't have blue eyes, and there are no mirrors, so if a single blue-eyed person lives in the village, she will never find this out from one of her fellow residents, and life will go on as before, with the blue-eyed person believing her eyes, like those of all the other villagers, are brown.
>
> Every morning at sunrise, the villagers meet on the village common. One Monday morning, a representative of the king shows up at the meeting and announces, 'At least one of you has blue eyes', and then leaves. Life continues normally for a week, with everyone attending the sunrise meeting, until Wednesday of the week following, when only one person shows up. Explain how this happened.

### 6.1.4 An inductive proof of the summation identity

Here we describe a method for seeing that Conjecture 6.1.1 is in fact correct. Look at the successive left-hand sides of the original equations that led us to the conjecture:

$$1 \xrightarrow{+3} 1+3 \xrightarrow{+5} 1+3+5 \xrightarrow{+7} 1+3+5+7.$$

The general step, passing from the left-hand side of the $n^{th}$ equation in the sequence to the left-hand side of the $(n+1)^{th}$ equation, is

$$1+3+\cdots+(2n-1) \xrightarrow{+(2n+1)} 1+3+\cdots+(2n-1)+(2n+1).$$

What about the right-hand sides? To get from $1^2$ to $2^2$, we add $2^2 - 1^2$. To get from $2^2$ to $3^2$, we add $3^2 - 2^2$. In general, on the $n^{th}$ step, we have

$$n^2 \xrightarrow{+\,(n+1)^2-n^2} (n+1)^2.$$

The crucial observation is a consequence of this little bit of algebra:

$$(n+1)^2 - n^2 = (n^2 + 2n + 1) - n^2 = 2n + 1.$$

Thus at each step, *we add the same thing to both sides.* We know the identity in the conjecture holds for $n = 1, 2, 3$. To produce further instances of the identity, we add the same quantity to both sides of the equation, so the result is still a valid equation. This means that the identity holds for *all* positive integers $n$.

We can bootstrap this summation identity to obtain others, without redoing the argument. For example, we can sum the first $n$ *even* integers:

$$\begin{aligned} 2 + 4 + 6 + \cdots + 2n &= (1+1) + (3+1) + (5+1) \cdots + ((2n-1)+1) \\ &= (1 + 3 + 5 + \cdots + (2n-1)) + (1 + 1 + \cdots + 1) \\ &= n^2 + n \end{aligned}$$

If we divide both sides by 2, we obtain the important identity for the sum of the first $n$ positive integers:

$$\sum_{j=1}^{n} j = 1 + 2 + \cdots + n = \frac{n^2 + n}{2}.$$

Many algorithms (for instance, elementary sorting algorithms like Selection Sort algorithm discussed in Chapter 3 ) are structured as a loop nested within a loop, where the outer loop is executed $n$ times on inputs of size $n$, and the inner loop is executed $j$ times on the $j^{th}$ pass through the outer loop. The total number of times a statement within the inner loop is executed is then $1 + 2 + \cdots + n$. The fact that this sum can be written as $(n^2 + n)/2$ is important in predicting the performance of the algorithm as the input size grows.[2]

### 6.1.5 An inductive solution to the inequality problem

Let's write down the last three inequalities that we can extract from the table in Figure 6.2.

$$\begin{aligned} 1.5^{13} &> 13^2 \\ 1.5^{14} &> 14^2 \\ 1.5^{15} &> 15^2 \end{aligned}$$

[2]We found the same expression for the running time of Selection Sort back in Chapter 3 by a quite different analysis, using binomial coefficients.

The question is whether this inequality holds for all subsequent lines. Once again, we analyze what happens when we pass from one instance of the inequality to the next, by looking at how we transform each side. On the left-hand side, we always multiply by the same amount:

$$1.5^{13} \xrightarrow{\times 1.5} 1.5^{14} \xrightarrow{\times 1.5} 1.5^{15}.$$

On the right-hand side, the general step is passing from $n^2$ to $(n+1)^2$, and to do this we multiply by $\frac{(n+1)^2}{n^2}$:

$$n^2 \xrightarrow{\times (n+1)^2/n^2} (n+1)^2.$$

*Now* the crucial fact is that as long as $n \geq 13$,

$$\begin{aligned}
\frac{(n+1)^2}{n^2} &= \left(\frac{n+1}{n}\right)^2 \\
&= \left(1+\frac{1}{n}\right)^2 \\
&\leq \left(1+\frac{1}{13}\right)^2 \\
&< 1.16 \\
&< 1.5
\end{aligned}$$

So, we start out with a valid inequality for $n = 13$. At each step we multiply the left-hand side by 1.5, and the right-hand side by something less than 1.5. The result is therefore still a valid inequality, so *all* the inequalities are valid.

### 6.1.6 Solution to the puzzle of the blue-eyed villagers.

Suppose only one villager has blue eyes. On hearing the announcement of the king's representative, she looks around and sees that all her fellow villagers have brown eyes. Being perfectly logical (as the characters in such puzzles always are) she realizes that she is the only person with blue eyes, and exiles herself at sundown.

What happens if exactly two villagers, $A$ and $B$, have blue eyes? On the first day, $A$ looks around and thinks, 'poor $B$—she's the only person with blue eyes, and she will leave the village at sundown'. When $A$ sees $B$ appear at the meeting the next morning, he realizes that $B$ must have concluded that she is not the only person with blue eyes, and thus $A$ has blue eyes as well. So $A$ will leave the village at sundown on the second day. Of course, $B$ makes the identical calculation, so they both leave the village, and will be absent from the meeting on the third day.

If $A, B, C$ have blue eyes, then $A$ sees only the blue eyes of $B$ and $C$. If they are they only people with blue eyes, then they will go into exile at sundown of the second day. When they show up at the third sunrise meeting, $A$ knows he has blue eyes as well, and exiles himself at sundown of the third day. $B$ and $C$ do the same.

And so it goes, for each new person. Nine people with blue eyes will believe that the only blue-eyed people are the eight they can see. It is only on the morning of the ninth day that they

all realize they have blue eyes, and they leave the village at sundown. On day ten, the single brown-eyed villager arrives alone at the sunrise meeting (and learns as a result that his eyes are brown).

## 6.2 Ordinary Induction

### 6.2.1 Principle of Mathematical Induction

The arguments in the preceding examples may seem like simple common sense, but in fact, a new reasoning principle, based on the structure of the set of integers, was employed. The Principle of Mathematical Induction says, in effect, that every integer greater than or equal to $k$ is eventually reached by starting from $k$ and repeatedly adding 1. Here we state the principle precisely. (This might at first look a bit obscure, but will become clearer as we see how it is applied in examples.)
**Principle of Mathematical Induction.** $P$ denotes a property of integers. We write $P(n)$ to mean 'the integer $n$ has property $P$'. Suppose $k$ is an integer such that

- $P(k)$
- for all $n \geq k$, $P(n) \to P(n+1)$.

Then $P(n)$ for all integers $n \geq k$.

The first hypothesis says that $P$ holds for $k$. The second hypothesis then implies, in particular, that $P$ holds for $k+1$. Apply it again, and you find that $P$ holds for $k+2$. And likewise $k+3, k+4, \ldots$. The Principle of Induction says that the property then holds for all subsequent integers.

A *proof by mathematical induction* is a proof of a proposition of the form '$P(n)$ for all $n \geq k$' that uses this principle.

### 6.2.2 Some proofs by mathematical induction.

Let's repackage the proofs of our summation identity and of our inequality as 'official' proofs by induction. Such a proof has two parts: a 'base step' in which $P(k)$ is proved for the starting value $k$, and an 'inductive step' in which we show $P(n) \to P(n+1)$ for all $n \geq k$. Pay careful attention to the organization and the language of the proofs; these constitute a kind of template for proofs by mathematical induction.

**Theorem 6.2.1.** *For all positive integers $n$,*

$$\sum_{j=1}^{n}(2j-1) = n^2.$$

*Proof.* We prove this by induction on $n$.

*Base step.* For $n = 1$, we have

$$\sum_{j=1}^{1}(2j-1) = 1 = 1^2,$$

so the theorem holds in this case.

*Inductive step.* Suppose the identity holds for some $n \geq 1$. (This is called the *inductive hypothesis*.) We need to show that the identity also holds for $n+1$. We have

$$\begin{aligned}\sum_{j=1}^{n+1}(2j-1) &= \left(\sum_{j=1}^{n}(2j-1)\right) + 2(n+1) - 1 \\ &= n^2 + 2(n+1) - 1 \text{ (by the inductive hypothesis)} \\ &= n^2 + 2n + 1 \\ &= (n+1)^2,\end{aligned}$$

so the identity does indeed hold for $n+1$. This completes the proof. □

**Theorem 6.2.2.** *For all positive integers $n \geq 13$,*

$$1.5^n > n^2.$$

*Proof.* We prove this by mathematical induction.
*Base step.* If $n = 13$, we have

$$1.5^{13} > 194.6 > 169 = 13^2,$$

so the inequality holds.
*Inductive step.* Now suppose $1.5^n > n^2$ holds for some $n \geq 13$. We have to show $1.5^{n+1} > (n+1)^2$. We have

$$\begin{aligned}1.5^{n+1} &= 1.5 \cdot 1.5^n \\ &> 1.5n^2 \text{ (by the inductivehypothesis)} \\ &> \left(1 + \frac{1}{13}\right)^2 \cdot n^2 \\ &\geq \left(1 + \frac{1}{n}\right)^2 \cdot n^2 \text{ (since } n \geq 13) \\ &= \left(\frac{n+1}{n}\right)^2 n^2 \\ &= \frac{(n+1)^2}{n^2} \cdot n^2 \\ &= (n+1)^2,\end{aligned}$$

completing the proof. □

It is a little harder to put the reasoning behind the puzzle of the blue-eyed villagers into this framework, but it is based on the same principle. Let us call the day the king's representative makes his revelation Day 1, and the subsequent days Day 2, Day 3, *etc.* Here the proposition $P(n)$ is:

*If exactly n villagers have blue eyes, then they will simultaneously exile themselves at sunset of Day n.*

Our theorem is that $P(n)$ holds for all $n \geq 1$. We have already gone through the base case: If exactly one villager has blue eyes, he learns this by looking at all his brown-eyed neighbors on the first day, and exiles himself at sunset of Day 1.

For the inductive step, assume $P(n)$ is true for some $n$. Now suppose exactly $n+1$ villagers have blue eyes. Consider the reasoning of one of them. He sees that $n$ of his neighbors have blue eyes. By the inductive hypothesis, he knows that if these $n$ neighbors were the only ones with blue eyes, they would have left the village on sundown of the $n^{th}$ day. So now he knows that he himself has blue eyes. He could not have known this on an earlier day, because if the $n$ people he sees are the only ones with blue eyes, they would not have left the village earlier than the evening of the $n^{th}$ day, so he gets no information about whether he has blue eyes until that evening. Everyone in the village can repeat this reasoning, and so all $n+1$ villagers with blue eyes exile themselves that evening. Thus $P(n+1)$ is true.

### 6.2.3 'Aren't You Assuming What You're Trying to Prove?'

This objection inevitably comes up whenever proofs by induction are introduced, because of the suspicious-looking clause in every proof by induction: 'suppose $P(n)$ holds for some $n \geq k$'. But we are not assuming that $P(n)$ holds for *any* $n$ at all, we are just proving that *if* $P(n)$ holds for some $n$, then it also holds for the successor of $n$. In other words, we are proving that the property $P$ is preserved in passing from $n$ to $n+1$. Another way to say this is that in the inductive step we are proving

$$\forall n((n \geq k \wedge P(n)) \rightarrow P(n+1)).$$

Recall from Chapter 1 that $\mathbf{F} \rightarrow \mathbf{F}$ has the value **true**: So it is possible for the formula above to be true, while

$$\forall n(n \geq k \rightarrow P(n))$$

is false. That is why we need both a base step and an inductive step.

For instance, here is an inductive 'proof' that for every positive integer $n$, $n = n+6$. Suppose it is true for some positive integer $n$. Then it is true for $n+1$, because

$$n+1 = (n+6)+1 = (n+1)+6.$$

The conclusion is ridiculous, but this is a perfectly correct proof of the perfectly useless fact

$$\forall n(n = n+6 \rightarrow n+1 = (n+1)+6).$$

The inductive step of our 'proof' is properly constructed, but the absence of a base case prevents us from applying the Principle of Mathematical Induction and concluding that the equation holds for all positive integers. In fact it holds for **no** positive integers.

Another misconception about mathematical induction is caused by confusion with an entirely different use of the word 'induction': 'Inductive reasoning' is used in the philosophy of science to describe the process of inferring general principles from specific examples, and is opposed to 'deductive reasoning', which is logical proof. Mathematical induction is *not* an instance of this kind of inductive reasoning; like all our proofs, it is entirely deductive.

### 6.2.4 Cardinality of the Power Set of an $n$-element Set

Here we will give a new proof of something we proved earlier in Theorem 3.3.1: The power set of $\{1, \ldots, n\}$ has $2^n$ elements. Our earlier proof gave a bijection between

$$\underbrace{\{0,1\} \times \{0,1\} \times \cdots \times \{0,1\}}_{n \quad \text{times}}$$

and

$$\mathcal{P}(\{1, \ldots, n\}).$$

Here we will give a proof of the same fact by induction on $n$.

The base case, when $n = 0$, is established by observing

$$\mathcal{P}(\emptyset) = \{\emptyset\},$$

which has $1 = n^0$ elements. For the inductive step we need to show that each time we add a new element to the set, the number of subsets *doubles*. To understand why this occurs, look at what happens when we add the element 3 to $\{1, 2\}$. The subsets of $\{1, 2, 3\}$ naturally separate into two types: those that contain 3 as an element:

$$\{1, 2, 3\}, \{1, 3\}, \{2, 3\}, \{3\},$$

and those that do not:

$$\{1, 2\}, \{1\}, \{2\}, \emptyset.$$

You can see that there is a one-to-one correspondence between the two types of subsets: You get from a subset of the first kind to one of the second kind by removing 3.

Let's turn this insight to into a proof by induction on $n$:

*Base step.* We have already verified that the identity holds for the case $n = 0$ (where we take $\{1, \ldots, n\}$ to be the empty set).

*Inductive step.* Now assume that for some $n \geq 0$,

$$|\mathcal{P}(\{1, \ldots, n\})| = 2^n.$$

We will use this to show

$$|\mathcal{P}(\{1, \ldots, n+1\})| = 2^{n+1}.$$

We have

$$\begin{aligned} \mathcal{P}(\{1, \ldots, n+1\} &= \{X \subseteq \{1, \ldots, n+1\} : n+1 \notin X\} \cup \{X \subseteq \{1, \ldots, n+1\} : n+1 \in X\} \\ &= \mathcal{P}(\{1, \ldots, n\}) \cup \{X \subseteq \{1, \ldots, n+1\} : n+1 \in X\} \end{aligned}$$

Moreover, the union is disjoint, so $|\mathcal{P}(\{1, \ldots, n+1\})|$ is the sum of the cardinalities of the two sets whose union is on the right-hand side. If we can show a one-to-one correspondence between these two sets of subsets, then by the inductive hypothesis we will have

$$|\mathcal{P}(\{1, \ldots, n+1\})| = 2^n + 2^n = 2^{n+1}.$$

The correspondence maps each subset $V$ of $\{1,\ldots,n\}$ to $V \cup \{n+1\}$, and each $X \subseteq \{1,\ldots,n+1\}$ with $n+1 \in X$ to $X \backslash \{n+1\}$. You can see that if you compose these two maps in either direction, you get back the set you started with, so they are mutually inverse bijections, which gives the desired correspondence. $\square$

## 6.3 Exponential, Polynomial, and Logarithmic Growth

### 6.3.1 Exponential versus polynomial growth

The popular media have appropriated the term 'exponential growth' to describe anything that is growing very rapidly. This is the sort of thing that can set your teeth on edge if you know something about mathematics. In scientific circles, at least, 'exponential growth' is a technical term with a very precise meaning.

So, for the record, a sequence of real numbers

$$x_0, x_1, x_2, \ldots$$

grows exponentially if the ratio $c = x_{i+1}/x_i$ is a *constant* greater than 1 for every value of $i$. This means that for every $i$, $x_i = x_0 \cdot c^i$. The sequence $1.5, 1.5^2, 1.5^3, \ldots$ that we saw earlier is an example of an exponentially-growing sequence. And so is the balance in a bank account that earns one-tenth of one percent interest every year, since in each successive year the balance is 1.001 times what it was in the previous year.

The sequence

$$1^k, 2^k, 3^k, \ldots,$$

where we fix the exponent but increase the base is said to grow *polynomially,* as long as $k > 0$. For instance, the sequence $\{n^2\}$, for $n \geq 1$, grows *quadratically,* which is a particular kind of polynomial growth, $\{n^3\}$ grows *cubically.* We shortly run out of special names for the such power sequences as $k$ gets larger, but they all grow polynomially. An excited news commentator, observing the graph of $n^3$ and the very rapid growth of its values, might declare that it grows exponentially, but it does not. *And we will prove it.* [3]

We will first use mathematical induction to establish a basic inequality:

**Theorem 6.3.1.** *Let $a$ be a positive real number. For all positive integers $n$,*

$$(1+a)^n \geq 1 + na.$$

[3] In the interests of simplicity, we have given definitions of exponential and polynomial growth that are somewhat overly restrictive. For exponential growth of the sequence $\{x_n\}_{n\geq 0}$, we don't need the ratios $x_{i+1}/x_i$ of successive terms to be a constant greater than 1; it is sufficient to have these ratios converge to a constant greater than 1 as $n$ grows larger, or even to be bounded below by such a constant. Similarly, for polynomial growth, we do not require the sequence to be exactly $\{n^k\}_{n\geq 1}$ for some positive $k$, but only that the ratios $x_n/n^k$ converge to some positive constant as a limit as $n$ grows larger. For example, the sequence $\{5 \cdot 2^n + 40\}_{n\geq 0}$ grows exponentially, and $\{6n^2 + n + 3\}_{n\geq 1}$ grows polynomially, and in fact quadratically.

*Proof.* We will prove this by induction. The base case $n = 1$ is true, because in this instance the two sides of the inequality are equal. Now suppose for some $k$ that $(1+a)^k \geq 1+ka$. To complete the proof by induction, we have to show $(1+a)^{k+1} \geq 1+(k+1)a$. We deduce this from a sequence of inequalities:

$$\begin{aligned}(1+a)^{k+1} &= (1+a)\cdot(1+a)^k \\ &\geq (1+a)(1+ka) \quad \text{inductive hypothesis} \\ &= 1+a+ka+ka^2 \\ &= 1+(k+1)a+ka^2 \\ &> 1+(k+1)a,\end{aligned}$$

which completes the proof. □

Theorem 6.3.1 doesn't appear to say very much, but it has a very important consequence. Let's look at an exponential sequence with a base that is only very slightly larger than 1, like our very slowly-growing bank balance:

$$1.001, 1.001^2, 1.001^3, \ldots.$$

Now let's take the fourth root of 1.001. We don't really need to evaluate it; we simply observe that it, too, is greater than 1, so we write it as $1+b$, where $b > 1$. (That is, $b = \sqrt[4]{1.001} - 1$.)

By Theorem 6.3.1

$$(1+b)^n \geq 1+nb,$$

so raising both sides to the fourth power gives

$$1.001^n = (1+b)^{4n} \geq (1+nb)^4 > (nb)^4 = n^4b^4.$$

The number $b^4$ is a very small positive constant, but no matter how small it is, there are still integers $n$ such that $1/n < b^4$. Thus if we choose $n$ large enough, we get

$$1.001^n > n^4b^4 > n^4/n = n^3.$$

We didn't bother to find out *how* large $n$ has to be in order for $1.001^n$ to be larger than $n^3$, we simply showed that this will eventually occur if we pick large enough $n$. (Exercise 8 asks you to provide an exact estimate.)

Of course there is nothing special about the constants 0.001 and 4 that we used in this example. The same argument shows

**Theorem 6.3.2.** *Let $c > 1$ be a real number, and $k \geq 0$ an integer. Then for all sufficiently large values of $n$,*

$$c^n > n^k.$$

Thus, *any* exponential sequence, no matter how tiny the base, will eventually overtake *any* polynomial sequence, no matter how large the power.

Here is another way to look at this result. Consider an exponentially-growing sequence

$$a, ac, ac^2, ac^3 \ldots,$$

where $a > 0$ and $c > 1$ and a polynomially-growing sequence

$$0, b \cdot 1^k, b \cdot 2^k, b \cdot 3^k, \ldots,$$

where $b > 0$, $k \geq 1$, and let's look at the sequence of *ratios* of the corresponding terms. The $n^{th}$ such ratio is

$$\frac{bn^k}{ac^n} = \frac{b}{an} \cdot \frac{n^{k+1}}{c^n}.$$

Since $k + 1 \geq 0$, Theorem 6.3.2 applies, so the expression

$$\frac{n^{k+1}}{c^n}$$

is less than 1 for sufficiently large values of $n$. But the expression $\frac{b}{an}$ converges to 0 as $n$ gets larger, and thus we conclude that the product converges to 0:

$$\lim_{n \to \infty} \frac{bn^k}{ac^n} = 0.$$

### 6.3.2 Archimedean Principle

The Principle of Mathematical Induction is based on a property of the ordering on the integers: If we start from 0 and repeatedly add 1, we can reach any nonnegative integer.

In the preceding section, we used *another* ordering property without explicit justification. We wrote that no matter how small the positive real number $b^4$ is, there are still integers $n$ such that $1/n < b^4$. Another way to put this is

> For every positive real number $x$, there is an integer $N$ such that $x < N$.

In the application we cited above, we are using this principle with $x = 1/b^4$.

In other words, no matter how high you go in the real numbers, there is an integer that is even higher. This is sometimes called the *Archimedean property*. Observe that it concerns the relationship between the ordering on the integers and the ordering on the real numbers. It cannot be deduced from either the basic properties of the real numbers that we have already cited, nor from the Principle of Induction.

### 6.3.3 Efficient algorithms versus inefficient algorithms

The contrast between exponential and polynomial growth figures importantly in the analysis of algorithms: Algorithms whose running time is exponential in the size of the input are usually considered inefficient, because they scale up very badly as the input size increases. In contrast, algorithms whose running time grows polynomially in the input size are generally considered efficient (provided the power $k$ is not too large).

As an example, consider the problems of solving and checking the correctness of a Sudoku puzzle. In the standard puzzles, there are 9 different entries, the small subgrids are $3 \times 3$, and the complete grid has $3^4 = 81$ cells. But one can of course construct Sudoku puzzles with $4^4$ cells and 16 different entries, $5^4$ cells and 25 different entries. *etc.*

To check that a proposed solution of a puzzle with $N = n^4$ cells is correct, we need to check that there is no repeat in each of the $n^2$ rows, each of the $n^2$ columns, and each of the $n^2$ subgrids. If we allow a little auxiliary storage for bookkeeping, we can check if a given row, column, or subgrid contains a repeat with a single scan, which takes $n^2$ steps. So all in all, we'll use $3n^4 = 3N$ steps to check correctness of a completed grid. If you view the running time as a function of the number $N$ of cells, then it grows proportionally to $N$—that is, linearly. If you view the running time as a function of the parameter $n$, then it grows proportionally to $n^4$. But in either case, it grows polynomially. For example, with $n = 10$, the complete grid has 10,000 cells, and a computer can check this for correctness very quickly.

In contrast, consider how long the naïve brute-force algorithm for *solving* a puzzle will take. The algorithm consists of trying out every possible configuration. With $N$ cells, there are $\sqrt{N}$ different values to populate the cells, and thus $\sqrt{N}^N$ different configurations to check, and each configuration requires $3N$ steps. So the running time for a grid of size $N$ grows faster than

$$N \times \sqrt{N}^N > 2^N,$$

that is, *faster* than exponentially. This is bad news for the brute-force algorithm! For the standard-sized puzzle, the total number of completed grids is $9^{81} \approx 2 \times 10^{77}$, which is already much too large to be checked by a computer.

It is quite correct to argue that the problem here is simply that we've chosen a very bad algorithm for solving these puzzles: You may have even written a program that solves all $9 \times 9$ Sudoku puzzles in a reasonable amount of time. But the worst-case running time of *all* such algorithms (at least those that are known) grows at least exponentially in the size of the puzzle, and are rendered impractical with only a modest increase in the puzzle size.

### 6.3.4 Polynomial versus logarithmic growth

Closely related to the comparison of exponential and polynomial growth is the connection between polynomial and *logarithmic* growth. Recall that if $b > 0$, then $\log_b n$ is the real number $x$ that solves the equation $b^x = n$. So, for example, $\log_{10} 1000 = 3$, $\log_2 32 = 5$, $0 < \log_{10} 2 < 1$, and more precisely, $\log_{10} 2 \approx 0.301$. Observe that the base of the logarithm is in a sense irrelevant when we talk about the growth of a sequence, since, for example, for any $x > 0$

$$\log_{10} x = \log_{10} 2 \cdot \log_2 x = 0.301 \cdot \log_2 x,$$

so the sequence of logarithms $\log_b n$ at one base is just a constant multiple of the sequence of logarithms at another base.

The logarithm function grows without bound, but it grows very slowly, and even if we boost the rate of growth by raising it to a high power, we will still eventually see the rate of growth slowing as $n$ increases. Functions of the form $n^a$, where $0 < a < 1$, such as the square root and cube root functions, show the same kind of slowing growth. Figure 6.3 superimposes the graphs of $(\log_2 n)^2$ and $\sqrt{n}$ for small values of $n$. It looks as if the logarithmic function is winning, but Figure 6.4, which plots the two functions for larger values of $n$, shows the real story.

We can deduce this behavior from our previous comparison of polynomial and exponential growth. Look at the sequence of ratios

$$\frac{(\log_2 n)^2}{\sqrt{n}}.$$

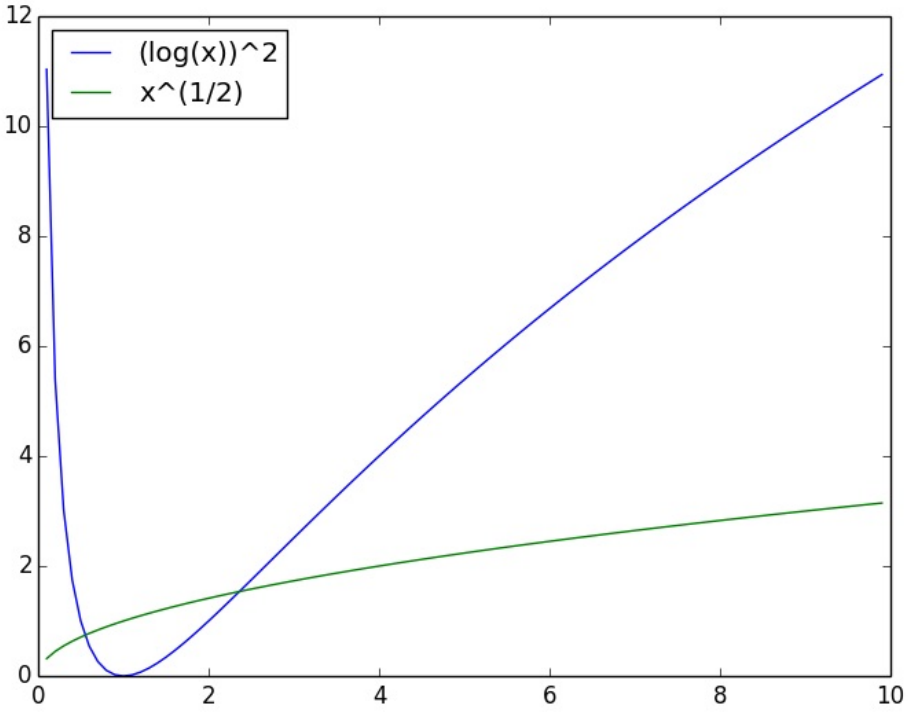



Figure 6.3: The graphs of $\log_2 x$ and $\sqrt{x}$ for small values of $x$: the logarithmic function starts out with much faster growth.

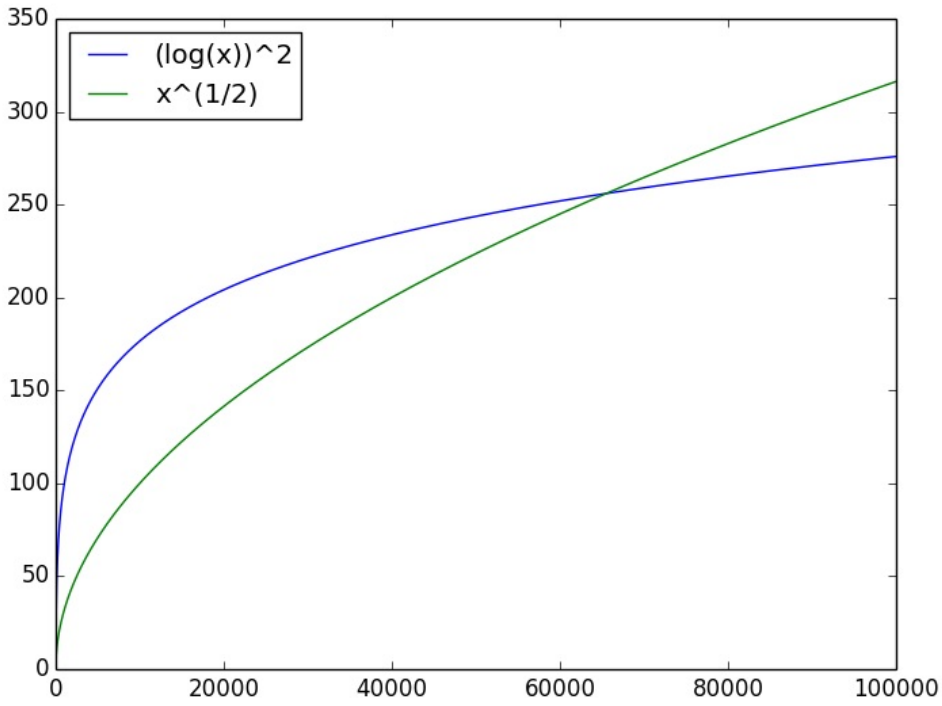



Figure 6.4: The same graphs for larger values of $x$: As $x$ grows larger, $x^c$ grows faster than $(\log x)^b$ for any positive $c$, no matter how small, and any positive $b$, no matter how large.

Square, and set $u = \log_2 n$:

$$\left[\frac{(\log_2 n)^2}{\sqrt{n}}\right]^2 = \frac{u^4}{n} = \frac{u^4}{2^u}.$$

We can make $u$ as large as we like by making $n$ sufficiently large, and we saw in our discussion of exponential and polynomial growth that $\frac{u^4}{2^u}$ converges to 0 as $u$ grows larger. We thus conclude that the square root of this expression, which is the ratio we started with, also converges to 0 as $n$ grows larger. The same argument shows that this holds over a wide range of parameters–for example we could have used the ninth power of the logarithm in the numerator and the cube root of $n$ in the denominator, and come up with same result. We state this conclusion formally as a theorem:

**Theorem 6.3.3.** *Let $a > 1$, $b, c > 0$. Then*

$$\lim_{n\to\infty} \frac{(\log_a n)^b}{n^c} = 0.$$

# 6.4 Strong Induction

## 6.4.1 The Unstacking Game

You have a stack of $n$ blocks. On each turn, you break the stack into two smaller stacks, of sizes $m_1$ and $m_2$, respectively. For this move, you receive $m_1 m_2$ points. You continue making moves, receiving points for each stack you break up, until all the stacks have just a single block. The goal is to do this in such a manner that you receive the highest possible score.

Let's illustrate this with a stack of 5 blocks. We can begin by breaking the stack into two stacks with 3 blocks and 2 blocks each, which gets us 6 points. After that, we don't have much choice, Here is a transcript of the whole game:

| | | | | |
|---|---|---|---|---|
| 5 | → | 3+2 | 6 points | score: 6 |
| 3+2 | → | 2+1+2 | 2 points | score: 8 |
| 2+1+2 | → | 1+1+1+2 | 1 point | score: 9 |
| 1+1+1+2 | → | 1+1+1+1+1 | 1 point | score: 10 |

On the other hand, we could have adopted a different strategy, removing just one block at each move:

| | | | | |
|---|---|---|---|---|
| 5 | → | 4+1 | 4 points | score: 4 |
| 4+1 | → | 3+1+1 | 3 points | score: 7 |
| 3+1+1 | → | 2+1+1+1 | 2 points | score: 9 |
| 2+1+1+1 | → | 1+1+1+1+1 | 1 point | score: 10 |

The new strategy did not change our score. A little playing around with the game will persuade you that no matter how you proceed, the total score for a stack of 5 blocks will always be 10. If you try it out for stacks of $n$ blocks for small $n$, you will always get $\frac{n(n-1)}{2}$ as the score. We've already seen that this expression is equal to the sum $1 + \cdots + (n-1)$, which is indeed the score you would get if you removed just one block at a time.

To see why this is the case, let's imagine we start with a stack of $n$ blocks, and divide it into two stacks with $m_1$ and $m_2$ blocks each, where $m_1 + m_2 = n$. The move gets us $m_1 m_2$ points.

Now the game breaks up into two completely separate games, one with the stack of $m_1$ blocks, and the other with the stack of $m_2$ blocks. If we have already established that our conjecture holds for the smaller stacks, then completing the game will get us

$$\frac{m_1(m_1-1)}{2}+\frac{m_2(m_2-1)}{2}$$

additional points. Thus our total score for the $n$-block game is

$$\begin{aligned} m_1m_2+\frac{m_1(m_1-1)}{2}+\frac{m_2(m_2-1)}{2} &= \frac{m_1^2+2m_1m_2+m_2^2-m_1-m_2}{2} \\ &= \frac{(m_1+m_2)^2-(m_1+m_2)}{2} \\ &= \frac{n^2-n}{2} \\ &= \frac{n(n-1)}{2} \end{aligned}$$

What we've shown is that if the conjecture holds for all stacks with fewer than $n$ blocks, then it holds for stacks with $n$ blocks. If you combine this with the fact that it holds for stacks with 1 block (where you get a total score of 0), we can conclude that it holds for all stacks: That is, knowing it holds for 1 implies that it holds for 2. Since we now know it holds for 1 and 2, we conclude it holds for 3. Now since it holds for 1, 2 and 3, it must hold for 4, etc.

### 6.4.2 The Principle of Strong Induction and the Least Integer Principle

This is a somewhat different use of induction from what we saw earlier. In the inductive step in this new version, we suppose $n > 0$ and that the claim that we are trying to prove holds for *all* $m < n$, (not just for $n-1$), and from this deduce the claim for $n$. From this and the base step (which is a collection of stacks of size 1) we concluded that the claim holds for all $n$. A formal statement of this principle is:

**Principle of Strong Induction.** If

- $P(k)$
- $(\forall n)((k < n \wedge \forall m(k \leq m < n \rightarrow P(m)) \rightarrow P(n))$

then

$$\forall n(k \leq n \rightarrow P(n)).$$

This version of induction is closely related to the least integer principle, which we already saw in the preceding chapter: *Every nonempty subset of the positive integers has a smallest element.* (See Exercise 13)

### 6.4.3 An application of strong induction: $\sqrt{2}$ revisited.

We have already used the least integer principle once: When we proved that the square root of 2 is irrational, we really proved that there is no positive integer $m$ such that $2m^2$ is a perfect square. To do this, we supposed that our claim was false, and considered the *least* $m$ for which $2m^2$ is a perfect square. The contradiction came in the conclusion that there was an still smaller value $q$ such that $2q^2$ is square. Let's rephrase the argument, giving it as a proof by strong induction:

**Theorem 6.4.1.** *For all $n \geq 1$, $2n^2$ is not a perfect square.*

*Proof.* By strong induction on $n$.
*Base step.* The claim is true for $n = 1$, since $2 \cdot 1^2 = 2$ is not a perfect square.
*Induction step.* Now suppose $n > 1$ and that the claim holds for every positive integer less than $n$. If $2n^2 = k^2$ for some $k$, then we conclude as before that $k$ is even. Thus $k = 2m$ for some positive integer $m$, and thus

$$2n^2 = (2m)^2 = 4m^2,$$
$$n^2 = 2m^2.$$

But then $m < n$, and our inductive hypothesis is that the claim holds for $m$, so $2m^2$ cannot be square. This contradiction shows that $2n^2 = k^2$ cannot hold, so the claim is true for $n$.

Thus, by the strong induction principle, the claim holds for all $n$, so that there is no positive integer $n$ for which $2n^2$ is square. □

## 6.5 Recursively Defined Sequences, Sets and Algorithms

You have all seen the idea of defining a sequence of numbers, sets, *etc.* by defining the first element in the sequence, and then defining the subsequent elements in terms of the preceding elements. Recursion is often introduced in programming courses via the following definition of the factorial function:

- $$0! = 1$$
- For $n > 0$,
$$n! = n \times (n-1)!$$

The pattern in the definition is the same as that in proofs by induction: There is a base case, in which we define the first elements of the sequence, and a recursive case, in which we define every subsequent element of the sequence in terms of the preceding element. The recursive *definition* of the sequence can be directly translated into a recursive *algorithm* for computing the factorial function. Here is what this looks like in Python:

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

### 6.5.1 Fibonacci Numbers

This is a sequence of numbers defined by starting with 0 and 1, and setting each subsequent element of the sequence to the sum of the two preceding. Thus the sequence begins

$$0, 1, 1, 2, 3, 5, 8, 13, 21 \ldots.$$

We can give the recursive definition precisely as:

- $F_0 = 0$
- $F_1 = 1$
- For $n > 1$,
$$F_n = F_{n-1} + F_{n-2}.$$

Since the recursive definition defines each succeeding element in terms of the preceding two, rather than just the immediate predecessor, this resembles proofs by strong induction rather than ordinary induction. As a result, properties of the Fibonacci numbers are often proved by an application of strong induction in which there are, in effect, two separate base cases.

We will give the proof of one such property in detail. This is the fact that the Fibonacci sequence grows exponentially.

**Theorem 6.5.1.** *For all $n \geq 3$,*
$$F_n \geq 2^{\frac{n-1}{2}}.$$

*Proof.* We will prove this by strong induction on $n$. *Base step:* If $n = 3$, then

$$F_3 = 2 = 2^1 = 2^{\frac{3-1}{2}},$$

as required.

*Inductive Step.* We assume $n > 3$ and that the inequality holds for all $m$ with $3 \leq m < n$. Generally speaking, in arguments such as this, we will use the inductive hypothesis together with the recurrence defining the Fibonacci numbers to deduce properties of $F_n$ from the corresponding properties for $F_{n-2}$ and $F_{n-1}$. In the case $n = 4$, however, this is not possible, because we do not have the property for $F_2$. So we have to verify the inequality for $n = 4$ directly:

$$F_4 = 3 = 2 \cdot 1.5 > 2\sqrt{2} = 2^{1.5} = 2^{\frac{4-1}{2}}.$$

So we can continue, supposing $n \geq 5$. We now have

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\geq 2^{\frac{n-2}{2}} + 2^{\frac{n-3}{2}} \quad \text{(by inductive hypothesis)} \\ &> 2^{\frac{n-3}{2}} + 2^{\frac{n-3}{2}} \\ &= 2 \cdot 2^{\frac{n-3}{2}} \\ &= 2^{1+\frac{n-3}{2}} \\ &= 2^{\frac{n-1}{2}}. \end{aligned}$$

□

For instance, the hundredth Fibonacci number is considerably more than $2^{49}$, which is about $10^{16}$.

If we wanted, we could use the definition to compute the Fibonacci numbers recursively:

```
def fibonacci(int n)
{
    if(n==0):
        return 0
    elif(n==1):
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2);
}
```

Let $C_n$ denote the number of separate calls that are made to the function `fibonacci` in order to compute $F_n$ by this method. We have

- $C_0 = C_1 = 1$
- for $n > 1$,
$$C_n = 1 + C_{n-1} + C_{n-2}.$$

The recurrence counts the original call to `fibonacci` plus all the calls generated by the two recursive calls. It is then easy to see (in fact, this would be done formally by induction, but it is pretty obvious from the form of the recurrence) that $C_n > F_{n+1}$ for all $n$. Thus the running time of this program is at least *exponential in* $n$, because of our previous result. Computing Fibonacci numbers by recursion is a *disaster.*

### 6.5.2 Recursive definitions of sets and structures, and proofs by structural induction

We saw recursive definitions at the very beginning of this book, when we defined the formulas of propositional logic. If you recall, the definition was this: A propositional formula is either a constant, a variable, or has the form $(\phi \wedge \psi)$, $(\phi \vee \psi)$, or $\neg\phi$, where $\phi$ and $\psi$ are propositional formulas. We are in effect, using recursion to define a sequence of sets

$$\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \cdots,$$

whose union

$$\mathcal{F} = \bigcup_{i \geq 0} \mathcal{F}_i$$

is the set of all propositional formulas. We could restate the definition as follows:

- $\mathcal{F}_0$ consists of variables and atomic formulas

- For $n > 0$,

$$\begin{aligned} \mathcal{F}_n = \mathcal{F}_{n-1} \quad &\cup \quad \{\neg\phi : \phi \in \mathcal{F}_{n-1}\} \\ &\cup \quad \{\phi \vee \psi : \phi, \psi \in \mathcal{F}_{n-1}\} \\ &\cup \quad \{\phi \wedge \psi : \phi, \psi \in \mathcal{F}_{n-1}\}. \end{aligned}$$

Stated this way, the definition looks exactly like our first recursive definitions, but sacrifices the simplicity of the original formulation. We can use that original formulation to prove properties of propositional formulas by *structural induction*: In such a proof, we first show the property holds for the atomic formulas, and then assuming it holds for formulas $\phi, \psi$, show that it holds for $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $\neg\phi$.

Here is a very simple example: If you look at a propositional formula and count the total number of occurrences of variables and the constants **T** and **F**, and then the total number of occurrences of the binary connectives $\vee$ and $\wedge$, you will find that the first count is always one more than the second. Let's use structural induction to prove that this property holds for all propositional formulas: For an atomic formula the number of variables and constants is 1 and the number of binary connectives is 0, so the claim holds for the atomic formulas. By the definition, a non-atomic formula has one of the three forms $(\phi \wedge \psi)$, $(\phi \vee \psi)$, or $\neg\phi$. We assume the property holds for $\phi$ and $\psi$. The number of constants, variables, and binary connectives in $\neg\phi$ is the same as that for $\phi$, so the property continues to hold for $\neg\phi$. Let $m_1$ be the number of constants and variables in $\phi$; by the inductive hypothesis, the number of binary connectives is $m_1 - 1$. Similarly, let $m_2$ be the number of constants and variables in $\psi$; by the inductive hypothesis, the number of binary connectives is $m_2 - 1$. Thus the total number of constants and variables in $(\phi \vee \psi)$ is $m_1 + m_2$, and the total number of binary connectives $(m_1 - 1) + (m_2 - 1) + 1 = (m_1 + m_2) - 1$. So the property holds for $(\phi \vee \psi)$. The same analysis works for $(\phi \wedge \psi)$.

The argument can be rephrased as a traditional proof by induction on $n$, showing that for each $n \geq 0$, all the formulas in $\mathcal{F}_n$ have the required property.

### 6.5.3 Dual formulas

We will now give a careful proof of the duality principle in Theorem 1.5.1 from Chapter 1. Our proof uses structural induction, and is based on the recursive definition of the semantics of propositional formulas that we gave in Chapter 2.

The heart of our proof is another property of duality, which is the subject of one of the exercises from Chapter 1. How do we get the truth table for the dual formula $\phi'$ of $\phi$ from the truth table for $\phi$? You may have found that the answer is: Change every **F** in the table to **T**, and vice-versa. (You might have to rearrange the rows to get the assignments in the order you want them to appear in, but this is not relevant.) This is illustrated in Table 6.1.

Let's give a precise statement and proof of this property. If $\alpha$ is an assignment of truth values to a set $V$ of variables, then we denote by $\alpha'$ the *dual assignment* defined by

$$\alpha'(p) = \neg\alpha(p),$$

for each variable $p \in V$.

| $p$ | $q$ | $p \wedge \neg q$ |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | F |
| F | F | F |

| $p$ | $q$ | $p \vee \neg q$ |
|---|---|---|
| F | F | T |
| F | T | F |
| T | F | T |
| T | T | T |

Table 6.1: Truth tables for a formula and its dual. The second is obtained from the first by changing every **F** to **T**, and vice-versa.

(That is, $\alpha'(p) =$ **true** if $\alpha(p) =$ **false**, and vice-versa.) [4]

**Lemma 6.5.2.** *If $\phi$ is a propositional formula and $\alpha$ an assignment of truth values to the variables in $\phi$, then*

$$[\phi'](\alpha) = [\neg\phi](\alpha').$$

*Proof.* We prove this by structural induction. For the base case, we show that the lemma holds for atomic formulas. $\mathbf{T}' = \mathbf{F} \equiv \neg\mathbf{T}$ so both these formulas have the value **false** on every assignment. Thus $[\mathbf{T}'](\alpha) = [\neg\mathbf{T}](\alpha')$, so the Lemma holds for **T**, and similarly for **F**. For atomic formulas consisting of a single variable $p$, we have $[p](\alpha) = \alpha(p)$. In particular,

$$[p'](\alpha) = [p](\alpha) = \alpha(p).$$

$$[\neg p](\alpha') = \neg([p](\alpha')) = \neg(\alpha'(p)) = \neg\neg\alpha(p) = \alpha(p),$$

so the Lemma holds here as well.

For the inductive step, suppose $\phi, \psi$ are formulas for which the Lemma holds. Then

$$\begin{aligned}
[(\neg\phi)'](\alpha) &= [\neg(\phi')](\alpha) \\
&= \neg([\phi'](\alpha)) \\
&= \neg\neg([\phi](\alpha')) \text{ by the inductive hypothesis} \\
&= \neg([\neg\phi](\alpha')) \\
&= [\neg\neg\phi](\alpha')
\end{aligned}$$

so the Lemma holds for $\neg\phi$. We also have

$$\begin{aligned}
[(\phi \vee \psi)'](\alpha) &= [\phi' \wedge \psi'](\alpha) \\
&= [\phi'](\alpha) \wedge [\psi'](\alpha) \\
&= \neg([\phi](\alpha')) \wedge \neg([\psi](\alpha')) \text{ by the inductive hypothesis}
\end{aligned}$$

while

$$\begin{aligned}
\neg([\phi \vee \psi](\alpha')) &= \neg([\phi](\alpha') \vee [\psi](\alpha')) \\
&= \neg([\phi](\alpha')) \wedge \neg([\psi](\alpha')) \text{ by DeMorgan's Law,}
\end{aligned}$$

[4] To make the proof more compact, we will use $\neg$, $\wedge$, and $\vee$ as operations on the truth values **true** and **false**, with the obvious meanings. That may not seem like news worth mentioning, but to be strictly correct about it, up until now we have only been using these operation symbols on formulas, rather than on values.

so

$$[(\phi \vee \psi)'](\alpha) = \neg([\phi \vee \psi](\alpha')).$$

Thus the Lemma holds for $\phi \vee \psi$. The proof that it holds for $\phi \wedge \psi$ is identical. This completes the proof of the Lemma. □

*Proof of Theorem 1.5.1.* Suppose $\phi$ and $\psi$ are equivalent formulas. Let $\alpha$ be any assignment of truth values to the variables in $\phi$ and $\psi$. The foregoing Lemma tells us that

$$[\phi'](\alpha) = [\neg\phi](\alpha') = \neg([\phi](\alpha')),$$

and likewise

$$[\psi'](\alpha) = \neg([\psi](\alpha')).$$

So

$$\begin{aligned} [\phi'](\alpha) &= \neg([\phi](\alpha')) \\ &= \neg([\psi](\alpha')) \text{ by the identity } \phi \equiv \psi \\ &= [\psi'](\alpha). \end{aligned}$$

That is, $\phi'$ and $\psi'$ have the same truth value on every assignment $\alpha$, so $\phi' \equiv \psi'$. □

### 6.5.4 Recursive Algorithms

We already saw an example of a (terrible) recursive algorithm–the one for computing Fibonacci numbers–and its analysis. Here is another such analysis. As we shall see, in one sense, the algorithm is just as terrible because its running time grows exponentially in the size of the puzzle, but in another sense the algorithm is optimal, because here we can actually prove that there is no better alternative.

Figure 6.5: The Towers of Hanoi puzzle, with 5 disks

You may be familiar with the Towers of Hanoi puzzle (Figure 6.5), since it is a standard example used in programming classes to illustrate recursion. The object of the game is to move the entire

```
To move n disks from post 1 to post 3:
        move n-1 disks from post 1 to post 2
        move 1 disk from post 1 to post 3
        move n-1 disks from post 2 to post 3.
```

Figure 6.6: The recursive algorithm for solving the Towers of Hanoi puzzle.

stack of disks, one at a time, to one of the other two posts, in such a manner that you never place a larger disk on top of a smaller one.

The recursive trick for solving the puzzle is that if you know how to solve it for $n-1$ disks, you can use this strategy to solve the $n$-disk puzzle: The strategy for the smaller game lets you move the top $n-1$ disks on post 1 to post 3 without disturbing the large bottom disk. You then move the bottom disk to post 2, and then again apply the strategy for the smaller game to move the $n-1$ disks from post 3 onto post 2. A pseudocode version of this recursive algorithm is given in Figure 6.6.

If we denote by $T_n$ the number of moves this algorithm makes to move a stack of $n$ disks, then we get the recurrence

$$T_n = 2T_{n-1} + 1.$$

Furthermore, the recursion bottoms out at 0 disks, so we can define

$$T_0 = 0.$$

This gives the sequence $T_1 = 1, T_2 = 2 \cdot 1 + 1 = 3, T_3 = 2 \cdot 3 + 1 = 7$, *etc.* Inspection of these numbers suggests $T_n = 2^n - 1$ for all $n$. Exercise 17 asks you to give the simple proof by induction that this is true. But this very time-consuming algorithm is also the *best* algorithm, because we can prove that any solution of the puzzle requires at least this many moves:

**Theorem 6.5.3.** *Let $n \geq 0$. Any solution of the Towers of Hanoi puzzle with $n$ disks takes at least $2^n - 1$ moves.*

*Proof.* We prove this by induction on $n$.
*Base Step.* If $n = 0$ there are no disks. We cannot do better than $2^n - 1 = 1 - 1 = 0$ moves.

*Inductive Step.* Let $n \geq 0$, and suppose that any solution of the puzzle with $n$ disks takes at least $2^n - 1$ moves, for some $n \geq 0$. We will show any solution of the puzzle with $n+1$ disks requires at least $2^{n+1} - 1$ moves.

Suppose we have a solution of the $(n+1)$-disk puzzle. Consider the situation *just before* the first time we move the largest disk. (See Figure 6.7) The largest disk is sitting on one post (say post 1) and the post it will be moved to (say post 2) must be empty, because we cannot put this disk on top of a smaller one. That means that the remaining $n$ disks are stacked on the remaining post (post 3). To get to this stage we had to solve the puzzle with $n$ disks (we just ignore the bottom disk, which has not moved in all this time). By the inductive hypothesis, this cost us at least $2^n - 1$ moves.

Now consider the situation *just after* the *last* time we move the largest disk. If we moved it from post 1 to post 2, then it is now sitting on post 2 and post 1 is empty. That means that the

Figure 6.7: The puzzle immediately prior to a move of the largest disk.

remaining $n$ disks are stacked on post 3. To complete the puzzle, we will move them all to post 2 without disturbing the bottom disk. By the inductive hypothesis, this requires $2^n - 1$ moves, at least.

So the entire solution requires $2^n - 1$ moves before we first touch the bottom disk, and $2^n - 1$ moves after we last touch it. In the best case, there is only one move of the bottom disk, so we will require at least

$$2^n - 1 + 2^n - 1 + 1 = 2^{n+1} - 1$$

moves. This completes the proof. □

## 6.6 Historical Notes

The modern description of Mathematical Induction was first given by De Morgan in a brief 1838 encyclopedia article. However, the underlying method of reasoning is very old, and is even implicit in some arguments in Euclid (4th century BC). These early works tacitly assume that there is no infinite descending chain $n_1 > n_2 > n_3 > \cdots$ of natural numbers, a property equivalent to strong induction, without identifying this as special principle.

In his geometric works, Archimedes (3rd century BC) gives as an axiom the following property: If two magnitudes are unequal, then by adding the difference to itself enough times, you arrive at a magnitude greater than the larger of the two magnitudes. In modern notation, this is saying, if $0 < a < b \in \mathbf{R}$ then there is some integer $n$ such that $n(b - a) > b$. That is, there is some integer $n > \frac{b}{b-a}$, which is the 'Archimedean principle' discussed in the text.

The use of recursive procedures appeared very early in the history of high-level programming languages, in LISP (1958) and ALGOL 60 (1960).

The Fibonacci numbers were introduced in a problem in the treatise *Liber Abaci* (1202) by Leonardo Pisano (also known as Fibonacci). (See the notes for Chapter 7.)

The Towers of Hanoi puzzle was invented by the French mathematician Édouard Lucas in 1883. It has nothing to do with the city of Hanoi in Vietnam. Lucas attached to his invention a story about priests in an Eastern temple moving the disks, apparently as a marketing gimmick.

## 6.7 Exercises

### 6.7.1 Basic identities

1. Evaluate the sums and products below for a few small values of $n$. Then make a conjecture about the value of the sum and product for all positive integers $n$, and prove your conjecture by induction.

   *(a)* $\sum_{j=1}^{n} \frac{1}{j(j+1)}$

   *(b)* $\sum_{j=0}^{n} 2^j$

   *(c)* $\left(\sum_{j=1}^{n} j\right)^2 - \sum_{j=1}^{n} j^3$. (HINT: To prove your conjecture, you will probably want to use the formula, given in the text, for the sum of the first $n$ positive integers.)

   *(d)* $\prod_{j=1}^{n}\left(1 - \frac{1}{j+1}\right)$.

   *(e)* $\prod_{j=1}^{n}\left(1 - \frac{1}{(j+1)^2}\right)$. (HINT: You may have to play around a bit with the numerators and denominators of the values you find in order to detect the pattern.)

2. Prove by induction that if $m \geq 0$ is an even integer, then $2^m - 1$ is divisible by 3. (HINT: What do you add in passing from $2^m - 1$ to $2^{m+2} - 1$?)

3. *(Sum of geometric series.)*

   *(a)* Prove by induction that if $r$ is any real number different from 1, then

   $$\sum_{j=0}^{n} r^j = \frac{r^{n+1} - 1}{r - 1}.$$

   Observe that part *(b)* of the previous problem is a special case.

   *(b)* Suppose that today you put 1000 dollars in a deposit account that pays you 1% interest every month. After $N$ months, the account balance will be $1000 \times 1.01^N$ dollars. Now consider a variant of this scenario where, beginning one month from today, you make a deposit of 100 dollars each month. So, for example, after 1 month your account will have $1000 \times 1.01 + 100 = 1110$ dollars; note that the interest is applied to the funds that have been sitting into the account for the past month, and not to the new deposit.

   Find an expression, using the summation notation, for the account balance after $N$ months under this new scenario. Then use the result of part *(a)* to find a simpler formula for the balance. What is the balance one year from today?

4. Prove by induction that

   $$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$$

   for all positive integers $n$.

5. Prove by induction on $n$ that

$$\sum_{j=1}^{2^n} \frac{1}{j} > \frac{n}{2}.$$

(As a consequence, the partial sums of the *harmonic series*

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$$

grow without bound, so this series does not converge to a finite limit.)

6. Prove that

$$\sum_{j=1}^{n} \frac{1}{j^2} \leq 2 - \frac{1}{n}$$

for all positive integers $n$. (As a consequence, the partial sums of the series

$$1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots$$

are bounded above by 2, and thus the series converges to a finite limit less than 2. In fact this limit is $\frac{\pi^2}{6}$.)

7. There is something particularly unsatisfying about induction problems like Exercise 4: A proof by induction requires that you *know the result* (in this case, the closed formula for the sum) before you can prove it, but how do you find the result in the first place? For summation identities like the sum of the first $n$ odd integers, or those in Exercise 1, it is not too hard to guess at the formula from inspection of a few examples, but how could you possibly guess the right-hand side of the equation in Exercise 4?

   Here we will see a method for deriving such identities from scratch, in effect eliminating the need for an inductive proof. We begin with a derivation of the sum for the first $n$ positive integers, a formula we've already seen. Consider first the sum

$$\sum_{j=0}^{n} ((j+1)^2 - j^2).$$

   If you write out the sum in reversed order, as

$$((n+1)^2 - n^2) + (n^2 - (n-1)^2) + \cdots + (2^2 - 1^2) + (1^2 - 0^2),$$

   you can see that everything cancels except for the $(n+1)^2$ at the beginning. On the other hand, we have

$$\sum_{j=0}^{n} ((j+1)^2 - j^2) = \sum_{j=0}^{n} (2j+1) = 2\Big(\sum_{j=0}^{n} j\Big) + n + 1.$$

   We thus get the equation

$$(n+1)^2 = 2\Big(\sum_{j=0}^{n} j\Big) + (n+1),$$

from which the identity for $\sum_{j=0}^{n} j$ follows quite easily.

Now for the problem: Use the same trick, beginning with the sum

$$\sum_{j=0}^{n}((j+1)^3 - j^3)$$

to derive a formula for

$$\sum_{j=1}^{n} j^2.$$

Repeat the procedure to find formulas for the sum of the first $n$ cubes and the first $n$ fourth powers.

### 6.7.2 Growth rate of functions

8. Using the argument in the text as a guide, find a specific $N$ such that for any $n > N$,

$$1.001^n > n^3.$$

9. *(Comparison of exponential growth at different bases.)* Show that if $a, b, c, d > 0$ are positive real numbers with $a > b > 1$, then

$$ca^n > db^n$$

for all sufficiently large values of $n$. (The point is that this holds even if $c$ is very small in comparison to $d$.)

10. If you tabulate values of the expression

$$\left(1 + \frac{1}{n}\right)^n$$

for positive integer values of $n$ you get the following results:

| $n$ | $(1+1/n)^n$ |
|---|---|
| 1 | 2 |
| 2 | 2.25 |
| 3 | 2.37 |
| 10 | 2.59 |
| 100 | 2.705 |
| 1000 | 2.717 |

This suggests that the sequence increases as $n$ grows larger, and that the values are not increasing without bound, but rather converge to some finite limit less than 3. Here you are asked to prove both these facts. Neither one is particularly simple to prove, so we will break the problem into baby steps.

Here is a practical consequence: $(1 + 1/n)^n$ represents the account balance after one year if you put 1 dollar in the bank at 100% interest compounded $n$ times a year. The claim that the sequence increases with $n$ says that the more often you compound the interest, the greater

the balance will be. But the second claim says that no matter how often you compound it, even every microsecond, the balance will never exceed three dollars.

*(a)* Prove that for all positive integers $n$ and any real number $a > 0$,

$$\frac{n-a}{n} < \frac{n+1-a}{n+1}.$$

(HINT: Resist the temptation to try to prove *everything* by induction: instead, try rewriting the two ends of the inequality in a different form.)

*(b)* Prove that for all integers $1 < k \leq n$,

$$\frac{\binom{n}{k}}{n^k} < \frac{\binom{n+1}{k}}{(n+1)^k}.$$

(HINT: Use the result of part *(a)* of this exercise together with induction on $k$ (not $n$).)

*(c)* Apply the Binomial Theorem and part *(b)* of this exercise to show that for all positive integers $n$,

$$\big(1+\frac{1}{n}\big)^n < \big(1+\frac{1}{n+1}\big)^{n+1}.$$

*(d)* Use the Binomial Theorem again to prove that for all integers $n \geq 2$,

$$\big(1+\frac{1}{n}\big)^n < \sum_{j=0}^{n} \frac{1}{j!} < 2 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}}.$$

Then apply the result of Exercise 3 to show that this is less than 3.

The limit of the sequence $\big(1+\frac{1}{n}\big)^n$, as you may know, is denoted $e$ and has value approximately 2.71828.

11. The aim of this problem is to have you prove that the factorial function grows faster than any exponential function, and then to pin down its rate of growth more precisely. Parts *(b)* and *(c)* of this problem depend on the results of the preceding problem, but you may assume these facts even if you have not proved them.

*(a)* Prove that for all real numbers $a > 1$,

$$n! > a^n$$

for sufficiently large values of $n$. (HINT: Pick an integer $m$ greater than $a$ and show that $n! > c \cdot m^n$ for $n > m$ and some positive constant $c$, which depends on $m$. Then use the result of Exercise 9.)

*(b)* Show by induction on $n$ that for all positive integers $n$,

$$n! > \frac{n^n}{e^n}.$$

(HINT: The result of the preceding problem implies that $\big(1+\frac{1}{n}\big)^n < e$ for all positive integers $n$.)

*(*c)* Let $c$ be a positive constant with $1 < c < e$. Show that for sufficiently large $n$,

$$n! < \frac{n^n}{c^n}.$$

(HINT: First choose a value $d$ between $c$ and $e$. The results of the preceding problem imply that $\left(1 + \frac{1}{n}\right)^n > d$ for sufficiently large values of $n$. Use this to show that $n! < n^n/(b \cdot d^n)$ for some constant $b$ and sufficiently large values of $n$, and then apply the result of Exercise 9.)

The results of this problem can be refined further. *Stirling's formula* gives the approximation

$$n! \approx \frac{n^n}{e^n} \cdot \sqrt{2\pi n}.$$

This is an exact asymptotic estimate in the sense that the ratio between the two sides of the inequality converges to 1 as $n$ grows larger.

12. Here is a series of refinements of Theorem 6.5.1, culminating in an astonishing formula for the Fibonacci numbers.

*(a)* Prove that $F_n > 1.5^n$ for sufficiently large values of $n$. (You may have to hunt a bit to find the first value of $n$ for which this is true. The trick that makes the proof work is the fact that $1 + 1.5 > 1.5^2$.)

*(b)* Prove that $F_n < 1.7^n$ for all positive integers $n$. (The trick that makes *this* work is that $1 + 1.7 < 1.7^2$.)

You should get the point now that exponential lower bounds for the Fibonacci numbers, like the result of *(a)*, hold for bases $\alpha$ with $1 + \alpha > \alpha^2$, and that upper bounds like *(b)* hold if $1 + \alpha < \alpha^2$. The magic number is the value of $\alpha$ such that $1 + \alpha = \alpha^2$.

*(c)* Show that if $\alpha > 1$ satisfies $1 + \alpha = \alpha^2$, then $\alpha = \frac{1+\sqrt{5}}{2}$. Compute this value to four decimal places.

*(d)* Show that for all $n \geq 1$,

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

What is remarkable here is that although the expression on the right-hand side is shot through with irrational numbers, its value for each $n$ is an integer. Observe that this expression has the form $c \cdot a^n - c \cdot b^n$, where $a > 1$ and $|b| < 1$. For large $n$, $b^n$ approaches 0 as a limit, so the $n^{th}$ Fibonacci number is just the nearest integer to $c \cdot a^n$.

### 6.7.3 Strong induction and the least integer principle

13. This is a rather abstract problem about ordered sets and the different flavors of induction. For its solution, you may assume a few basic properties of the set $\mathbf{N}$ natural numbers: $\mathbf{N}$ is linearly ordered, there is no natural number strictly between $m$ and $m + 1$ for any $m \in \mathbf{N}$, and 0 is the smallest natural number (that is, $0 \leq m$ for all $m \in \mathbf{N}$).

*(a)* Use ordinary induction to prove the *least integer principle:* Every nonempty subset of the natural numbers has a smallest element, that is, there exists $m \in S$ such that for all $m' \in S$, $m \leq m'$. (HINT: Prove by induction that if $S$ is a subset of the natural numbers without a smallest element, then for all $m \in \mathbf{N}$, $m \notin S$, and thus $S$ must be empty.)

*(b)* A linearly-ordered set that satisfies the least element principle is said to be *well-ordered.* So the result of part *(a)* says that $\mathbf{N}$ is well-ordered. Is $\mathbf{Z}$ well-ordered? Is $\mathbf{Z}^+$ well-ordered? Is the set of nonnegative rational numbers well-ordered?

*(c)* Let $X$ be a nonempty well-ordered set. (We denote the ordering on $X$ by $\leq$, and write $x_1 < x_2$ to mean $x_1 \leq x_2$ and $x_1 \neq x_2$.) Show that $X$ satisfies the strong induction principle. We state the principle as follows: Let $Y \subseteq X$ such that *(i)* $m \in Y$, where $m$ is the least element of $X$, and *(ii)* for every $x \in X$ such that $y \in Y$ for all $y < x$, $x \in Y$. Then $Y = X$. (HINT: If there were a $Y \subsetneq X$ satisfying *(i)* and *(ii),* consider the least element of $X - Y$ and derive a contradiction.)

*(d)* Consider the set $\mathbf{N} \times \mathbf{N}$ with the following ordering: $(m, n) \leq (m', n')$ if and only if $m \leq m'$, or $m = m'$ and $n \leq n'$. Show that this set is well-ordered.

*(e)* Suppose that $X$ is a well-ordered set without a greatest element. Show that every element $m$ has a *successor* $m'$: This is the smallest element such that $m < m'$. (So $m+1$ is the successor of $m$ on the ordering on $\mathbf{N}$.)

*(f)* The ordinary induction principle can be stated this way: Let $X$ be well-ordered, and let $Y \subseteq X$. If the smallest element of $X$ is in $Y$, and if for all $y \in Y$, the successor of $y$ is in $Y$, then $Y = X$. Show that it is *not* the case that every well-ordered set satisfies the ordinary induction principle. (HINT: Show that $\mathbf{N} \times \mathbf{N}$ of part *(d)* has a proper subset $Y$ that satisfies these two conditions.)

The moral is that 'strong induction' is a *weaker* principle than ordinary induction, because there are sets that satisfy the strong induction principle but not the ordinary one.

14. *(The chocolate bar theorem.)* A bar of chocolate is, usually, a rectangular array of individual square chocolate tablets. To break a chocolate bar, you separate it into two smaller bars by splitting along either a horizontal or a vertical line. How many breaks do you need to make in a chocolate bar with $n$ tablets in order to completely separate it into individual tablets?

    If you experiment with this a bit, you will find that the answer does not depend on the dimensions of the bar (*e.g.,* there is no difference between a $4 \times 3$ bar and a $6 \times 2$ bar), nor on the strategy chosen. Find the answer and prove that it is correct using strong induction.

## 6.7.4 Structural induction.

15.* (Generalized associative law.) The goal of this problem is to show that the associative law

$$(a + b) + c = a + (b + c)$$

really does imply that no matter how you introduce parentheses into a sum of many elements, the result does not change. The same holds for any operation on a set obeying an associative law. This is one of those things that appears obvious, but that is not so easy to prove.

Let us generate strings that we will call 'fully parenthesized sums' by the following grammar:

$$S \to \textbf{number}, S \to (S + S).$$

So, for example,

$$((34 + 2) + (51 + (47 + 16))$$

is an instance of such a string. We will also generate a subset of these sums that we will call 'left-parenthesized sums' by

$$T \to \textbf{number}, T \to (T + \textbf{number}).$$

An example is

$$((((34 + 2) + 51) + 47) + 16).$$

It is pretty easy to verify in each individual case that by repeated application of the associative law, you can convert a fully-parenthesized sum into a left-parenthesized sum with the same summands in the same order. Prove that this works in general: That is, use the ordinary associative law for addition, and strong induction, to show that every fully parenthesized sum has the same value as the corresponding left-parenthesized sum (the left-parenthesized sum with the same summands in the same order).

16.* In Chapter 1, we asked you to prove that neither $\{\neg, \oplus\}$ nor $\{\vee, \wedge\}$ is a complete set of connectives, but gave you absolutely no tools to prove such a thing! The tool to use is structural induction.

*(a)* Use structural induction to prove that any formula $\phi$ defined using only the connectives $\vee$ and $\wedge$ has the following property: Let $\alpha$ be an assignment of truth values to the variables in $\phi$, and let $\hat{\alpha}$ be an assignment that results by switching the value of $\alpha$ at one variable from **false** to **true**. (That is, there is a variable $p$ such that $\alpha(p) = \textbf{false}$ and $\hat{\alpha}(p) = \textbf{true}$, but $\alpha(q) = \hat{\alpha}(q)$ for all other variables $q$.) If $[\phi](\alpha) = \textbf{true}$ then $[\phi](\hat{\alpha}) = \textbf{true}$. The desired result follows because not every formula has this property (for example, $\neg\phi$).

*(b)* Use structural induction to prove that any formula $\phi$ defined using only the connectives $\neg$ and $\oplus$ has the following property: Whenever we change the value of an assignment $\alpha$ at a single variable $p$, then value of $[\phi](\alpha)$ changes if $p$ occurs an odd number of times in the formula, and is unchanged if $p$ occurs an even number of times in the formula. The result follows because not every formula (for example $p \vee q$) has this property.

### 6.7.5 Recursive algorithms

17. Let $T_n$ be the number of moves used by the algorithm to transfer $n$ disks in the Towers of Hanoi puzzle. Use the recursive description of $T_n$ to prove that $T_n = 2^n - 1$ for all $n \geq 1$.

18. Consider the Towers of Hanoi puzzle with an additional constraint: You can only move a disk from one post to an *adjacent* post. The object of the puzzle is to move all the disks from the left post to the right post. Observe that with two disks this now takes 8 moves instead of 3, namely

$$1 \mapsto 2, 2 \mapsto 3, 1 \mapsto 2, 3 \mapsto 2, 2 \mapsto 1, 2 \mapsto 3, 1 \mapsto 2, 2 \mapsto 3.$$

Find an optimal strategy for solving this new version of the puzzle. Derive a recurrence relation for the number of moves in your strategy for the puzzle with $n$ disks, and then find and prove a closed-form formula for this number.

# Chapter 7

# Basic Number Theory

*The nine Indian figures are:*

*9 8 7 6 5 4 3 2 1.*

*With these nine figures, and with the sign 0 which the Arabs call zephir, any number whatsoever is written, as is demonstrated below.*

—The opening sentences of *Liber Abaci* (1202) by Leonardo Pisano (Fibonacci)

This chapter is devoted to the arithmetic of the integers. We will begin with a study of positional number systems, an ancient subject of critical importance in understanding what modern computers do. We'll finish with an analysis of card-shuffling, and a surprising application to cryptography.

## 7.1 Divisibility

### 7.1.1 Quotient and Remainder

When you were in the third or fourth grade, and were first learning about division, your work probably looked something like this:

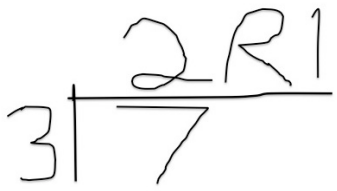



Figure 7.1: Third-grade division

A more grown-up way to write the result is the equation

$$7 = 2 \cdot 3 + 1.$$

This is an instance of what may be the most important fact about the arithmetic of the integers. It ought to be named the Fundamental Theorem of Arithmetic, but that name is already taken. Instead it is often called the Division Algorithm, although it is not really an algorithm.

**Theorem 7.1.1.** *Let $n \in \mathbf{Z}$, $d \in \mathbf{Z}^+$. There exists a unique pair of integers $q$ and $r$ such that $0 \leq r < d$ and*

$$n = qd + r.$$

Think $n$ = 'number', $q$ = 'quotient', $d$ = 'divisor', and $r$ = 'remainder'.

Observe that the divisor is always a positive integer, and the remainder is always nonnegative, but $n$ can be negative, and so can the quotient:

$$-7 = -3 \cdot 3 + 2.$$

You might have preferred the quotient in this case to be -2 and the remainder -1, but requiring the remainder to be nonnegative makes things simpler, as we'll see later on when we discuss congruences.

The quotient and remainder of integer division are built into just about every programming language. In Python[1] they can be treated as either the results of two separate operations, or of a single one that returns a pair of integers:

```
>>> 7/3
2
>>> 7%3
1
>>> divmod(-7,3)
(-3, 2)
```

Observe that Python treats the quotient and remainder for negative $n$ in the way we have defined them. Many other programming languages do not![2]

In standard mathematical notation we write the integer quotient using the 'floor' notation $\lfloor x \rfloor$, as

$$\lfloor n/d \rfloor,$$

since it is the largest integer less than or equal to the rational number $n/d$, and the remainder as

$$n \bmod d.$$

(There is also, as you might guess, a *ceiling* notation: $\lceil x \rceil$ denotes the smallest integer greater than or equal to $x$.)

[1] This use of the expression `7/3` is from Python 2.7, which performs integer division–the kind of division we are describing here–when both the arguments to `/` are of type `int`. This has been changed in Python 3, in which the operator `/` always treats its arguments as floats, and will give the result `2.3333` for this quotient. Integer division in Python 3 is performed by typing `7//3`.

[2] For example, Java evaluates `-7 / 3` as `-2` and `-7 % 3` as `-1` so as to maintain the relation $n = qd+r$. This behavior is also typical in implementations of the C programming language, from which Java is descended. In fact, the original specification of C says that whether `-7/3` evaluates to `-2` or -3 is machine-dependent, hardly the best idea if one is interested in program portability! There might be some practical benefit to the remainder of a division having the same sign as the dividend, but it makes the mathematical treatment more awkward. Python's design tends to adhere more closely to the common practice in mathematics.

*Proof of Theorem 7.1.1.* Let $d, n$ be as in the statement of the theorem. We'll leave the *existence* part of the proof—fact that every integer $n$ has such a representation as $qd + r$—as an exercise in mathematical induction. We prove the uniqueness part by contradiction: Suppose we had two such representations

$$n = qd + r = q'd + r'.$$

Either $r \leq r'$ or $r' \leq r$. We'll suppose that $r \leq r'$–the argument is the same in either case—and subtract to get

$$0 = q'd - qd + r' - r,$$

so

$$(q - q')d = r' - r.$$

The contradiction will be that the right-hand side of this equation must be less than $d$, but the left-hand side must be at least as large as $d$. Here are the details: If $r' = r$, both sides are 0. Thus $(q - q')d = 0$, and since $d > 0$, this means $q - q' = 0$, so $q = q'$. In this instance both $r = r'$ and $q = q'$, so the two representations are not different at all.

Thus $r < r'$, but since $r' < d$, this means $r' - r < d - r \leq d$. Since $(q - q')d = r' - r$ is a positive integer, we have to have $q - q'$ be at least 1. So

$$r' - r < d = 1 \cdot d \leq (q - q')d = r' - r,$$

which makes $r' - r$ less than itself, a contradiction. □

### 7.1.2 The 'divides' relation

When the remainder $n$ mod $d$ is 0, we simply have

$$n = qd.$$

In this case we write

$$d|n$$

and say '$d$ divides $n$', or '$n$ is a multiple of $d$', or '$n$ is divisible by $d$', or '$d$ is a factor of $n$'—these all mean the same thing.

Here are a few very basic facts about this relation on the integers. These are all easy to prove, and you will probably be able to convince yourself of their truth by just looking at a couple of examples.

**Theorem 7.1.2.** *Let $m_1, m_2, m_3 \in \mathbf{Z}$.*
*(a) If $m_1|m_2$ and $m_2|m_3$, then $m_1|m_3$.*
*(b) $m_1|m_1$.*
*(c) If $m_1|m_2$ and $m_2|m_1$, then $m_1 = \pm m_2$.*
*(d) If $m_1|m_2$ and $m_1|m_3$, then $m_1|(m_2 + m_3)$ and $m_1|(m_2 - m_3)$.*
*(e) If $m_1|m_2$ or $m_1|m_3$ then $m_1|m_2m_3$.*

```
def digits(n,d):
        if n<d:
                return [n]
        else:
                return digits(n/d,d)+[n%d]
```

Figure 7.2: An algorithm to generate the radix `d` representation of a positive integer `num`

In the terminology of relations, part *(a)* of this Theorem says that the 'divides' relation is transitive, part *(b)* says it is reflexive, and part *(c) almost* says it is antisymmetric. In fact it *is* antisymmetric when restricted to the set of nonnegative integers, and thus defines a partial order on **N**. Part *(d)* has already shown up for us in the special case $m_1 = 2$, in which case it says that the sum of two even numbers is even.

The *converse* of part *(e)* is: If $m_1|m_2m_3$ then $m_1|m_2$ or $m_1|m_3$. This is *false*–just try $m_1 = 4$, $m_2 = m_3 = 2$. But why it is false, and the special cases for which it is true, are a matter of some interest, which we will explore later.

## 7.2 Positional Number Systems

Our conventional system for writing numbers is called a *positional number system*, because the value of each individual symbol depends on its position in the expression. For example the '2' in '532' means two, but in '4327' it means twenty, and in '23069' it means twenty thousand. This particular positional number system has *radix,* or *base,* ten. The binary system used in computer arithmetic has radix two. The invention of positional number systems took place independently at several different times in human history (and with several different bases). What is now so familiar and obvious to us was in fact a great scientific advance: With a small fixed repertory of symbols, people were able to write in a matter of seconds integers far larger than any they could count up to, compare them at a glance, and do arithmetic with them using simple and efficient algorithms.

Theorem 7.1.1 will provide us with an explanation of why positional number systems work. The algorithm shown in Figure 7.2 repeatedly divides its first input by a fixed divisor `d`, and returns the sequence of remainders. (While the algorithm could have been encoded with a simple `for` loop, we have encoded it in a recursive fashion so that it more closely resembles the argument we make below in Theorem 7.2.1.)

We'll execute this algorithm with a few different pairs of inputs:

```
>>> digits(470,10)
[4, 7, 0]
>>> digits(123,2)
[1, 1, 1, 1, 0, 1, 1]
>>> digits(32,3)
[1, 0, 1, 2]
```

To understand what's going on, let's work in detail through the last example. The algorithm begins by dividing 32 by 3, giving a quotient of 10 and a remainder of 2. The remainder 2 will be

the last element of the list returned by the function; the remaining elements will be computed by applying the algorithm to the quotient 10. This yields a quotient of 3 and a remainder of 1, which becomes the next-to-last element of the list. We then pass 3 to the function and get a quotient of 1 and a remainder of 0. The recursion bottoms out when we pass the argument 1 to the function. So the sequence of divisions performed by the algorithm is:

$$\begin{aligned} 32 &= 3 \cdot 10 + 2 \\ &= 3 \cdot (3 \cdot 3 + 1) + 2 \\ &= 3 \cdot (3 \cdot (3 \cdot 1 + 0) + 1) + 2 \\ &= 1 \cdot 3^3 + 0 \cdot 3^2 + 1 \cdot 3 + 2 \end{aligned}$$

Thus the repeated division gives an expression for the original value as a weighted sum of powers of $d$, with the remainders as the weights.

We provide a formal statement and proof in following theorem, but keep in mind that the whole idea is contained in the calculation above. If $d > 1$, we call a sequence $(c_k, \ldots, c_0)$ a *radix $d$ representation of $n \in \mathbf{Z}^+$* if

$$n = c_k d^k + \cdots + c_1 d + c_0,$$

where $0 \leq c_i < d$ for all $i$, and $1 \leq c_k$.

**Theorem 7.2.1.** *Let $n, d \in \mathbf{Z}^+$ with $d > 1$. The function* `digits` *applied to the pair of inputs $(n, d)$ returns a radix $d$ representation of $n$. Furthermore, this representation is unique: there is only one radix $d$ representation of $n$.*

From a practical standpoint, the uniqueness of the representation is almost as important as the existence—we need to know that two different sequences of digits really represent two different numbers.

*Proof.* We prove the first claim by strong induction on $n$. If $1 \leq n < d$, then the claim holds, because the one-term sequence $(n)$ is a radix $d$ representation of $n$. Now suppose $n \geq d$ and that for every positive integer $n' < n$, `digits(n',d)` returns a radix $d$ representation of $n'$. By Theorem 7.1.1, `digits(n,d)` returns a sequence $(c_k, c_{k-1}, \ldots, c_0)$, where

$$n = qd + c_0,$$

with $0 \leq c_0 < d$. Since $n > d$, we have $k > 0$ and $q > 0$, and $(c_k, c_{k-1}, \ldots, c_1)$ is the sequence returned by `digits(q,d)`. By the inductive hypothesis, this sequence is a radix $d$ representation of $q$. Thus $1 \leq c_k < d$, $0 \leq c_i < d$ for $1 \leq i < k$, and

$$\begin{aligned} n &= qd + r \\ &= (c_k d^{k-1} + \cdots + c_1)d + c_0 \\ &= c_k d^k + \cdots + c_1 d + c_0. \end{aligned}$$

This shows that $(c_k, c_{k-1}, \ldots, c_0)$ is a radix $d$ representation of $n$, as required.

In particular, every positive integer $n$ has at least one radix $d$ representation. We prove again by induction that there is only one: If $1 \leq n < d$ then a radix $d$ representation of $n$ can have only one digit, and thus must be $(n)$ itself. Now suppose $n > d$ has two radix $d$ representations

$$n = c_k d^k + \cdots + c_1 d + c_0 = c'_m d^m + \cdots + c'_1 d + c'_0,$$

and that all positive integers smaller than $n$ have a unique representation. We can write

$$n = (c_k d^{k-1} + \cdots + c_1)d + c_0 = (c'_m d^{m-1} + \cdots + c'_1)d + c'_0.$$

Since $0 \leq c_0 < d$, the uniqueness part of Theorem 7.1.1 implies

$$c_0 = c'_0, c_k d^{k-1} + \cdots + c_1 = c'_m d^{m-1} + \cdots + c'_1,$$

and the inductive hypothesis implies $m = k$ and $c_i = c'_i$ for $1 \leq i \leq k$. Thus the two representations of $n$ are identical. □

### 7.2.1 Base conversion

We court confusion when we begin representing integers at different bases: Does '11' mean eleven, as we're accustomed to, or does it mean three, because we are working in base 2? To resolve the ambiguity, whenever we have several different bases under discussion and intend the representation to be at some base other than ten, we indicate this with a subscript next to the string of digits:

$$11_2 = 3.$$

The function `digits` above provides an algorithm for computing the radix $d$ representation of an integer. If $n < d^k$, then there are no more than $k$ digits in the representation. If $d^k$ is the smallest power of $d$ greater than $n$, we have

$$d^{k-1} \leq n,$$

so taking logarithms of both sides and adding 1 gives

$$k < 1 + \log_d n.$$

As we saw in the last chapter, the logarithm of $n$ (at any base) grows very much more slowly than $n$, which is why numbers of very large magnitude have such succinct representations. In particular, the algorithm for computing the representation requires only about $\log_d n$ divisions, and is thus very fast.

What if you wanted to convert in the other direction—that is, find the value of an integer $n$ given its representation at base $d$? We could, of course, perform all the additions and multiplications present in the representation. For instance, if we wanted to know what integer is represented by 42635 at base 7, we can write

$$n = 4 \cdot 7^4 + 2 \cdot 7^3 + 6 \cdot 7^2 + 3 \cdot 7 + 5$$

We need three multiplications to compute the table of powers of 7, four more to compute the products of these powers with the corresponding digits, then four additions to find the sum of the products. A more efficient way is to work from the most significant digit, repeatedly multiplying by 7 and adding each new digit, effectively reversing the procedure of the algorithm `digits`.

$$n = (((4 \cdot 7 + 2) \cdot 7 + 6) \cdot 7 + 3) \cdot 7 + 5.$$

```
def evaluateDigits(digit_list,d):
    val=0
    for digit in digit_list:
        val=d*val+digit
```

Figure 7.3: An algorithm to find the value represented by a sequence of radix `d` digits

This does only four multiplications and four additions. The general algorithm similarly requires $k-1$ multiplications and $k-1$ additions to evaluate a number whose representation has $k$ digits.

We can thus convert between the representations of an integer at two different radixes $d_1$ and $d_2$ by first using `evaluateDigits` to find the value $n$ represented by the radix $d_1$ representation and then using `digits` to calculate the radix $d_2$ representation of $n$.

An special case occurs when $d_2$ is a power of $d_1$, or vice-versa. In this instance we are able to convert between the two representations without going through the intermediate step of evaluating the representations. This arises in practice when we convert between the binary (radix 2) and *hexadecimal* (radix 16) representations. Hexadecimal digits are the integers from 0 to fifteen inclusive; we use the symbols A,B,C,D,E,F for the digits in the range ten through fifteen. Thus for example

$$\texttt{8A3}_{hex} = 8 \cdot 16^2 + 10 \times 16 + 3 = 2211.$$

To convert to binary, we write the binary representations of each hexadecimal digit as 4-bit sequences, including leading zeros, and string these sequences together. Thus `8` is replaced by 1000, `A` by 1010, `3` by 0011, giving the binary representation $100010100011_2$ of $\texttt{8A3}_{hex} = 2211$. To convert in the other direction, we break the binary representation into 4-bit segments, starting with the least significant bit, and then replace each segment by the hex digit corresponding to the value represented by the segment at radix 2. For example, to convert 10011010010 to hex we write

$$10011010010 \mapsto 100 \quad 1101 \quad 0010 \mapsto \texttt{4D2}.$$

Hexadecimal notation consequently serves as a human-readable version of binary: The correspondence between the hexadecimal digits of a number and 4-bit chunks of its binary representation allows us to quickly extract each individual bit of the binary representation from the hexadecimal, and it is far easier to visually distinguish `EA3` from `F53` than it is to tell `1110100100011` and `111101010011` apart.

We leave it to the reader to explain why this works. The process is analogous to the familiar practice of taking the ordinary decimal representation of an integer $n$ and inserting commas every three digits, which you can think of as giving the radix one thousand representation of the integer: For instance, 31427093 is written as

$$31,427,093 = 31 \cdot 1000^2 + 427 \cdot 1000 + 93.$$

## 7.3 Common factors and Euclid's Algorithm

### 7.3.1 A puzzle

You have two opaque containers, one holding exactly five gallons of water, and the other three gallons. There is a stream nearby from which you can fill the containers. How can you measure out exactly one gallon of water? Think a moment about the problem and try to solve it before looking at the answer below.

Here is one solution: Fill the three-gallon container and pour its contents into the five-gallon container. Then refill the three-gallon container and pour as much of it into the five-gallon container as will fit. Exactly one gallon remains in the five-gallon container.

And here is another solution: Fill the five-gallon container, pour its contents into the three-gallon container, and dump these three gallons of water back into the stream. Pour the two gallons that remain in the five-gallon container into the three-gallon container, refill the five-gallon container, and continue pouring until the three-gallon container is full, then dump the three-gallon container again. If you're keeping track, there are now four gallons remaining in the five-gallon container. Pour one last time into the three-gallon container and exactly one gallon remains in the five-gallon container.

These solutions appear to be the product of trial and error, and you might wonder how you could possibly solve the problem if, say, one of the containers held 439 gallons and the other 256. But there actually is a method behind this: The first solution is based on the observation

$$2 \cdot 3 - 1 \cdot 5 = 1,$$

so that filling the three-gallon container twice and using it to fill up the five-gallon container once leaves exactly one gallon. The second solution comes from the equation

$$2 \cdot 5 - 3 \cdot 3 = 1,$$

so we can fill the five-gallon container twice and use it to fill the three-gallon container three times, again leaving exactly one gallon.

But what if we were given, say, a 6-gallon container and a 15-gallon container? Every time we perform the operation of filling one container from the stream or from the other container, each one will wind up with a number of gallons divisible by 3. So if the capacities of the two containers have a *common factor* bigger than 1, the puzzle cannot be solved.

If the capacities have *no* common factor greater than 1, as in our original example with 3 and 5, can the puzzle always be solved? It turns out that the answer is yes, but we will need a little work first to show how to find the solution.

### 7.3.2 Euclid's Algorithm for Computing the GCD

We obtained the radix $d$ representation of an integer by repeatedly dividing a number by $d$ and returning the list of remainders. There is another way to do repeated division, using the remainder from each step as the divisor in the next step. For example, if we start with the values 273 and

```
def euclid(n1,n2):
    if n2==0:
        return n1
    else:
        return euclid(n2,n1%n2)
```

Figure 7.4: Euclid's Algorithm computes the greatest common divisor of its two inputs

196 we would compute

$$\begin{aligned} 273 &= 1 \cdot 196 + 77 \\ 196 &= 2 \cdot 77 + 42 \\ 77 &= 1 \cdot 42 + 35 \\ 42 &= 1 \cdot 35 + 7 \\ 35 &= 5 \cdot 7 + 0 \end{aligned}$$

We give a recursive Python implementation of this algorithm in Figure 7.3.2. This scheme forces the remainder to strictly decrease at each step, so the remainder eventually becomes 0. Our implementation returns the last nonzero remainder (which is 7 in the above example). This number has special properties:

**Theorem 7.3.1.** *Euclid's Algorithm applied to $n_1, n_2 \in \mathbf{N}$, with $n_1 \neq 0$, returns a positive integer $d$ such that $d|n_1$ and $d|n_2$. Further, if $d'$ is an integer with $d'|n_1$ and $d'|n_2$, then $d'|d$.*

The first conclusion of the Theorem says that the value $d$ returned by Euclid's Algorithm is a common divisor of the two inputs. The second says that any other common divisor of the inputs is a divisor of $d$. In particular, $d$ is the *greatest common divisor* of $n_1$ and $n_2$, and we denote it $\gcd(n_1, n_2)$. The reason is pretty simple: If you work from the last division in the example displayed above up to the first, you find that 7 divides every one of the values on the left-hand side of the equations. If you start from a value $d'$ that divides the two initial numbers 273 and 196, and work your way down, you find that $d'$ divides every single one of the left-hand sides, and in particular divides 7. Using the recursive description of the algorithm, we get a quick proof by strong induction:

*Proof.* We argue by induction on the second input $n_2$. We first note that the theorem holds whenever $n_2 = 0$. In this case the algorithm returns $n_1$, which obviously satisfies the two criteria.

Now suppose that $n_2 > 0$ and that the conclusion of the theorem holds whenever Euclid's Algorithm is applies to $n_1', n_2'$ with $0 \leq n_2' < n_2$. On inputs $n_1, n_2$, the algorithm recursively calls itself on inputs $n_2, r$, where $n_1 = qn_2 + r$, with $0 \leq r < n_2$. By the inductive hypothesis, the value $d$ that is returned is the $\gcd(n_2, r)$. So to complete the proof, we just need to show that $\gcd(n_1, n_2) = \gcd(n_2, r)$, in other words, that the common divisors of $n_1$ and $n_2$ are exactly the common divisors of $n_2$ and $r$. If an integer divides both $n_2$ and $r$, then if divides $n_1 = qn_2 + r$, and thus is a common divisor of $n_1$ and $n_2$. Conversely, if $d$ divides both $n_1$ and $n_2$, then it divides $r = n_1 - qn_2$, and thus is a common divisor of $n_2$ and $r$, completing the proof.

□

### 7.3.3 Extended Euclid's Algorithm

Let's elaborate a bit on the inductive step in the second part of the preceding proof. Our algorithm throws away the quotient at each division, and only works with the remainder. But we can use these quotients to get some additional information. Let's apply the algorithms to the integers 27 and 92. We have

$$\begin{aligned} 92 &= 3 \cdot 27 + 11 \\ 27 &= 2 \cdot 11 + 5 \\ 11 &= 2 \cdot 5 + 1. \end{aligned}$$

Now we rewrite the last equation as

$$1 = 11 - 2 \cdot 5.$$

The equation just before this expresses 5 in terms of 11 and 27:

$$5 = 27 - 2 \cdot 11,$$

and if we substitute this into the preceding equation, we get

$$1 = 11 - 2 \cdot (27 - 2 \cdot 11) = 5 \cdot 11 - 2 \cdot 27.$$

Similarly, we have

$$11 = 92 - 3 \cdot 27,$$

so

$$1 = 5 \cdot (92 - 3 \cdot 27) - 2 \cdot 27 = 5 \cdot 92 - 17 \cdot 27.$$

We can thus use Euclid's algorithm to express the gcd (in this example, 1) in terms of the two starting values. Here is the general statement:

**Theorem 7.3.2.** *Let $n_1, n_2 \in \mathbf{N}$ with $n_1 \neq 0$. There are integers $a, b$ such that*

$$an_1 + bn_2 = \gcd(n_1, n_2).$$

*Proof..* Figure 7.5 revises our original implementation of Euclid's algorithm so that it returns two such integers $a$ and $b$ along with the gcd $d$. We prove by induction on $n_2$ that if this algorithm returns $(a, b, d)$, then $d = an_1 + bn_2$. If $n_2 = 0$, then we have $a = 1, b = 0$, and the claim holds. Now assume $n_2 > 0$, and that the claim holds whenever the algorithm is applied to $n_1', n_2'$, with $0 \leq n_2' < n_2$. We have

$$n_1 = qn_2 + r,$$

where $0 \leq r < n_2$, and the algorithm applied to inputs $n_2, r$, gives integers $a, b$, with

$$d = an_2 + br.$$

Applied to $n_1, n_2$, the algorithm returns $(b, a - bq, d)$, and we have

```
#return (a,b,d), where d is the gcd
#of the arguments and d=an1+bn2

def extended_euclid(n1,n2):
    if n2==0:
        print 1,0
        return (1,0,n1)
    else:
        (q,r)=divmod(n1,n2)
        (a,b,d)=extended_euclid(n2,r)
        print b,a-b*q
        return (b,a-b*q,d)
```

Figure 7.5: Extended Euclid's Algorithm

$$\begin{aligned} bn_1 + (a - bq)n_2 &= b(qn_2 + r) + (a - bq)n_2 \\ &= an_2 + r \\ &= d, \end{aligned}$$

so the claim holds for $n_1, n_2$.
□

We say that two integers $n_1, n_2$ are *relatively prime* if $\gcd(n_1, n_2) = 1$. In this case Theorem 7.3.2 tells us that there are integers $a$ and $b$ such that $an_1 + bn_2 = 1$.

This provides a complete solution to the water jug problem for any pair of containers: If the two capacities have a factor greater than 1 in common, then we know we can't solve it. On the other hand, if they have no such common factor–that is, if they are relatively prime–we now have an algorithm that generates a solution. For instance, if you have a 92-gallon container and a 27-gallon container (and are very strong), fill the 92-gallon container 5 times, repeatedly pouring its contents into the 27-gallon container and dumping the latter each time it fills up. At the end you will have filled the 27-gallon container 17 times, and have exactly one gallon left in the 92-gallon container.

### 7.3.4 Speed of Euclid's Algorithm

Much like our algorithm for computing the radix $d$ representation of an integer, Euclid's Algorithm, in both its basic and extended forms, is very fast, and will terminate rapidly even when applied to very large integers. Let's see why this is so. Consider one step of the algorithm:

$$r_1 = qr_2 + r_3.$$

Since the remainders decrease at every step, $r_2 < r_1$, so $q \geq 1$. Thus

$$r_1 = qr_2 + r_3 \geq 1 \cdot r_2 + r_3 > r_3 + r_3 = 2r_3.$$

Thus at each step, the remainder is less than half what it was two steps before, so after $2k$ steps the remainder decreases by a factor of at least $2^k$. If we apply Euclid's Algorithm to a pair $n_1 > n_2$ of positive integers and the algorithm performs $m$ divisions, we have

$$2^{m/2} < n_1,$$

so

$$m < 2 \log_2 n_1.$$

We showed in Theorem 6.3.3 that $\log_2 n$ grows much more slowly than $n$. If $n$ is very large, we might expect the time required to perform division to grow as the square of the number of digits, so that the time required to carry out the extended Euclid algorithm is at worst proportional to $(\log_2 n_1)^3$, which is still much smaller than $n_1$ for large values of $n_1$.

As we'll see later, cryptographic algorithms in wide use apply Euclid's Algorithm to extremely large integers, sometimes as large as $2^{1024}$. Our argument above justifies the claim that this can actually be carried out in practice.[3]

## 7.4 Prime Numbers

### 7.4.1 Prime factorization

An integer $d > 1$ is *prime* if its only positive divisors are 1 and $d$. The first few primes are 2,3,5,7, and 11. Every integer $n > 1$ is either prime or a product of primes. For example,

$$84 = 2 \cdot 2 \cdot 3 \cdot 7.$$

To see why, we'll start dividing by 2, then 3, *etc.,* until we find a divisor of $n$. Eventually we *will* find some divisor $p$ of $n$, because $n|n$. This smallest divisor $p$ of $n$ must be prime, because if it were not, the divisors of $p$ would give us smaller divisors of $n$. So we can write $n = p \cdot (n/p)$. If $p \neq n$, then $n/p > 1$, so we can apply the same process to $n/p$, each time extracting another prime factor, until we reduce to 1. We could formalize this informal argument as a proof by strong induction, or by appeal to the least integer principle. We have implemented it as a Python function in Figure 7.4.1. But the point we wanted to make is that the *existence* of a prime factorization of every $n > 1$ really is obvious.

What is far less obvious is the *uniqueness* of the prime factorization of $n$. Is it possible for some integer $n$ to have more than one prime factorization? Of course, we can write

$$84 = 2 \cdot 2 \cdot 3 \cdot 7 = 7 \cdot 2 \cdot 3 \cdot 2,$$

but that shouldn't count as two different factorizations. What we are asking is whether it is possible for an integer to have two prime factorizations that are not merely permutations of one another. Somehow, it *feels* like it shouldn't be possible, and you may already be used to the expression '*the* prime factorization' of an integer. But it is not so easy to explain why this should be so.

[3] To see that this is so not just in theory, but in practice, you can run the Python code above with numbers hundreds of digits long, and see the result come back in a flash.

```
def prime_factorization(num):
    factorlist=[]
    while num>1:
        divisor=2
        while num%divisor != 0:
            divisor=divisor+1
        factorlist.append(divisor)
        num=num/divisor
    return factorlist
```

Figure 7.6: Computing a Prime Factorization

We will officially state and prove the theorem on unique factorization, which is called 'the Fundamental Theorem of Arithmetic', below. But first let's give the intuition behind the argument. Imagine some integer $n$ with two distinct prime factorizations

$$n = p_1 \cdots p_k = q_1 \cdots q_r,$$

where the $p_i$ and $q_j$ are all primes. If these two factorizations have a prime factor in common, we can cancel the common prime and get a smaller integer with two distinct prime factorizations. We can keep doing this until no common primes are left, so we arrive at an integer with two prime factorizations which have *no* primes in common. This means, in particular, that $p_1$ is not equal to any of the $q_j$. But

$$p_1 | n = q_1 \cdots q_r.$$

Somehow we have split the prime $p_1$ among these other primes. That seems wrong, but there is nothing in the definition of prime numbers that allows us to rule this out immediately. For this, we'll use something we learned from Euclid's Algorithm:

**Theorem 7.4.1.** *Let $m, n$ be integers and $p$ prime. If $p|mn$, then $p|m$ or $p|n$.*

*Proof.* Suppose $p|mn$, and $p \not| m$. We'll show $p|n$. Since $p$ is prime, $\gcd(p, m) = 1$. By Theorem 7.3.2, there are integers $a$ and $b$ such that $ap + bm = 1$. We now have

$$\begin{aligned} n &= (ap + bm)n \\ &= apn + bmn \end{aligned}$$

Since $p|apn$ and $p|bmn$ (by the hypothesis $p|mn$) we have $p|apn + bmn = n$. □

By repeated application of the above result, we get the following corollary.

**Corollary 7.4.2.** *If $p, m_1, \ldots, m_r$ are integers with $p$ prime and $p|m_1 \cdots m_r$, then $p|m_i$ for some $i$.*

This lets us change the musings above into a proof.

**Theorem 7.4.3.** *(Fundamental Theorem of Arithmetic) Every positive integer $n > 1$ has a unique factorization into primes. More precisely, suppose $n > 1$ and*

$$n = p_1 \cdots p_k = q_1 \cdots q_r,$$

*where all the $p_i$ and $q_j$ are prime and*

$$p_1 \leq p_2 \leq \cdots \leq p_k, q_1 \leq q_2 \leq \cdots \leq q_r.$$

*Then $k = r$ and $p_i = q_i$ for all $1 \leq i \leq k$.*

*Proof.* Suppose to the contrary that there is an integer $n > 1$ with prime factorizations as above that are not identical. By the Least Integer Principle, there is a smallest such $n$. The two factorizations of $n$ can have no prime in common, otherwise we could cancel the common prime and get a smaller integer with two distinct factorizations. In particular $p_1$ does not appear among the $q_i$. But

$$p_1 | n = q_1 \cdots q_r,$$

so by Corollary 7.4.2, $p_1 | q_i$ for some $i$. Since $q_i$ is prime, that means $p_1 = q_i$, so the two factorizations have a factor in common after all, a contradiction. □

What is the time complexity of the algorithm in Figure 7.4.1? The largest number of divisions will be performed in the case where the outer loop is executed only once but the inner loop is executed $n - 1$ times on input $n$. This happens when $n$ is itself a prime. We can improve this performance by noting that if $n$ is not a prime, then $n$ has a factorization $n = m_1 m_2$ into two smaller integers. Suppose $m_1$ is the minimum of the two factors. Then $n \geq m_1^2$, so we only have to test potential divisors up to the square root of $n$. As big as this speedup is, it is not at all helpful when dealing with very large numbers. A product of two primes, each of which has one hundred decimal digits, cannot be factored by this method, since we would have to perform about $10^{100}$ divisions.

As you go to higher and higher values, primes appear to get rarer. This raises the question of whether there might be only finitely many primes. The following argument shows this is not the case: Take any finite collection of primes:

$$p_1, \ldots, p_n.$$

Form their product and add 1:

$$M = p_1 \cdot p_2 \cdots p_n + 1.$$

We know from our discussion above that $M$ has a prime divisor, so let $q$ be a prime such that $q | M$. Thus $M \bmod q = 0$, but for $i = 1, \ldots, n$, $M \bmod p_i = 1$. So $q$ is not equal to any of the $p_i$. This tells us that for any finite set of primes, there must be a prime that is not in the set, so the set of all primes must be infinite. This proves:

**Theorem 7.4.4.** *There are infinitely many primes.*

It is important in applications not just to know that there are infinitely many primes, but the rate at which they thin out as we test larger integers. We will return to this point in Section 7.5.3.

## 7.5 Congruence

What is

$$(4329 \cdot 2111) \bmod 7 \quad ?$$

We can evaluate the product, divide by 7, and find the remainder. But it is easier to divide the factors 4329 and 2111 by 7, find the remainders (in this case 3 and 4), and take the remainder of the product. This entails a couple of extra divisions, but we only have to work with rather small numbers, which makes the computation by hand much quicker:

$$3 \cdot 4 \bmod 7 = 12 \bmod 7 = 5.$$

Why does this work? The rule we have applied is

$$(ab) \bmod d = ((a \bmod d) \cdot (b \bmod d)) \bmod d.$$

Here is the proof: Let's write $a = q_1 d + r$, $b = q_2 d + r'$, where $r = a \bmod d$ and $r' = b \bmod d$, and $rr' = q_3 d + r''$, where $r'' = (rr') \bmod d$. We want to show that $r''$ is also the remainder $ab$ leaves on division by $d$. This is because

$$ab = (q_1 d + r) \cdot (q_2 d + r') = (q_1 q_2 + r + r')d + rr' = (q_1 q_2 + r + r' + q_3)d + r''.$$

An even simpler computation shows that this method works for evaluating the remainders left by sums and differences: You can reduce the summands mod $d$, then add, then reduce the result mod $d$, as an alternative to adding large summands and then reducing. The take-away is that we can do arithmetic with remainders more or less as if they were ordinary integers. You already saw this with the rules for adding and multiplying even and odd integers:

| | | | | |
|---|---|---|---|---|
| even | + | even | = | even |
| even | + | odd | = | odd |
| odd | + | odd | = | even |
| odd | × | odd | = | odd, |

and so on. We are saying that this sort of thing works with any divisor $d$, not just 2. We formalize this below, and introduce an important new notation.

### 7.5.1 Definition and basic properties

Let $x, y \in \mathbf{Z}$, $n \in \mathbf{Z}^+$. We write

$$x \equiv y \pmod{n}$$

if $x$ and $y$ leave the same remainder upon division by $n$; that is, if $x \bmod n = y \bmod n$. We say in this case that '$x$ is congruent to $y$ modulo $n$'. For example,

$$8 \equiv -2 \pmod{5}.$$

since both of these integers leave a remainder of 3 upon division by 5.

This definition makes it obvious that congruence modulo $n$ is an equivalence relation on the integers. The following simple fact, which gives an alternative characterization of congruence, is often used as the definition.

**Proposition 7.5.1.** *Let $x, y \in \mathbf{Z}$, $n \in \mathbf{Z}^+$. Then*

$$x \equiv y \pmod{n}$$

*if and only if*

$$n|(x-y).$$

*Proof.* First suppose $x \equiv y \pmod{n}$. Let $r = x \bmod n$. By assumption, we also have $r = y \bmod n$. This means there are integers $q, q'$ such that

$$x = qn + r, y = q'n + r,$$

and thus

$$x - y = (q - q')n,$$

so $n|(x-y)$.

Conversely, suppose $n|(x-y)$. Let $r_1 = x \bmod n$, $r_2 = y \bmod n$. Now we have

$$x = qn + r_1, y = q'n + r_2.$$

We want to show $r_1 = r_2$. We can suppose without loss of generality that $r_1 \geq r_2$ (the argument will be identical if $r_1 \leq r_2$, just with $x$ and $y$ interchanged). Then

$$x - y = (q - q')n + (r_1 - r_2).$$

We have $0 \leq r_1 - r_2 \leq r_1 < n$, so that $r_1 - r_2$ is the remainder $x - y$ leaves on division by $n$, and thus $r_1 - r_2 = 0$, which is what we wanted to prove. $\square$

To illustrate, in our example above, $8 - (-2) = 10$, which is divisible by 5.

The next theorem formalizes what we meant above when we said we can do arithmetic with remainders as though they were ordinary integers.

**Theorem 7.5.1.** *Let $x, y, x', y' \in \mathbf{Z}$, $n \in \mathbf{Z}^+$. Suppose*

$$x \equiv y \pmod{n}$$

$$x' \equiv y' \pmod{n}.$$

*Then*

$$x + x' \equiv y + y' \pmod{n}$$

$$x - x' \equiv y - y' \pmod{n}$$

$$xx' \equiv yy' \pmod{n}.$$

*Proof.* We already proved the third conclusion, concerning the product, at the start of this section:

$$(xx') \bmod n = ((x \bmod n)(x' \bmod n)) \bmod n = ((y \bmod n)(y' \bmod n)) \bmod n = (yy') \bmod n,$$

so

$$xx' \equiv yy' \pmod{n}.$$

The conclusions for addition and subtraction work similarly, and are easier. $\square$

In our example at the beginning of the section, we found

$$4329 \equiv 3 \pmod 7,$$

$$2111 \equiv 4 \pmod 7,$$

so

$$4329 \cdot 2111 \equiv 3 \cdot 3 \equiv 5 \pmod 7.$$

Let's give a few more examples.
*Modular exponentiation.* What is $5^{2392}$ mod 7? We can start by computing the first few powers of 5 modulo 7, reducing at each step so we have only small values to work with:

$$\begin{array}{rcrcrcl} 5^2 & = & 25 & \equiv & 4 & & \pmod 7 \\ 5^3 & \equiv & 4 \cdot 5 & = & 20 & \equiv -1 & \pmod 7 \\ 5^4 & \equiv & -1 \cdot 5 & = & -5 & \equiv 2 & \pmod 7 \\ 5^5 & \equiv & 2 \cdot 5 & = & 10 & \equiv 3 & \pmod 7 \\ 5^6 & \equiv & 3 \cdot 5 & = & 15 & \equiv 1 & \pmod 7 \end{array}$$

Since $2392 = 6 \cdot 398 + 4$, this gives

$$5^{2392} = 5^{6 \cdot 398 + 4} = (5^6)^{398} \cdot 5^4 \equiv 1^{398} \cdot 2 = 2 \pmod 7.$$

We'll observe a similar thing whenever we work with a small modulus $n$ (in this case$n = 7$). Since there are no more than $n$ possible remainders, the powers $a^k$ mod $n$ will start to repeat in a cycle after just a few steps. We can then use this repetition to compute $a^k$ mod $n$ for very large values of $k$. Below we will find that we can compute $a^k$ mod $n$ fairly quickly even when $a$ and $b$ are quite large. *Casting out nines.* Since

$$10 \equiv 1 \pmod 9,$$

we have

$$10^k \equiv 1^k = 1 \pmod 9$$

for every $k \geq 0$. This means, for example, that

$$3247 = 3 \cdot 10^3 + 2 \cdot 10^2 + 4 \cdot 10 + 7 \equiv 3 + 2 + 4 + 7 \equiv 7 \pmod 9.$$

The general principle is that *every positive integer is congruent modulo 9 to the sum of its decimal digits.* Moreover, we can simplify the process of computing the sum of the digits mod 9 by looking for subsets of the digits whose sum is 9 and crossing them out. For instance, in the example above we can cross off the digits 3,2,and 4, leaving the result 7.

This is the basis for a method of checking the correctness of computations done by hand. Suppose we find, after some laborious arithmetic, that

$$24315 \times 7432 = 180709080.$$

If we cast out nines in the first factor we get

$$\cancel{24}315 \equiv 1 + 5 = 6 \pmod 9,$$

and in the second

$$7\cancel{432} \equiv 7 \pmod 9.$$

So the product should be congruent to $7 \times 6 \equiv (-2) \cdot (-3) = 6$ modulo 9. For the product we have

$$\cancel{1807090}80 \equiv 7 + 8 = 15 \equiv 6 \pmod 9,$$

which suggests that the multiplication is correct. Of course this is not a guarantee of correctness, but the method will catch 8 out of 9 random errors.

### 7.5.2 Fast Modular Exponentiation

In the modular exponentiation example above, we were able to rapidly compute $5^{2392} \bmod 7$ by hand, without having to compute the gigantic number $5^{2392}$, because the powers of 5 modulo 7 cycle as $5, 4, 6, 2, 3, 1, \ldots$, completing a cycle every six powers. What if we wanted to compute

$$39^{43} \bmod 67 \quad ?$$

We can avoid having to deal with large numbers by reducing modulo 67 after each multiplication by 39, but we would still have to perform 42 multiplications and reductions. There is a nice trick for speeding up the calculation. Let's represent the exponent 43 in binary, as a sum of distinct powers of 2:

$$43 = 2^5 + 2^3 + 2 + 1.$$

We can compute the powers $39^{2^k} \bmod 67$ for $k \leq 5$ by repeatedly squaring and reducing mod 67:

$$\begin{array}{rcrcrcrl}
39^2 & = & 1521 & & & \equiv & 47 & \pmod{67} \\
39^4 & \equiv & 47^2 & = & 2209 & \equiv & 65 \equiv -2 & \pmod{67} \\
39^8 & \equiv & (-2)^2 & & & = & 4 & \pmod{67} \\
39^{16} & \equiv & 4^2 & & & = & 16 & \pmod{67} \\
39^{32} & \equiv & 16^2 & = & 256 & \equiv & 55 & \pmod{67}
\end{array}$$

We then have

$$\begin{aligned}
39^{43} &= 39^{32} \cdot 39^8 \cdot 39^2 \cdot 39 \\
&\equiv 55 \cdot 4 \cdot 47 \cdot 39 \pmod{67} \\
&\equiv 54
\end{aligned}$$

which requires just three multiplications and reductions to compute. All told, we have performed 8 multiplications and reductions modulo 67 instead of 42.

The Python function displayed in Figure 7.7 implements this repeated-squaring algorithm. To compute $a^b \bmod c$, the inner loop is executed $\lceil \log_2 b \rceil$ times, and there are two multiplications and reductions in each step. As with computing the radix $d$ representation of a number and the gcd of two numbers, this algorithm's running time is roughly logarithmic in the size of its input (in this case, the input $b$), and thus can be executed in practice with extremely large numbers. This will be important below when we discuss cryptography.

Incidentally, fast modular exponentiation is performed by the built-in Python function `pow` applied to three integer inputs.

```
def repeated_squaring(base,exponent,modulus):
    power=1
    basepower=base
    while exponent>0:
        (exponent,remainder)=divmod(exponent,2)
        if remainder==1:
            power=(power*basepower)%modulus
        basepower=(basepower**2)%modulus
    return power
```

Figure 7.7: Fast modular exponentiation by repeated squaring

### 7.5.3 Fermat's Theorem and Primality Testing

In an example we worked above, we saw that the powers of 5 mod 7 formed cycles of length 6. Let's see what happens when we compute the successive powers mod 7 of all the integers $1 \leq a < 7$. In the tabulation below, the $j^{th}$ entry of the $i^{th}$ line is $i^j$ mod 7.

```
for j in range(1,7):
    for k in range(1,7):
        print '%2d,'% ((j**k)%7),
        print


 1,  1,  1,  1,  1,  1,
 2,  4,  1,  2,  4,  1,
 3,  2,  6,  4,  5,  1,
 4,  2,  1,  4,  2,  1,
 5,  4,  6,  2,  3,  1,
 6,  1,  6,  1,  6,  1,
```

The powers of 2 and 4 form cycles of length 3. The powers of 6 form cycles of length 2. We get 6 different values among the powers of 3 and 5, but these too will cycle if we continue to compute larger powers: since the last value in the sequence is 1, the values will repeat. In all cases we have $a^6 \bmod 7 = 1$. This always happens when the modulus is a prime.

**Theorem 7.5.2.** *(Fermat's Theorem) Let $p$ be prime and $1 \leq a < p$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Furthermore, we will always see the same kind of cyclic behavior, so that the smallest $k$ such that $a^k \bmod p = 1$ will always be a divisor of $p - 1$. You can see this in the example above where we have cycles of length 1,2,3, and 6.

The proof of this is elementary, but a bit involved, so we won't give it here. Instead, we will mention an important application.

Suppose we want to test whether a large value, say $n = 15248792147$, is prime. This number is larger than $10^{10}$, and the algorithm we gave earlier requires us to test potential divisors that may be as large as $\sqrt{n} > 10^5$. Instead, let's evaluate $a^{n-1} \bmod n$ for a randomly selected value of $a$.

```
>>> a=random.randint(2,15248792146)
>>> print pow(a,15248792146,15248792147)
13603881447
```

If $n$ were prime, then Theorem 7.5.2 tells us that the result would be 1, so this computation proves that $n$ is composite. Thanks to fast modular multiplication, the computation required no more than $2 \cdot \lceil \log_2 n \rceil = 68$ multiplications and reductions, far fewer than the more than one hundred thousand multiplications we would perform with the naïve test. We have found a way to prove that a number is composite that gives no hint of its factorization.

What if we tested a value $n$ like this and found that $a^{n-1} \bmod n = 1$? Would that prove that $n$ is prime? Not necessarily. But if we tested a few different values of $a$ at random and found that $a^{n-1} \bmod n = 1$ for all of them, then $n$ is almost certain to be prime. The matter is somewhat more complicated than this—for instance, there are some rare composite values of $n$ that will always give 1 as the result, and some extra care is required to rule these out—but Fermat's Theorem is still the basis of such *probabilistic* primality tests. The take-away is that in practical terms, *we can reliably and efficiently test whether a given integer is prime or composite.*

### 7.5.4 Card shuffling

Take a standard deck of 52 cards. A *perfect shuffle,* also called a *Faro shuffle,* is the result of splitting the pack of cards into two equal-sized packs and perfectly interleaving them. Actually, there are two kinds of perfect shuffles: *in-shuffles* and *out-shuffles:* In the out-shuffle, the top card of the original deck is used as the top card of the shuffled deck (and thus the bottom card remains the bottom card). In an in-shuffle, which is more appropriate if you're shuffling to be fair in the game, the top card of the original deck will become the second card of the shuffled deck.

We want to answer the following question: If you repeatedly shuffle the cards in this manner, will the deck ever return to its original order? If so, how many shuffles will this require?

We'll start by analyzing the out-shuffle. Let's number the cards 0 to 51. The order of the cards after the first shuffle is

$$0, 26, 1, 27, 2, 28, \ldots, 25, 51.$$

The last card will always be 51, and the first 0, so we don't need to write these. Moreover, we might notice that $2 \cdot 26 \bmod 51 = 1$, $3 \cdot 26 \bmod 51 = 27$, *etc.*, so we can write the order of the shuffled deck as

$$1 \cdot 26 \bmod 51, 2 \cdot 26 \bmod 51, 3 \cdot 26 \bmod 51, \ldots, 50 \cdot 26 \bmod 51.$$

(To see why this works, imagine we arranged the original deck in a circle, with cards 0 and 51 coinciding. Shuffling the deck is the same thing as repeatedly advancing 26 cards around the circle

to find the next card in the shuffled deck.) We repeat the process, so that after 2 shuffles, the order of the deck (not counting the first and last cards) is

$$26^2 \bmod 51, 2 \cdot 26^2 \bmod 51, \ldots, 50 \cdot 26^2 \bmod 51.$$

And after $k$ shuffles the deck is in the order

$$26^k \bmod 51, 2 \cdot 26^k \bmod 51, \ldots, 50 \cdot 26^k \bmod 51.$$

We can solve the problem of restoring the deck to its original order if we can find the least integer $k$ such that

$$26^k \equiv 1 \pmod{51}.$$

We can make this a little easier if we observe that

$$26^k \times 2^k \equiv (26 \times 2)^k = 52^k \equiv 1^k = 1 \pmod{51}.$$

It follows that if $2^k \bmod 51 = 1$ then we will also have $26^k \bmod 51 = 1$.

Now, before we calculate a thing, we can get an idea of the answer: The prime factorization of 51 is $17 \times 3$. By Fermat's theorem, we know that

$$2^{16} \equiv 1 \pmod{17}$$

and

$$2^2 \equiv 1 \pmod{3},$$

so that

$$2^{16} = (2^2)^8 \equiv 1^8 = 1 \pmod{3}.$$

Thus both 17 and 3 are prime divisors of $2^{16} - 1$, which means that their product 51 also divides this number, and thus

$$2^{16} \equiv 1 \pmod{51}.$$

So we know that 16 shuffles suffice. But we can do better. As we saw in our discussion of Fermat's Theorem, the *smallest* integer $k$ such that $2^k \bmod 17 = 1$ is a divisor of 16, and thus will be 2,4,8 or 16. We'll test these by repeated squaring:

$$2^2 = 4$$
$$2^4 = 4^2 = 16 \equiv -1 \pmod{17}.$$
$$2^8 \equiv (-1)^2 \equiv 1 \pmod{17}.$$

Since we have $2^8 \bmod 17 = 1$, and $2^8 \bmod 3 = 1$, we know that both 3 and 17 are prime factors of $2^8 - 1$, and thus 51 divides $2^8 - 1$. So $2^8 \bmod 51 = 1$, and this is the best possible answer. Eight shuffles bring the deck back to its original order.

What about the in-shuffle? The analysis is pretty much the same, coupled with the observation that an in-shuffle with a 52-card deck is the same as an out-shuffle with a 54-card deck, since the top and bottom cards don't move during the out-shuffle. We thus have to find the least $n$ such that

$$2^n \equiv 1 \pmod{53}.$$

Since 53 is prime, we know by Fermat's Theorem that $n = 52$ is a solution, and that the best solution is a divisor of 52, either 52 itself, or 2,4,13,26. It's clear that 2 and 4 don't work. Since $2^8 = 256 = 5 \times 53 - 9$ we have

$$2^8 \equiv -9 \pmod{53}$$
$$2^4 \equiv 16 \pmod{53}$$

so

$$2^{13} = 2^8 \cdot 2^4 \cdot 2 \equiv (-9) \cdot 16 \cdot 2 = -288 \equiv 30 \pmod{53}$$
$$2^{26} = (2^{13})^2 \equiv (30)^2 = 900 = 17 \cdot 53 - 1 \equiv -1 \pmod{53}$$

So none of these works, which means that 52 shuffles is the optimal answer.

# 7.6 Public-Key Cryptography

## 7.6.1 Symmetric and asymmetric cryptography

We will put some pieces of the earlier sections of this chapter together in order to describe an important application to cryptography.

Two parties, Alice and Bob, want to communicate privately, so that their messages cannot be read by an eavesdropper, Eve. Alice and Bob thus decide to encrypt their communications. To do this, they previously agreed on some secret information —a *key*. To send a message $M$ (the *plaintext*) to Bob, Alice combines it with the key $K$ and *encrypts* it using an encryption algorithm $E$. Bob receives the output of this algorithm, the *ciphertext* $C = E(M, K)$. To recover the plaintext, Bob combines the ciphertext with the key and applies a decryption algorithm $D$, and finds $M = D(C, K)$. This setup is called *symmetric encryption*: Alice and Bob share the same secret information $K$, and Bob can reply to Alice by encrypting his own plaintext messages $M$ using the encryption algorithm $E$. (The encryption and decryption algorithms $D$ and $E$ themselves are *not* secret.)

This begs the question of exactly how Alice and Bob agree on the key $K$ in the first place. In the kinds of practical applications of cryptography that you use all the time, the two parties are typically an online retailer and a customer who have not previously had any contact. The retailer cannot simply send the key $K$ to the customer in unencrypted form, since then any eavesdropper will be able to recover $K$ and decrypt all subsequent communications between the two parties.

The amazing solution to this dilemma is that the retailer actually *does* send $K$ to the customer. $K$ is public information. The trick is that encryption and decryption are not carried out using the same key. Instead there are two keys, the public one $K$ used for encryption, and a secret key $K'$ used for decryption. The customer encrypts a plaintext message $M$ by computing

$$C = E(M, K),$$

and the retailer decrypts it by computing

$$M = D(C, K').$$

This setup is called *asymmetric,* or *public-key* cryptography.

How is such a thing possible? If both the encryption algorithm and the encryption key are publicly known, why can't the adversary reverse-engineer the algorithm to recover the plaintext $M$ from $E(M, K)$?

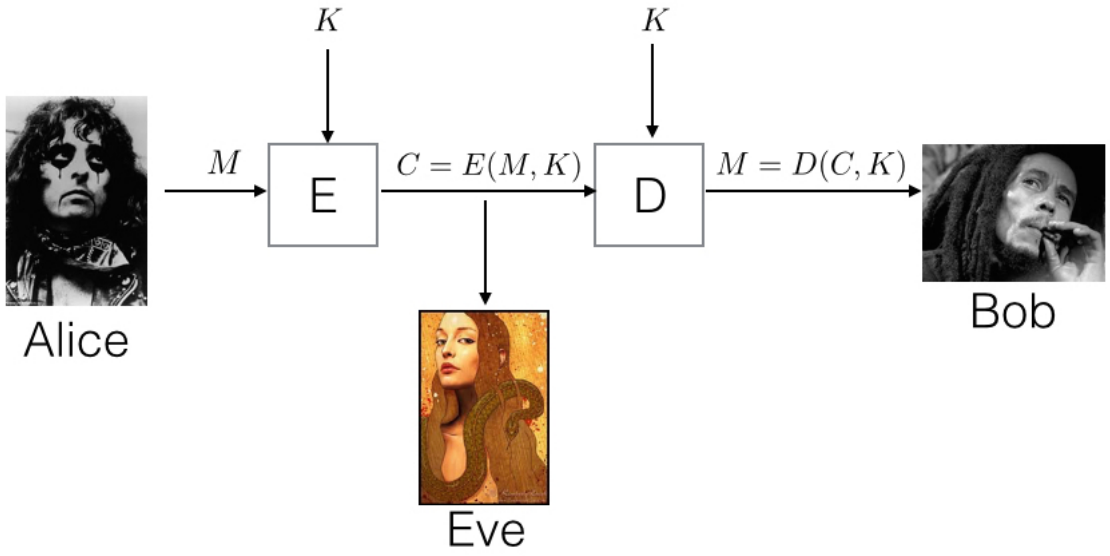



Figure 7.8: Symmetric encryption: Eavesdropper Eve intercepts the ciphertext $C$ and knows the encryption and decryption algorithms $E$ and $D$, but without access to the shared secret key $K$, she cannot recover the plaintext message $M$.

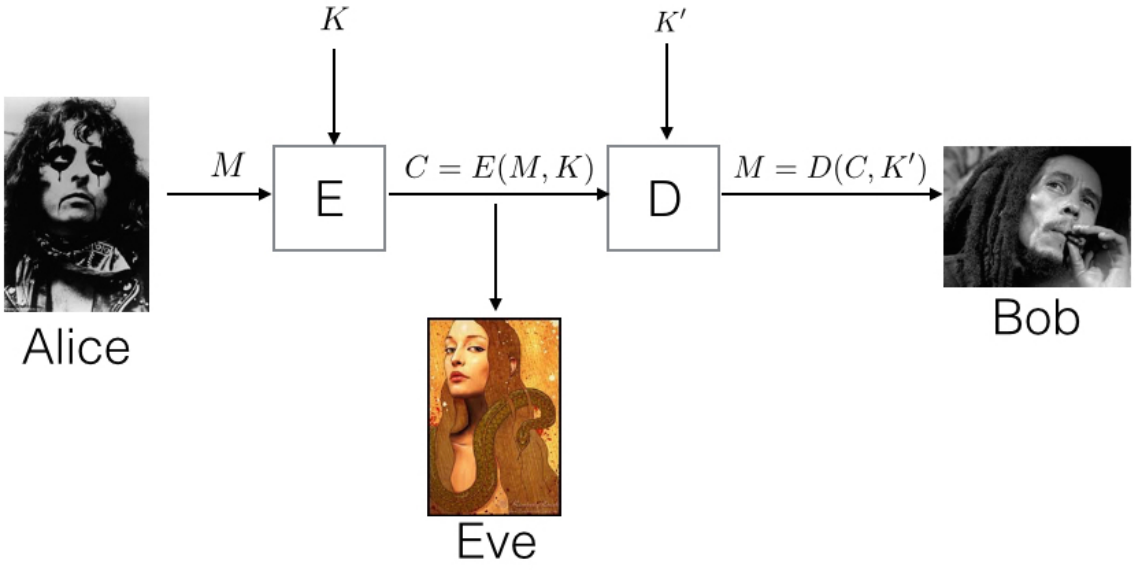



Figure 7.9: Public key encryption: Eavesdropper Eve intercepts the ciphertext $C$ and knows both encryption and decryption algorithms $E$ and $D$, and Bob's public encryption key $K$, but without access to Bob's secret decryption key $K'$, she cannot recover the plaintext message $M$.

### 7.6.2 Easy and hard problems

The answer was found in another problem that, as far as we can tell, cannot be reverse-engineered. As we saw above, it is *easy* to test whether a given integer $N$ is prime. By 'easy', we mean that the time required for the test is proportional to $(\log N)^k$ for some small power $k$, so the running time scales up like the number of digits in $N$. In practice, this means that we can quickly test for primality of numbers with hundreds, or even thousands, of decimal digits.

It is not only easy to tell if a large number is prime; it is also easy to *find* large primes. As we saw above in Theorem 7.4.4, there is an infinite supply of primes. A more difficult result concerns the distribution of primes: The number of primes less than $n$ is approximately $n/\ln n$, where $\ln n$ denotes the natural logarithm of $n$. The precise statement of this fact is known as the *Prime Number Theorem.* The Prime Number Theorem allows us to estimate the number of primes in a given interval. For example, suppose we wanted to find a prime with exactly 100 decimal digits. The number of primes with fewer than 100 decimal digits is approximately

$$\frac{10^{100}}{100 \cdot \ln 10} \approx \frac{10^{100}}{230}.$$

So roughly about 1 out of every 230 100-digit integers is prime. If we test integers at random and discard the even ones, we will only have to test about 115 integers before we find a prime. Thus, without much computational effort, we can generate large prime numbers.[4]

On the other hand, factoring a number into primes appears to be *hard.* The algorithm we saw earlier to find a divisor of $N$ took about $\sqrt{N}$ divisions in the worst case, so for a 100-digit integer this would mean $10^{50}$ divisions, a number of operations far too large for computers to perform. There are actually much faster algorithms for factoring, but all of them scale up badly with the size of the integer. This problem is believed to be intrinsically hard—that there is no algorithm for efficiently factoring an integer.

So here is the problem that cannot be reverse-engineered: Generate two large primes $p, q$ of approximately 200 digits each. Multiply them together (another easy problem). There is no computer program that can resolve $N = pq$ into its factors without running for many years. In a sense, $N$ holds a secret that cannot be unlocked.

### 7.6.3 The RSA algorithm

How can we harness the difficulty of factoring to create a public-key cryptographic system? The first proposal for doing so, and a method that remains in wide use, is called RSA (after its inventors Rivest, Shamir, and Adleman). We'll describe the steps of the algorithm, work through an example with artificially small parameters, and then prove that the algorithm works.

#### Key Generation

The recipient, Bob, secretly generates two large primes $p \neq q$ and forms the products

$$N = pq, K = (p-1)(q-1).$$

[4] And in fact, we can generate large prime numbers that *have never been generated before.* Flip a coin 300 times, and record the sequence of heads and tails as 1's and 0's. The result is a 300-bit integer, and the probability that the same sequence of 300 coin tosses will ever appear again is for all practical purposes zero.

He chooses an integer $e$ so that $e$ is relatively prime to $K$, and determines integers $d$ and $c$ such that $de = cK + 1$. This is an easy problem in light of Euclid's Algorithm and the extended Euclid Algorithm: we may have to test several different candidate values of $e$ before we find one that is relatively prime to $K$. Observe that $d$ can be chosen so that $0 < d < K$, since if $d$ were outside this range, we can add multiples of $K$ to ensure this. So we can assume $d > 0, c \geq 0$.

Bob publishes the pair $(e, N)$: this is the public key. The integer $d$ is the private key.

### Encryption

Alice copies Bob's public key. The message to be sent must be encoded as an integer $M < N$. Alice then uses the public information and fast modular exponentiation to compute

$$C = M^e \bmod N.$$

The ciphertext $C$ is sent to Bob.

### Decryption

Bob uses his secret information and fast modular exponentiation to compute

$$C^d \bmod N.$$

We will prove below that this is identical to the original plaintext message $M$.

### Example

We'll illustrate the algorithm with an example that uses small integers, so you can see what every step looks like. Of course, in practice, the algorithm is never used with such small values.

Let's choose $p = 53, q = 61$ as our primes. Then

$$N = pq = 3233, K = (p-1)(q-1) = 3120.$$

A little trial and error shows $\gcd(7, K) = 1$, and we can apply the extended Euclid algorithm to find

$$\begin{aligned} 3120 &= 7 \cdot 445 + 5 \\ 7 &= 1 \cdot 5 + 2 \\ 5 &= 2 \cdot 2 + 1 \end{aligned}$$

so

$$\begin{aligned} 1 &= 5 - 2 \cdot 2 \\ &= (3120 - 7 \cdot 445) - 2 \cdot (7 - (3120 - 7 \cdot 445)) \\ &= 3 \cdot 3120 - 1337 \cdot 7 \end{aligned}$$

This would give $d = -1337$, so we adjust by adding 3120, and get $d = 1783$. This is the secret key. The pair $(7, 3233)$ is the public key.

For purposes of this small example let's suppose that the sender encrypts two-letter pairs, first by encoding the pair as a four-digit integer. For example, the pair 'TB' would be encoded as 2002, since T is the 20th letter and B the 2nd letter of the alphabet.

To encrypt this using the public key, the sender computes

$$2002^7 \bmod 3233.$$

We have already seen how to do this quickly using repeated squaring. We'll let the built-in Python function `pow` do the work here:

```
>>> pow(2002,7,3233)
2817
```

The recipient takes the ciphertext 2817 and uses the secret information to decrypt:

$$2817^{1783} \bmod 3233.$$

We use `pow` again to get the result, which is the original plaintext.

```
>>> pow(2817,1783,3233)
2002
```

### Proof of correctness of RSA

We have to show that encryption followed by decryption recovers the original plaintext, in other words, that

$$C^d \bmod N = M^{de} \bmod N = M.$$

To do this we will show

$$M^{de} \equiv M \pmod{p}$$

and

$$M^{de} \equiv M \pmod{q}.$$

This implies that both $p$ and $q$ divide $M^{de} - M$. Since $p$ and $q$ are distinct primes, we have $N = pq$ also divides $M^{de} - M$. Thus

$$M^{de} \equiv M \pmod{N},$$

so $M^{de} \bmod M = M$.

To establish the claims above, note that if $p|M$, then $p|M^{de}$, so

$$M^{de} \equiv 0 \equiv M \pmod{p}.$$

If $p \not| M$, then $M \bmod p = a$, where $1 \leq a < p$. Since $de = cK + 1$ for some integer $c$, we have:

$$\begin{aligned} M^{de} &= M^{cK+1} \\ &= M^{c(p-1)(q-1)}M \\ &= (M^{p-1})^{c(q-1)} \cdot M \\ &\equiv (a^{p-1})^{c(q-1)} \cdot M \\ &\equiv 1 \cdot M (\text{by Fermat's Theorem}) \\ &= M \pmod{p} \end{aligned}$$

The proof that $M^{de} \equiv M \pmod{q}$ is identical.

### Security and Efficiency of RSA

Since the exponent $N$ in RSA is part of the public key, anyone who can find the prime factors of $N$ will be able to reproduce the computation of the secret exponent $d$ and decipher all subsequent messages. Knowledge of the secret exponent can also be used to factor $N$. So RSA is only secure to the extent that factoring is infeasible. Currently, the best factoring algorithms, coupled with massive computational effort, can factor integers of approximately 200 decimal digits, or about 660 bits in binary. Public RSA keys in wide use are typically 1024 or 2048 bits long, with the latter becoming more common.

The conjecture that factoring is intrinsically hard does not itself guarantee the security of RSA: no one has ruled out the possibility that there is some efficient method for decryption that does not recover the secret exponent.

Thanks to tools like fast modular exponentiation, Euclid's algorithm, and probabilistic primality testing, RSA is 'fast', in the sense that the algorithms for key generation and encryption can be carried out reasonably quickly on very large numbers. Still, encryption with RSA is much slower than with conventional symmetric encryption, so it is not well-suited for large volumes of traffic. For this reason, encryption on the Internet is a two-phase process: when you contact a secure website, your browser downloads the RSA public key $(e, N)$, generates a symmetric key $K$, encrypts $K$ with the RSA key and sends this to the server. Now both you and the server share the symmetric key $K$, which is used to encrypt all subsequent communication for the session.

## 7.7 Historical Notes

The first positional number system, using 60 as a base, was invented by the Babylonians around 2000 BC. Another system, with 20 as the base, was developed in Central America by the Maya and their neighbors, and dates at least as far back as the first century AD. The base 10 system in use today originated in India as early as 400 AD, then spread to the Muslim world by the 8th century. The treatise *Liber Abaci* ('The Book of Calculation', 1202) by Leonardo Pisano, *aka* Fibonacci, is the earliest known account of the system written in Europe. The method of casting out nines for checking the result of a calculation is described by Fibonacci.

Leibniz (1703) described the binary (base 2) representation. As late as the 19th century, some writers seriously suggested replacing the decimal system of arithmetic with a base 8 system. (See, in particular, a very long-winded article by A. B. Taylor, in which the author goes so far as to propose a new set of symbols and new names for numbers. In his scheme, 'one thousand one hundred eighty-nine' would be 'dutyder duder foty pa'.)

A calculating machine using balanced ternary notation was invented by Thomas Fowler in England in 1840, and described in an article by DeMorgan, who saw a demonstration of a working version of the machine. Balanced ternary notation was also used in an early experimental electronic computer called Setun, developed in the Soviet Union in the 1950's.

Euclid's algorithm, the proof of the infinitude of primes, and Theorem 7.4.1 (which is the basis for the Fundamental Theorem of Arithmetic) are all described in *The Elements* of Euclid, c. 300 BC. Fermat's Theorem (sometimes called 'Fermat's Little Theorem') was stated by Pierre de Fermat in a 1640 letter (in which he wrote, 'I would send you the proof, if I weren't afraid that it would be too long'.)[5]

[5]This should not be confused with 'Fermat's Last Theorem': thirty years after his death, it was discovered

Congruences were systematically investigated by Gauss, in *Disquisitiones Arithmeticae* (1801). In the same book, Gauss wrote, 'The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic.' From the perspective of computer science, we now understand these to be two very different problems, one of which admits efficient solutions, and the other of which is, apparently, computationally intractable. Efficient probabilistic tests for primality were developed by Miller and Rabin, and Solovay and Strassen. These tests have a microscopically small probability of erroneously identifying a composite number as prime. The first efficient *deterministic test*, in which the answer is guaranteed to be correct, was discovered by Agrawal, Kayal and Saxena in 2002.

The possibility of public-key cryptography was first suggested by Diffie and Hellman in 1976, although there is some evidence that the idea was discovered independently by British and American intelligence agencies and kept secret. It was not long before Rivest, Shamir and Adelman described the RSA algorithm. The widespread use of 'Alice' and 'Bob' as the names of the two communicating parties originates with their paper.

You can watch video of eight perfect out-shuffles restoring a deck to its original order.

## 7.8 Exercises

### 7.8.1 Quotient and remainder

1. Find the decomposition $n = qd + r$, where $0 \leq r < d$ for the following pairs of values:
   (a) $n = 8, d = 2$
   (b) $n = 2, d = 8$
   (c) $n = 372, d = 10$
   (d) $n = 372, d = 100$
   (e) $n = -372, d = 100$
   (f) $n = 6, d = 1$

2. Prove the existence part of Theorem 7.1.1. That is, show for $n \in \mathbf{Z}$, $d \in \mathbf{Z}^+$, there exist integers $q, r$ such that $n = qd + r$ with $0 \leq r < d$. (HINT: First prove this in the case $n \geq 0$ using induction on $n$, then employ this case to obtain the result for negative integers as well.)

3. Prove all five parts of Theorem 7.1.2. (These are all quite simple and direct from the definition.)

4. *(a)* Write a Python function that takes as its argument an integer $n$ and returns a list of all the divisors of $n$. Write your program in such a fashion that it works with negative as well as positive arguments. Why is an argument of 0 a problem?

that Fermat had written in the margin of his copy of an ancient mathematics text the statement of *another* theorem, to which he claimed he had a proof that he was not writing down, because the margin of the book was too small! Fermat's Last Theorem remained one of the most famous unsolved problems in mathematics, until it was finally proved, by Andrew Wiles, in 1994.

*(b)* Use your solution from part *(a)* to find the positive integers less than 1000 with the largest number of divisors.

*(c)* Find all the positive integers less than 1000 with an odd number of positive divisors. For example, the first positive integer produced by this computation will be 1, the next 4, because 4 has three divisors 1, 2 and 4.

*(d)* You should notice something about your answer to (c)! Use this to formulate and prove a theorem about positive integers that have an odd number of divisors

5. *(One hundred lockers)* This puzzle appeared in the 'Numberplay' blog on the website of the *New York Times*. The men's locker room at a local gym has exactly one hundred lockers, numbered 1 through 100. Every night after closing, the janitor performs the following ritual: He first opens every locker. He then walks along the row of lockers again and closes every second locker (that is, lockers 2,4,6, *etc.*) He then repeats and visits every third locker, closing it if it is open (for example locker 3) and opening it if it is closed (for example, locker 6). He repeats his procedure with every fourth locker, every fifth locker, and so on until on the hundredth pass he visits locker 100. The problem is to find how many lockers are open at the end. (You can find the answer by simulating with a computer program, but you can also reason it out using some ideas from the previous exercises.)

6. Prove, by exhibiting a single example, that if $m_1, m_2, n \in \mathbf{Z}$ with $m_1|n$ and $m_2|n$, it is not necessarily true that $m_1 m_2|n$.

7. Here you are asked about some formal properties of the 'divides' relation considered as a partial order on $\mathbf{N}$. An element $m$ of a partially ordered set $X$ with order relation $\preceq$ is said to be *minimal* if $m \preceq x$ for all $x \in X$ such than $x \neq m$. It is said to be a *minimum element* if it is the only minimal element. The terms *maximal* and *maximum* are defined similarly. An element $m'$ is *subminimal* if the only elements $m$ such that $m \preceq m'$ are minimal elements. For example, in the usual ordering on $\mathbf{N}$, 0 is the minimum element, 1 is the only subminimal element, and there is no maximal element.

   *(a)* Show that there is a minimum element for the division relation.

   *(b)* Is there a maximal element? A maximum element?

   *(c)* What are the subminimal elements for the division relation? How many subminimal elements are there?

## 7.8.2 Positional number systems

8. Find the decimal representations of

   *(a)* $263_7$

   *(b)* $1010010001_2$

   *(c)* $\mathrm{A6D}_{hex}$

9. Find the binary representation of the number in (c) above and the hexadecimal representation of the number in (b), *without* using the decimal representations you found.

10. Find the representation of one hundred at all bases $b$ with $2 \leq b \leq 9$. (You should note that you really only have to work at this for bases 2,3,5,6,7; finding the representations at bases 4,8 and 9 is then much easier—why?)

11. *(a)* How many bits are in the binary representation of an integer with ten decimal digits? Give an exact range for the answer, both the minimum and maximum possible.

    *(b)* How many bits are in the binary representation of an integer with $m$ decimal digits? Give the answer in terms of $m$–you will need the floor and ceiling functions, as well as logarithms.

12. Why does the definition of a radix $d$ representation, given just prior to the statement of Theorem 7.2.1. include the requirement that the leading digit $c_k$ be greater than 0? Where in the proof of the theorem is this requirement used? (It really is used.)

13. The algorithms for adding, subtracting, multiplying and dividing integers at radix $b$ are *exactly* the same ones that you learned for radix ten. Of course, the addition and multiplication tables are different (since, for example, at base 5, $4+2=11$ and $4\times 3=22$) but the process of carrying and shifting is the same. With that in mind, compute the following sum, product, and quotient, working entirely in binary, and then check your answer by converting the operands and your answer.

    *(a)* 1100101 + 10101101

    *(b)* 10101 × 1011

    *(c)* 10101/100.

### 7.8.3 Exotic number systems

14. *Base -2 representation.* One of the drawbacks of the positional systems we have studied is that sequences of digits represent only positive values. We can use a modified binary notation to represent negative as well as positive integers (and dispense with the minus sign) if we let the positions be weighted by powers of -2. For instance, in this system 1101 represents

$$(-2)^3 + (-2)^2 + (-2)^0 = -8 + 4 + 1 = -3.$$

    *(a)* Find representations for the integers $\pm 1, \pm 2, \pm 3$. Trial and error works fine here (but see below).

    *(b)* What are the greatest and least integers that can be represented in this system with 4 bits? With $n$ bits?

    *(c)* Find an algorithm for computing a base -2 representation for every integer. (HINT: The lowest-order bit is still the remainder upon division by 2.) Use the idea in your algorithm to write a proof that every nonzero integer has a unique base -2 representation with highest-order bit equal to 1.

    *(d)* Write Python functions for converting integers to and from base -2 representation.

15. *(Balanced ternary representation)* Balanced ternary is like ordinary base 3 in the sense that the positions are weighted by the powers $3^0, 3^1, 3^2, \ldots$. However, the digits represent, not 0,1 and 2, but 0, 1 and -1. We will write $\bar{1}$ for the digit representing -1. Thus for example $1\bar{1}001\bar{1}$ represents

$$3^5 - 3^4 + 3^1 - 3^0 = 243 - 81 + 3 - 1 = 164.$$

The result is that both negative and positive values can be represented in this system.

Repeat parts (a)-(d) of the preceding problem for this system.

16. *(Two's complement representation)* The usual way that negative integers are represented in computers is *not* base -2, nor balanced ternary, nor is a minus sign used. Let us say that we are using $n$ bits to represent an integer. In the *unsigned* interpretation, the bit sequences represent each integer between 0 and $2^n - 1$ inclusive, just as we have described in the text. In the two's-complement system, bit sequences in which the leftmost bit is 0 represent the integers between 0 and $2^{n-1} - 1$ in the usual way, but sequences with the leftmost bit equal to 1 represent the negative integers between $-2^{n-1}$ and $-1$ inclusive: For $1 \leq x \leq 2^{n-1}$, the $n$-bit two's complement representation of $-x$ is the unsigned representation of $2^n - x$. For instance, in 3-bit two's complement, 101 represents -3.

*(a)* Find the 4-bit two's complement representations of 5,0 and -7.

*(b)* What are the integers whose 4-bit two's complement representations are 0111, 1101, 1000 and 1111?

*(c)* What are the integers whose 5-bit two's complement representations are the values given in *(a)*? What is the procedure for passing from the $n$-bit representation of a number $x$ to the $(n+1)$-bit representation? (Explain why this works.)

*(d)* Two's complement representation has the convenient property that the algorithm for adding two numbers is essentially *the same* as that for the conventional representation. If we want to add two integers in the conventional representation, using only a fixed number of bits, then we just perform the usual addition algorithm. As long as the correct answer is within the range of representable numbers, then this algorithm gives it as a result. Here, for example, is the addition of 9 and 5 as 4-bit integers.

$$\begin{array}{cccccc} & 1 & 0 & 0 & 1 \\ + & 0 & 1 & 0 & 1 \\ \hline & 1 & 1 & 1 & 0 \end{array}$$

The result is the binary representation of fourteen, as it should be. Now if we interpret the summands and the answer as two's complement representations, then this is the addition of -7 and 5, giving the result -2, *which is still correct.* Show that this works in general: if two $n$-bit sequences represent $x$ and $y$ in the standard interpretation, and $x'$ and $y'$ in two's complement, and if $x + y$ is representable in the ordinary representation, and $x' + y'$ in two's complement, then the standard representation of $x + y$ is the same as the two's complement representation of $x' + y'$.

*(d)* This is a continuation of part *(c)*. It may happen that the sum of the two summands is not representable in the unsigned representation, but is representable at two's complement. For example, in the addition below

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 1 | 0 |
| + | 0 | 1 | 1 | 1 |
| c | 0 | 0 | 0 | 1 |

there is a *carry out* of the most significant digit (indicated above with the letter c). If the summands are given the unsigned interpretation, we have 10+7=17, which does not fit into 4 bits. The carry out tells us that the addition *overflowed.* But if we interpret the addition in two's complement, the summands are -6 and 7 and the result represents 1, so the addition is still correct. Show that this, too, always works: As long as $x$, $y$, and $x+y$ are all representable in $n$-bit two's complement, then the standard algorithm gives the correct result, provided we ignore any carry out of the most significant bit.

*(e)* As the example above shows, a carry out of the most significant bit does not necessarily indicate overflow in two's complement addition. Give an example where two's complement arithmetic results in overflow (that is, the true sum is not representable in two's complement), but the ordinary interpretation does not. Then, find a criterion for detecting overflow in two's complement notation.

17. *(Sign-magnitude representation)* A seemingly more natural—and certainly easier to understand—way to represent both positive and negative values with a fixed number of bits is to use the highest-order bit as a sign, with 0 for a plus sign and 1 for a minus sign. Thus with four bits `0101` represents 5 and `1101` represents -5. The least number representable in this notation is -7, represented by `1111`, and the largest is 7, represented by `0111`. Now there are 15 integers between -7 and 7 inclusive, but $2^4 = 16$ different 4-bit patterns! What is going on? (This is one of the drawbacks of sign-magnitude representation.)

### 7.8.4 Euclid's algorithm

18. Use Euclid's Algorithm to find $d = \gcd(28, 147)$ as well as a pair of integers $a, b$ such that $28a + 147b = 1$.

19. If $m$ and $n$ are relatively prime, then the Extended Euclid Algorithm finds a pair of integers $a$ and $b$ such that
$$ab + mn = 1.$$
Is this pair of integers unique?

20. If $m_1, m_2, m_3 \in \mathbf{Z}$ we define $\gcd(m_1, m_2, m_2)$ to be a nonnegative integer $d$ that divides all three $m_i$, and such that any other common divisor $d'$ of the $m_i$ also divides $d$.

*(a)* Show that
$$\gcd(m_1, m_2, m_2) = \gcd(\gcd(m_1, m_2), m_3),$$
and use this to conclude that there exist integers $a, b, c$ such that
$$am_1 + bm_2 + cm_3 = \gcd(m_1, m_2, m_3).$$

*(b)* Use part *(a)* and Euclid's algorithm to find integers $a, b, c$ such that
$$15a + 10b + 6c = 1.$$

*(c)* Explain how to get exactly one gallon of water if you have a 15-gallon container, a 10-gallon container, and a 6-gallon container.

21. Show that if $\gcd(m_1, m_2) = 1$ and $n$ is an integer with $m_1|n$ and $m_2|n$, then $m_1 m_2|n$. (Note the contrast with Problem 6.) HINT: Use the fact that there are integers $a, b$ with $am_1 + bm_2 = 1$ to obtain an expression for $n$ that is a multiple of $m_1 m_2$.

22. The purpose of this exercise is to show that the Fibonacci numbers form a worst case for Euclid's algorithm, and in so doing to get a sharper version of the logarithmic bound on the number of divisions performed by the algorithm. Recall the definition of the Fibonacci numbers given in Chapter 6: $F_0 = F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for $k > 1$.

    *(a)* Let $d(m, n)$ denote the number of divisions performed when Euclid's algorithm is applied to two integers $m < n$. So for example, if $m|n$, then $d(m, n) = 1$, because we get a remainder of 0 after one division. Prove that if $k \geq 1$, then $d(F_k, F_{k+1}) = 1$.

    *(b)* Let $k \geq 1$. Show that if $0 < m < n < F_{k+1}$, then $d(m, n) < k$. (HINT: Show that either $m < F_k$, or that the remainder after the first step is less than $F_{k-1}$, then use mathematical induction.)

    *(c)* Use the result of part *(b)* and the estimate from Section 6.5.1 of the size of $F_k$ to conclude that if $0 < m < n$, then

    $$d(m, n) < \log_{1.6} n \approx 1.47 \log_2 n.$$

23. Write a computer program that generates a pair $(n, n')$ of random integers of several thousand digits and applies the extended Euclid algorithm to find their greatest common divisor $d$ as well as a pair $(a, b)$ of integers such that $an + bn' = d$. (If you are working in Python, which has built-in support for arbitrary-precision integer arithmetic, you can simply use the code given in the text along with the `randint` function of the `random` package. If you work in a different programming language, you will have to investigate what is available for doing arithmetic with large integers.) This is meant to convince you that these algorithms really are fast.

### 7.8.5 Primes

24. Is 113 prime? Find the answer by hand, using as little computation as possible.

25. Find the prime factorization of 438.

26. We can tell at a glance whether two integers $m$ and $n$ are relatively prime, and more generally, determine $\gcd(m, n)$, if we have prime factorizations of $m$ and $n$. Explain carefully how this is done and why it works. In particular, why is unique factorization crucial here? (If you don't see it, find the prime factorizations of 300, 280 and of their gcd, and you will probably get the idea.) Why is it always suggested that we use Euclid's algorithm to compute the gcd rather than this method?

27. Redo Exercise 21 using unique factorization to obtain the result. This makes the problem much simpler.

28. Earlier, we proved that the square root of 2 is irrational by using properties of odd and even integers along with a form of mathematical induction. Here we revisit the irrationality of square roots (and cube roots, and fourth roots, ...) using unique factorization.

*(a)* Let $m > 1$. How many factors are in the prime factorization of $m^2$? It depends on $m$, of course, but we can say *something* in general about the number of factors, such as whether this number is even or odd. What about the prime factorization of $2m^2$?

*(b)* Conclude from the above that the equation $2m^2 = n^2$ is impossible for nonzero integers $m$ and $n$, and thus that $\sqrt{2}$ is irrational.

*(c)* Generalize. Show that the same argument proves that $\sqrt{p}$ is irrational when $p$ is prime.

*(d)* Generalize further! We can elaborate part *(a)* a bit: We not only know something about the number of factors in the prime factorization of $m^2$, but the number of times each prime factor appears. Use this to show that if $n$ is not a perfect square, then $\sqrt{n}$ is irrational.

*(e)* And further still! Adapt this argument to show that $\sqrt[k]{n}$ is either an integer or an irrational number.

### 7.8.6 Congruences

29. We observed in the text that for each $d > 0$, the relation $x \equiv y \pmod{d}$ is an equivalence relation. What is the equivalence class of 3 if $d = 5$?

30. *(a)* Show that if $x \equiv y \pmod{mn}$ then $x \equiv y \pmod{m}$. (This is easy to deduce from the characterization of congruence given in Proposition 7.5.1.)

*(b)* Conclude from this and the casting-out-nines example in the text that for every integer $n$, $n \equiv s(n) \pmod{3}$, where $s(n)$ denotes the sum of the decimal digits of $n$.

*(c)* Use the result of (b) to find

$$((492 + 9728)^2 + 279 \cdot 89) \bmod 3$$

preferably in your head, but in any case without writing down any number larger than 2.

*(d)* Since that was *so* easy, why do you suppose casting out nines was recommended as a method for checking arithmetic, rather than 'casting out threes'?

31. Here is a test for divisibility by 11: Take the an integer $n$ and find the *alternating* sum of its digits. For instance, if you start with 45873, you would compute

$$4 - 5 + 8 - 7 + 3 = 3.$$

If the result has more than one digit, take the alternating sum again, and repeat until you get a one-digit answer. Then $n$ is divisible by 11 if and only if the result of this algorithm is 0. Explain why this algorithm works.

32. Use Fermat's Theorem to find $10^{6000} \bmod 7$.

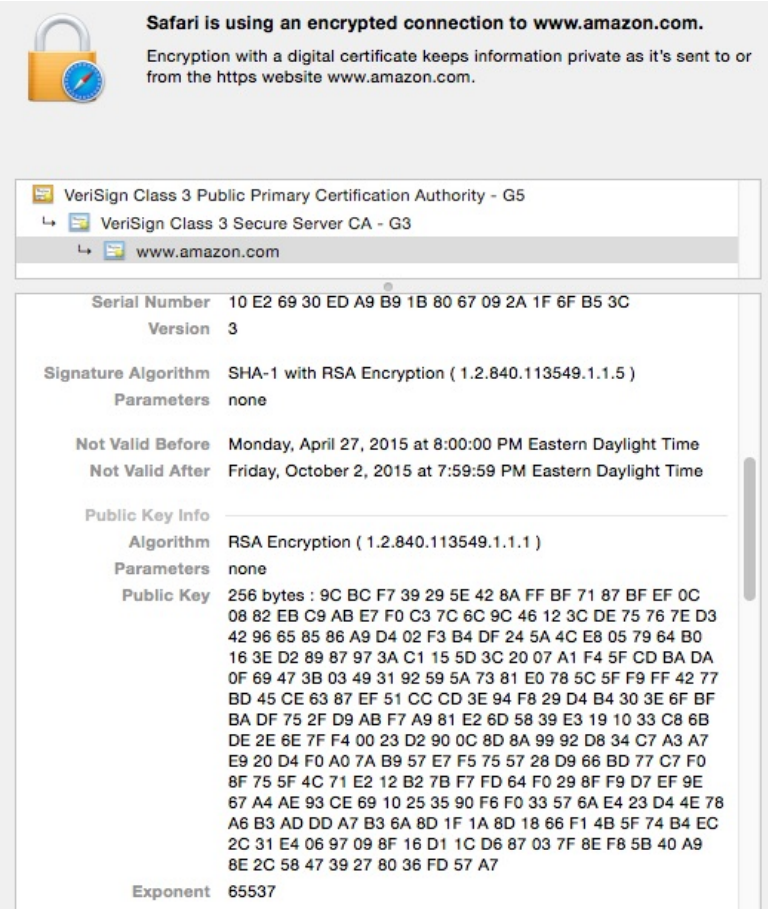



Figure 7.10: The public key of Amazon.com. The key can be obtained from the site's certificate, which can be viewed in any browser.

33. Find $10^{6000}$ mod 6 doing as little arithmetic as possible.

34. Use Fermat's Theorem to prove that 10001 is composite. (You should use a computer to do the modular exponentiation required in this problem.)

35. Our algorithms for base conversion, computation of the gcd, computation of the prime factorization, and modular exponentiation were implemented using iteration, but all of them lend themselves very naturally to recursion. Write recursive versions of these functions.

### 7.8.7 Public-key cryptography

36. Figure 7.10 shows the RSA public key of Amazon.com. The component of the key that we have been calling the modulus is here called the public key, and is given in hexadecimal. The encryption exponent 65537 is given in decimal.

*(a)* If the modulus were written *in decimal,* determine exactly how many decimal digits it would contain, and what the leading digit is.

*(b)* The value 65537 shows up frequently as an exponent in RSA public keys. This number is a prime and is equal to $2^{16} + 1$. Both these features make it particularly useful for RSA encryption. Explain why. (Think of what we have to do with this exponent, both in key generation and in encryption.)

The following exercises, in ascending order of difficulty, are meant to be worked with a computer. You will be given an RSA key and an encrypted message; your job is to decrypt it. In each case the plaintext message consists of a five-letter word. Each letter of the word will be encrypted as a pair of decimal digits, giving the letter's position in the alphabet. The result is that the plaintext is encoded as an integer with nine or ten decimal digits. For example, the plaintext LOGIC is encoded as the integer 1215070903. (In practice, RSA is never used with such short plaintexts and small moduli.)

37. You receive the encrypted message 395415567933. The modulus is 472221735719, and the secret decryption exponent is 14699465729. Find the five-letter plaintext.

38. You intercept the encrypted message 58565857117. You only have the public key information: the modulus is 150306660227 and the encryption exponent is 257. You do not have the secret decryption exponent, but the modulus is so small that it can be factored easily by trying out candidate divisors. Use this to find the decryption exponent and decrypt the message.

39. You intercept the message 27284682555982882069237 which was encrypted using a public modulus of 124137798108168664109413 and an encryption exponent 257. The modulus is now too large to be factored by testing successive candidate divisors (although it can be done using more sophisticated algorithms). However Bob, who is the intended recipient of the message, runs several online businesses. and the public key information for his other business is modulus 283967477199546905990801 and encryption exponent 257.

    You have heard a rumor that Bob has a 'lucky prime', which he uses as one of the prime factors for all the moduli he generates. Verify that this rumor is true (how do you check if the two moduli have a common prime factor?) and then use this information to decrypt the message.

# Chapter 8

# Finite State Machines

## 8.1 Finite Automata and Regular Expressions

### 8.1.1 Finding patterns in text

UNIX-based systems, including Mac OS, include a text file called `words` with a list of English words, used for purposes of spell-checking. The one on my Mac is in the directory `/usr/share/dict` and consists of close to a quarter million words, one per line. A typical extract is

```
awedness
awee
aweek
aweel
aweigh
Awellimiden
awesome
awesomely
awesomeness
awest
aweto
awfu
awful
awfully
```

It makes for awfully awesome reading!

Suppose you are playing (actually cheating at) Scrabble, and want to know whether 'prout' is a word. UNIX systems also include a tool called `grep`, which can be used to locate occurrences of patterns in text files. The version of the `grep` command that used in the examples below has the syntax

`grep -E` *pattern* `file`

If you type

```
grep -E 'prout' /usr/share/dict/words
```

you get the result:

```
asprout
nonsprouting
resprout
sprout
sproutage
sprouter
sproutful
sprouting
sproutland
sproutling
undersprout
unsprouted
unsproutful
unsprouting
uproute
upsprout
```

No 'prout', which answers the question, but you are obliged to wade through all the incorrect results in order to find that the pattern does not occur. This is because `grep` prints every line of the file in which it finds an instance of the pattern. To avoid having to look at irrelevant output, you can use special metasymbols `^` and `$` to indicate, respectively, the start and end of a line, and instead type

```
grep -E '^prout$' /usr/share/dict/words
```

The program responds by printing nothing at all, which tells us that the word is not in the dictionary.

Perhaps you wish to search for instances of that most prized item in Scrabble, words that contains a single `q` that is not immediately followed by a `u`. Here the pattern is not a specific string, but a class of strings. This requires the use of a special pattern-specification language. For a taste of how the language works, here is a partial solution to the problem:

```
grep -E '^[^q]*q[^qu][^q]*$' /usr/share/dict/words
```

Let's unpack this cryptic notation: Once again, `^` at the start of the expression, and `$` at the end, match the beginning and end of the line. A set of individual characters is indicated by enclosing the characters in square brackets, so that, for example, the pattern

```
[abcde]
```

will be matched by any of the five lower-case letters *a* through *e*. In our example, we've used a variant of this notation: if we put `^` in front of the list of letters inside the brackets, then this subexpression will be matched by any individual character *except* the ones we indicate. (Note that this is an entirely different use of the symbol `^` from the one used to indicate the start of a line.) The `*` occurring after a subexpression means 'zero or more repetitions of' the pattern described by the subexpression. So the interpretation of `[^q]*` is any, possibly empty, string of characters different from `q`. The whole expression then is, zero or more non-`q` characters, followed by a `q`, followed by a character other than `q` or `u`, followed again by zero or more non-`q` characters.

The reason that this is only a partial solution is that our pattern specification requires a character after the `q`—we have neglected words whose last letter is `q`. To capture these, we rewrite our specification as

```
grep -E '^[^q]*q([^qu][^q]*$|$)' /usr/share/dict/words
```

and obtain the more expansive list

```
Iraq
Iraqi
Iraqian
Louiqa
miqra
nastaliq
Pontacq
q
qasida
qere
qeri
qintar
qoph
Saqib
shoq
Tareq
```

[1] The new element that we have added is the symbol `|`, which means 'or'. We also introduced some parentheses to set off subexpressions. So this expression means: zero or more non-`q` characters, followed by a `q`, followed by either the end of the line, or by a character different from `q` and `u`, followed by zero or more non-`q` characters and the end of the line.

One more example: Suppose we wish, because we are fond of such word games, to search the dictionary for words that contain exactly 6 vowels (we won't count `y`) all of which are `i` or all of which are `o`. We call `grep` with the expression

```
^(([^aeiou]*[o]){6}|([^aeiou]*[i]){6})[^aeiou]*$
```

[1] Real Scrabble players will be shocked–shoqed!–at the inclusion of proper nouns, the pointless one-letter word `q`, and most of all, the absence of their best friend `qi`. What is permitted or forbidden in Scrabble is determined by the official Scrabble tournament word list, which is quite different from the list we are using here.

and get the result

```
indivisibility
odontonosology
proctocolonoscopy
```

(The `{6}`, pretty plainly, is a shortcut for saying 'exactly 6 occurrences of' whatever subexpression precedes it, and allows us to avoid having to type the subexpression six times in succession.) If, on the other hand, we relax the restriction to allow `o` and `i` in the same word, then we can use the expression

```
^([^aeiou]*[oi]){6}[^aeiou]*$
```

and get a bonanza of 72 words, including the prosaic `microbiologist` and the scary `voodooistic`, along with the even scarier `proctocolonoscopy`.

The name `grep` is an acronym for 'get regular expression and print'. The strings we use to specify patterns are called *regular expressions* and they are used in a variety of applications (among them compiler construction and natural language processing) quite apart from our frivolous word games.

### 8.1.2 Regular expressions defined

*Syntax.* Since software tools like `grep` require input that consists exclusively of standard keyboard characters, the exact form of regular expressions used in such tools represents a departure from the way regular expressions are ordinarily described in books and papers on compilation, theory of computation, and the like. Here we will adopt this more standard 'math book' notation, but we should stress that the differences are merely superficial—we really are describing the same sorts of expressions that `grep` uses, just in somewhat different notation.

In a nutshell, regular expressions are built from individual letters, using three operations, corresponding to concatenation ('followed by'), union (the `|` of `grep`) and 'zero or more occurrences of' (the `*` of `grep`).

More formally, we begin with a finite alphabet $A$ of symbols. In our examples in this section, we will have either $A = \{a, b\}$ or $A = \{a, b, c\}$. In the preceding section, $A$ was, more or less, the set of ASCII characters. Regular expressions over $A$ are defined by the grammar given by the following rules:

$$\begin{aligned} E &\rightarrow \emptyset|\epsilon|a|b|c\cdots \\ E &\rightarrow (E\cdot E)|(E+E)|E^* \end{aligned}$$

That is, the atomic regular expressions are the letters of the input alphabet $A$, along with special symbols $\emptyset$ and $\epsilon$. Then regular expressions are built recursively using the two binary operations $\cdot$ and $+$, and the unary operation $^*$. We will do some standard parenthesis de-cluttering by giving $^*$ precedence over $\cdot$, and $\cdot$ precedence over $+$. We will usually leave out the $\cdot$ and simply juxtapose two expressions, and we will further exploit the fact that (as we shall see when we get to the semantics) $+$ and $\cdot$ are both associative operations. With all this in mind, here is an example of a typical regular expression over the alphabet $\{a, b, c\}$.

$$(a+c)^*b(b+c(a+c)^*b)^*(\epsilon+c(a+c)^*).$$

*Semantics.* A regular expression describes a pattern, which is matched by any string having the pattern described. Put otherwise, a regular expression $e$ represents a set $L(e)$ of strings over the input alphabet $A$.

The set of all strings over a finite alphabet $A$ is denoted $A^*$. If, for example, $A = \{a, b\}$. then $A^*$ includes the elements $a, b, aab, bab, babbabbaaab$, and infinitely many others. A synonym for *string* is *word*, and we will use the two terms interchangeably. As always, it is indispensable to have an *empty string*, which we denote $\epsilon$. [2]

If $u$ and $v$ are words over $A$, then we denote by $uv$, or $u \cdot v$, the word formed by concatenating the two together, so for instance if

$$u = abaa, v = bab,$$

then

$$uv = abaabab, vu = bababaa.$$

Observe that this operation is not commutative, although it is associative. For any word $u$,

$$\epsilon \cdot u = u \cdot \epsilon = u.$$

Thus this operation of concatenation of words works something like multiplication, complete with an identity element. We will push the multiplication analogy even further, and write things like

$$(aba^2b)^3$$

for the word

$$abaababaababaab.$$

We denote the length of a word $w$ by $|w|$. So, for example, $|abaa| = 4$. (And what is $|\epsilon|$?)

A subset of $A^*$ is called a *language* over $A$. The reason for this terminology is that we refer to things like the set of legal propositional formulas, or legal formulas of predicate logic, or grammatical sentences of English, as languages, and each of these is a set of strings over some base alphabet of symbols. (Be careful here—the analogy is between *words* over the alphabet, and *sentences* of the language!) For our purposes, the text patterns we searched for earlier using `grep` are to be thought of as languages.

Since languages over an alphabet $A$ are subsets of $A^*$, we can perform the usual set operations of union, intersection, and complement on them.

In addition, the concatenation operation on words can be extended to sets of words, so we can define the concatenation of two languages

$$L_1L_2 = \{uv \in A^* : u \in L_1, v \in L_2\}.$$

For example, let $L_1$ be the set of words over $A = \{a, b\}$ that begin with $bb$, and let $L_2$ be the set of words that end with $bb$. Then $L_1L_2$ consists of all words that can be written as $uv$, where $u$ starts with $bb$ and $v$ ends with $bb$. This is not quite the same thing as the set of words that start and end with $bb$, because, for example, $bbb$ cannot be written as a product of an element in $L_1$ followed by an element of $L_2$. However $L_1L_2$ is the set of all words of length at least 4 that start and end

[2] The notation for the empty string is not completely standard. Some writers like $\lambda$. Others write 1, because the empty word is an identity for concatenation.

with $bb$. Observe that there may be many different ways to write a word as the concatenation of words in $L_1$ and $L_2$. For instance

$$bbababb = bb \cdot ababb = bba \cdot babb = bbab \cdot abb = bbaba \cdot bb.$$

In our example, $L_1$ itself is a concatenation of two languages:

$$L_1 = \{bb\} \cdot A^*,$$

where the first factor is the language consisting of the single word $bb$ and the second factor is the language consisting of all words over the input alphabet.

Concatenation of languages is an associative operation, so we can write $L_1L_2L_3$ without fear of ambiguity. We will usually write $L^2$ in place of $L \cdot L$, $L^3$ in place of $L \cdot L \cdot L$, *etc.*

If $L \subseteq A^*$ then $L^*$ denotes the set of all words obtained by concatenating together zero or more elements of $L$. More precisely,

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots$$

Here is an example: Let $A = \{a, b\}$ again, and let $L = \{aa, ab, ba, bb\}$, the set of words of length 2. Then $L^*$ consists of all the words that can be written as products of words of length 2—that is, all the words of even length. Note that this includes the words of length 2 themselves, as well as the empty word, which has length 0.

With that, we can finally give the recursive definition of the semantics of regular expressions. To each regular expression over a finite alphabet $e$ we associate a language $L(e) \subseteq A^*$ according to the following rules:

- $L(\emptyset) = \emptyset$.
- $L(\epsilon) = \{\epsilon\}$.
- If $a \in A$, then $L(a) = \{a\}$.
- $L(e_1 + e_2) = L(e_1) \cup L(e_2)$.
- $L(e_1 \cdot e_2) = L(e_1)L(e_2)$.
- $L(e^*) = L(e)^*$.

If $L = L(e)$ for some regular expression $e$, then we say that $L$ is a *regular language.*

*Examples.* We give a few examples, all supposing that the alphabet is $A = \{a, b, c\}$. The regular expression

$$a + b + c$$

represents the language

$$\{a\} \cup \{b\} \cup \{c\} = \{a, b, c\},$$

that is, the set of all one-letter words over $A$. (In `grep` we would write `[abc]`.) The expression

$$(a + b + c)^*$$

represents the language $\{a, b, c\}^*$, which is the set of all strings over the alphabet $A$. Let us now write a regular expression representing the set $L$ of all strings over $A$ in which the letter $b$ appears

somewhere. We will actually find three different regular expressions representing this language. The simplest idea is to recognize that every string $w \in L$ has a factorization

$$w = ubv,$$

where $u, v$ can be any words over $A$ at all, and that conversely, any string that can be factored this way contains a $b$. (You see how useful the empty string is—if we didn't have it around, then this characterization would be false, because a string like $ab$ would not hav such a factorization.) Thus $L$ is represented by the regular expression

$$(a + b + c)^* b (a + b + c)^*.$$

Observe that a word $w$ in $L$ may have many factorizations of the required kind, one for each occurrence of $b$ in $w$. Another way to approach the problem is to look at the factorization you get from the leftmost occurrence of $b$. Looked at this way, $L$ is the set of all strings of the form $ubv$, where $v$ is arbitrary, but $u$ is a string over the smaller alphabet $\{a, c\}$. We thus obtain the following equivalent regular expression:

$$(a + c)^* b (a + b + c)^*.$$

Before we give the third example, we introduce a little notational shortcut: If $e$ is a regular expression, then $e^+$ is an abbreviation for $ee^*$. Think of it as denoting 'one or more' where the $^*$ denotes 'zero or more'. $L(e^+)$ differs from $L(e^*)$ only in that the latter language always contains the empty string, whereas the former contains the empty string only if $L(e)$ itself does.

With that, suppose $w \in A^*$ is a word that contains a $b$. Instead of isolating an arbitrary $b$, or the first $b$, we pick out *all* occurrences of $b$, and factor $w$ as

$$w = u_0 b u_1 b ... u_{k-1} b u_k,$$

where each $u_i$ is a string over the alphabet $\{a, c\}$. Each piece $u_i b$ belongs to the language represented by the regular expression

$$(a + c)^* b,$$

which consists of words containing exactly one occurrence of $b$, at the end of the word. Thus the language itself is given by

$$((a + c)^* b)^+ (a + c)^*.$$

Let's redo our '`q` not followed by a `u`' example in this notation. We can write an expression for the set of all strings containing some occurrence of $b$ that is not followed by an $a$, by noting that any such string either has the form $ub$ or $ubxv$, where $u$ and $v$ are arbitrary strings and $x$ is a letter different from $a$. This gives the regular expression

$$(a + b + c)^* b + (a + b + c)^* b (b + c)(a + b + c)^*.$$

It's easy to see that a distributive law of concatenation over union holds for languages, so we can write this somewhat more compactly as

$$(a + b + c)^* b (\epsilon + (b + c)(a + b + c)^*).$$

Note that such a string can contain occurrences of $ba$, it's just that there must be *some* $b$ that is not followed by an $a$. What if we wanted to find an expression for the set of strings that contain

exactly one occurrence of $b$, and that this occurrence of $b$ is not followed by $a$? The same reasoning gives

$$(a+c)^*b(\epsilon + c(a+c)^*).$$

What if we wanted a regular expression for the language consisting of strings in which $ba$ does not occur at all? The expression above does not do this, because it rules out any string with more than one $b$. It is tempting to write something like

$$(a+b+c)^* - (a+b+c)^*ba(a+b+c)^*,$$

but complementation is not allowed in regular expressions (for a good reason, as it turns out). The problem is not so easy, and it is not clear that there even is a solution. To solve it, we need to look at a completely different way of representing regular languages.

### 8.1.3 Deterministic finite automata

Consider the four labeled directed graphs pictured in Figure 8.1. The nodes in the graph are called *states,* and for each state and each letter of $A = \{a, b, c\}$, there is exactly one arrow labeled by that letter originating in that state. Think of these as little machines—when you start the machine, it is in its *initial state*, the one with the blank incoming arrow. You then proceed to enter symbols from the alphabet $A$, and at each entry, the machine enters a new state, as indicated in the diagram. Every sequence of entries is a string over $A$, and every string over $A$ traces out a unique path from the initial state to another state (which might also be the initial state). For example, in the top left diagram, the string *acabbac* goes from state 0 to state 1, the same string goes from state 0 to state 2 in the bottom left and bottom right diagrams, and from state 0 to state 0 in the top right diagram. In all the diagrams, the empty string $\epsilon$ goes from state 0 to state 0.

When entry of the input string is completed, the machine answers 'yes' or 'no', depending on whether the state it lands in is one of the nodes indicated with a doubled circle. For example, in the top left diagram, the string *acabbac* causes the machine to answer 'yes'–we say it is *accepted* by the machine; in the other three diagrams, the same string is rejected.

Such a machine is called a *deterministic finite automaton,* or a *DFA.* [3]

Here is the formal definition of a DFA. An automaton has five components, and thus is a 5-tuple

$$\mathcal{M} = (Q, A, \delta, q_0, F).$$

Let's see what these things are:

- $Q$ is a finite set, the set of *states.*
- $A$ is the input alphabet.

[3] And, let's get this out of the way, the plural of *automaton* is *automata*, a peculiar Greek-based way of forming plurals that this word shares with *criterion/criteria* and *phenomenon/phenomena.* Irregular plurals with a foreign origin understandably cause speakers a good deal of trouble (while *mouse/mice* and *foot/feet* trouble only toddlers), and so you will hear many educated people use *criteria* and *phenomena* as singular, and even some mathematicians use *automata* as a singular. Getting it wrong this way may become so commonplace that before long it will be right.

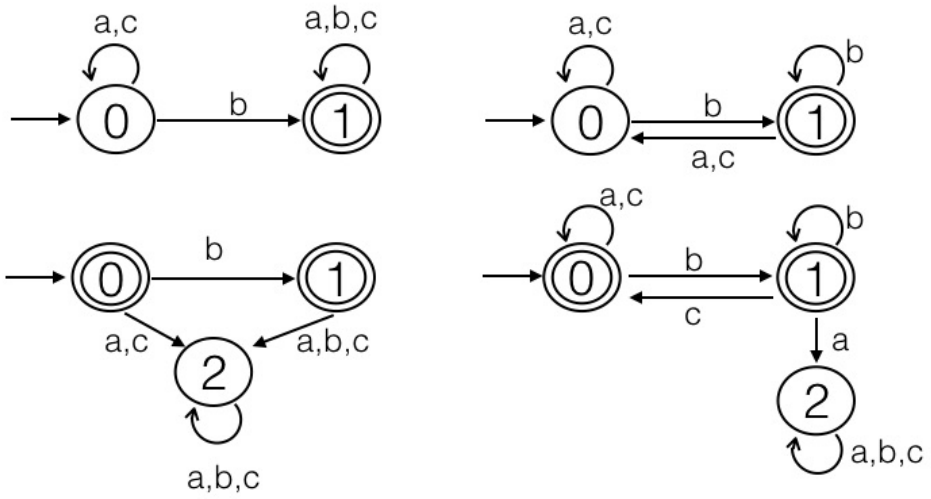



Figure 8.1: Four DFAs. The one at top left recognizes the set of strings containing an occurrence of the letter $b$. At top right, the set of strings that end in $b$. At bottom left, the set $\{b\}$, and at bottom right, the set of strings containing no occurrence of $a$ immediately after $b$.

- $\delta$ is the *state-transition function*, also called the *next-state function*. It is a function

$$\delta : Q \times A \to Q,$$

  so that given a state $q \in Q$ and an input letter $a \in A$, $\delta$ gives us the next state $q' = \delta(q, a)$.

- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *accepting states*

So, in the example at the bottom right in Figure 8.1, we have

$$Q = \{0, 1, 2\}$$

$$A = \{a, b, c\}$$

$$q_0 = 0$$

$$F = \{0, 1\}.$$

And the state-transition function $\delta$ is given in the following table:

| $q$ | $a$ | $b$ | $c$ |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 2 | 1 | 0 |
| 2 | 2 | 2 | 2 |

There is a convenient notation for the next-state function: We will write

$$q \cdot a,$$

or even

$$qa$$

in place of $\delta(q, a)$. If the machine is in state $q$, and we feed it a sequence of letters, say $abc$, then instead of writing

$$\delta(\delta(\delta(q, a), b), c)$$

for the state it winds up in after reading these letters, we can just write

$$q \cdot abc.$$

This notation is unambiguous, since it is not hard to show that

$$(q \cdot v) \cdot w = q \cdot (vw)$$

for any $q \in Q$ and $v, w \in A^*$. Observe that if $w$ is the empty word, then we just have $q \cdot w = q$.

We will say that a word $w \in A^*$ is *accepted* by the DFA $\mathcal{M}$ if, when we start the automaton in state $q_0$ and read $w$, we wind up in an accepting state in $F$. That is, $\mathcal{M}$ accepts $w$ if and only if $q \cdot w \in F$. The *language recognized* by $\mathcal{M}$ is the set of all words accepted by $\mathcal{M}$, that is,

$$\{w \in A^* : q \cdot w \in F\}.$$

We denote this language $L(\mathcal{M})$.

In the examples in the figure, the automaton at top left recognizes the set of strings that contain an occurrence of $b$. At bottom left, the only string accepted is $b$. At top right, the automaton recognizes exactly the strings that end in $b$. At bottom right, the automaton recognizes the set of strings that have no occurrence of an $a$ after a $b$. The first three languages are represented by the regular expressions

$$(a + b + c)^* b (a + b + c)^*, b, (a + b + c)^* b,$$

respectively. We earlier raised the question of whether there was a regular expression that represents the last of these languages. The answer is 'yes', and in fact we have the following important theorem:

**Theorem 8.1.1.** *Let $A$ be a finite alphabet and let $L \subseteq A^*$. $L$ is recognized by a DFA if and only if $L$ is represented by a regular expression.*

This result is often called *Kleene's Theorem*. We will outline the proof in the next two sections. The proof is quite concrete: We will exhibit two algorithms, one for finding a DFA corresponding to a given regular expression, and the other for finding a regular expression corresponding to a given automaton.

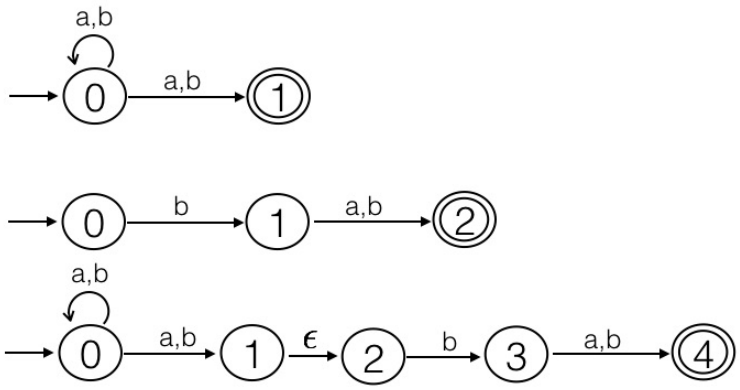



Figure 8.2: Three NFAs with $\epsilon$-transitions: The top recognizes $(a+b)^+$, the middle $b(a+b)$, and the bottom their concatenation.

### 8.1.4 From regular expressions to automata

Before describing the algorithm in general, we give a detailed example: Let us consider the set of strings represented by the regular expression

$$(a+b)^+b(a+b).$$

This consists of all strings over $\{a,b\}$ of length at least 3 whose second-to-last letter is $b$. It is pretty easy to construct a DFA that recognizes this set of strings without going through the fuss of the algorithm we are about to describe, but we want to lay out what the general procedure is. Look at Figure 8.2. This depicts three 'automata', the top one recognizing $L((a+b)^+)$, the second $L(b(a+b))$ and the third our language $L((a+b)^+b(a+b))$. There's a lot that's wrong with these pictures!

In the top figure, there are two different arrows out of state 0 labeled $a$, and two different transitions labeled $b$, and there are no arrows out of state 1. In the middle figure, there is no arrow labeled $a$ leaving state 0, and none at all leaving state 1. We've broken all of our rules, and in the bottom diagram, even more outrageously, we just glued these two 'automata' together and joined them with an '$\epsilon$-transition', allowing the machine to spontaneously jump from one state to another without consuming any input.

Yet, there's much that's right with these pictures: In each case, the set of labels of paths from the initial state to an accepting state is exactly the set of strings that we want to accept. (Keep in mind that $\epsilon$ is absorbed by any neighboring letter, so, for example, the path 'labeled' $a\epsilon ba$ starting from the initial state in the bottom diagram actually has the label $aba$.) These machines are called *nondeterministic finite automata,* or NFAs. Like the deterministic model, an NFA has a finite set $Q$ of states, an initial state $q_0$, and a set of accepting states $F \subseteq Q$. But we have dropped the requirement of a next-state function $\delta$ that gives a unique value for each pair $(q,a) \in Q \times A$, even allowing the same letter to yield two or more different next states, along with the spontaneous $\epsilon$-transitions.

There is a simple trick for putting the determinism back, called the *subset construction.* We will construct a new DFA whose states are *sets* of states of the original NFA, and that accepts the

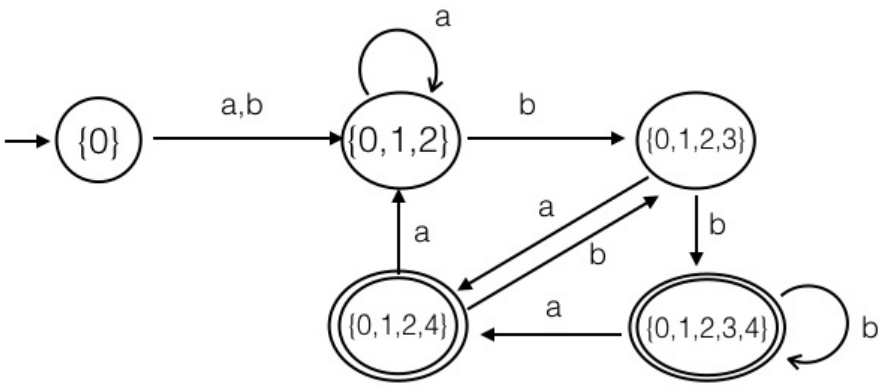



Figure 8.3: The DFA constructed with the subset construction.

same set of strings. The idea is simply that if $P$ is a set of states in the NFA, and $a \in A$ is an input letter, then we set $\delta(P, a)$ to be the set of all states of the NFA that we can reach from states in $P$ with a path labeled $a$—such a path might include numerous $\epsilon$-edges, but only one edge labeled $a$. We set the initial state of our new DFA to be $\{q_0\}$, and the set of accepting states to be all the sets $P \subseteq Q$ such that $P \cap F \neq \emptyset$.

The result is that a string $w$ in the DFA goes from $\{q_0\}$ to the *set* of all NFA states that can be reached by a path labeled $w$, and thus the DFA accepts exactly the strings that label a path from $q_0$ to $F$. We will illustrate the procedure with the example in the figure. The NFA contains 5 states, so we might need as many as 32 states in the DFA we construct. In fact, we can go about this construction in a lazy fashion, only adding new states as we need them. We will find in the end that the number of states in the DFA is quite small.

We begin with our initial state $\{0\}$ and compute

$$\delta(\{0\}, a) = \delta(\{0\}, b) = \{0, 1, 2\}.$$

That is, there are paths labeled $a$ (and also paths labeled $b$) from 0 to 0, from 0 to 1, and from 0 to 2. We then have

$$\delta(\{0, 1, 2\}, a) = \{0, 1, 2\}, \delta(\{0, 1, 2\}, b) = \{0, 1, 2, 3\}.$$

Let's pause a moment and confirm this calculation: We already know that we can get from 0 to 0,1, and 2 by following a path labeled $a$, and we can't get *anywhere* from states 1 or 2 by applying $a$, which gives the first of the two equations above. However we can get from 1 to 3 and from 2 to 3 by following a path labeled $b$, so $\delta(\{0, 1, 2\}, b)$ includes the state 3. We continue the process with our new state $\{0, 1, 2, 3\}$. In the end we will get only two more states, $\{0, 1, 2, 4\}$ and $\{0, 1, 2, 3, 4\}$. These are the accepting states of our DFA, since they contain the accepting state 4 of the NFA. You should carry out the rest of the calculation, and confirm that the resulting DFA is given by the state-transition graph depicted in Figure 8.3.

So this is the algorithm: Start from the atomic subexpressions of the given regular expression and construct automata recognizing each of them, then glue them together to build nondeterministic

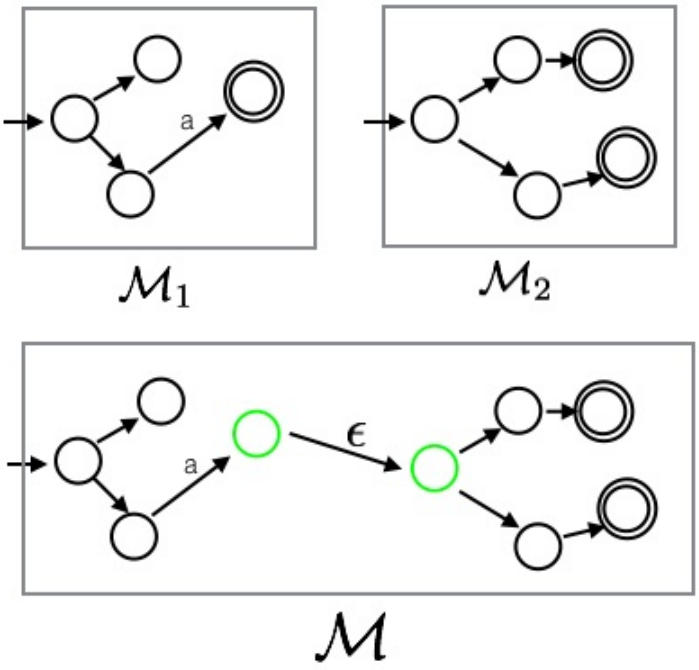



Figure 8.4: Construction of an NFA recognizing the concatenation of two languages from automata for the component languages. We add $\epsilon$-transitions from the accepting states of the first automaton to the initial state of the second.

automata that recognize the larger subexpressions, until you have an NFA that recognizes the same language as the original expression. Finally, apply the subset construction to produce an equivalent NFA. It only remains to show exactly how the gluing works.

The three figures that follow show how to accomplish this: Starting from automata, either deterministic or nondeterministic, recognizing $L_1$ and $L_2$, we construct new automata recognizing $L_1 \cup L_2$, $L_1 L_2$ and $L_1^*$. For example, in the construction for the concatenation operation, we make the accepting states of the automaton for $L_1$ nonaccepting, and add an $\epsilon$ transition from each of these to the initial state of the automaton for $L_2$ (which now is no longer initial). It is easy to see why this works: Any path from the initial state to an accepting state in the new automaton must cross one of the new $\epsilon$ transitions, so it must first pass from the initial state to an accepting state of the first automaton, and then from the initial state to an accepting state of the second automaton. The other constructions work similarly.

### 8.1.5 From automata to regular expressions

We now show the converse direction of Theorem 8.1.1: that every regular language is represented by a regular expression. In fact, we will give an algorithm for producing a regular expression representing the language recognized by any given automaton, which can be either deterministic or nondeterministic. Before entering the details, we will take a look at how the algorithm works in a

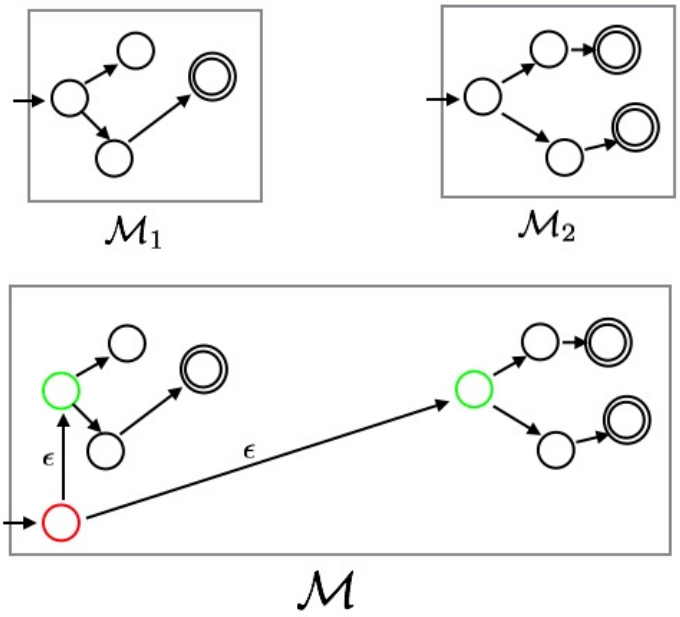



Figure 8.5: Construction of an NFA recognizing the union of two languages from automata for the component languages. The accepting states remain the same. A new initial state is added, along with $\epsilon$-transitions from this new state to the original intial states.

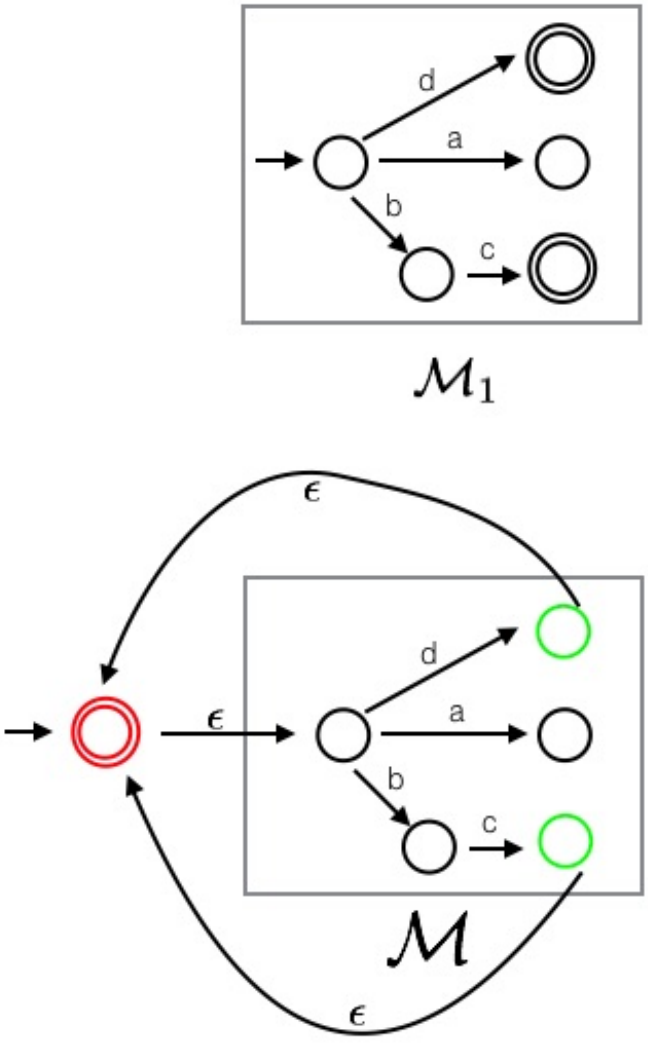



Figure 8.6: Construction of an NFA recognizing $L^*$ from an automaton recognizing $L$. A new state, which will be both the initial state and the sole accepting state, is added, along with $\epsilon$ transitions from this state to the original initial state, and from the original accepting state to the new state.

relatively simple instance. Consider the language recognized by the automaton at the bottom right of Figure 8.1. As we observed earlier, this automaton recognizes the set of all strings over $\{a, b, c\}$ in which no $b$ is followed immediately by an $a$.

Let us denote this language by $L$, and further denote by $L_0$ and $L_1$ the sets of strings that lead from the initial state 0 to the accepting states 0 and 1, respectively. Then

$$L = L_0 \cup L_1,$$

so we obtain a regular expression for $L$ by finding expressions for $L_0$ and $L_1$ and joining them with the + operator.

The crucial observation is that $L_0 = K^*$, where $K$ is the set of strings that label paths from 0 to 0, but that do not pass *through* state 0. That is, these paths begin at state 0, and, eventually, return to state 0, but once they return, they do not subsequently leave state 0 again. For example *bbc* is such a string, but *bbcc* is not, because it labels a path from 0 to 0 that passes through the state. It is easy to see that $K$ is given by the regular expression

$$a + c + bb * c$$

and thus $L_0$ is represented by

$$(a + c + bb * c)^*.$$

Finally, note that $L_1 = L_0P$, where $P$ is the set of strings that label paths from 0 to 1 that do not pass through 0. Now $P$ is given by the regular expression

$$bb^*$$

so that $L$ itself is represented by

$$(a + c + bb^*c)^* + (a + c + bb^*c)^*bb^*,$$

which we can simplify a bit to

$$(a + c + b^+c)^*b^*.$$

Once you see it, it's not too hard to understand why this expression works, although it is not the easiest thing to come up with from scratch.

The general algorithm for producing a regular expression from an automaton works along the same lines. Let $\mathcal{M}$ be an automaton, which can be either deterministic or nondeterministic, and let us number the states of $\mathcal{M}$ as $1, \ldots, n$, where 1 is the initial state. For any $1 \leq i, j \leq n$, $0 \leq k \leq n$, we define $L(i, j, k)$ to be the set of words over the input alphabet $A$ that label a path from $i$ to $j$ that passes through no state larger than $k$. For example $L(i, j, 0)$ denotes the set of words that label a path from $i$ to $j$ on which there is no intermediate state at all, so this would just include letters labeling arrows from $i$ to $j$, as well as the empty word if $i = j$. On the other hand, $L(i, j, n)$ is the set of all words labeling a path from $i$ to $j$, since no state number is higher than $n$.

We will show by induction on $k$ that *every* $L(i, j, k)$ *is represented by a regular expression,* which we will denote $r(i, j, k)$. This claim implies the result we want, since the language recognized by

$$\mathcal{M} = (\{1, \ldots, n\}, A, \delta, 1, F)$$

is

$$\bigcup_{j \in F} L(1, j, n)$$

and thus we get a regular expression for the language by forming the sum of all the $r(1, j, n)$.

If $k = 0$, we have already observed that $L(i, j, 0)$ is represented either by a sum of letters (if $i \neq j$ and there are arrows from $i$ to $j$), or by a sum of letters and $\epsilon$ if $i = j$. There is a third possibility, that $i \neq j$ and there is no arrow from $i$ to $j$, in which case $L(i, j, 0)$ is represented by the regular expression $\emptyset$.

Now suppose that $k < n$ and every $L(i, j, k)$ is represented by a regular expression $r(i, j, k)$. We claim

$$L(i, j, k+1) \quad = \quad L(i, j, k) \cup L(i, k+1, k)L(k+1, k+1, k)^* L(k+1, j, k), \tag{8.1.1}$$

and thus, by the inductive hypothesis, $L(i, j, k+1)$ is represented by the regular expression

$$r(i, j, k) + r(i, k+1, k)r(k+1, k+1, k)^* r(k+1, j, k).$$

Why does equation (8.1.1) hold? A path from state $i$ to state $j$ that passes through no state higher than $k+1$ might in fact pass through no state higher than $k$, in which case its label belongs to $L(i, j, k)$. Otherwise, the path contains an initial segment that goes from $i$ to $k+1$ and passes through no state higher than $k$ (this is the first time the path visits the state $k+1$), then 0 or more segments that go from $k+1$ to $k+1$ without passing through an intermediate state higher than $k$, up until the last time the path visits $k+1$, and finally a segment that goes from $k+1$ to $j$ without passing through a state higher than $k$ :

$$i \xrightarrow{s} k+1 \xrightarrow{t_1} k+1 \xrightarrow{t_2} \cdots \xrightarrow{t_p} k+1 \xrightarrow{u} j.$$

Thus a word in $L(i, j, k+1)$ can be factored as

$$w = st_1 \ldots t_p u,$$

where $s \in L(i, k+1, k)$, $u \in L(k+1, j, k)$, and each $t_i \in L(k+1, k+1, k)$. (Note that it is possible for $p$ to be 0; this is the case where the path passes through state $k+1$ but does not visit it a second time.) Consequently $w$ belongs to the language on the right-hand side of 8.1.1. Similarly any word $w$ in this language admits a factorization of this form, and thus labels a path from $i$ to $j$ with no intermediate state higher than $k+1$.

This proof provides an algorithm for obtaining a regular expression for a language directly from an automaton that recognizes it. The recurrence relation given by equation 8.1.1 suggests a recursive algorithm. In practice it is inefficient to implement this recursively, since doing so will generate a very large number of repeated calls with identical parameters to the function for computing $L(i, j, k)$. Instead, it is smarter to compile a table of the $L(i, j, k)$ beginning with $k = 0$, and using the recurrence to fill in cells of the table for successively larger values of $k$. (This is an instance of a technique called *dynamic programming*.)

We illustrate this procedure in detail for an automaton that recognizes the set $L$ of strings over $\{a, b\}$ that contain no occurrence of the segment *bab*. Observe that it is easy to find a regular expression for the *complement* of this language, namely $(a+b)^* bab(a+b)^*$. However, complementation is not among the operations that we use to build regular expressions, so the calculation of a regular expression for $L$ will be considerably more complicated.

We begin with a DFA recognizing $L$. This has four states, but we can quickly reduce the number to three by removing the 'reject' state at the end. The DFA and its trimmed version are shown

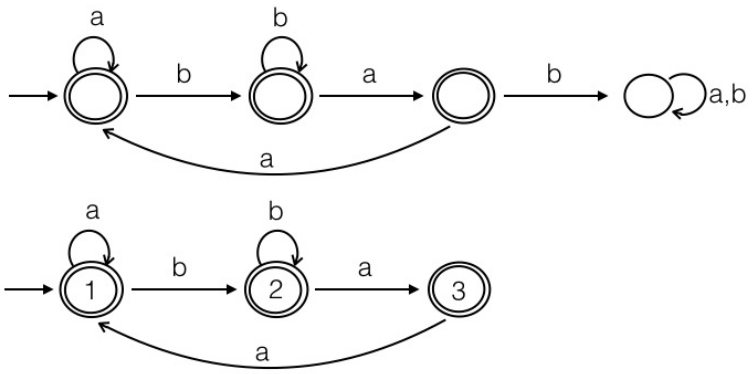



Figure 8.7: A DFA recognizing the set of words with no occurrence of *bab*, and a trimmed version that will be used to calculate a regular expression

in Figure 8.1.5. We have complete liberty as to how we number the states (we do not need, for example, to make the initial state 1). The order in which we number them affects the calculation of the regular expression, and might yield a different, although equivalent, expression from what we get with a different ordering. Here we have tried to choose the ordering to make the calculation of the regular expression as simple as possible.

In Table 8.1.5 we tabulate the values of $L(i, j, k)$ for $k = 1, 2$. Strictly speaking, we should start at $k = 0$, but it is easy to read off the row for $k = 1$ directly from the diagram. In fact, we could do this for $k = 2$, but we want to show how the recurrence relation is applied in these simple cases. What we are really after, of course, are the entries for $i = 1$ and $k = 3$. These are too big to fit in the table, so we will compute these separately in the subsequent discussion.

| $i, j$ | 1,1 | 1,2 | 1,3 | 2,1 | 2,2 | 2,3 | 3,1 | 3,2 | 3,3 |
|---|---|---|---|---|---|---|---|---|---|
| $k = 1$ | $a^*$ | $a^*b$ | $\emptyset$ | $\emptyset$ | $\epsilon + b$ | $a$ | $a^+$ | $a^+b$ | $\epsilon$ |
| $k = 2$ | $a^*$ | $a^*b^+$ | $a^*b^+a$ | $\emptyset$ | $b^*$ | $b^*a$ | $a^+$ | $a^+b^+$ | $\epsilon + a^+b^+a$ |

Table 8.1: *Table showing the regular expressions* $L(i, j, k)$ *for* $k = 1, 2$ *in our example.*

The $1, 1$ entry in row $k = 1$ gives us a regular expression for the set of words that lead from state 1 to 1, passing through no state labeled higher than 1, so this is $a^*$. Similarly, the $1, 2$ entry is $a^*b$; observe that while a word like *abb*, which does not match this pattern, does label a path from 1 to 2, this path both enters and leaves state 2, so *abb* is not in $L(1, 2, 1)$. There are also paths from 1 to 3, but all of them must go through state 2, so $L(1, 3, 1) = \emptyset$. There *is* a path from state 3 to itself that does not pass through any state higher than 1—this is the path labeled by the empty string $\epsilon$.

We can apply the recurrence formula 8.1.1 to the entries in the first row to obtain those in the

second. For example,

$$\begin{aligned} r(1,1,2) &= r(1,1,1) + r(1,2,1)(r(2,2,1))^* r(2,1,1) \\ &= a^* + a^* b(\epsilon + b)^* \emptyset \\ &= a^* + \emptyset \\ &= a^*, \end{aligned}$$

while

$$\begin{aligned} r(1,2,2) &= r(1,2,1) + r(1,2,1)(r(2,2,1))^* r(2,2,1) \\ &= a^* b + a^* b(\epsilon + b)^* (\epsilon + b)\emptyset \\ &= a^* b + a^* b b^* (\epsilon + b) \\ &= a^* b(\epsilon + b^* + b^* b) \\ &= a^* b^+. \end{aligned}$$

The other entries in the second row can be computed similarly,. Finally, our recurrence relation and the tabulated entries give:

$$\begin{aligned} L(1,1,3) &= a^* + a^* b^+ a(a^+ b^+ a)^* a^+ \\ L(1,2,3) &= a^* + a^* b^+ a(a^+ b^+ a)^* a^+ b^+ \\ L(1,2,3) &= a^* + a^* b^+ a(a^+ b^+ a)^* (a^+ b^+ a + \epsilon). \end{aligned}$$

The sum of these three is a regular expression representing $L$. We can use the distributive law and a little further simplification to get the expression

$$a^* + a^* b^+ a(a^+ b^+ a)^* (a^* + a^+ b^* a).$$

### 8.1.6 Closure under boolean operations

If you take a DFA that recognizes a language $L \subseteq A^*$, and make the accepting states non-accepting and the non-accepting states accepting—that is, if you replace

$$(Q, A, q_0, \delta, F)$$

by

$$(Q, A, q_0, \delta, Q - F),$$

you get a DFA that recognizes the complement $A^* - L$. Thus the complement of a regular language is regular. We observed earlier, with the construction used for obtaining an NFA for the union of two languages, that the class of regular languages is closed under union as well. Therefore, by DeMorgan's Law, the intersection of two regular languages is regular. It is a bit of a surprise that for any regular expression $e$ there is a regular expression representing the complement of $L(e)$—there is no simple way of transforming the regular expression to see this.

You get much more compact expressions for languages if you allow intersection and complementation. So why don't we include these operations when we work with regular expressions? (Observe that the symbol `^` used in `grep`-style regular expressions is not a general complementation operation, but only applies to individual characters.) The reason, in part, is the algorithmic

cost of constructing an automaton corresponding to a given expression. The constructions we saw for realizing the operations $+, \cdot$ and $^*$ with NFAs, were quite efficient, and did not lead to a substantial blowup in the number of states. But the trick of interchanging the accepting and nonaccepting states in order to recognize the complement of a language only works for DFAs, so to realize complementation and intersection we would first have to apply the subset construction, and this could lead to an exponential blowup in the number of states. In practice, software that synthesizes automata from regular expressions can avoid the expensive step of constructing a DFA, and instead simulate a run of the typically much smaller NFA when searching for instances of a pattern in texts.

### 8.1.7 A non-regular language

Everything that a DFA knows about the inputs it has read must be remembered in its state. We cannot require the automaton to keep track of more than a fixed amount of information about its input, because we are only allowed finitely many states. To make this vague idea precise, consider the language $L \subseteq \{a, b\}^*$ consisting of all words with an equal number of $a$'s and $b$'s. An automaton that recognizes this language will have to remember how many $a$'s and $b$'s it has seen, or at any rate the difference between the number of $a$'s and $b$'s. Imagine that this language *is* recognized by a DFA $(Q, \{a, b\}, \delta, q_0, F)$. If we have only finitely many states, then the states

$$q_0 \cdot a, q_0 \cdot a^2, q_0 \cdot a^3, \ldots$$

cannot all be different. So there must be a repeated state

$$q = q_0 \cdot a^j = q_0 \cdot a^k$$

for some $1 \leq j < k$. But then

$$q \cdot b^j = q_0 \cdot a^j b^j \in F,$$

because our automaton must accept the word $a^j b^j \in L$. On the other hand

$$q \cdot b^j = q_0 \cdot a^k b^j \notin F,$$

because it must reject the word $a^k b^j \notin L$. This contradiction shows that such a DFA could not have existed in the first place, so $L$ is not regular.

## 8.2 Sequential circuits

Watch this space.

## 8.3 Exercises

### 8.3.1 Regular expressions

1. Find regular expressions representing the following languages. Unless specified otherwise, the alphabet is $\{a, b\}$. (This can be done systematically by constructing automata and applying

the algorithm for extracting regular expressions from the automata. But it is much easier in these examples to construct the regular expressions directly.)

*(a)* $\{ab, ba\}$.

*(b)* The set of words that contain exactly one $a$.

*(c)* The set of words that contain at least one $a$.

*(d)* The set of words that contain at least two $b$'s.

*(e)* The set of words that contain at least two $b$'s and at least two $a$'s. (This is tricky– remember that you cannot use intersection.)

*(f)* The set of words that contain no occurrence of either $aa$ or $bb$. (HINT: Such words must alternate $a$'s and $b$'s.)

*(g)* The set of words over $\{a, b, c\}$ that are in correct alphabetical order (that is, $a$'s before $b$'s and $b$'s before $c$'s).

### 8.3.2 Deterministic finite automata

2. Draw state-transition diagrams of DFA's that recognize the following languages. Except where noted specifically, the input alphabet is $\{a, b\}$.

   *(a)* $\emptyset$.

   *(b)* $\{\epsilon\}$.

   *(c)* $\{bb\}$

   *(d)* $\{aa, bb\}$.

   *(e)* The set of words that contain no segment of the form $aa$ or $bb$.

   *(f)* The set of words that contain an $a$ followed somewhere later (not necessarily in the next position) by a $b$, followed still later by an $a$. For example *bbaabbabbb* satisfies this property, but *bbbbaaaab* does not.

   *(g)* The set of words in which the number of $a$'s and the number of $b$'s are congruent modulo 3. (You can do this with 9 states by keeping separate mod 3 counts of both the number of $a$'s and the number of $b$'s. But a more efficient implementation is to keep track of the value mod 3 of the *difference* between the number of $a$'s and the number of $b$'s.

   *(h)* The set of words over the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ that are decimal representations of numbers divisible by 3. For instance, the word 126 is accepted by this DFA, but 124 is not. (HINT: You can use the fact that the value represented is congruent mod 3 to the sum of its digits.)

3. Show that every finite subset $L$ of $A^*$, where $A$ is a finite alphabet, is a regular language.

4. Consider words over the alphabet consisting of letters of the form

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

where $i, j, k \in \{0, 1\}$. We can think of a word over this alphabet as an addition problem in binary. For example

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

represents the addition problem

$$\begin{array}{cccc} & 1 & 1 \\ + & 1 & 0 \\ \hline 1 & 0 & 1 \end{array}$$

which happens to be correct. Show that the set of words over this alphabet that represent correct additions is a regular language. (HINT: It is much easier to do this problem if you read the input from right to left, the way you would ordinarily add two numbers. This, in conjunction with the result from Exercise8, will give the result when the input is read from left to right.

5. Now consider words over the alphabet $\{0, 1, +, =\}$. Some of these words, for example,

$$10 + 01 = 100$$

represent the addition of two numbers in binary, although in this case the addition happens to be incorrect. A still smaller subset of this set of words represents correct additions.

*(a)* Show that the set of all words representing the addition of two numbers in binary, whether correct or not, is a regular language.

*(b)* In contrast, show that the set of all words representing correct additions is *not* a regular language. (HINT: Look at strings of the form

$$1^k + 0 = 1^{k'}.$$

These represent correct additions if and only if $k = k'$. You can argue as in the text that a DFA cannot check for this condition.)

### 8.3.3 Nondeterministic automata and operations on languages

6. Draw state transition diagrams for NFAs recognizing the languages of Exercise 1. In each case, you should take advantage of the nondeterminism–whenever possible, the total number of states and arrows in your NFA should be smaller than that for a DFA recognizing the same language.

7. Now apply the subset construction for converting each of the NFAs you found in the preceding problem into equivalent DFAs.

8. The reversal $w^{rev}$ of a word $w$ is what you think it is: for example, $(abcca)^{rev} = accba$. The reversal $L^{rev}$ of a language $L \subseteq A^*$ is just the set of reversals of elements of $L$. That is,

$$L^{rev} = \{w^{rev} : w \in L\}.$$

Show that if $L$ is a regular language, then so is $L^{rev}$. (HINT: Start with the state-transition diagram of an NFA recognizing $L$ and do something very simple to construct an NFA recognizing $L^{rev}$. Part (a) of the preceding exercise is helpful here.)

9. The construction in the textbook shows that every NFA with $n$ states is equivalent to a DFA with at most $2^n$ states. Thus the elimination of nondeterminism is achieved at the expense of an exponential blowup in the number of states. Can we do better? In this problem you will show that the answer is 'no'. More precisely, for each $n$ there is a language recognized by an $NFA$ with $n+1$ states, but by no DFA with fewer than $2^n$ states.

*(a)* Let $A = \{a, b\}$, $n \geq 1$, and consider the language $L_n = A^*aA^{n-1}$, that is, the set of all words whose $n^{th}$ letter from the left is $a$. (For example, if $n = 1$, this is just the language $A^*a$ of words whose last letter is $a$. If $n = 2$, it is the language recognized by the NFA in Figure **??**. Show that $L_n$ is recognized by an NFA with $n+1$ states.

*(b)* Now suppose $L_n$ is recognized by a DFA $\mathcal{M}$ with initial state $q_0$. Let $u, v \in A^n$ be words of length $n$ such that $u \neq v$. Prove that $q_0u \neq q_0v$. (HINT: Show that there must be a word $w$ such that $uw \notin L_n$, and $vw \in L_n$, or vice-versa.)

*(c)* Conclude that any $DFA$ recognizing $L_n$ must have at least $2^n$ states.

### 8.3.4 From automata to regular expressions

10. Try out the algorithm for extracting a regular expression from an automaton on a number of examples. You should choose examples for which it is not immediately clear how to represent the language recognized by a regular expression. Here are some suggestions:

*(a)* The language in part *(g)* of Exercise 2.

*(b)* The set of words over $\{a, b, c\}$ that do not contain two consecutive occurrences of the same letter. (That is, a $c$ is always followed by $a$ or $b$, a $b$ by $a$ or $c$, etc.

11. Give a different proof of the result in Exercise 8 using the equivalence of regular languages and regular expressions. That is, show that if $L$ is a regular language, then $L^{rev}$ is a regular language, by finding a regular expression for $L^{rev}$. (Use structural induction to show that for every regular expression there is a corresponding reversed expression.)

# Chapter 9

# Turing Machines, Computability, and Undecidability

## 9.1 The Turing Machine

We earlier saw that state machines can be viewed as concrete realizations of computer programs. The state machines that we studied communicated with a few external devices like switches and lights. But they could just as well communicate with a large external memory. For example, the traffic light controller that we saw in the preceding chapter could be redesigned to log the arrival of each pedestrian and record the amount of time that automobile traffic did not move, then periodically fetch and organize the stored data and use this information to modify the timing of the lights. Or the external memory could store two large numbers in binary and compute their sum, or product, writing the successive digits of the answer to another area of the memory. The 'state' of such a machine is not the contents of the large memory, but rather the current section of program instructions, analogous to those in Figure **??**. A section of the program to compute the sum of two integers in binary might look like the code in Figure 9.1, which instructs the machine how to compute a bit of the sum when there is no carry out of the previous bit.

Well before engineers discussed such things as 'state machines' and 'sequential circuits', the British mathematician A. M. Turing proposed just such a model of computation. In fact, Turing contended that *anything* that we might reasonably call a computation could be carried out by such a device. Moreover, he showed that it was possible to design a single device, a 'universal machine', that alone was capable of performing any computation, in essence a general-purpose stored-program computer. Turing's immediate purpose was to solve a problem in logic—whether there is an algorithm to decide if a sentence of predicate logic is logically valid or not. But his method of solution showed, among other things, that one could capture the intuitive philosophical notions of 'computation' and 'algorithm' by a precise mathematical model.

In Turing's conception, the large external memory is represented by a long tape, divided into cells, that extends infinitely in both directions. The tape holds all of the program's input, and all the output is written to this tape. Each tape cell contains a single symbol. Throughout the program's execution, only finitely many cells of the tape are in use, so all but finitely many of the cells contain a special blank symbol. The fact that the tape is infinite simply means that there is no *a priori* limit on the quantity of memory available; we can always call for more storage if we

```
if state==nocarry:
    if input1==0 and input2==0:
        write 0
        state=nocarry
    elif (input1==0 and input2==1) or (input1==1 and input2==0):
         write1
         state=nocarry
    elif (input1==1 and input2==1):
        write0
        state=carry
    advance to next bit of input1
    advance to next bit of input2
    advance to next bit of output area
```

Figure 9.1: Fragment of a state machine-based program that performs binary addition.

need it. At each step, the finite-state control reads its input from, and can write its output to, only one of the tape cells. This is illustrated in Figure 9.2, where the current cell is depicted as a read/write head.

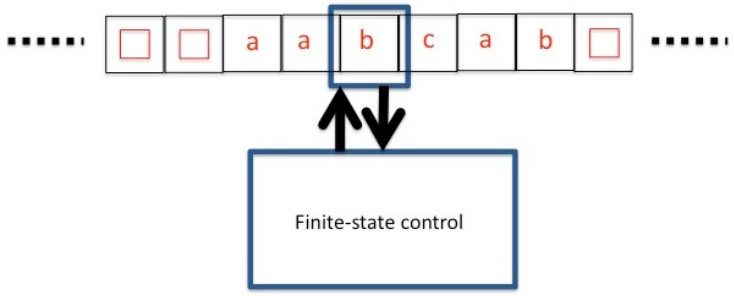



Figure 9.2: A Turing Machine. The current cell is the one containing the first $b$. Signals from the controller write a symbol to the current cell and advance the tape one cell to the left or right. Input to the controller is the symbol in the current cell.

### 9.1.1 An example

Let's give a simple example of a Turing machine in action. We will design a machine to tell us whether a word $w$ over a finite alphabet $\{a, b\}$ contains an equal number of $a$'s and $b$'s. As we saw in Section **??** this is a nonregular language, so we cannot recognize words with this property simply by making a single scan of the input and changing the state accordingly, which is all that a DFA can do.

Instead, our machine will make repeated scans of the input, checking off occurrences of $a$'s and $b$'s in pairs. At each scan it will cross off the first $a$ that it finds and the first $b$ that it finds, then return to the start of the input. If it ever completes a scan in which it is only able to cross out one letter, then the algorithm halts and rejects the input. If it completes a scan and is unable to cross out any letter, then the algorithm halts and accepts the input. For instance, suppose the word

$$aabab$$

is initially written on the tape. After the first left-to-right scan, the tape contents will be

$$cacab.$$

(The letter $c$ represents a crossed-out input letter.) After the second, it will be

$$cccac.$$

After the third scan, the tape wil be

$$ccccc,$$

but the machine will be in the state 'I have crossed out an $a$, but am looking for a $b$ to cross out'. Having found no $b$, the algorithm rejects.

In Figure 9.3 we give a precise description of the operation of this machine in the form Python-like pseudocode.

### 9.1.2 Formal definition

A *Turing Machine* (TM) is a 7-tuple

$$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where

- $Q$ is a finite set (the set of *states*),
- $\Sigma$ and $\Gamma$ are finite alphabets, with $\Sigma \subseteq \Gamma$. These are called the *input alphabet* and the *tape alphabet,* respectively. We require that $\Gamma$ contain a special *blank symbol* $\square$, where $\square \notin \Sigma$.
- $q_0$, $q_{\text{accept}}$, $q_{\text{reject}}$ are distinct elements of $Q$ (the *initial* state, *accepting* state, and *rejecting* states, respectively).
- $\delta$ is a function (the *next-state function*).

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

### 9.1.3 Operation and acceptance Behavior

At each step of execution, the *configuration* of the machine is given by three pieces of information:

- the state

- the contents of the tape: this can be given by a finite string over $\Gamma$, since the other cells of the tape are assumed to contain the blank symbol
- the location of the current cell of the tape

A configuration is also called an *instantaneous description* (ID). We can denote such a configuration as a string over $\Gamma \cup Q$, where exactly one of the letters belongs to $Q$. For example, the string

$$aa\mathbf{q}bcab$$

means that the tape contains *aabcab*, with blanks elsewhere, the current cell is the one containing the first $b$, and the state is $q$. This notation allows for configurations like

$$aab\mathbf{q},$$

in which the current cell is the one following the cell containing $b$, and thus contains a blank symbol.

The meaning of

$$\delta(p, \gamma_1) = (q, \gamma_2, R)$$

is 'in state $p$, if the current symbol is $\gamma_1$, write $\gamma_2$ in the current cell, change to state $q$, and advance the tape one cell to the right' (*i.e.,* the new current cell is the one just to the right of the original current cell). We similarly move the tape one unit to the left if the third component of $\delta(p, \gamma_1)$ is $L$. Thus given any configuration $C$ of the machine, unless the current state is $q_{\text{accept}}$ or $q_{\text{reject}}$, there is a unique next configuration $C'$ that results upon applying this next state function. We write $C \Rightarrow C'$. For example, if the current configuration is

$$C = ab\mathbf{p}ac$$

and

$$\delta(p, a) = (q, c, L),$$

then

$$C' = a\mathbf{q}bcc.$$

If the current configuration is

$$C = abac\mathbf{p},$$

and

$$\delta(p, \square) = (q, a, R),$$

then the next configuration is

$$C' = abaca\mathbf{q},$$

which we can also write as

$$C' = abaca\mathbf{q}\square,$$

If the machine is in a configuration where the state is $q_{\text{accept}}$ or $q_{\text{reject}}$, then there is no next configuration, so the machine *halts.*

Let $w \in \Sigma^*$. Suppose we start the TM $\mathcal{M}$ in configuration

$$C_0 = \mathbf{q_0}w,$$

(*i.e.,* in the initial state, with $w$ written on the tape, and all other cells blank, and with the current cell containing the first symbol of $w$) and there is some sequence of configurations

$$C_0 \Rightarrow C_1 \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_m,$$

where the state in $C_m$ is $q_{\text{accept}}$. This means that starting from the initial state with input $w$, the machine eventually halts in the accepting state. We say $\mathcal{M}$ *accepts* $w$.

We similarly say that $w$ is *rejected* by $\mathcal{M}$ if there is such a sequence with $C_m$ in the state $q_{\text{reject}}$.

The *language recognized* by $\mathcal{M}$, denoted $L(\mathcal{M})$, is the set $L \subseteq \Sigma^*$ of strings accepted by $\mathcal{M}$.

We return to the example we gave above, and see how it fits into this definition. There are six states: the initial state $q_0$, which we called `find_letter` in the pseudocode, states $q_a, q_b$ which we called `find_a` and `find_b`, a state $q_r$, which was called `return_to_start` in the example, and the states $q_{\text{reject}}, q_{\text{accept}}$. The input alphabet is $\{a, b\}$, and the tape alphabet is $\{a, b, c, \square\}$.

Just as we did for finite automata, we can depict the state-transition function by a diagram. The idea is that if $\delta(q, \gamma) = (q', \gamma', D)$, where $D \in \{L, R\}$, then the arrow from $q$ to $q'$ in the diagram will be labeled $\gamma \to \gamma', D$. We observe several conventions to simplify these diagrams: If $\gamma = \gamma'$ (so that the machine leaves the tape cell unchanged on this transition), then we leave off $\gamma'$. If $\delta(q, \gamma)$ has the same value for several different letters $\gamma$, then we include all of these letters in the label. If the transition is into an accepting state, we do not write the tape direction $D$, as this information is not relevant to the question of whether the input is accepted. We do not draw the rejecting state at all, and instead suppose that any state transition $\delta(q, \gamma)$ that is not explicitly given leads to the rejecting state. Figure 9.4 gives the complete TM recognizing the set of words with equal numbers of $a$'s and $b$'s.

Let's trace the run of the machine on a couple of different input words, the first of which is accepted, and the other rejected.
Input *abba*:

$$\begin{array}{lllllll}
\mathbf{q}_0abba & \Rightarrow & c\mathbf{q}_bbba & \Rightarrow & \mathbf{q}_rccba & \Rightarrow \\
\mathbf{q}_r\square ccba & \Rightarrow & \mathbf{q}_0ccba & \Rightarrow & c\mathbf{q}_0cba & \Rightarrow \\
cc\mathbf{q}_0ba & \Rightarrow & ccc\mathbf{q}_aa & \Rightarrow & cc\mathbf{q}_rcc & \Rightarrow \\
c\mathbf{q}_rccc & \Rightarrow & \mathbf{q}_rcccc & \Rightarrow & \mathbf{q}_r\square cccc & \Rightarrow \\
\mathbf{q}_0cccc & \Rightarrow & c\mathbf{q}_0ccc & \Rightarrow & cc\mathbf{q}_0cc & \Rightarrow \\
ccc\mathbf{q}_0c & \Rightarrow & cccc\mathbf{q}_0\square & \Rightarrow & \texttt{accept} &
\end{array}$$

Input *bab*:

$$\begin{array}{lllllll}
\mathbf{q}_0bab & \Rightarrow & c\mathbf{q}_aab & \Rightarrow & \mathbf{q}_rccb & \Rightarrow \\
\mathbf{q}_r\square ccb & \Rightarrow & \mathbf{q}_0ccb & \Rightarrow & c\mathbf{q}_0cb & \Rightarrow \\
cc\mathbf{q}_0b & \Rightarrow & ccc\mathbf{q}_a\square & \Rightarrow & \texttt{reject} &
\end{array}$$

Observe how we applied one of our conventions in the last step: Since there was no transition out of $q_a$ labeled with input $\square$ (the blank symbol), the machine switched to the reject state and halted.

### 9.1.4 Variants of the Turing machine model

**Minor Tweaks.** Different texts give slightly different definitions of 'Turing machine'. For instance, in our definition the tape is infinite in both directions, while many authors restrict to a tape that has

a leftmost cell and extends infinitely to the right. (This requires you to specify what the machine is supposed to do if it is positioned on the leftmost cell of the tape, and the state-transition function requires it to move left.) Our definition has the machine both write a tape symbol and move the read-write head to an adjacent position in each step, while other definitions require that only one or the other of these actions is performed in a single step. All of these variants are *equivalent* in the sense that any computation that you can carry out in one model can be carried out in the others. This is proved by showing how to simulate any one of these TM models by any one of the others. We will see an example of such a simulation proof shortly, when we discuss multitape TMs.

**TMs for decision problems versus TMs for input/output problems** We have defined our Turing machines to execute algorithms that read an input string and give an output of 'Yes' or 'No'. In other words, our TMs solve *decision problems*: For example, 'given a formula of propositional logic, determine whether it is satisfiable', or 'given a graph, determine whether it is connected'. But of course, other kinds of computational problems are important: 'given two integers encoded in binary, find the binary representation of their product', or 'given a graph and two vertices $v$ and $w$, find the shortest path from $v$ to $w$'. Turing machines can also solve these kinds of problems if we modify the definition slightly: In place of special accept and reject states, the machine has a single *halt* state. Execution stops when the machine enters this state. The machine begins with an input word $w$ as the tape contents, and ends with an output word $f(w)$ as the tape contents. There are other possibilities as well: in Turing's original paper, he imagined machines that started on an initially blank tape, running forever, and writing the infinite binary expansion of a real number. (For example, there is a machine for computing $\pi$.)

**Multitape Machines** As the example above showed, Turing machines have to do an awful lot of work to perform even a simple computation. A Turing machine to add two numbers in binary would have to repeatedly shuttle back and forth over its input to fetch and remember the individual bits of the summands, and to mark its place. You might start to wonder if it really is true that any computation can be carried out by a TM, because it seems, at first glance, that we have not provided them with sufficient power.

Things become considerably easier if we allow our machines to have multiple tapes. For example, if we write each of the two binary summands on separate tapes, and write the answer on a third tape, then the machine just has to advance both reading heads to the rightmost bit of each summand, and then compute the successive bits of the sum in a single scan. This is, in essence, what the pseudocode in Figure 9.1 is doing.

This represents a considerable savings in computing time over a one-tape solution. Nonetheless, *a one-tape machine can do anything that a multitape machine can.* We will outline a proof of this fact for machines that have two tapes, but the idea is the same for machines with $k$ tapes for any $k > 1$. First, let us say exactly what we mean by a two-tape TM. The definition is quite similar to that of a TM with a single tape, except that now the next-state function has the form

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \times \Gamma \to Q \times \Gamma \times \{L, R, S\} \times \Gamma \times \{L, R.S\}.$$

In words, at each step the machine changes state depending on the original state and the scanned symbols on the two tapes. It also writes symbols to the current locations on the two tapes, and moves the current position on each tape. The two tapes move independently (one might move right at the same time the other moves left, for example). Observe that we have added a third possibility for the motion of the read-write head: $S$ means 'stay put'. This makes it possible for one of the two heads to remain in place while the other moves to the left or right. The machine

is started with the input string written on the first tape, the second tape blank. Initially the first symbol of the input is the current position of the first tape, while the initial current position of the second tape is arbitrary.

Here is how we can simulate a two-tape machine with a one-tape machine: If the tape alphabet of the two-tape machine is $\Gamma$, then the alphabet of the one-tape machine is

$$\Gamma \times \{0,1\} \times \Gamma \times \{0,1\} \cup \square.$$

The idea is that each non-blank tape symbol of the one-tape machine is a *pair* of symbols of the original machine, each with an optional mark. Typically, there is no mark (indicated by 0); when a mark is present (indicated by 1), it signals the current position on the relevant tape of the two-tape machine. Figure 9.5 depicts this simulation.

The one-tape machine makes several passes over the input for each transition of the two-tape machine. On the first pass it moves right to locate the marks on each half of the tape cells, and records in its state the pair of symbols in $\Gamma$ associated with the marks, and then returns to the left-hand end of its tape. On the second pass it again moves right until it finds the mark in the first component of a tape cell, corresponding to the current position of the first tape. It modifies the contents of the cell according to the transition of the original two-tape machine, then moves right or left and marks the first component of the cell, and returns to the left-hand end of the tape. On the third pass, it does the same thing for the second component. Thus a single transition of the multi-tape machine generates a procedure consisting of about a dozen phases (move right to find the first mark, change the first component, move left, *etc.*), and each such phase gives rise to a new state of the new one-tape machine. The result is a one-tape TM that perfectly mimics the behavior of the two-tape machine. [1]

Obviously, one-tape Turing machines are rather awkward devices to deal with in practice, and you might wonder why we bother with them at all. In fact, we really *won't* use them in practice. Whenever we have to show that something can be done by a Turing machine, we will adopt a 'high-level' approach, freely availing ourselves of extra tapes and similar shortcuts and not troubling ourselves with detailed specifications. The utility of one-tape machines comes when you want to show that something *cannot* be computed by a Turing machine, in which case it helps to have as simple a model as possible. Our argument above shows that if a computation cannot be carried out by a one-tape machine, then it cannot be carried out by any fancier model like a multitape Turing machine.

## 9.2 Recursively enumerable and recursive languages

From here on out, we will treat Turing machines largely as devices for solving decision problems. Again, this means that the machine takes a word $w \in \Sigma^*$ as input, and answers either yes or no, depending on whether or not the input is accepted.

[1] It may seem as though we have somehow cheated in the way we have expanded the tape alphabet—essentially taking a pair of symbols and calling it a single symbol. In fact we could keep the tape alphabet the same size and increase the number of tape cells in the one-tape machine–For example, we can represent each aligned pair of cells in the two-tape machine by four cells on the single tape: two to hold the original tape symbols, and two more to indicate the presence or absence of a mark, The behavior of the one-tape machine becomes somewhat more complicated, because of the necessity of moving back and forth a lot within each group of four cells.

But things are not quite so simple: For Turing machines, 'rejected' does not mean the same thing as 'not accepted'. As is the case for the computer programs with which you are familiar, it is easy to design Turing machines that enter 'infinite loops' on certain inputs. So there are really three possible outcomes for a Turing machine on a given input: accept and halt, reject and halt, or run forever.

We say that a language $L \subseteq \Sigma^*$ is *Turing-recognizable,* also *recursively enumerable* if $L = L(\mathcal{M})$ for some TM $\mathcal{M}$. That is, $w \in L$ if and only if $\mathcal{M}$ accepts $w$.

We say that a language $L \subseteq \Sigma^*$ is *Turing-decidable,* and also *recursive* if there is a Turing machine $\mathcal{M}$ such that for all $w \in \Sigma^*$, $\mathcal{M}$ accepts $w$ if $w \in L$ and $\mathcal{M}$ rejects $w$ if $w \notin L$. We then say that $\mathcal{M}$ *decides* $L$.

Observe that every Turing-decidable language is Turing-recognizable—that is, decidability is a stronger condition. Is it strictly stronger? That is, are there Turing-recognizable languages that are not Turing-decidable? Or is there a way to bridge the gap between decidability and recognizability by designing machines so that they detect when they are stuck in an infinite loop, and in that instance reject? We will see the answer below.

In the meantime, let's establish some properties of decidable and recognizable languages. The arguments below illustrate the 'high-level' approach to showing that properties can be checked by Turing machines.

**Theorem 9.2.1.** *A language $L \subseteq \Sigma^*$ is Turing-decidable if and only if both $L$ and $\Sigma^* - L$ are Turing-recognizable.*

*If $L_1, L_2 \subseteq \Sigma^*$ are Turing-decidable, then so are $L_1 \cup L_2$ and $L_1 \cap L_2$.*

*If $L_1, L_2 \subseteq \Sigma^*$ are Turing-recognizable, then so are $L_1 \cup L_2$ and $L_1 \cap L_2$.*

*Proof.* For the first claim, suppose first that $L$ is Turing-decidable. Then by definition $L$ is Turing-recognizable. Take a machine $\mathcal{M}$ that decides $L$ and interchange the accepting and rejecting states. This new machine accepts exactly the elements of $\Sigma^* - L$, and thus $\Sigma^* - L$ is Turing-recognizable. (Observe that this argument only works because $\mathcal{M}$ halts on every input, either accepting it or rejecting it. Conversely, suppose both $L$ and $\Sigma^* - L$ are Turing-recognizable. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Turing machines that recognize $L$ and $\Sigma^* - L$, respectively. We can now construct a two-tape machine that simulates both $\mathcal{M}_1$ and $\mathcal{M}_2$ in parallel: At the outset, the machine copies the input on its first tape to the second tape. Then in each step, it performs a step of $\mathcal{M}_1$ on the first tape, and a step of $\mathcal{M}_2$ on the second tape (in particular, the state of this two-tape machine is a pair consisting of a state of $\mathcal{M}_1$ and a state of $\mathcal{M}_2$. If $w \in L$, then the simulation will enter an accepting state in the $\mathcal{M}_1$ component, in which case our new machine accepts. If $w \notin L$ then the simulation will enter an accepting state in the second component, in which case the new machine rejects. So our two-tape machine decides $L$, either accepting or rejecting each input. Since we can simulate the two-tape machine with a one-tape machine, $L$ is Turing-decidable.

For the second and third claims, we will just treat the case of the union operation, and leave intersection as an exercise. First let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Turing machines that decide $L_1$ and $L_2$ respectively. Again, we simulate the two machines in parallel with a two-tape machine. If the new machine enters an accepting state of either $\mathcal{M}_1$ or $\mathcal{M}_2$, it accepts. If it enters a rejecting state for one of the two machines (say $\mathcal{M}_1$) it will continue to run until it gets a result for the other machine, which will be accept or reject, and then accepts or rejects accordingly. The resulting machine thus accepts any input that is accepted by either $\mathcal{M}_1$ or $\mathcal{M}_2$, and rejects otherwise. Thus it decides

$L_1 \cup L_2$. We perform exactly the same simulation for the case where $\mathcal{M}_1$ and $\mathcal{M}_2$ merely recognize the two languages. Now the new machine may fail to halt on an input, but it any case, it accepts a word $w \in \Sigma^*$ if and only if $w \in L_1 \cup L_2$.

□

### 9.2.1 The Church-Turing Thesis

Is it true, as Turing claimed, that anything that we might reasonably call a computation (or, as we are more likely to say today, an *algorithm*) can be carried out by one of these machines? In terms of the categories of problems we gave above, this means that there is an algorithm to decide, yes or no, if a string of symbols has a particular property if and only if the set of symbols with the property is a Turing-decidable language. Turing-recognizable languages represent properties for which there are one-sided algorithms: these eventually terminate and answer 'yes' on all instance of strings that have the property in question. For strings that do not have the property, the algorithm may or may not terminate.

This claim is called the *Church-Turing thesis*. (Alonso Church formulated a different mathematical model of computation, provably equivalent to Turing's, and made the same claim for his model.) This is something that we cannot really prove or disprove, because it claims to provide a precise mathematical formulation of an intuitive philosophical concept. Turing himself advanced several arguments for the correctness of the thesis. One of these was based on a careful analysis of what the intuitive notion of computation really entails. The other was based on proving its equivalence to other models. For example, one can prove that anything computable by a program in a high-level language like Python (or C, or Java, ...) can be computed by a Turing machine. There is nothing, to our knowledge, that anyone has proposed that we would all agree to call an algorithm, but that cannot be carried out by a Turing machine.

In our subsequent arguments, we will assume that the Church-Turing thesis is correct. This means that when we want to prove that a language is Turing-decidable or Turing-recognizable, we will content ourselves with providing a description of an algorithm (or a one-sided algorithm) for determining membership in the language. We are in effect treating 'algorithm' and 'Turing machine' as synonymous. Rest assured that in all the cases that we consider, it is possible to carry out all the tedious details and provide precise descriptions of the underlying Turing machines.

As an example of this approach, we revisit Theorem 9.2.1: We will use the Church-Turing thesis to prove the first of the claims: If both $L$ and $\Sigma^* - L$ are Turing-recognizable, then there are one-sided algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ for each of these languages. Now consider the algorithm $\mathcal{A}$ that runs the two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ in parallel on the same input word $w$. More precisely, $\mathcal{A}$ consists of running a step of $\mathcal{A}_1$, a step of $\mathcal{A}_2$, then a step of $\mathcal{A}_1$, *etc.* Since any $w \in \Sigma^*$ belongs to either $L$ or to $\Sigma^* - L$, eventually one of the $\mathcal{A}_i$ will answer yes. If it is $\mathcal{A}_1$, we accept $w$, otherwise we reject. Thus the algorithm $\mathcal{A}$ decides $w$. The Church-Turing thesis implies that $L$ is Turing-decidable.

## 9.3 Universal Turing Machine

### 9.3.1 Encoding Turing machines by strings

We can encode a Turing machine by a word over a finite alphabet: The specification of the Turing machine is given by a table in which each row has the form

$$q \quad \gamma \quad q' \quad \gamma' \quad D \ ,$$

where $q, q'$ are states, $\gamma, \gamma'$ are tape symbols, and $D \in \{L, R\}$. We can make a convention that the first state entry in the first row is the initial state, and that missing transitions lead to the reject state. In case of a conflict (say, two rows that begin with the same pair of symbols $q, \gamma$) we require that the transition followed be the one specified in the earlier row. We can encode every state by a fixed-length string of bits, and reserve the bit patterns $00\cdots0$ for the rejecting state, and $11\cdots1$ for the accepting state: The number of bits we use depends on the number of states of the machine.

Similarly, we will encode each tape symbol by a fixed-length string of bits. The length $k$ of these encodings of tape symbols again depends on the size of the tape alphabet: We will suppose that our tape alphabet includes at least the five symbols $0, 1, \$, \#, \Box$ and reserve five bit patterns, say, $0^{k-3}000$, $0^{k-3}001$, $0^{k-3}010$, $0^{k-3}011$, $0^{k-3}100$ to encode these symbols.

We encode the direction $L$ or $R$ by a single bit: 0 or 1, respectively.

We can then encode a row of this table by concatenating together the encodings of all the fields of the row, using the symbol \$ to separate the fields. We can then encode the entire Turing machine $\mathcal{M}$ by concatenating all of these row encodings, again separating rows by the \$ symbol. We thus obtain a word $enc(\mathcal{M})$ over the alphabet $\{0, 1, \$\}$.

**Example.** We will calculate the encoding under this scheme of the Turing machine whose state-transition diagram appears in Figure 9.4. For the sake of economy, we will treat the input symbols $a$ and $b$ as synonymous with 0 and 1. A complete description of the machine in tabular form is given in Table 9.1

We encode the each of the six states by a 3-bit field as

$$q_{reject} \mapsto 000, q_0 \mapsto 001, q_a \mapsto 010, q_b \mapsto 011, q_r \mapsto 100, q_{accept} \mapsto 111,$$

and each of the six tape symbols (including the unused # and \$ by a 3-bit field as

$$0 \mapsto 000, 1 \mapsto 001, \$ \mapsto 010, \# \mapsto 011, \Box \mapsto 100, c \mapsto 101.$$

Thus, for example, the first row of the table is encoded by the string

001\$000\$011\$101\$1

and the final row by

100\$101\$100\$101\$0.

The entire machine is thus encoded by concatenating together all thirteen of these strings, separating them by the \$ symbol, giving $enc(\mathcal{M})$ as a string of length 233.

We presented this example so that we could provide a concrete instance of the encoding scheme, but we must stress that *the details of the encoding are entirely irrelevant.* What is important is that one should be able to take $enc(\mathcal{M})$, use it to reconstruct the transition table for $\mathcal{M}$, and then proceed to simulate $\mathcal{M}$ on any input.

| | | | | |
|---|---|---|---|---|
| $q_0$ | 0 | $q_b$ | $c$ | R |
| $q_0$ | 1 | $q_a$ | $c$ | R |
| $q_0$ | c | $q_0$ | $c$ | R |
| $q_0$ | $\square$ | $q_{accept}$ | $\square$ | R |
| $q_a$ | 0 | $q_r$ | $c$ | L |
| $q_a$ | 1 | $q_a$ | 1 | R |
| $q_a$ | c | $q_a$ | $c$ | R |
| $q_b$ | 0 | $q_b$ | 0 | R |
| $q_b$ | 1 | $q_r$ | $c$ | L |
| $q_b$ | c | $q_a$ | $c$ | R |
| $q_r$ | 0 | $q_r$ | 0 | L |
| $q_r$ | 1 | $q_r$ | 1 | L |
| $q_r$ | c | $q_r$ | $c$ | L |

Table 9.1: State-transition table for a Turing machine deciding if its input string has an equal number of 0's and 1's. Unspecified transitions go to the reject state.

### 9.3.2 Turing machines as inputs to Turing machines; a universal machine

The Church-Turing thesis implies that since you can reconstruct the table and simulate the TM $\mathcal{M}$ from the specification $enc(\mathcal{M})$, then there is a Turing machine that can do the same thing. More precisely, there is a Turing machine $\mathcal{U}$ that on input

$$enc(\mathcal{M})\#w,$$

(where $w \in \Sigma^*$ and $\#$ is another separator symbol) simulates the behavior of the Turing machine $\mathcal{M}$ on the input word $w$, accepting if $\mathcal{M}$ accepts $w$, rejecting if $\mathcal{M}$ rejects $w$, and running forever otherwise.

The TM $\mathcal{U}$ is a *universal* Turing machine, in the sense that it can simulate any Turing machine. This might seem a bit magical, but in fact the universal machine is not as unfamiliar as it may first appear. Consider that it is possible to to write an *interpreter* for Python programs. The interpreter takes a text string as input, parses it to determine if it is a syntactically correct Python program, and, if it passes the syntax check, proceeds to execute it. The interpreter itself can be written in Python, and that makes it a precise analogue of our universal Turing machine $\mathcal{U}$: A single Python program that can simulate any Python program. Of course, there is nothing special here about Python; the same thing can be carried out in any programming language. We discussed this back in 3.4.4, when we first raised the possibility of a problem that no Python program could solve. Here we are repeating the same argument in the context of the formal model of Turing machines.

Here is another way to think about $\mathcal{U}$. An ordinary Turing machine is realization of a special-purpose computer, a machine that executes only a single program. The universal Turing machine is a realization of a general-purpose computer, a machine that can execute *any* program that is presented to it as input. The key insight here is that the general-purpose computer is just another special-purpose computer, one whose special purpose is executing arbitrary programs!

## 9.4 An Undecidable Problem

The universal TM $\mathcal{U}$ recognizes the set $L(\mathcal{U})$ of all strings of the form

$$enc(\mathcal{M})\#w,$$

where $M$ accepts the word $w$. (We can include in $\mathcal{U}$ an initial scan that checks if the portion of its input preceding the # is indeed a legal encoding of a Turing machine, and rejects if it is not.) $L(\mathcal{U})$ is, of course, a Turing-recognizable language. We thus come to the crucial result:

**Theorem 9.4.1.** *$L(\mathcal{U})$ is not Turing-decidable.*

Viewed through the lens of the Church-Turing thesis, this says that there is no algorithm for determining, given a Turing machine $\mathcal{M}$ and an input string $w$, whether $\mathcal{M}$ accepts $w$. Of course, we can walk through a step-by-step simulation $\mathcal{M}$ on $w$; if we reach an accepting state, we know that $\mathcal{M}$ accepts $w$, and, if we reach a rejecting state, that $\mathcal{M}$ does not accept $w$. However, we cannot reliably determine whether we are seeing the third type of behavior, in which $\mathcal{M}$ is stuck in an infinite loop. Is the machine truly stuck in a loop, in which case $\mathcal{M}$ does not accept $w$, or is it merely taking a very long time to give an answer?

*Proof.* We will prove the theorem by contradiction. Let us suppose then, contrary to what we are trying to prove, that $L(\mathcal{U})$ is Turing-decidable, and that there is some Turing machine $\mathcal{N}$ that decides it. We now construct a new Turing machine $\mathcal{N}_1$, which behaves as follows: $\mathcal{N}_1$ first ascertains if the input has the form $enc(\mathcal{M})$ for some TM $\mathcal{M}$. If it does not, then the input is rejected. If it does, then $\mathcal{N}_1$ first duplicates its input, so that the tape, which originally contained $enc(\mathcal{M})$, now contains $enc(\mathcal{M})\#enc(\mathcal{M})$. These preliminaries concluded, $\mathcal{N}_1$ now proceeds to simulate $\mathcal{N}$ on the input. To summarize, if the input to $\mathcal{N}_1$ is not a legal encoding of a Turing machine $\mathcal{M}$, then $\mathcal{N}_1$ rejects. If the input is a legal encoding of a Turing machine $\mathcal{M}$, then $\mathcal{N}_1$ accepts if $\mathcal{M}$ accepts $enc(\mathcal{M})$, and rejects otherwise. So $\mathcal{N}_1$ decides the language

$$\{enc(\mathcal{M}) : \mathcal{M} \text{ accepts } enc(\mathcal{M})\}.$$

Now let's modify $\mathcal{N}_1$. The initial phase, where we check that the input is a legal encoding of a Turing machine, is the same. In the subsequent phase, we modify the transitions so that the new machine accepts when $\mathcal{N}_1$ rejects, and vice-versa. We'll call the resulting machine $\mathcal{N}_2$. $\mathcal{N}_2$ decides the language

$$\{enc(\mathcal{M}) : \mathcal{M} \text{ does not accept } enc(\mathcal{M})\}.$$

The TM $\mathcal{N}_2$ is like Bertrand Russell's barber, who shaves all the men in the village who do not shave themselves. Again, we ask, 'who shaves the barber'? Does $\mathcal{N}_2$ accept $enc(\mathcal{N}_2)$? If it does, then $\mathcal{N}_2$ does not accept $enc(\mathcal{N}_2)$, and if it doesn't it does. We seem to be trapped in Russell's Paradox again.

But of course there is no paradox; this is a proof by contradiction, and we have found our contradiction. Our original assumption, that $L(\mathcal{U})$ is Turing-decidable, is false.

□

## 9.5 The halting problem and other undecidable problems about computer programs

### 9.5.1 Reductions

We have at last found an undecidable computational problem, one for which no algorithm exists: We cannot determine membership in $L(\mathcal{U})$. That is, we cannot determine if a given Turing machine accepts a given input.

We now proceed to bootstrap this result to find other undecidable problems. There is essentially one technique for doing this: To prove that a language $L_1$ is undecidable, show that if it were decidable, then we would have a decision procedure for another language $L_2$ that we already know is undecidable. This is called a *reduction* of $L_2$ to $L_1$.

The first undecidable problems that we will find in this manner are minor variants of the undecidability of $L(\mathcal{U})$, all of which concern, in one way or another, the problem of predicting the long-term behavior of computer programs. But our subsequent examples appear to have nothing directly to do with computer programs.

### 9.5.2 Halting problems

As we discussed above, the crucial point in the undecidability of $L(\mathcal{U})$ is our inability to detect when a TM is caught in an infinite loop. Here we give this a more formal statement. Let $HALT_1$ denote the set of strings of the form

$$enc(M)\#w,$$

where $M$ eventually halts when started on input $w$, either accepting or rejecting.

**Theorem 9.5.1.** *$HALT_1$ is not Turing-decidable.*

*Proof.* We will reduce $L(\mathcal{U})$ to $HALT_1$. That is, we will assume that we have an algorithm $\mathcal{A}$ that decides $HALT_1$, and use it to produce an algorithm $\mathcal{A}'$ that decides membership in $L(\mathcal{U})$. The algorithm $\mathcal{A}'$ works as follows: Given an input $enc(\mathcal{M})\#w$, we first apply the algorithm $\mathcal{A}$ to the input to check whether $M$ halts on $w$, and if the answer is no, we reject the input. If it the answer is yes, we then proceed to run $\mathcal{U}$ on $enc(\mathcal{M})\#w$. We now have a guarantee that $\mathcal{U}$ will eventually halt on this input, and we accept or reject accordingly. □

We now know that we cannot test whether a TM will eventually halt on a given input. A reasonable question to ask about a computer program is whether it can *ever* enter an infinite loop; that is, whether or not every input will cause it to eventually halt. Intuition would suggest that this problem is harder than determining if the machine halts on a single fixed input, so that this version of the halting problem ought to be undecidable as well. Let us define $HALT_2$ to be the set of all strings

$$enc(\mathcal{M})$$

such that $\mathcal{M}$ eventually halts when started on any input.

**Theorem 9.5.2.** *$HALT_2$ is not Turing-decidable.*

*Proof.* We show this by reducing $HALT_1$ to $HALT_2$. By the previous theorem, $HALT_1$ is Turing-undecidable, so this will show that $HALT_2$ is undecidable.

Suppose then that we have an algorithm $\mathcal{A}$ that decides membership in $HALT_2$. We use this to give an algorithm $\mathcal{A}'$ for $HALT_1$. $\mathcal{A}'$ takes the input $enc(\mathcal{M})\#w$, and creates a new Turing machine that we will call $M_w$. $M_w$ begins by erasing its input, and writing the word $w$ in its place. (That is, the word $w$ itself is hard-wired into the specification of the machine $M_w$; we get a different machine for each $w$.) After this initial phase, $M_w$ then works identically to $M$, and mimics the run of $M$ on $w$. As a result, $M_w$ halts on every input if and only if $M$ halts on $w$. Thus if we apply the algorithm $\mathcal{A}$ to $M_w$, we decide whether $M$ halts on $w$. $\square$

Our last example of a basic undecidable question about computer programs is the very practical one of determining whether a program does what it is alleged to do. If someone proposes to you a new program for testing, say, whether a formula of propositional logic is satisfiable, or for multiplying two integers, can you prove that the programs solve these problems correctly? In specific instances you can, but we will show that there is no general procedure that will tell you whether two programs have the same behavior. Let $EQUIV$ denote the set of strings of the form

$$enc(\mathcal{M})\#enc(\mathcal{N})$$

where $\mathcal{M}, \mathcal{N}$ are TMs with the same input alphabet $\Sigma$, such that $L(\mathcal{M}) = L(\mathcal{N})$; that is, $\mathcal{M}$ and $\mathcal{N}$ accept exactly the same strings.

**Theorem 9.5.3.** *$EQUIV$ is not Turing-decidable.*

*Proof.* We will show how to reduce $L(\mathcal{U})$, our original undecidable problem, to $EQUIV$. Suppose $\mathcal{A}$ is an algorithm for deciding $EQUIV$. Here is an algorithm $\mathcal{A}'$ for deciding $L(\mathcal{U})$. Given a Turing machine $M$ and an input word $w$, we construct two new machines $\mathcal{M}_1$ and $\mathcal{M}_2$. $\mathcal{M}_1$ and $\mathcal{M}_2$ begin the same way: They check and see if the word written on the input tape is $w$, and if it is not, the input is rejected. If the input *is* $w$, $\mathcal{M}_1$ accepts, while $\mathcal{M}_2$ behaves identically to $\mathcal{M}$. Thus $\mathcal{M}_1$ and $\mathcal{M}_2$ both reject all inputs other than $w$. They accept the same set of strings if and only if $\mathcal{M}$ accepts $w$. Thus we can decide whether $\mathcal{M}$ accepts $w$ by feeding $enc(\mathcal{M}_1)\#enc(\mathcal{M}_2)$ to the algorithm $\mathcal{A}$ that decides $EQUIV$. $\square$

## 9.6 *An Undecidable Problem from Logic

Up to this point, our project of finding undecidable problems may seem like an exercise in navel-gazing: All of the problems that we have seen involve determining something about the long-term behavior of Turing machines—that is, of computer programs. Our fundamental argument boils down to: a computer program can't be designed to determine this, because it would have to be able to say something about its *own* long-term behavior...

But in fact there are many computational problems that have nothing directly to do with computer programs that turn out to be undecidable. The reason is that the problems are sufficiently complex that if we were able to answer them, we would also have the means for answering our undecidable questions about Turing machines.

The example we give is in fact the very same problem that Turing originally studied. The paper in which Turing introduced his computing machines was titled, 'On computable numbers, with an

application to the *Entscheidungsproblem*'. '*Entscheidungsproblem*' means 'decision problem'; Turing used this word because it was the name given to the problem when it was first described, in a 1928 text in German by David Hilbert and Wilhem Ackermann. We mentioned this problem earlier: Given a sentence $\psi$ of first-order predicate logic, determine whether $\psi$ is logically valid, that is, whether $\psi$ is true in every interpretation. We will prove that this is undecidable. Contrast this with the situation in propositional logic, where the method of truth tables gives an algorithm for determining if a formula is a tautology.

To prove undecidability, we will show there is a reduction of the halting problem to the *Entscheidungsproblem.* We exhibit an algorithm $\mathcal{A}$ that takes as input a Turing machine $\mathcal{M}$ and constructs a sentence $\psi_\mathcal{M}$ of predicate logic with the following properties: If there is an interpretation in which $\psi_\mathcal{M}$ is true, then $\mathcal{M}$ runs forever when started on an empty tape, and conversely, if $\mathcal{M}$ runs forever when started on an empty tape, then there is an interpretation in which $\psi_\mathcal{M}$ is true.

Put another way, $\mathcal{M}$ halts when started on the empty tape if and only if $\psi_\mathcal{M}$ is *false* in *every* interpretation; that is, if and only if $\neg\psi_\mathcal{M}$ is logically valid. So if we had an algorithm for determining logical validity, whe would have one for the undecidable problem of determining whether a given Turing machine halts on any input[2]: Use the algorithm $\mathcal{A}$ to construct $\psi_\mathcal{M}$, and then test whether $\neg\psi_\mathcal{M}$ is logically valid.

So we need to show how, given $\mathcal{M}$, we construct $\psi_\mathcal{M}$. The idea is that $\psi_\mathcal{M}$ essentially says, '$\mathcal{M}$ runs forever when started on an empty tape.' The details are as follows: $\psi_M$ will contain binary relation symbols $Su$, $Pr$, $Cur$, and $Sym_\gamma$, for each symbol $\gamma$ in the tape alphabet of $\mathcal{M}$. It also has unary relation symbols $Pos$ and $State_q$ for each non-halted state $q$ in the state set of $\mathcal{M}$. $\psi_\mathcal{M}$ itself is the conjunction of a number of sentences, the first five of which are the following:

$$\forall x \exists y (Su(x,y) \wedge x \neq y).$$

$$\forall x \forall y \forall z ((Su(x,y) \wedge Su(x,z)) \rightarrow y = z).$$

$$\forall x \exists y (Pr(x,y) \wedge x \neq y).$$

$$\forall x \forall y \forall z ((Pr(x,y) \wedge Pr(x,z)) \rightarrow y = z).$$

$$\forall x \forall y (Pr(x,y) \leftrightarrow Su(y,x)).$$

Read $Su(x,y)$ as '$y$ is a successor of $x$', and $Pr(x,y)$ as '$y$ is a predecessor of $x$'. The first sentence of says that every element of the domain has a successor that is different from itself, and the second sentence says that every element has a unique successor. The next two sentences assert the same properties for predecessor. That is, 'predecessor' and 'successor' define *functions* on the domain, and the last sentence says that they are inverse functions of one another. Let us denote by $\Phi_1$ the conjunction of these five sentences. What does a model of $\Phi_1$ look like? That is, suppose we try to draw the directed graph corresponding to the successor relation. Figure 9.6 shows several examples. The domain can be finite, with the elements arranged in a circle, or infinite, with the elements in a line, or contain several components, not connected to each other, that satisfy the same set of properties. However, the graph cannot contain a configuration like the one in the last panel of the figure, where two different elements have the same successor, since that would mean that one element has more than one predecessor.

We let $\Phi_2$ be the conjunction of sentences that say that some vertices are 'positive':

[2]See Exercise 7 for this variant of the halting problem.

$$\forall x((Pos(x) \wedge Su(x,y)) \rightarrow Pos(y)),$$

$$\exists x(Pos(x) \wedge \forall y(Pr(x,y) \rightarrow \neg Pos(y))).$$

Thus if $x$ is positive, its successor is positive, and there is some positive $x$ that is the 'first' positive value, because its predecessor is not positive. Such an $x$ cannot occur in a component of the domain graph that is finite. because the sequence of positive successors would eventually wrap around to the predecessor of $x$. Thus any interpretation in which $\Phi_1 \wedge \Phi_2$ is true must contain a component that 'looks like' the positive and negative integers. (See Figure 9.7.) From here on out we will write $y = x + 1$ and $y = x - 1$ to denote the successor and predecessor relations. We will also pick a domain element that satisfies the second sentence above and denote it by 0, and also write $Pos(x)$ as $x \geq 0$. This is just a device to simplify the notation and is not, strictly speaking, necessary. We can. if we wish, write everything in terms of the original relation symbols.

The remaining clauses that we use to construct $\psi_{\mathcal{M}}$ are derived from the description of $\mathcal{M}$. Let $Q'$ denote the set of non-halted states of $\mathcal{M}$. If $q \in Q'$, then we will write $State_q(t)$ informally to mean 'at time $t$, the machine is in state $q$'. We require that the machine be in exactly one state at any positive instant of time. Thus we include the clauses

$$\forall t(t \geq 0 \rightarrow \bigvee_{q \in Q'} State_q(t)).$$

$$\bigwedge_{q_1 \neq q_2} \forall t \neg(State_{q_1}(t) \wedge State_{q_2}(t)).$$

We write $Sym_\gamma(x,t)$ to mean that at time instant $t$, tape cell $x$ contains $\gamma$. Again, we need sentences to say that at every positive time, every tape cell contains a unique symbol, and also that at time 0, every tape cell is blank:

$$\forall x \forall t(t \geq 0 \rightarrow (\bigvee_{\gamma \in \Gamma} Sym_\gamma(x,t) \wedge \bigwedge_{\gamma \neq \gamma'} \neg(Sym_\gamma(x,t) \wedge Sym_{\gamma'}(x,t))).$$

$$\forall x(Sym_\square(x,0).$$

We will write $Cur(x,t)$ to mean that at time $t$, the current position on the tape is tape cell $x$. Our sentences say that at every positive time instant, there is exactly one current position:

$$\forall t(t \geq 0 \rightarrow \exists x Cur(x,t)).$$

$$\forall t \forall x \forall y((Cur(x,t) \wedge Cur(y,t)) \rightarrow x = y).$$

Finally, for each transition

$$\delta(q,\gamma) = (q',\gamma',D)$$

of $\mathcal{M}$, where $q, q'$ are non-halting states, $\gamma, \gamma'$ tape symbols, and $D \in \{L,R\}$ we have a the following sentence, which describes the operation of $\mathcal{M}$ from one time instance to the next:

$$\forall x \forall t((Cur(x,t) \wedge State_q(t) \wedge Sym_\gamma(x,t)) \rightarrow (Cur(x \pm 1, t+1) \wedge State_{q'}(t+1) \wedge Sym_{\gamma'}(x \pm 1, t+1))),$$

where we interpret $x \pm 1$ as $x + 1$ if $D = R$, and as $x = 1$ if $D = L$.

Our sentence $\psi_{\mathcal{M}}$ is the conjunction of $\Phi_1 \wedge \Phi_2$ with the sentences involving the predicates $Cur$, $State_q$ and $Sym_\gamma$. (The number of these sentences depends on the numbers of states and tape symbols of $\mathcal{M}$.)

If $\mathcal{M}$ runs forever when started on an empty tape, then we interpret the sentences in $\mathbf{Z}$. We interpret $Su(x, y)$ as $y = x + 1$, $Pos(x)$ as $x \geq 0$, $Cur(x, t)$ as saying that the position at time $t$ is tape cell $x$, *etc.* Since the machine does not halt, all of the clauses that we joined to make $\psi_{\mathcal{M}}$ are true in this interpretation, so $\psi_{\mathcal{M}}$ is true. Conversely, suppose we have an interpretation in which $\psi_{\mathcal{M}}$ is true. We have designed our sentences so that any interpretation in which $\psi_{\mathcal{M}}$ is true contains the integers, along with the relations $x \geq 0$, and the operations $x \mapsto x \pm 1$. The domain of the interpretation might contain other elements as well. The sentences derived from $\mathcal{M}$ tell us that when we restrict the interpretations of $Cur$, $State_q$ and $Sum_\gamma$ to those domain elements that are ordinary integers, we get an infinite sequence of configurations corresponding to the operation of the Turing machine $\mathcal{M}$ on an initially blank tape. This can only happen if $\mathcal{M}$ runs forever when started on a blank tape.

Note that the construction of $\psi_{\mathcal{M}}$ from the specification of $\mathcal{M}$ is entirely mechanical: We have given an algorithm that starts from $\mathcal{M}$ and produces a sentence that is true in some interpretation if and only if $\mathcal{M}$ runs forever when started on an empty tape, as we set out to do.

## 9.7 Historical Notes

## 9.8 Exercises

### 9.8.1 Low-level problems on Turing machines

1. Write a detailed specification of a Turing machine with input alphabet $\{a, b\}$. that erases its input, writes *aab* on the tape, and halts. (This is an input-output problem, rather than a decision problem, so there is a single halt state.) You should give the specification in the form of a state diagram. Observe that this construction was used in the reduction of $HALT_1$ to $HALT_2$ in the discussion in the text.

2. Write a detailed specification of a Turing machine with input alphabet $\{a, b\}$ that duplicates its input and then halts. In other words, if the input is originally *abbab*, then the machine halts with *abbababbab* on the tape.

3. Write a detailed specification of a Turing machine with input alphabet $\{a, b, \#\}$ that accepts inputs of the form $w\#w$, where $w \in \{a, b\}^*$. For example, the machine accepts

   $$ab\#ab$$

   but rejects

   $$ab\#aab.$$

   (If you want more of a challenge, try designing a machine that accepts exactly the inputs of the form $ww$, without the #.)

### 9.8.2 High-level problems on Turing machines

For these problems, you don't have to produce explicit Turing machine specifications: you may take for granted the Church-Turing thesis, so that a language is Turing-decidable if there is an algorithm to decide membership, and Turing-recognizable if there is a *partial algorithm* for membership—that is, an algorithm that answers 'yes' for all strings in the language, but either answers 'no' or simply fails to answer for strings not in the language.

4. Prove that the intersection of two Turing-decidable languages is Turing-decidable. (Given algorithms to decide each language, describe an algorithm to determine if a string belongs to the intersection. This is easy.)

5. Do the same for Turing-recognizable languages. This is harder, because you cannot run one of the two algorithms on the input and wait for an answer—it might not give an answer!

6. Consider Turing machines with tape alphabet $\{a, b, \#, \square\}$ that are started on an initially blank tape. Such a machine might produce a sequence like the following:

   $$ab\#abba\#babbba\#a\#\#\cdots$$

   Thus the output of this machine is a language. In the example above, the output language contains as a subset $\{\epsilon, ab, abba, babbba, a\}$. If the machine halts, the output language is finite, but typically the machine will run forever and output an infinite language.

   (a) Design such a Turing machine that outputs the language $(ab)^*$. (This is a 'low-level' problem.)

   (b) Show that a language $L$ over $\{a, b\}$ is Turing-recognizable if and only if it is the output language of such a Turing machine. (That is, there is an algorithm that enumerates the elements of $L$, which explains the terminology 'recursively enumerable'.)

   (c) Show that a language $L$ over $\{a, b\}$ is Turing-decidable if and only if it is the output language of such a Turing machine with the additional property that if $w_1, w_2 \in \{a, b\}^*$ and $w_1$ is written on the tape *before* $w_2$, then $|w_1| < |w_2|$. (That is, there is an algorithm that enumerates the elements of $L$ in order from shortest elements to the longer ones.)

### 9.8.3 Undecidable problems

7. *(a)* Prove that there is no algorithm to decide if a given TM eventually halts if it is started on a blank tape. (HINT: Reduce $HALT_1$ to this problem.)

   *(b)* Prove that there is no algorithm to decide if a given TM eventually writes the symbol `` `a' `` if started on a blank tape. (HINT: Reduce the previous problem to this one.)

   *(c)* In contrast, prove that there is an algorithm to decide if a given TM eventually writes a non-blank symbol if started on a blank tape.

```
state=find_letter
if state==find_letter:
        if current_input==a:
                write c
                move right
                state=find_b
        elif current_input==b:
                write c
                move right
                state=find_a
        elif current_input==c:
                move right
        elif current_input==blank:
                accept and halt
elif state==find_a:
        if current_input==a:
                write c
                move left
                state=return_to_start
        elif current_input in [b,c]:
                move right
        elif current_input==blank:
                reject and halt
elif state==find_b:
        if current_input==b:
                write c
                move left
                state=return_to_start
        elif current_input in [a,c]:
                move right
        elif current_input==blank:
                reject and halt
elif state==return_to_start:
        if current_input==blank:
                move right
                state=find_letter
        else:
                move left
```

Figure 9.3: A Turing machine program that determines whether its input contains equal numbers of $a$'s and $b$'s.

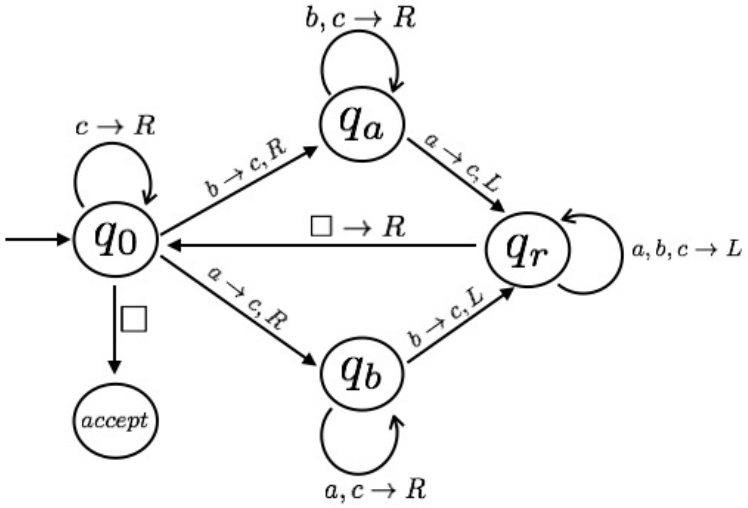



Figure 9.4: State-transition Diagram for a Turing machine that decides the set of strings with an equal number of $a$'s and $b$'s.

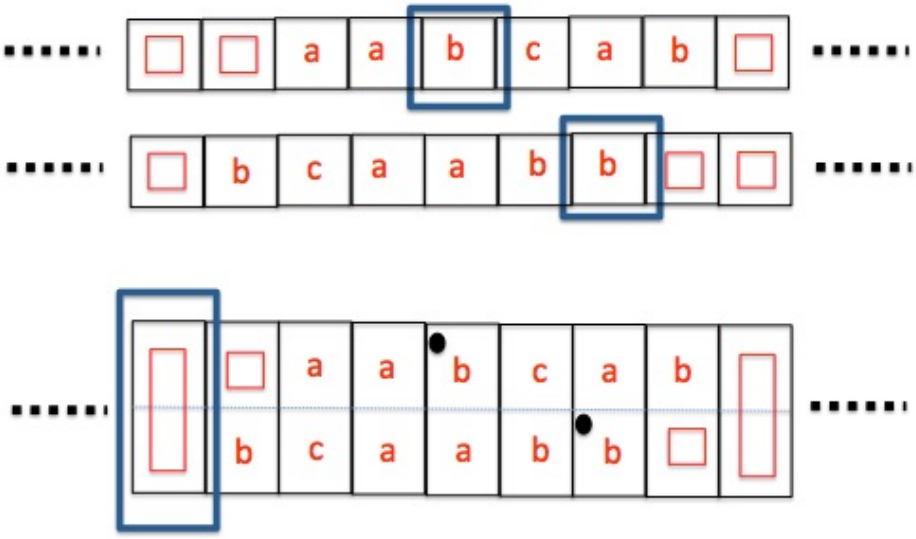



Figure 9.5: Simulation of a two-tape Turing machine by a one-tape machine. Each tape cell of the one-tape machine is labeled by a pair of tape symbols of the original two-tape machine, with marks to indicate the location of the two read-write heads.

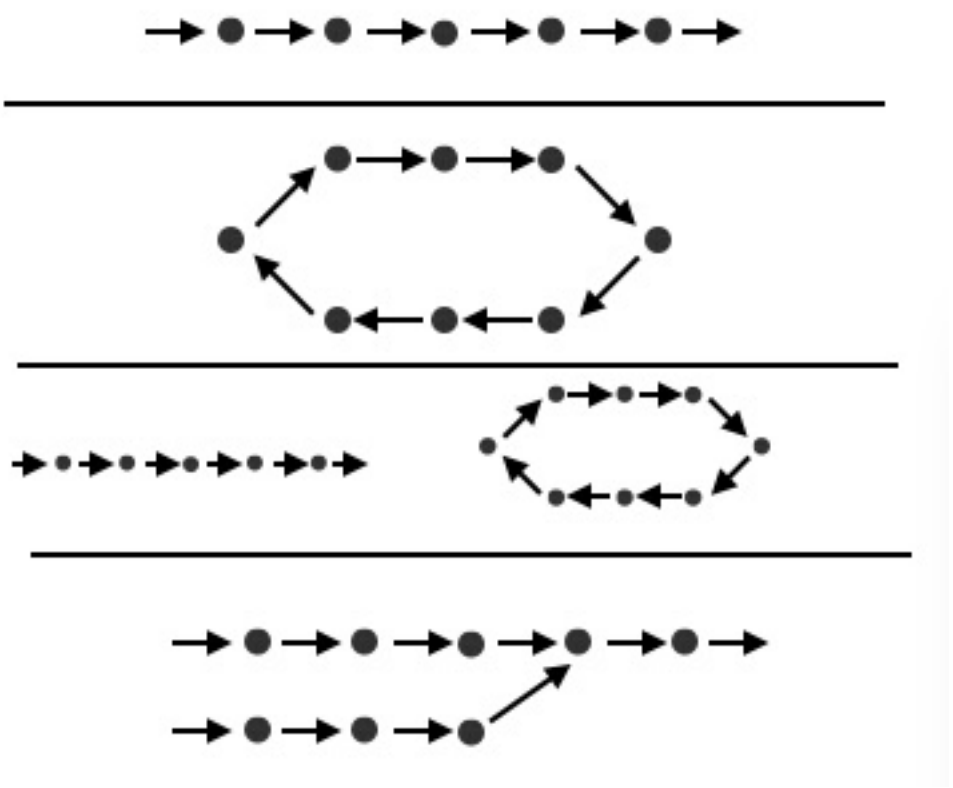


Figure 9.6: The top three panels show the $Su$ relation in three different models of $\Phi_1$, one infinite, one finite, and one containing both finite and infinite components. The bottom panel cannot be from a model of $\Phi_1$, since it violates the condition of unique successors.

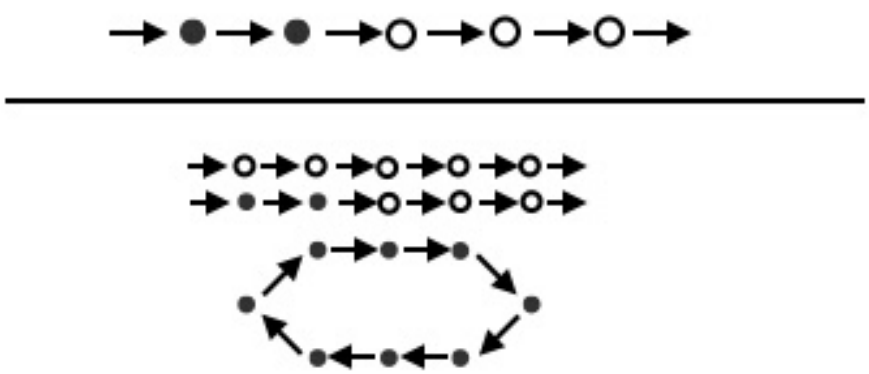


Figure 9.7: *Su* and *Pos* relation (indicated by the open circles) in two different models of $\Phi_1 \wedge \Phi_2$. Every such model must have a component that looks like the integers, with the nonnegative integers marked. This is what allows us to describe an infinite run of a Turing machine in predicate logic.

# Index


$A^*$, 219
$X - Y$ (operation on sets), 64
$X \backslash Y$, 64
$X^Y$ (operation on sets), 70
$\Pi$ (product), 146
$\Sigma$ (sum), 146
$\bar{X}$ (operation on sets), 64
$\cap$, 64
$\cup$, 64
$\emptyset$, 62
$\epsilon$, 219
$\equiv$
    congruence of integers, 193
    propositional logic, 18
$\forall$, 103
gcd, 186
$\in$, 61
$\lceil x \rceil$ (ceiling), 180
$\leftrightarrow$, 17
$\lfloor x \rfloor$ (floor), 180
$\neg$, 10
$\notin$, 61
$\oplus$, 18
$\rightarrow$
    in function notation, 70
    propositional connective, 17
$\sqrt{2}$, irrationality, 140
$\subseteq$, 62
$\subsetneq$, 62
$\times$ (operation on sets), 63
$\vee$, 10
$\wedge$, 10
$d|n$ (divides), 181
$f \circ g$ (operation on functions), 74
$f^{-1}$, 76
    extended function notation, 77
$n!$, 80
$\binom{n}{k}$, 81
$\mathbf{N}$, 62
$\mathbf{Q}$, 62
$\mathbf{R}$, 62
$\mathbf{Z}$, 62
$\mathbf{id}_X$, 76
$\mathcal{P}$, 63
$|X|$ (cardinality of set), 77

absolute value, 135
ALGOL 60, 170
algorithm
    efficient, 157
    recursive, 163, 168
Archimedean principle, 157
Aristotle, 27, 141
arithmetic mean, 133
automaton
    deterministic finite, 222
    nondeterministic, 225
axiom, 131

Babbage, Charles, 28
balanced ternary representation, 208
base (of number system), 182
    conversion, 184
base step, 151
bijection, 72
bijective function, 72
binary number system, 39
binomial coefficient, 81
Binomial Theorem, 84
bipartite directed graph, 70
bit, 39, 53
blue-eyed villagers, puzzle, 148, 150
Boole, George, 27, 93
bound variable, 103

Cantor, Georg, 90, 93

card shuffling, 198
cardinality, 77
Cartesian product, 63
    cardinality, 78
cases (in proof), 135
casting out nines, 195
Church, Alonso, 247
Church-Turing thesis, 247
closed formula, 102
CNF, *see* conjunctive normal form
Codd, E. F., 118
codomain (of function), 70
complement, 64
    absolute complement, 64
    relative complement, 64
composition of functions, 74
Condorcet (Nicolas de Caritat, marquis de Condorcet), 118
Condorcet's Paradox, 126
congruence, 193
conjunctive normal form, 25
conradiction
    in propositional logic, 16
consistent formula, 16
contrapositive, 138
converse, 139
Coq, 141
countable set, 85–88
counterexample, 137
cryptograph
    symmetric, 200
cryptography, 200
    public-key, 200

De Morgan's Law, 20
    for sets, 65
De Morgan, Augustus, 28, 170
deduction principle, 131
deterministic finite automaton, 222
DFA, 222
diagonal argument, 89
digital logic, 39
digraph, 64
directed graph, 64
    bipartite, 70
disjunctive normal form, 24
Division Algorithm, 180
DNF, *see* disjunctive normal form
domain
    of function, 70
    of interpretation, 110
domain relational calculus, 107
DPLL algorithm, 52, 54
dual of a formula, 21
duality theorem, 21

empty set, 62
*Entscheidungsproblem*, 253
equivalence
    in predicate logic, 112
    in propositional logic, 18
equivalence class, 116
equivalence relation, 115
Euclid, 141, 170
Euclid's algorithm, 186
    extended, 188
    speed, 189
Eudoxus, 141
Euler, Leonhard, 137
exclusive-or, 18
existential generalization, 132
existential instantiation, 132
existential quantifier, 104
exponential growth, 146, 155
extended Euclid's algorithm, 188
extended function notation, 77

factorial, 80
Fermat prime, 137
Fermat's Theorem, 197
Fermat, Pierre de, 137
Fibonacci (Leonardo Pisano), 170, 179
Fibonacci numbers, 164
first-order predicate logic, *see* predicate logic
Frege, Gottlob, 28, 118, 141
function, 70–77
    bijective, 72
    codomain, 70
    composition, 74
    domain, 70
    extended notation, 77
    formal definition, 70

identity, 76
injective, 72
inverse, 76
one-to-one, 72
onto, 72
partial, 70
range, 70
surjective, 72
function symbol, 113
Fundamental Theorem of Arithmetic, 190

gate, *see* logic gate
Gentzen, Gerhard, 141
geometric mean, 133
geometric series, 171
Goldbach's Conjecture, 122
grammar
for propositional logic, 12
greatest common divisor (gcd), 186

halting problem, 251
hand-waving, 23
harmonic mean, 142
harmonic series, 172
hexadecimal, 185
Hilbert, David, 141

identities
in propositional logic, 19
of sets, 65
identity function, 76
if and only if, 17, 139
inconsistent formula, 16
indirect proof, 138
induction, *see* mathematical induction
structural, 165
inductive step, 151
infinite set, 84–90
injective function, 72
interpretation, 110
intersection, 64
inverse function, 76

Jevons, William Stanley, 28

Kant, Immanuel, 27

language, 219
recursive, 246
recursively enumerable, 246
Turing-decidable, 246
Turing-recognizable, 246
least integer principle, 140, 162
Leibniz, Gottfried, 27
lexicographic order, 118
linear order, 117
LISP, 170
literal, 24
logarithmic growth, 158
logic gate, 41
logically valid sentence, 112, 253
Lucas, Édouard, 170

mathematical induction, 143–170
strong, 161–163
mean
arithmetic, 133
geometric, 133
harmonic, 142
Mirzakhani, Maryam, 127
mod, 180
modular exponentiation, 195
fast, 196
modus ponens, 131
multitape Turing machine, 244

NAND, 26
NFA, 225
nondeterministic finite automaton, 225
NOR, 38
number system
balanced ternary, 208
base -2, 208
hexadecimal, 185
positional, 182
sign-magnitude, 210
two's complement, 209

one-to-one correspondence, 72
one-to-one function, 72
onto function, 72
order
lexicographic, 118

linear, 117
partial, 117
total, 117

parse tree, 29
partial function, 70
partial order, 117
Pascal's Triangle, 83
Pascal, Blaise, 83, 93
Peano, Giuseppe, 93
Peirce, Charles Sanders, 28, 118
permutation, 80
Plato, 141
polynomial growth, 146, 155
positional number system, *see* number system, positional
power set, 63
cardinality, 78, 154
predicate logic, 101–114
bound variable, 103
closed formula, 102
constant, 102
equivalent formulas, 112
existential quantifier, 104
free variable, 103
function symbol, 113
interpretation, 110
logically valid sentence, 112, 253
open formula, 103
relation symbol, 102
semantics, 110
sentence, 102
syntax, 108
term, 108, 113
universal quantifier, 103
variable, 103
preorder, 116
prime, 114, 190
distribution of primes, 202
factorization, 190
infinitude, 192
testing primality, 197
Prime Number Theorem, 202
Principia Mathematica, 127
Principle of Mathematical Induction, 151
Principle of Strong Induction, 162
proof, 127–143
by cases, 135
by contradiction, 138
counterexample, 137
indirect, 138
reductio ad absurdum, 138
proper subset, 62
propositional logic, 9–28
applications, 39–54
consistent formula, 16
contradiction, 16
duality principle, 21
equivalent formulas, 18
grammar, 12
identities, 19
inconsistent formula, 16
parentheses in formulas, 12
parse tree, 29
satisfiable formula, 16
semantics, 13, 71
syntax, 10
tautology, 16
truth table, 15
valid formula, 16
variable, 11
puzzles
blue-eyed villagers, 148, 150
Lady or the Tiger, 55
liar, 42–47
river-crossing, 58
sudoku, 47
Pythagoreans, 141

quantifier
existential, 104
universal, 103

radix, 182
range (of function), 70
recursive
algorithm, 163, 168
definition, 12, 163
recursive language, 246
recursively enumerable language, 246
reductio ad absurdum, 138
reduction, 251

reflexive relation, 115
regular expression, 218
    definition, 218
    syntax, 218
regular expressions
    semantics, 220
relation, 115–118
    equivalence, 115
    linear order, 117
    partial order, 117
    preorder, 116
    reflexive, 115
    symmetric, 115
    total order, 117
    transitive, 115
relation symbol, 102, 108
relational algebra, 107
relational database, 118
relatively prime, 189
repeated squaring, 196
RSA Algorithm, 202
Russell's Paradox, 90, 250
Russell, Bertrand, 28, 90, 93, 127, 250

SAT solver, *see* satsifiability solver
satisfiability solver, 50
    DIMACS format for input, 50
    DPLL algorithm, 52, 54
satisfiable formula, 16
scheduling, 47
sentence, 102
set, 61–69
    cardinality, 77
    Cartesian product, 63
    complement, 64
    comprehension notation, 62
    countable, 85–88
    empty, 62
    identities, 65
    infinite, 84–90
    intersection, 64
    power set, 63
    subset, 62
    uncountable, 88
    union, 64
set comprehension notation, 62
    in Python, 65
Shannon, Claude, 53
Shestakov, V. I., 53
sign-magnitude representation, 210
Smullyan, Raymond, 43, 53
Stibitz, George, 53
Stockton, Frank, 54
string, 219
    empty, 219
structural induction, *see* induction, structural
subset, 62
    proper, 62
subset construction (to convert NFA to DFA), 225
substitution in formulas of predicate logic, 131
sudoku, 47
surjective function, 72
symmetric relation, 115

tautology, 16
term, 108
topological sort, 118
total order, 117
Towers of Hanoi, 168
transitive relation, 115
triangle inequality, 135
truth table, 15
Tukey, John, 53
Turing machine, 239–255
    definition, 241
    encoding of, 248
    multitape, 244
    universal, 249
Turing, Alan, 28
Turing, Alan M., 239
Turing-decidable language, 246
Turing-recognizable language, 246
two's complement representation, 209

uncountable set, 88
undecidable, 250
undecidable problem, 90–93
union, 64
universal generalization, 131
universal instantiation, 132
universal quantifier, 103

universal Turing machine, 249

valid formula
    in propositional logic, 16
variable
    bound, 103
    in propositional logic, 11
Venn diagram, 66
Venn, John, 93

Whitehead, A. N., 127
WLOG (without loss of generality), 136
word, 219
    empty, 219