Mathematics, Reconnected: Volume 3, Discrete and Probabilistic Mathematics

Chris Collins
Version 0.1, 2026-10-04

Table of Contents

How to Use This Book

The Knowledge Map

Part VI: Discrete Mathematics

- Chapter 21. Logic and Sets
- Chapter 22. Proof and Induction
- Chapter 23. Integers, Divisibility, and Modular Arithmetic
- Chapter 24. Counting and Combinatorics
- Chapter 25. Graph Theory

Part VIII: Probability and Statistics

- Chapter 29. Probability
- Chapter 30. Random Variables and Distributions
- Chapter 31. Statistics and Inference

Part X: Horizons

- Chapter 33. Horizons: Optimisation, Machine Learning, and Information
- Appendix A: Cheat Sheet
- Appendix B: Glossary
- Appendix C: Notation and Symbols
- Appendix D: The US Mathematics Curriculum
- Appendix E: Interactive Lab Inventory

How to Use This Book

This book is meant to take you from rusty foundational mathematics to the mathematics a strong computer scientist reads and uses every day. Calculus is one milestone on that journey, not the finish line.

Who this book is for

You probably met most of this material once. Maybe it was UK sixth-form "Maths with Mechanics" decades ago, maybe an American Algebra II class. Since then, some of it has stuck and much of it has faded. When you see

$$ax^2 + bx + c = 0$$

it looks familiar, but words like *discriminant*, *vertex* or *completing the square* may not come back straight away.

So the book assumes **prior exposure with substantial gaps and rusty recall**. It won't treat you as someone who has never seen an equation. It also won't wave a section through with "you already know this". Each topic is complete enough that a real gap won't leave you stuck, and the explanations are built so you can move quickly through anything that comes back easily.

There is a second reader in mind: a teenager in the American school system. Every major topic carries a short **US curriculum** note saying roughly where an American student would meet it. If someone tells you "he's doing Algebra II", you'll know what mathematics that covers, and the explanations here can be shared with them directly.

What this book is, and is not

This book is the **canonical knowledge path**, the core of a larger project that can grow to include experiments, Python and Mathematica notebooks, interactive visualisations, and eventually a companion application with mastery tracking.

The book deliberately avoids becoming:

- **a school textbook**, padded out to fill semesters;
- **a formula encyclopedia**, where formulas are known but not understood;
- **a pure mathematics text**, where rigour comes before intuition and application;
- **a programming book in disguise**, since code reinforces the mathematics but never replaces it;
- **a calculus cram course**, which reaches derivatives fast and leaves holes underneath;
- **an endless prerequisite trap**, since you don't need to relearn everything before reaching interesting material;
- **an exercise treadmill**, because mastery matters more than volume.

How each topic is taught

Every substantial topic follows the same arc:

> *Motivation → intuition → formalisation → mechanics.*

The topic starts with a problem worth solving. Next comes what the idea **means** in plain language, then how to state it precisely, and only after that how to calculate with it efficiently. Formulas are **derived when derivation illuminates**, and theorems are **proved when proof teaches**. Procedures are shown **step by step with nothing hidden**, then summarised as algorithms, because much of mathematics really is algorithmic and you already think in algorithms.

The book keeps returning to four representations of the same idea:

words ⟷ symbols ⟷ pictures ⟷ computation

Being able to move between them is much closer to understanding than memorising any one of them.

It also keeps asking two questions: *Where would I use this?* and *When should I use this, and when shouldn't I?* Knowing when a technique fails is part of knowing the technique.

The boxes

You will see the same kinds of box throughout the book. Here is what each one is for.



Refresher boxes

Short definitions of vocabulary and notation, written to be scanned quickly. Together they form the glossary and the cheat sheet in the appendices: the "I remember learning this, remind me what it means" reference.

Example 1. Procedure boxes

Input: a technique you have just understood.

Steps: the technique written as an explicit algorithm.

Output: something you could translate into code, and often the book does.



Watch-out boxes

Failure modes, common mistakes and limitations. These are not footnotes; they are part of understanding the idea.



Interesting boxes

Where a word came from, who solved the problem first, or an unexpected modern use. Kept short.

Interactive Lab boxes

Places where a hands-on experiment would help, such as dragging a point, changing a matrix or picking a starting value. They are collected in Appendix E as the design inventory for a future companion app.

Computer-science connection sidebars

Where the mathematics shows up in computing: algorithms, graphics, databases, machine learning, information retrieval, numerical software.

US curriculum sidebars

Roughly where an American student would meet the topic. Treat these as a cross-reference, not a rule; it varies by state, district and track.

Test out where possible; slow down where necessary

There is no big entrance exam. Instead, each chapter ends with a **mastery check**: a few problems and a **decision rule**. If they are easy, move on. If they are hard, that tells you exactly which foundation needs more time, and the decision rule points you back to the relevant section.

Each chapter also opens with a **Where this chapter sits** sidebar that lists its prerequisites. If a mastery check exposes a gap, follow those links backwards.

Understand, calculate, implement, experiment

Computation in this book follows a fixed order:

1. understand the idea;
2. calculate a few cases by hand;
3. implement it (usually in a few lines of Python);
4. experiment: break it, vary it, plot it.

Asking a computer algebra system for an answer is not understanding. Writing ten lines of Python *after* doing three iterations by hand is a good way to make the understanding stick. All code uses only Python's standard library and NumPy.

The companion code

The `companion/` folder that accompanies this book has one Python folder per chapter (`ch01_numbers/` to `ch33_horizons/`), and each one follows that same order:

- `walkthrough.py` reproduces the chapter's worked examples line by line, with the reason for each step, then implements the procedure and experiments with it, including the ways it fails;
- `exercises.py` holds functions for you to implement, and `test_exercises.py` tells you when you've got them right; `solutions.py` has reference answers with commentary;
- `lab_*.py` builds the chapter's Interactive Labs as interactive plots you can drag and adjust;
- `python3 -m mathbook.mastery chNN` runs the chapter's mastery check as a quiz and tells you which sections to revisit.

A good rhythm is to read a section, run the matching part of the walkthrough, and do the exercises before moving on. See `companion/README.md` for setup.

The three volumes

The book comes in three volumes that follow the knowledge map:

Volume	Contents	Chapters
1: Foundations to Precalculus	Numbers, algebra, geometry, functions, trigonometry, sequences, complex numbers. Everything the other two volumes build on.	1–15 (Parts I–IV)
2: Continuous Mathematics	Calculus, linear algebra, numerical mathematics.	16–20, 26–28, 32 (Parts V, VII, IX)
3: Discrete and Probabilistic Mathematics	Logic, proof, number theory, combinatorics, graphs, probability, statistics, and the Horizons capstone.	21–25, 29–31, 33 (Parts VI, VIII, X)

Volumes 2 and 3 both build on Volume 1 but not on each other, so you can read them in either order or side by side. Chapter numbers are the same in every edition, which is why Volume 2 jumps from Chapter 20 to Chapter 26: a reference such as "Chapter 17" always means the same chapter, and so does the companion folder `ch17_derivatives/`. A reference to a chapter in another volume names the volume, as in "Volume 2, Chapter 17". In the PDF it links to that chapter in the online edition.

You are reading Volume 3. Each volume has the full cheat sheet, notation guide and US-curriculum map, plus a glossary and Interactive Lab inventory covering its own chapters.

How to read the rest of the book

The chapters are numbered, but the book is not a straight line. The Knowledge Map shows how the topics depend on one another. Discrete mathematics can run alongside algebra. Probability can begin early. Linear algebra does not need calculus. Choose your route, use the mastery checks, and go back

when you need to.

Building the book

The source is AsciiDoc. From the repository root:

```shell
make            # regenerate all diagrams, then build build/book.html
make html       # just the HTML
make check      # fail on any AsciiDoc warning
```

Mathematics is typeset with LaTeX notation (`:stem: latexmath`) and rendered by MathJax in the HTML output. Diagrams are SVGs generated from the scripts in `diagrams/`, so they can be regenerated and changed.

The Knowledge Map

Mathematics isn't a ladder. It's a web of ideas, each depending on a few others. This chapter is that web: the route-planner for the rest of the book, and the skeleton of the future skill tree.

Why a map rather than a list

School mathematics in the US is usually taught as a sequence of year-long courses:

Pre-Algebra → Algebra I → Geometry → Algebra II → Precalculus → Calculus

That sequence is a scheduling decision, not a law of nature. The real constraint is **dependency**: you can't understand the derivative without limits, and you can't understand limits without functions. But you **can** learn graph theory long before calculus, and elementary linear algebra needs no calculus at all.

The map below shows each chapter and the chapters it builds on directly. An arrow from A to B means "B uses ideas from A". The dashed arrow marks a dependency that only matters for the optional parts of a chapter.

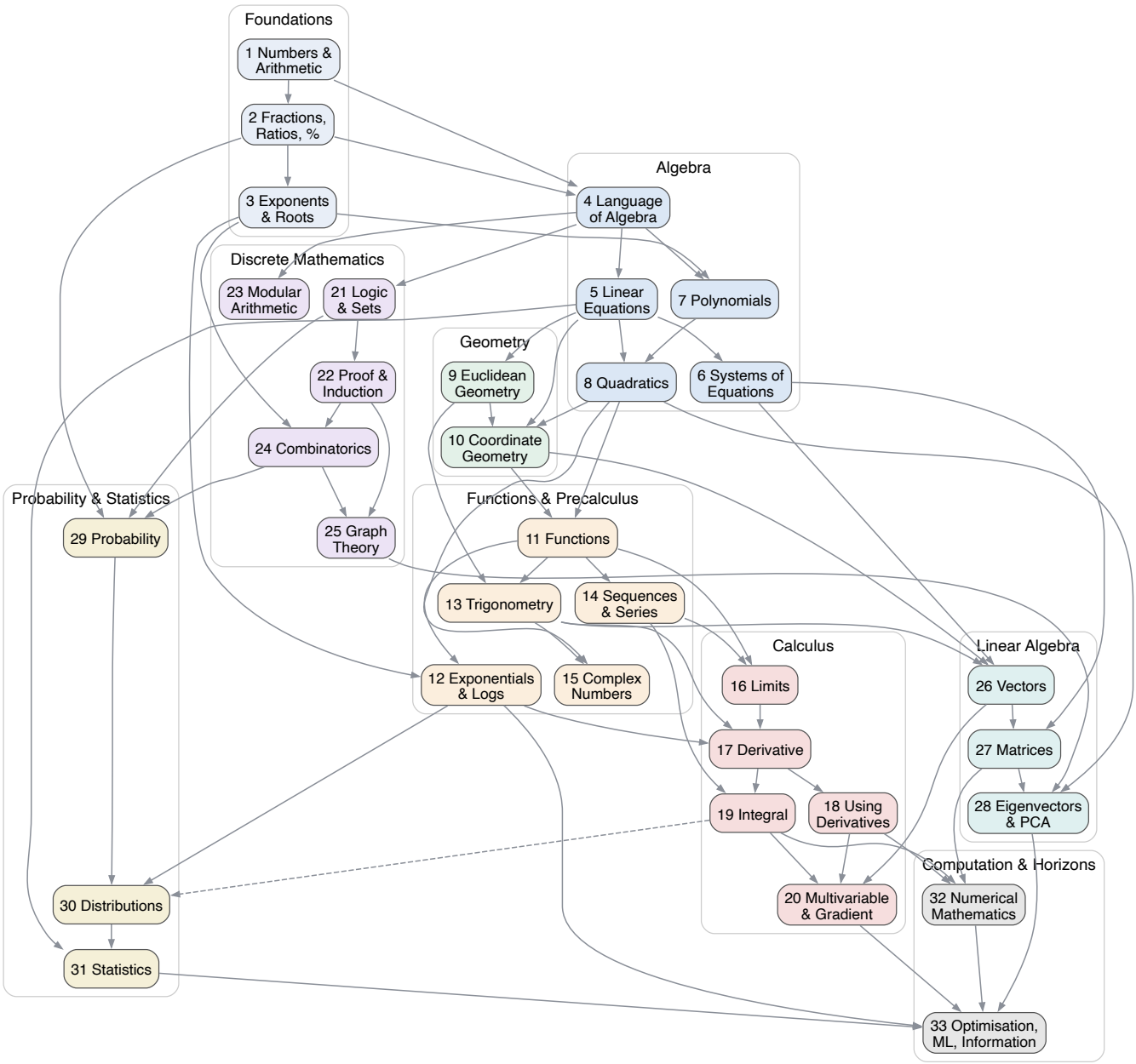


Figure 1. The knowledge map. Arrows point from prerequisite to dependent chapter.

The three volumes cut the map along its natural seams. Volume 1 is the foundation every other chapter rests on (Parts I–IV). Volume 2 holds the continuous branches: calculus, linear algebra and numerical mathematics. Volume 3 holds the discrete and probabilistic branches and ends with the Horizons capstone, which draws on both.

Computer-science connection: this map is a DAG

The map is a *directed acyclic graph*. "Find an order to study the chapters so every prerequisite comes first" is exactly **topological sorting**, the same problem a build system solves when it decides which files to compile first, or a package manager solves when it installs dependencies. You'll implement it in Chapter 25. The numbering of this book is one valid topological order, but there are many others.

The spine and the branches

The **spine** is the classical route to calculus:

Arithmetic → Algebra → Geometry → Functions → Trigonometry → Precalculus → Calculus

For a computer scientist, several **branches** deserve equal standing:

Branch	Why it matters in computing
Calculus (Volume 2, Chapter 16 –Volume 2, Chapter 20)	Continuous change, optimisation, physics simulation, the training of machine-learning models.
Linear algebra (Volume 2, Chapter 26 –Volume 2, Chapter 28)	Graphics, transformations, embeddings, search, machine learning.
Probability (Chapter 29, Chapter 30)	Uncertainty, randomised algorithms, hashing, machine learning.
Statistics (Chapter 31)	Inference, experimentation, A/B testing, metrics.
Logic and sets (Chapter 21)	Programming languages, databases, verification.
Proof (Chapter 22)	Algorithm correctness, loop invariants, reasoning about code.
Number theory (Chapter 23)	Hashing, checksums, cryptography.
Combinatorics (Chapter 24)	Algorithm analysis, probability, complexity.
Graph theory (Chapter 25)	Networks, search, compilers, dependency resolution, distributed systems.
Numerical mathematics (Volume 2, Chapter 32)	How computers actually approximate mathematical answers.

Chapter 33 brings the branches together in optimisation, machine learning and information theory.

Suggested routes

These are starting points, not prescriptions. Use the mastery checks to skip what you already know.

The calculus route

Volume 1, Chapter 1 → Volume 1, Chapter 2 → Volume 1, Chapter 3 → Volume 1, Chapter 4 → Volume 1, Chapter 5 → Volume 1, Chapter 7 → Volume 1, Chapter 8 → Volume 1, Chapter 10 → Volume 1, Chapter 11 → Volume 1, Chapter 12 → Volume 1, Chapter 13 → Volume 1, Chapter 14 → Volume 2, Chapter 16 → Volume 2, Chapter 17 → Volume 2, Chapter 18 → Volume 2, Chapter 19 .

The computer-science core

Foundations and Algebra, then Chapter 21 → Chapter 22 → Chapter 23 → Chapter 24 → Chapter 25 → Chapter 29. These can run alongside the calculus route.

The machine-learning route

The calculus route through Volume 2, Chapter 18 , plus Volume 2, Chapter 26 → Volume 2, Chapter 27 → Volume 2, Chapter 28 , then Chapter 29 → Chapter 30 → Chapter 31, then Volume 2, Chapter 20 → Chapter 33.

The "helping with homework" route

Follow the US-curriculum sidebars. Appendix D maps each school course to the chapters that cover it.

From map to skill tree

Each chapter breaks down into skills. In a future companion app, the map becomes a skill tree whose progress reflects **demonstrated understanding**, not the number of problems ground through:

```
Algebra                          Geometry
  ● Linear equations               ● Coordinates
  ● Quadratics                     ○ Vectors
  ● Functions                      ○ Transformations
  ○ Exponents
  ○ Logarithms
```

Because the tree records dependencies, it can tell you more than "wrong answer". For example: *You understand equations but seem rusty on exponent laws. Refresh those two ideas before logarithms.* That's the purpose of the "Where this chapter sits" sidebar at the top of every chapter and the decision rule at the bottom.

US curriculum

A rough correspondence, detailed in Appendix D:

- **Elementary school**: Volume 1, Chapter 1 , Volume 1, Chapter 2 , early Volume 1, Chapter 9
- **Middle school / Pre-Algebra**: Volume 1, Chapter 2 , Volume 1, Chapter 3 , Volume 1, Chapter 4
- **Algebra I**: Volume 1, Chapter 5 , Volume 1, Chapter 6 , early Volume 1, Chapter 7 , Volume 1, Chapter 8 , Volume 1, Chapter 11
- **Geometry**: Volume 1, Chapter 9 , Volume 1, Chapter 10
- **Algebra II**: Volume 1, Chapter 7 , Volume 1, Chapter 12 , Volume 1, Chapter 15 , Volume 1, Chapter 14
- **Precalculus**: Volume 1, Chapter 11 , Volume 1, Chapter 13 , Volume 1, Chapter 14 , introduction to Volume 2, Chapter 16
- **AP Calculus**: Volume 2, Chapter 16 to Volume 2, Chapter 19
- **AP Statistics**: Chapter 29, Chapter 30, Chapter 31
- **University (CS / math)**: discrete mathematics, linear algebra, multivariable calculus, numerical methods

Part VI: Discrete Mathematics

Chapter 21. Logic and Sets

Every `if` *statement you have ever written is a small piece of logic, and every database query is a small piece of set theory. This chapter makes that implicit knowledge explicit: propositions and the connectives that combine them, the slippery conditional "if … then", the laws (De Morgan's above all) that let you rewrite conditions safely, the quantifiers "for all" and "there exists", and then sets, relations and the special relations called functions. By the end you will have the vocabulary that the rest of discrete mathematics, and every proof in this book, is written in.*

Where this chapter sits

Prerequisites: comfort with numbers and basic algebra from Volume 1, Chapter 1 , Volume 1, Chapter 2 and Volume 1, Chapter 4 . Nothing from calculus, geometry or trigonometry is needed. Discrete mathematics can be studied alongside algebra.

Unlocks: Chapter 22 (proof techniques are logic in action), Chapter 23 (congruence is an equivalence relation), Chapter 24 (counting is set theory with numbers), Chapter 25 (a graph is a set with a relation on it), and Chapter 29 (events are sets). It also sharpens your reading of Volume 1, Chapter 11 , where functions are defined as special relations.

CS payoff: boolean expressions, short-circuit evaluation, refactoring conditionals, digital logic, SQL, Python sets, hashing contracts, dependency ordering and SAT solving.

Why this chapter matters. Outside code, the same few ideas decide whether a contract clause has been met, whether a positive medical test means what the patient hopes it means, which blood a patient can safely receive, and how many people on a survey ticked "neither". Logic is the grammar of precise rules; sets are the bookkeeping of who or what a rule applies to.

21.1. Propositions and connectives

21.1.1. The problem

A product manager writes in a ticket: "Show the discount banner if the user is a member or has spent over $100, but not if they're logged out." You translate that into code. A week later a bug report arrives: logged-out members are seeing the banner. Was the ticket ambiguous, or was the code wrong?

English is a wonderful language for poetry and a terrible one for specifications. "Or" sometimes means "one or the other or both" and sometimes "one or the other but not both". "But" means "and" with an attitude. "Unless" means something different to almost everyone who reads it. Logic is the discipline of removing that ambiguity: it gives each connecting word one precise meaning, and then gives rules for combining and rewriting statements without changing what they say.

21.1.2. Intuition

Think of a statement as a wire that carries either **true** or **false**. Connectives are little machines that take one or two wires in and produce one wire out. "Not" flips its wire. "And" outputs true only if both inputs are true. "Or" outputs true if at least one input is true. Because each wire has only two possible values, you can describe any machine completely by listing what it does for every combination of inputs. That list is a **truth table**, and it is the whole definition: no further interpretation is needed or allowed.

21.1.3. Vocabulary and notation



Proposition

A **proposition** (or **statement**) is a declarative sentence that is either true or false, but not both. "7 is prime" is a proposition (true). "7 is even" is a proposition (false). "Close the door" (a command) and "$x > 3$" (whose truth depends on an unknown x) are **not** propositions. We name propositions with lowercase letters p, q, r. The **truth value** of a proposition is T (true) or F (false), sometimes written 1 and 0.



Connective

A **logical connective** (or **operator**) builds a new proposition from one or more existing ones. The five standard connectives are:

Symbol	Name	Read as	In code
$\neg p$	negation	"not p"	`not p`, `!p`
$p \wedge q$	conjunction	"p and q"	`p and q`, `p && q`
$p \vee q$	disjunction	"p or q"	`p or q`, `p || q`
$p \to q$	conditional (implication)	"if p then q"	`(not p) or q`
$p \leftrightarrow q$	biconditional	"p if and only if q"	`p == q`

A proposition built from others using connectives is **compound**; one that cannot be broken down further is **atomic**. Some books write $\sim p$ or $\overline{p}$ for negation, $\Rightarrow$ for $\to$, and $\Leftrightarrow$ or "iff" for $\leftrightarrow$.

The symbols $\wedge$ (and) and $\vee$ (or) are easy to mix up. A mnemonic: $\wedge$ looks like the "A" in *A*nd with the crossbar missing; $\vee$ comes from the Latin *vel*, "or".

21.1.4. The truth tables

Here are the definitions, all in one table. Each row is one possible combination of truth values for p and q; with two inputs there are $2^2 = 4$ rows.

Table 1. Truth tables of the five connectives

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \to q$	$p \leftrightarrow q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Try it: The conditional's truth table .

Three of these columns match everyday intuition. Two need comment.

Or is inclusive. $p \vee q$ is true when both are true. If you mean "one or the other but not both", that is a different connective, **exclusive or**, written $p \oplus q$ (and `p != q` or `p ^ q` in most languages). Mathematics, logic and programming languages all default to the inclusive reading.

The conditional is the subject of the next section; its last two rows surprise almost everyone.



Exclusive or (XOR)

$p \oplus q$ ("p xor q") is true when exactly one of p, q is true. It equals $(p \vee q) \wedge \neg(p \wedge q)$, and it is the negation of the biconditional: $p \oplus q \equiv \neg(p \leftrightarrow q)$.

21.1.5. Worked example: building a truth table

Let us settle the banner ticket. Let m = "user is a member", s = "user spent over $100", ℓ = "user is logged in". The most reasonable reading is *Try it:* Build the banner truth table .

$$(m \vee s) \wedge \ell.$$

With three atomic propositions there are $2^3 = 8$ rows. The systematic way to list them is to count down in binary: the first column alternates in blocks of four, the second in blocks of two, the third every row. Then compute intermediate columns left to right, exactly as you would evaluate a sub-expression before the expression that uses it.

m	s	ℓ	$m \lor s$	$(m \lor s) \land \ell$
T	T	T	T	**T**
T	T	F	T	F
T	F	T	T	**T**
T	F	F	T	F
F	T	T	T	**T**
F	T	F	T	F
F	F	T	F	F
F	F	F	F	F

Rows 2 and 4 are the bug report: a member who is logged out. The specification says F. The most likely culprit is code written as `is_member or big_spender and logged_in`, which Python parses as $m \lor (s \land \ell)$ because `and` binds more tightly than `or`. In rows 2 and 4 that gives $T \lor (\ldots) = T$: the banner appears. A truth table makes this kind of disagreement impossible to miss.



> *Precedence is not a matter of taste*
>
> The conventional precedence, from tightest to loosest, is $\neg$, then $\land$, then $\lor$, then $\to$, then $\leftrightarrow$. So $\neg p \lor q \land r$ means $(\neg p) \lor (q \land r)$. Programming languages agree on "not before and before or", but whenever an expression mixes `and` with `or`, add parentheses. They cost nothing and they document intent.

Example 2. Procedure — building a truth table

> **Input:** a compound proposition with n distinct atomic propositions.
>
> **Steps:**
>
> 1. List the atomic propositions as the first n columns.
> 2. Write 2^n rows. Fill column k (counting from 1) with alternating blocks of T and F of length 2^{n-k}. This enumerates every combination exactly once.
> 3. Identify the sub-expressions, innermost first, and give each its own column.
> 4. Fill each new column row by row, using the definitions of the connectives and the columns already computed.
> 5. The last column is the truth value of the whole proposition.
>
> **Output:** the complete truth table; the rows marked T are exactly the situations in which the proposition holds.

Try it: Truth table, step by step .

The procedure is an algorithm, so here it is as code. `itertools.product([True, False], repeat=n)` generates the 2^n rows in exactly the order described in step 2.

PYTHON

```python
from itertools import product

def truth_table(f, names):
    """Print the truth table of f, a function of len(names) booleans."""
    print(" ".join(names), "| result")
    for row in product([True, False], repeat=len(names)):
        cells = " ".join("T" if v else "F" for v in row)
        print(cells, "|", "T" if f(*row) else "F")

def implies(p, q):
    return (not p) or q

truth_table(lambda p, q: implies(p, q), ["p", "q"])
```

```
p q | result
T T | T
T F | F
F T | T
F F | T
```

Truth tables are exhaustive, which is both their virtue and their limit: n variables need 2^n rows. Ten variables means 1,024 rows; forty means about a trillion. That exponential wall is exactly why the algebraic laws later in this chapter, and the SAT solvers mentioned at the end, matter.

Logic gates: connectives in silicon

Every connective is also a physical circuit. A **logic gate** is a tiny arrangement of transistors whose output voltage is high (1) or low (0) according to a truth table.

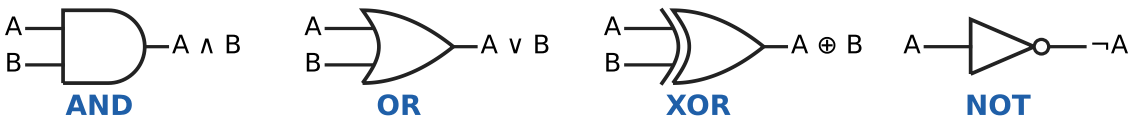


Figure 2. The four basic gates and the connective each computes

Wire gates together and you can compute arithmetic. Adding two one-bit numbers A and B gives a sum bit and a carry bit: $1 + 1 = 10_2$, so the sum bit is 0 and the carry is 1. Reading the truth table, the sum bit is $A \oplus B$ and the carry is $A \wedge B$. That two-gate circuit is a **half adder**; chain full adders (which also accept an incoming carry) and you have the adder inside every CPU.

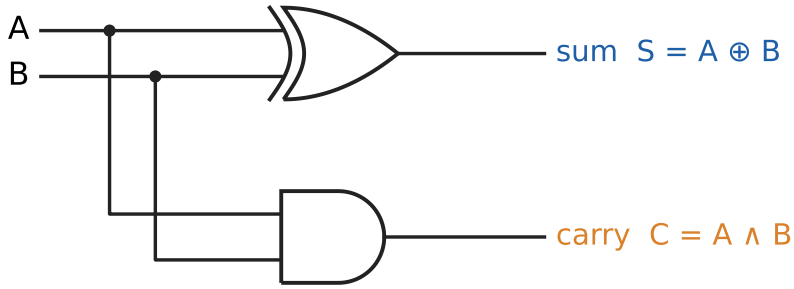


Figure 3. A half adder: logic that does arithmetic

A striking fact: the single connective NAND, $\neg(p \wedge q)$, can build every other connective. $\neg p$ is p NAND p, and $p \wedge q$ is the NAND of $(p$ NAND $q)$ with itself. Chip fabrication exploits this: a whole processor can be laid out from one kind of gate.



> *Boole, Shannon and the most important master's thesis*
>
> George Boole published *The Laws of Thought* in 1854, treating logic as algebra with values 0 and 1. For eighty years it was a philosophical curiosity. Then in 1937 a 21-year-old MIT student, Claude Shannon, noticed in his master's thesis that relay switching circuits obey exactly Boole's laws. Simplifying the algebra simplified the circuit. That thesis is often called the most consequential master's thesis ever written: it is the reason the type is called `bool`.

21.1.6. Where you'd use this

Everyday: rules with several conditions. A shop offers "free delivery if you are a member, or if the order is at least $40 and not oversized". With m, f (at least $40) and o (oversized), that is $m \vee (f \wedge \neg o)$. The brackets matter: $(m \vee f) \wedge \neg o$ is a different policy that charges members for oversized parcels.

Writing the formula is how you settle the argument at the checkout.

Medicine: "at least two of three". The qSOFA bedside screen for sepsis flags a patient when at least two of these hold: breathing rate at least 22 per minute (r), altered mental state (a), systolic blood pressure at most 100 mmHg (b). As a formula: $(r \wedge a) \vee (r \wedge b) \vee (a \wedge b)$, the **majority** function. Its truth table is T in exactly 4 of the 8 rows (TTT, TTF, TFT, FTT), and the same function is the carry-out bit of a full adder.

Engineering: interlocks. An industrial press may cycle only if the guard is closed (g) and both of the operator's hands are on the two buttons (ℓ, r): $g \wedge \ell \wedge r$. Two-hand controls exist so that no hand can be under the ram; the safety argument is literally a truth table with one T row out of eight.

Computing. Search boxes that accept `jaguar AND (engine OR gearbox) NOT cat` are evaluating a propositional formula per document; firewall and feature-flag rules are the same thing.

21.1.7. Exercises

1. Which of the following are propositions? (a) "17 is prime." (b) "Is 17 prime?" (c) "$x^2 \geq 0$ for every real number x." (d) "$x + 1 = 5$."
2. Build the truth table for $\neg(p \wedge q)$ and for $\neg p \wedge \neg q$. Are they the same?
3. A login is allowed if the password is correct and either the device is trusted or a one-time code was entered. Write this as a formula using w (password correct), d (trusted device), c (code entered), and find every row of its truth table that is T.

▼ **Answers**

1. (a) Yes, true. (b) No, a question. (c) Yes, true (the variable is bound by "for every", so the sentence as a whole is true or false; see Section 21.4). (d) No: its truth depends on the unknown x. It is a *predicate*.
2. $\neg(p \wedge q)$ has column F, T, T, T (rows TT, TF, FT, FF). $\neg p \wedge \neg q$ has column F, F, F, T. They differ in rows TF and FT, so they are **not** the same. (The correct partner of $\neg(p \wedge q)$ is $\neg p \vee \neg q$; see De Morgan's laws.)
3. $w \wedge (d \vee c)$. It is T exactly when w is T and at least one of d, c is T: rows (w, d, c) = (T,T,T), (T,T,F), (T,F,T). Three of eight rows.

US curriculum

Truth tables and connectives are not a standard part of the core US high-school sequence. They appear briefly in many **Geometry** courses (grade 9–10) in a unit on "reasoning and proof", in AP Computer Science Principles and AP Computer Science A (boolean expressions), and in full in a first-year college **Discrete Mathematics** course.

21.2. The conditional: "if ... then"

21.2.1. The problem

A parent says: "If you finish your homework, you can play video games." The child does not finish the homework and plays anyway. Did the parent lie? Most people say: no, the parent made no promise about what happens if the homework is unfinished.

Now look at the truth table for $p \to q$ again. When p is false, $p \to q$ is true, whatever q is. "False implies anything." This feels like a cheat when you first meet it. Why is it the right definition?

21.2.2. Intuition: a conditional is a promise

Read $p \to q$ as a promise or a contract: "whenever p holds, I guarantee q". The only way to **break** such a promise is for p to happen and q not to: row T, F. In every other row the promise has not been broken, and in two-valued logic "not false" means true.

A better test case than homework is a rule that has to hold for many cases at once. Take "for every whole number n, if n is divisible by 4 then n is even". Everyone agrees this is true. Now check it at $n = 3$: "if 3 is divisible by 4 then 3 is even" has a false hypothesis and a false conclusion. At $n = 6$: "if 6 is divisible by 4 then 6 is even" has a false hypothesis and a true conclusion. If either of these instances counted as **false**, then the general rule, which is the conjunction of all its instances, would be false. So for universal rules to work the way everyone wants, the conditional with a false hypothesis must be true. Logic is designed so that general statements behave; the truth table follows.



Vacuous truth

A conditional whose hypothesis is false is **vacuously true**: true because there is no case that could break it. "Every unicorn in this room is purple" is vacuously true, since there are no unicorns to check. In code, `all(x > 100 for x in [])` returns `True` for the same reason.



Hypothesis, conclusion, sufficient, necessary

In $p \to q$, p is the **hypothesis** (or **antecedent**, or **premise**) and q is the **conclusion** (or **consequent**). All of the following say exactly $p \to q$:

- if p then q; q if p;
- p implies q;
- p only if q;
- p is **sufficient** for q;
- q is **necessary** for p;
- q whenever p; not p unless q.

The biconditional $p \leftrightarrow q$ ("p if and only if q", often "iff") means $(p \to q) \wedge (q \to p)$: each is necessary and sufficient for the other.

"Only if" deserves a second look because it trips everyone. "You can vote only if you are 18 or over" means voting $\rightarrow$ 18-or-over: being 18 is **necessary**. It does not say every 18-year-old can vote (they might not be registered).

21.2.3. Why it works: the conditional as an "or"

Comparing columns shows that $p \rightarrow q$ has exactly the same truth table as $\neg p \vee q$:

p	q	$\neg p$	$\neg p \vee q$	$p \rightarrow q$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

"If it is raining, I take an umbrella" says the same as "either it is not raining, or I take an umbrella". This is how you implement implication in a language that has no implication operator, and it is the form that algebraic simplification works with.

21.2.4. Converse, inverse, contrapositive

From one conditional $p \rightarrow q$ you can form three relatives by swapping and/or negating the parts.

Name	**Form**	**Example (p: "it is a square", q: "it is a rectangle")**
original	$p \rightarrow q$	If it is a square, it is a rectangle. (true)
converse	$q \rightarrow p$	If it is a rectangle, it is a square. (false)
inverse	$\neg p \rightarrow \neg q$	If it is not a square, it is not a rectangle. (false)
contrapositive	$\neg q \rightarrow \neg p$	If it is not a rectangle, it is not a square. (true)

The pattern in the example is universal: **a conditional is always equivalent to its contrapositive**, and the converse is always equivalent to the inverse (the inverse is the contrapositive of the converse). But a conditional is **not** equivalent to its converse. Check the contrapositive with a truth table:

p	q	$p \to q$	$\neg q$	$\neg p$	$\neg q \to \neg p$
T	T	T	F	F	T
T	F	F	T	F	F
F	T	T	F	T	T
F	F	T	T	T	T

The third and last columns agree in every row. The contrapositive is not a curiosity: it is a whole proof technique in Chapter 22, used whenever "not q" is easier to work with than p.

A picture makes the asymmetry obvious. Treat p and q as properties and draw the **set** of things having each. "Every square is a rectangle" means the square region sits entirely inside the rectangle region. The converse would need the reverse containment, and the non-square rectangle on the right shows it fails.

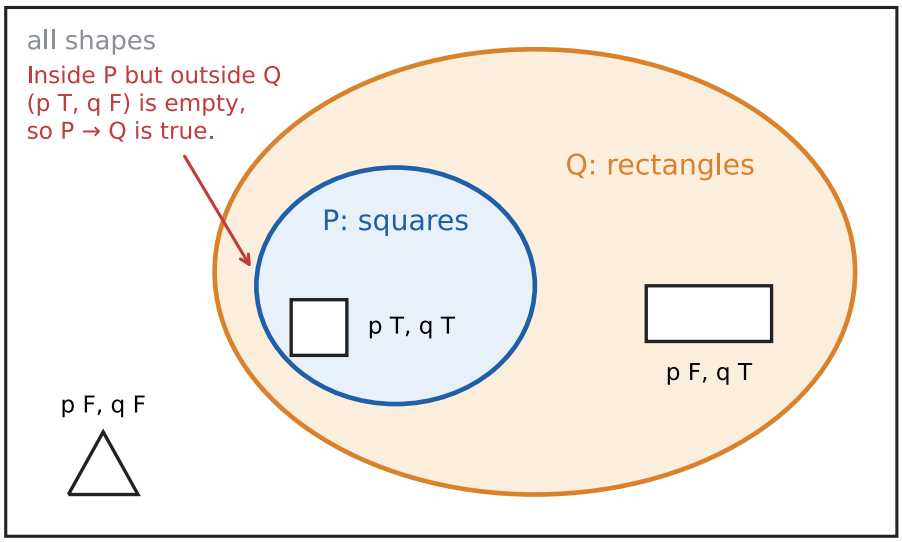


Figure 4. $p \to q$ as containment: the only row that would break the promise (p true, q false) is the empty crescent of P lying outside Q



Affirming the consequent

Confusing a statement with its converse is the most common logical error in everyday reasoning, in debugging, and in medicine. "If the server is overloaded, requests time out. Requests are timing out. So the server is overloaded." This is invalid: it uses $q \to p$, which was never established. Timeouts could come from a network partition. The valid inference patterns are

- **modus ponens:** from $p \to q$ and p, conclude q;
- **modus tollens:** from $p \to q$ and $\neg q$, conclude $\neg p$ (no timeouts, so not overloaded).

The invalid ones are **affirming the consequent** (from q, conclude p) and **denying the antecedent** (from $\neg p$, conclude $\neg q$).

21.2.5. Worked example: the four relatives of a code comment

A comment says: "If the cache is warm, the request returns in under 10 ms." Let w = "cache is warm", f = "request is fast". Then:

statement:	$w \to f$	**warm ⇒ fast**
contrapositive:	$\neg f \to \neg w$	**slow ⇒ cold (equivalent; useful for diagnosis)**
converse:	$f \to w$	**fast ⇒ warm (NOT implied: a cold cache can still be fast)**
inverse:	$\neg w \to \neg f$	**cold ⇒ slow (NOT implied, equivalent to the converse)**

If you observe a slow request, the contrapositive lets you conclude the cache was cold, assuming the comment is true. If you observe a fast request, you can conclude nothing about the cache.



The four-card puzzle

Psychologists have measured how bad people are at this. In the Wason selection task (1966), subjects see four cards showing E, K, 4, 7 and are told "if a card has a vowel on one side, it has an even number on the other". Which cards must you turn over to test the rule? Fewer than 10% answer correctly: E and 7. The 7 is the contrapositive check (odd, so must not be a vowel). Most people pick E and 4, and the 4 is useless: it tests the converse.

21.2.6. Where you'd use this

Everyday: reading offers and rules. "If you spend over $50, delivery is free" is broken only by a customer who spends $60 and is charged for delivery. The shop that also gives free delivery on a $30 order has not broken it (false hypothesis). "Only if" clauses are the ones people misread: "the warranty is valid only if the device has not been opened" means valid $\rightarrow$ not opened. Not opening it is necessary, but not sufficient: a two-year-old device may be out of warranty anyway.

Medicine: a test and its converse. "If you have the disease, the test is positive" (a test with high **sensitivity**) is the statement labs measure. The patient wants the converse, "if the test is positive, I have the disease", and that can be badly false when the disease is rare. Quantifying how false is a job for Bayes' theorem in Chapter 29; recognising that it is a different statement is a job for this section.

Engineering: fault-finding with the contrapositive. "If the fuse is intact and the switch is closed, the lamp lights": $(f \wedge s) \rightarrow L$. The lamp is dark, so by modus tollens $\neg(f \wedge s)$, which by De Morgan (next section) is $\neg f \vee \neg s$: the fuse has blown **or** the switch is open. Check the switch first; it is cheaper.

Computing. An `assert` or a function's documented precondition is a conditional: "if the input is sorted, the result is the index of the target". Calling it on unsorted input does not make the function wrong; it makes the contract vacuous, and the result meaningless.

21.2.7. Exercises

1. Write the converse, inverse and contrapositive of "If n is divisible by 6, then n is divisible by 3." Which are true for all whole numbers n?
2. Rewrite "The build passes only if all tests pass" as an "if … then" sentence.
3. Is "if $2 + 2 = 5$, then the moon is made of cheese" true or false? Why?
4. Express $p \leftrightarrow q$ using only $\neg$, $\wedge$ and $\vee$.

▼ **Answers**

1. Converse: if n is divisible by 3, then n is divisible by 6 (false: $n = 9$). Inverse: if n is not divisible by 6, then n is not divisible by 3 (false: $n = 9$ again). Contrapositive: if n is not divisible by 3, then n is not divisible by 6 (true, equivalent to the original, which is true because $6 = 2 \cdot 3$).
2. "If the build passes, then all tests pass." "Only if" introduces the **conclusion**.
3. True, vacuously: the hypothesis $2 + 2 = 5$ is false.

4. $p \leftrightarrow q \equiv (p \to q) \land (q \to p) \equiv (\neg p \lor q) \land (\neg q \lor p)$. An equally good answer is $(p \land q) \lor (\neg p \land \neg q)$: "both true or both false".

> **US curriculum**
>
> Conditionals, converse, inverse and contrapositive are a standard unit in high-school **Geometry** (grade 9–10), under "conditional statements" and "biconditionals", where they set up two-column proofs. Vacuous truth is usually postponed to college **Discrete Mathematics**.

21.3. Logical equivalence and De Morgan's laws

21.3.1. The problem

Here is a real-world conditional from a code review:

```python
if not (user.is_admin or (user.is_owner and not doc.locked)):
    raise PermissionDenied
```

What does it actually say? Under what conditions is the user **allowed**? And can it be written so that a tired reviewer at 6 p.m. can check it? We need a way to transform logical expressions while guaranteeing the meaning does not change.

21.3.2. Intuition

Two expressions are **equivalent** if no situation can tell them apart: in every row of the truth table they give the same answer. Just as algebra has laws such as $a(b + c) = ab + ac$ that let you rewrite expressions, logic has laws that let you rewrite propositions. Each law can be verified once, by truth table, and then used forever without checking again.



Logical equivalence, tautology, contradiction

Two propositions A and B are **logically equivalent**, written $A \equiv B$, if they have identical truth tables. Equivalently, $A \leftrightarrow B$ is true in every row.

A **tautology** is a proposition that is true in every row (for example $p \lor \neg p$). A **contradiction** is false in every row (for example $p \land \neg p$). A proposition that is neither is **contingent**. A proposition is **satisfiable** if at least one row makes it true.

$\equiv$ is a statement *about* two propositions (a "meta" claim), not a connective inside one.

21.3.3. The laws

Table 2. The laws of propositional logic ($\mathbf{T}$ is any tautology, $\mathbf{F}$ any contradiction)

Name	"And" form	"Or" form
Identity	$p \wedge \mathbf{T} \equiv p$	$p \vee \mathbf{F} \equiv p$
Domination	$p \wedge \mathbf{F} \equiv \mathbf{F}$	$p \vee \mathbf{T} \equiv \mathbf{T}$
Idempotent	$p \wedge p \equiv p$	$p \vee p \equiv p$
Complement	$p \wedge \neg p \equiv \mathbf{F}$	$p \vee \neg p \equiv \mathbf{T}$
Double negation	$\neg\neg p \equiv p$	
Commutative	$p \wedge q \equiv q \wedge p$	$p \vee q \equiv q \vee p$
Associative	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
De Morgan	$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Absorption	$p \wedge (p \vee q) \equiv p$	$p \vee (p \wedge q) \equiv p$
Conditional	$p \to q \equiv \neg p \vee q \equiv \neg q \to \neg p$	
Negated conditional	$\neg(p \to q) \equiv p \wedge \neg q$	

Notice the pattern: every law comes in a pair, obtained by swapping $\wedge$ with $\vee$ and $\mathbf{T}$ with $\mathbf{F}$. This is the **duality principle**. Also notice that unlike ordinary algebra, **both** distributive laws hold: "and" distributes over "or" (like multiplication over addition), and "or" also distributes over "and" (which has no arithmetic counterpart: $a + bc \neq (a + b)(a + c)$ in general).

21.3.4. De Morgan's laws, in words

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

To push a "not" inside a bracket, negate each piece **and flip the connective**. "It is not the case that the server is both up and healthy" means "the server is down or it is unhealthy". "It is not the case that it's raining or snowing" means "it is not raining and it is not snowing".

Why must the connective flip? "Not both" fails only when both hold, so it is true as soon as **one** of them fails: that is an "or" of failures. "Neither" requires **every** option to fail: that is an "and" of failures. You saw the truth tables in the first exercise set; the Venn diagram in Section 21.5 gives the same picture for sets.

Example 3. Procedure — simplifying a boolean expression

Input: a boolean expression, typically a messy condition.

Steps:

1. Replace every $\rightarrow$ by $\neg p \lor q$ and every $\leftrightarrow$ by $(\neg p \lor q) \land (\neg q \lor p)$, so that only $\neg, \land, \lor$ remain.
2. Push every $\neg$ inward with De Morgan's laws, flipping $\land \leftrightarrow \lor$ each time, until negations sit directly on atomic propositions.
3. Remove double negations: $\neg\neg p \equiv p$.
4. Simplify with the complement, identity, domination, idempotent and absorption laws, using distribution or factoring where it shortens the expression.
5. Check the result: either compare truth tables (for few variables) or test it mechanically, for example with the `equivalent` function below.

Output: an equivalent, simpler expression, usually with negations only on single variables.

21.3.5. Worked example: the permission check

Let a = `user.is_admin`, o = `user.is_owner`, ℓ = `doc.locked`. The code raises when *Try it:* Push the negation inward.

$$\neg\big(a \lor (o \land \neg \ell)\big).$$

Push the negation inward one step at a time:

$$\begin{aligned} \neg\big(a \lor (o \land \neg \ell)\big) &\equiv \neg a \land \neg(o \land \neg \ell) && \text{(De Morgan on } \lor\text{)} \\ &\equiv \neg a \land (\neg o \lor \neg\neg \ell) && \text{(De Morgan on } \land\text{)} \\ &\equiv \neg a \land (\neg o \lor \ell) && \text{(double negation)} \end{aligned}$$

In words: **deny if the user is not an admin, and either is not the owner or the document is locked.** The code can be rewritten

```python
if not user.is_admin and (not user.is_owner or doc.locked):
    raise PermissionDenied
```

Even better is to name the positive condition, since the original's inner expression was the **allow** rule all along:

```python
can_edit = user.is_admin or (user.is_owner and not doc.locked)
if not can_edit:
    raise PermissionDenied
```

Which is better is a judgment call; the point is that you can now move between the forms with certainty.

21.3.6. A second worked example: absorption

Simplify $(p \wedge q) \vee (p \wedge \neg q)$. *Try it:* Check the absorption example .

$$\begin{aligned} (p \wedge q) \vee (p \wedge \neg q) &\equiv p \wedge (q \vee \neg q) && \textbf{(distributive law, read right to left: factor out } p\textbf{)} \\ &\equiv p \wedge \mathbf{T} && \textbf{(complement)} \\ &\equiv p && \textbf{(identity)} \end{aligned}$$

Sanity check in words: "p and q, or p and not q" is true exactly when p is true, whatever q does. The variable q was irrelevant, and the code `if (p and q) or (p and not q)` is a smell.

21.3.7. Alternative approaches and choosing a method

You have three ways to decide whether two expressions are equivalent.

- **Truth table.** Mechanical and certain. Best for up to three or four variables. Hopeless beyond about twenty.
- **Algebraic laws.** Produces a **simplified** form and an explanation of why, which a truth table never gives you. Requires some skill to choose which law to apply.
- **Mechanical check in code.** The same as a truth table but delegated to a computer, which pushes the limit to about 20–25 variables:

PYTHON

```python
from itertools import product

def equivalent(f, g, n):
    """True if f and g agree on every one of the 2**n input rows."""
    return all(f(*row) == g(*row) for row in product([True, False], repeat=n))

def implies(p, q):
    return (not p) or q

# De Morgan
print(equivalent(lambda p, q: not (p and q), lambda p, q: (not p) or (not q), 2))
# contrapositive vs converse
print(equivalent(lambda p, q: implies(p, q), lambda p, q: implies(not q, not p), 2))
print(equivalent(lambda p, q: implies(p, q), lambda p, q: implies(q, p), 2))
# distributive law, three variables
print(equivalent(lambda p, q, r: p and (q or r),
                 lambda p, q, r: (p and q) or (p and r), 3))
```

Try it: De Morgan, row by row ; Contrapositive versus conditional ; Converse is not equivalent ; Distributive law, three variables .

```
True
True
False
True
```

Rule of thumb: use the laws to **find** a simpler form, then use a truth table or `equivalent` to **confirm** it. When refactoring a production conditional, write the confirmation as a unit test.



The classic De Morgan slip

The two errors that cause real bugs:

- Negating the pieces but forgetting to flip the connective: writing `not (a and b)` as `not a and not b`. That is "neither", not "not both".
- Negating a range check incorrectly. "x is in range" is $0 \le x \land x < n$. Its negation is $x < 0 \lor x \ge n$, with an **or**. A guard written `if x < 0 and x >= n` can never fire.

Computer-science connection: short-circuit evaluation

Mathematically $p \wedge q \equiv q \wedge p$. In code, `p and q` and `q and p` can behave differently, because almost every language evaluates `and` / `or` **lazily**: left to right, stopping as soon as the answer is known.

- `False and X`: the answer is False whatever `X` is (domination law), so `X` is never evaluated.
- `True or X`: the answer is True whatever `X` is, so `X` is never evaluated.

```python
def expensive():
    print("  expensive() was called")
    return True

print("False and expensive():")
r = False and expensive()
print("True or expensive():")
r = True or expensive()
print("True and expensive():")
r = True and expensive()
```

```
False and expensive():
True or expensive():
True and expensive():
  expensive() was called
```

This is why `if user is not None and user.is_admin` is safe while the reversed order crashes, and why you should put cheap, likely-to-decide tests first. Commutativity holds for the **values**; it does not hold for **side effects, exceptions or cost**. When you apply the laws of logic to code, check that no operand has side effects or can raise, or keep guarding conditions on the left.

Python adds one more twist: `and` / `or` return one of their operands, not necessarily a `bool`. `0 or "default"` is `"default"`, the basis of the `x = value or default` idiom (which misfires when a legitimate value is falsy, such as `0` or `""`).



Interactive Lab: truth-table playground

Type two boolean expressions in up to five variables. The lab draws both truth tables side by side, highlights rows where they disagree, and offers a "push negation inward" button that applies De Morgan one step at a time, showing which law fired. A challenge mode gives a messy condition from real code and asks for the shortest equivalent.

Try it: Open the truth-table playground

21.3.8. Where you'd use this

Everyday: rules phrased in the negative. A cinema says "no entry unless you are over 15 or with an adult". Entry is allowed when $a \vee w$; by De Morgan you are refused exactly when $\neg a \wedge \neg w$, under 15 **and** alone. Spreadsheets obey the same law: `=NOT(OR(B2>100, C2="late"))` and `=AND(B2<=100, C2<>"late")` filter the same rows, and the second is easier for a colleague to read.

Engineering: fewer gates. Each gate costs chip area, power and a few picoseconds of delay. The absorption example above turned $(p \wedge q) \vee (p \wedge \neg q)$ (two ANDs, a NOT and an OR, four gates) into p (no gates: a wire). Circuit synthesis tools apply exactly these laws, millions of times, to every chip design.

Computing. Database query optimisers rewrite `WHERE NOT (status = 'done' OR archived)` into `status <> 'done' AND NOT archived` so that each part can use an index; compilers push negations inward the same way. The Watch out box above is the bug you get when a human does the rewrite by hand and forgets to flip the `or`.

21.3.9. Exercises

1. Use De Morgan's laws to negate: (a) "the file exists and is readable"; (b) $x < 0 \vee x > 10$; (c) `not (a or not b)`.
2. Simplify $\neg(p \to q) \vee q$ using the laws, showing each step. *Try it:* Step through this simplification.
3. Show that $p \wedge (p \vee q) \equiv p$ (absorption) with a truth table.
4. Is $(p \to q) \vee (q \to p)$ a tautology, a contradiction, or contingent?
5. A colleague "simplifies" `if not (is_weekend or is_holiday):` to `if not is_weekend or not is_holiday:`. Find inputs where the two disagree and give the correct rewrite. *Try it:* Find where they disagree.

▼ **Answers**

1. (a) The file does not exist **or** it is not readable. (b) $x \geq 0 \wedge x \leq 10$, i.e. $0 \leq x \leq 10$. (c) $\neg(a \vee \neg b) \equiv \neg a \wedge \neg\neg b \equiv \neg a \wedge b$, i.e. `not a and b`.
2. $\neg(p \to q) \vee q \equiv (p \wedge \neg q) \vee q$ (negated conditional) $\equiv (p \vee q) \wedge (\neg q \vee q)$ (distribute $\vee$ over $\wedge$) $\equiv (p \vee q) \wedge \mathbf{T}$ (complement) $\equiv p \vee q$ (identity).
3. Rows (p, q): TT gives $p \vee q = T$, so $p \wedge T = T$; TF: $T \wedge T = T$; FT: $F \wedge T = F$; FF: $F \wedge F = F$. The column T, T, F, F equals the p column.
4. A tautology. $(\neg p \vee q) \vee (\neg q \vee p) \equiv (\neg p \vee p) \vee (q \vee \neg q) \equiv \mathbf{T} \vee \mathbf{T} \equiv \mathbf{T}$. Surprising but correct: of any two propositions, one implies the other, purely by the truth-table definition.
5. With `is_weekend = True`, `is_holiday = False`: the original gives `not True = False`; the "simplification" gives `False or True = True`. Correct rewrite: `if not is_weekend and not is_holiday:` ("neither").

US curriculum

De Morgan's laws appear in **AP Computer Science A** (boolean expressions unit) and sometimes in **Geometry** logic units. The full set of equivalence laws is standard in college **Discrete Mathematics**, often taken in the first or second year of a CS degree.

21.4. Predicates and quantifiers

21.4.1. The problem

"$x > 3$" is not a proposition, because its truth depends on x. Yet mathematics and code are full of statements like "every element of the list is positive", "some user has no password", "for every input there is a matching output". Propositional logic, with its fixed T/F atoms, cannot express any of these. We need statements with variables in them, and a way to say "all" and "some".

21.4.2. Intuition

A **predicate** is a proposition with a hole in it, a function that returns a boolean: `is_even(n)`. Fill the hole with a value and you get a proposition. A **quantifier** fills the hole differently: instead of naming one value, it makes a claim about **all** values or about **at least one**. In code, the universal quantifier is `all(…)` and the existential quantifier is `any(…)`.



Predicate and domain

A **predicate** $P(x)$ is a statement involving a variable x that becomes a proposition when x is given a value. The set of allowed values is the **domain** (or **universe of discourse**). With domain the integers, $P(x)$: "x is even" gives $P(4)$ true and $P(7)$ false. Predicates can have several variables: $L(x, y)$: "$x < y$".



Quantifiers

- The **universal quantifier** $\forall$ ("for all", an upside-down A): $\forall x\, P(x)$ is true if $P(x)$ is true for **every** x in the domain.
- The **existential quantifier** $\exists$ ("there exists", a backwards E): $\exists x\, P(x)$ is true if $P(x)$ is true for **at least one** x in the domain.
- $\exists!\, x\, P(x)$ means "there exists exactly one x".

The domain is often written into the quantifier: $\forall x \in \mathbb{Z}$ ("for every integer x"). A variable governed by a quantifier is **bound**; one that is not is **free**. Only a statement with no free variables is a proposition.

Over a finite domain $\{a_1, \ldots, a_n\}$, quantifiers are just long connectives:

$$\forall x\, P(x) \;\equiv\; P(a_1) \wedge P(a_2) \wedge \cdots \wedge P(a_n), \qquad \exists x\, P(x) \;\equiv\; P(a_1) \vee P(a_2) \vee \cdots \vee P(a_n).$$

This view explains vacuous truth again: over an **empty** domain, a "for all" is an empty "and", which is true (the identity for $\wedge$), and a "there exists" is an empty "or", which is false (the identity for $\vee$). It is the same reason the empty sum is 0 and the empty product is 1.

```python
nums = [3, 8, 12, 5]
print(all(n > 0 for n in nums))        # for all n, n > 0
print(any(n % 2 == 0 for n in nums))   # there exists an even n
print(all(n % 2 == 0 for n in nums))   # for all n, n is even
print(any(n % 2 != 0 for n in nums))   # there exists an n that is not even
print(all(n > 100 for n in []))        # vacuous truth
print(any(n > 100 for n in []))
```

```
True
True
False
True
True
False
```

Lines three and four are negations of each other, which is the next idea.

21.4.3. Negating quantified statements

To deny "every swan is white" you do not need to show that every swan is non-white. You need **one** black swan. To deny "some student passed" you need to show that **every** student failed. That is De Morgan's law again, applied to the long "and" and the long "or":

$$\neg\,\forall x\, P(x) \;\equiv\; \exists x\, \neg P(x) \qquad\qquad \neg\,\exists x\, P(x) \;\equiv\; \forall x\, \neg P(x)$$

Example 4. Procedure — negating a quantified statement

Input: a statement with a string of quantifiers followed by a quantifier-free part.

Steps:

1. Write $\neg$ in front of the whole statement.
2. Move the $\neg$ rightward past each quantifier in turn, **flipping** it: $\forall$ becomes $\exists$ and $\exists$ becomes $\forall$. Domains ($x \in S$) stay unchanged.
3. When the $\neg$ reaches the quantifier-free part, negate it with the propositional rules: De Morgan for $\wedge/\vee$, $\neg(p \to q) \equiv p \wedge \neg q$ for conditionals, and flip comparisons ($\neg(a < b)$ is $a \geq b$).
4. Read the result back in English as a check.

Output: an equivalent statement with the negation pushed all the way in.

21.4.4. Worked example: negating a nested statement

Take the statement "every request gets a response within 2 seconds of being sent". With R the set of requests and S the set of responses:

$$\forall r \in R \;\; \exists s \in S \;\; \big(\mathbf{answers}(s, r) \land t(s) - t(r) \leq 2\big).$$

What does a monitoring alert need to detect, i.e. what is the negation?

$$\begin{aligned}
&\neg\, \forall r \; \exists s \; \big(\mathbf{answers}(s, r) \land t(s) - t(r) \leq 2\big) \\
&\equiv \exists r \; \neg\, \exists s \; \big(\mathbf{answers}(s, r) \land t(s) - t(r) \leq 2\big) && (\textbf{flip } \forall \textbf{ to } \exists) \\
&\equiv \exists r \; \forall s \; \neg\big(\mathbf{answers}(s, r) \land t(s) - t(r) \leq 2\big) && (\textbf{flip } \exists \textbf{ to } \forall) \\
&\equiv \exists r \; \forall s \; \big(\neg \mathbf{answers}(s, r) \lor t(s) - t(r) > 2\big) && (\textbf{De Morgan, flip } \leq)
\end{aligned}$$

In English: "there is some request such that every response either does not answer it or arrives more than 2 seconds later". Equivalently: some request was not answered in time. You can also write the final disjunction as a conditional, $\mathbf{answers}(s, r) \to t(s) - t(r) > 2$, "any response that answers it is late".

21.4.5. Order of quantifiers matters

Compare, over the integers, *Try it:* Now play ∃∀ ; Play the ∀∃ game .

$$\forall x \; \exists y \; (y > x) \qquad \textbf{versus} \qquad \exists y \; \forall x \; (y > x).$$

The first says "for every number there is a bigger one": true (take $y = x + 1$). The second says "there is one number bigger than every number": false (it would have to be bigger than itself). The only difference is the order. In the first, y is chosen **after** x and may depend on it; in the second, y must be chosen first, once, and work for all x.

A game helps. Read $\forall$ as your opponent's move and $\exists$ as yours, in left-to-right order. The statement is true if you have a winning strategy. In $\forall x \; \exists y$, your opponent names x, then you reply: easy. In $\exists y \; \forall x$, you must commit first, and your opponent picks $x = y$ to beat you.



Swapping quantifiers silently

"Every user has a password" ($\forall u \; \exists p$) is not "there is a password every user has" ($\exists p \; \forall u$); the second would be a security disaster. Quantifiers of the **same** kind commute ($\forall x \; \forall y \equiv \forall y \; \forall x$), but $\forall$ and $\exists$ do not. Whenever an English sentence has two quantifiers, write it symbolically and check the order. In Volume 2, Chapter 16 the definition of a limit has the shape "$\forall \varepsilon \; \exists \delta \; \ldots$" and the order is the whole point.

Computer-science connection: quantifiers everywhere

- `all()` and `any()` in Python, `.every()` and `.some()` in JavaScript, `allMatch` / `anyMatch` in Java streams, `ALL` / `EXISTS` in SQL subqueries are all quantifiers over a finite domain. Like `and` / `or`, they short-circuit: `any` stops at the first true element.
- A **test** checks $P(x)$ for some x. Passing tests show $\exists x\, P(x)$; a specification claims $\forall x\, P(x)$. One failing test (a counterexample) refutes the universal claim; a million passing tests do not prove it. Chapter 22 takes this up.
- Property-based testing (Hypothesis in Python, QuickCheck in Haskell) makes the quantifier explicit: you write the predicate, the library samples the domain hunting for a counterexample.
- Type signatures are quantified statements: a generic function `def first(xs: list[T]) → T` claims to work **for all** types `T`.

21.4.6. Where you'd use this

Everyday: one counterexample is enough. A dish is safe for a guest with coeliac disease when **every** ingredient is gluten-free: $\forall i\, G(i)$. Its negation is $\exists i\, \neg G(i)$: one ingredient with gluten (soy sauce is the classic hidden one) makes the whole dish unsafe. Allergen labels, "all passengers must have a passport" and "every room has a smoke alarm" are universal claims that a single inspection can refute.

Science and engineering: universal claims and finite tests. A specification such as "for every load up to 5 kN the beam deflects less than 2 mm" is a $\forall$ statement over infinitely many loads. Testing at 1, 2, 3, 4 and 5 kN gives five instances, not the universal claim; the claim rests on a calculation (a proof) plus a safety factor. Scientific laws have the same shape, which is why one reproducible counterexample outweighs any number of confirmations.

Computing. A foreign-key constraint, "every order belongs to some customer", is $\forall o\, \exists c\, \mathbf{owns}(c, o)$. The query that finds violations is its negation, $\exists o\, \forall c\, \neg \mathbf{owns}(c, o)$: an orphan order, written in SQL as `NOT EXISTS`.

21.4.7. Exercises

1. Translate into symbols with domain the integers, and decide truth: (a) every integer has a negative; (b) some integer is its own square; (c) there is an integer that divides every integer.
2. Negate, pushing the negation all the way in: $\forall x\, \big(x > 0 \to \exists y\, (y^2 = x)\big)$.
3. Write a Python expression for "some row in `table` has every entry zero", and for its negation.

▼ **Answers**

1. (a) $\forall x\, \exists y\, (x + y = 0)$: true (take $y = -x$). (b) $\exists x\, (x^2 = x)$: true ($x = 0$ or $x = 1$). (c) $\exists d\, \forall n\, (d \text{ divides } n)$: true ($d = 1$, or $d = -1$).

2. $\exists x\,(x > 0 \wedge \forall y\,(y^2 \neq x))$: "some positive number is not a perfect square". Steps: flip $\forall$ to $\exists$; negate the conditional $\neg(p \to q) \equiv p \wedge \neg q$; flip $\exists y$ to $\forall y$; negate $=$ to $\neq$. (Over the integers this negation is true, e.g. $x = 2$, so the original is false; over the reals the original is true.)
3. `any(all(v == 0 for v in row) for row in table)`. Negation: `all(any(v != 0 for v in row) for row in table)`: every row has at least one nonzero entry.

US curriculum

Formal quantifier notation ($\forall$, $\exists$) is generally not taught in US high schools, though the ideas appear informally in **Algebra II** and **Precalculus** ("for all real x"). It is core material in college **Discrete Mathematics** and in an introductory proofs course.

21.5. Sets

21.5.1. The problem

A marketing team wants "customers who bought in 2025 but not in 2026, excluding anyone who unsubscribed". A security team wants "every IP address that appears in both the firewall log and the threat list". A compiler wants "the variables that are live at the end of this block". Each of these is a question about collections: combining them, intersecting them, removing one from another. The language for collections is set theory, and it turns out to be logic wearing different clothes.

21.5.2. Intuition

A set is a bag of distinct things where order does not matter and duplicates do not count. The only question you can ask of a set is "is this thing in it?" Everything else, including union, intersection and complement, is defined by how it answers that question. And the answer to "is x in $A \cup B$?" is "is x in A **or** is x in B?" So each set operation is a logical connective applied to membership.

21.5.3. Vocabulary and notation



Set, element, membership

A **set** is a well-defined collection of distinct objects, called its **elements** or **members**. We write $x \in A$ for "x is an element of A" and $x \notin A$ for "x is not". Sets are written with braces: $A = \{2, 3, 5, 7\}$. Order and repetition are irrelevant: $\{1, 2, 2, 3\} = \{3, 2, 1\}$. Two sets are **equal** when they have exactly the same elements.

The **cardinality** $|A|$ is the number of elements of a finite set: $|\{2, 3, 5, 7\}| = 4$.



Standard number sets

$\mathbb{N}$ natural numbers $\{0, 1, 2, \ldots\}$ (some authors start at 1; computer scientists usually start at 0); $\mathbb{Z}$ integers (from German *Zahlen*, "numbers"); $\mathbb{Q}$ rationals (*quotient*); $\mathbb{R}$ reals; $\mathbb{C}$ complex numbers. See Volume 1, Chapter 1.

Set-builder notation

Set-builder notation describes a set by a rule rather than a list:

$$\{\, x \in \mathbb{Z} \mid x > 0 \text{ and } x \text{ is even} \,\} = \{2, 4, 6, \ldots\}$$

Read the bar $\mid$ (sometimes a colon) as "such that". The left part says where elements come from and what shape they have; the right part is a predicate they must satisfy. You can also put an expression on the left: $\{\, n^2 \mid n \in \mathbb{N} \,\} = \{0, 1, 4, 9, \ldots\}$. Python's set comprehension `{n*n for n in range(10)}` is set-builder notation, almost symbol for symbol.

Empty set, subset, universal set

- The **empty set** $\varnothing$ (or $\{\}$) has no elements; $|\varnothing| = 0$. There is only one empty set.
- A is a **subset** of B, written $A \subseteq B$, if every element of A is also in B: $\forall x\,(x \in A \to x \in B)$. A **proper subset**, $A \subsetneq B$ (often written $A \subset B$), is a subset that is not equal to B.
- $A = B$ exactly when $A \subseteq B$ and $B \subseteq A$. This "two inclusions" approach is the standard way to prove two sets equal.
- The **universal set** U is the set of everything under discussion in a given context (all customers, all integers).

Two consequences of the definition of subset surprise people. First, $\varnothing \subseteq A$ for every set A: the condition "every element of $\varnothing$ is in A" is vacuously true. Second, $\in$ and $\subseteq$ are different relations: $2 \in \{1, 2\}$ but $2 \not\subseteq \{1, 2\}$ (2 is not a set of elements of $\{1, 2\}$); $\{2\} \subseteq \{1, 2\}$ but $\{2\} \notin \{1, 2\}$.

21.5.4. Operations on sets

Union, intersection, difference, complement

For sets A, B inside a universal set U:

Operation	Definition	Logic behind it
Union $A \cup B$	$\{x \mid x \in A \vee x \in B\}$	or
Intersection $A \cap B$	$\{x \mid x \in A \wedge x \in B\}$	and
Difference $A \setminus B$ (also $A - B$)	$\{x \mid x \in A \wedge x \notin B\}$	and not
Complement $\overline{A}$ (also A^c, A')	$\{x \in U \mid x \notin A\}$	not
Symmetric difference $A \triangle B$	$(A \setminus B) \cup (B \setminus A)$	xor

Sets with no common element, $A \cap B = \varnothing$, are **disjoint**.

A **Venn diagram** draws the universal set as a rectangle and each set as a region inside it. Shading shows the result of an operation.

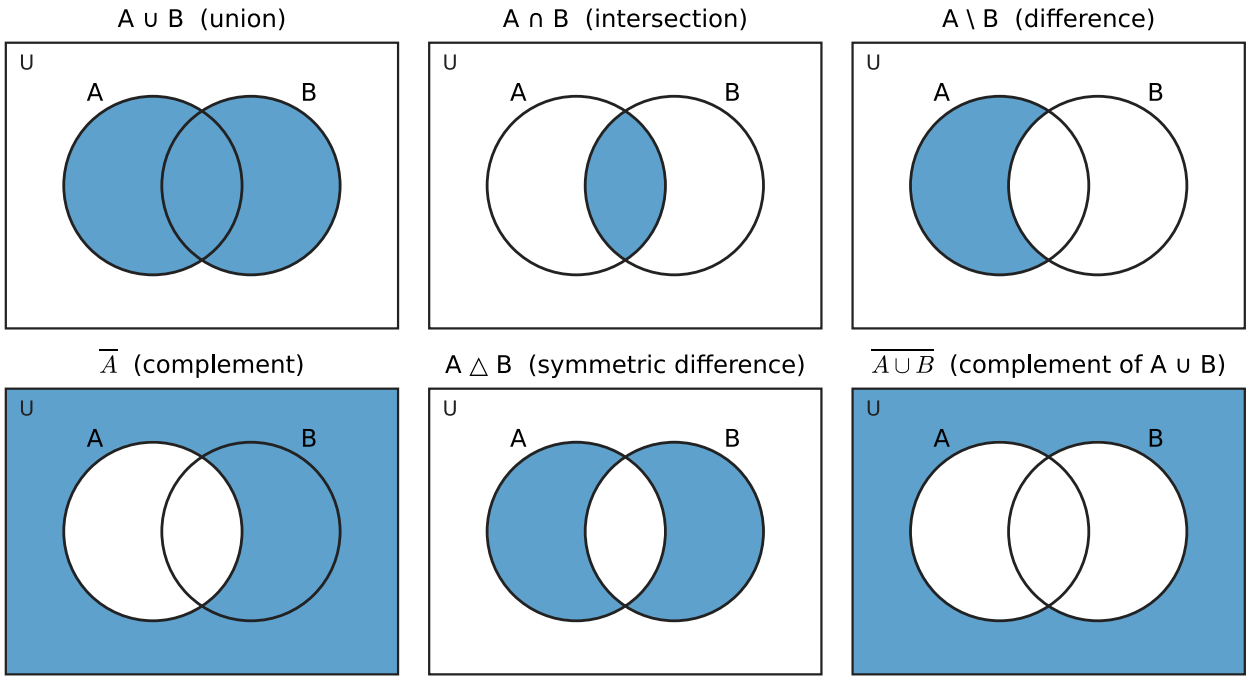


Figure 5. Venn diagrams of the main set operations

Because each operation is a connective in disguise, **every law of logic gives a law of sets**. Replace $\vee$ by $\cup$, $\wedge$ by $\cap$, $\neg$ by complement, $\mathbf{T}$ by U and $\mathbf{F}$ by $\varnothing$:

Law	Logic	Sets
Distributive	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
De Morgan	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\overline{A \cup B} = \overline{A} \cap \overline{B}$
De Morgan	$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\overline{A \cap B} = \overline{A} \cup \overline{B}$
Complement	$p \vee \neg p \equiv \mathbf{T}$	$A \cup \overline{A} = U$
Absorption	$p \wedge (p \vee q) \equiv p$	$A \cap (A \cup B) = A$

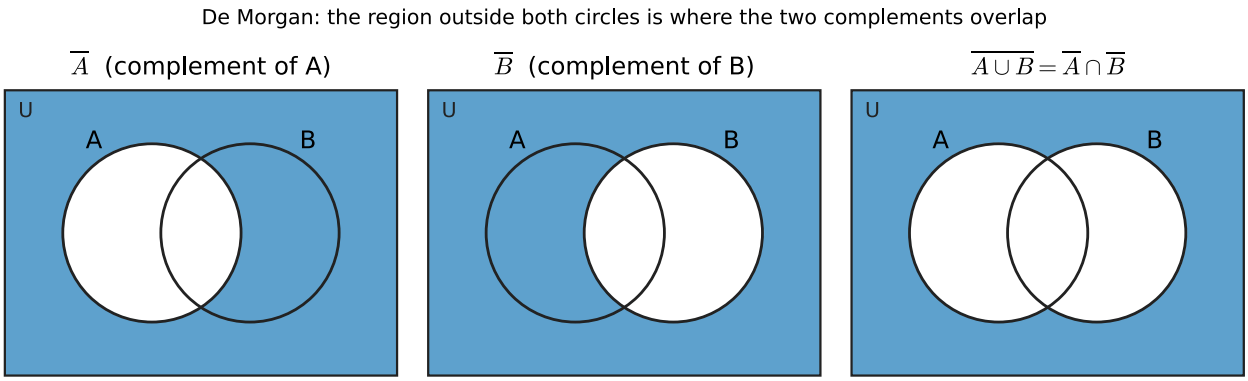


Figure 6. De Morgan for sets: outside both circles is the overlap of the two outsides

A Venn diagram is a good way to **discover** or **check** a set identity. To **prove** one, use the membership argument below, because diagrams with three or more sets get hard to draw correctly and easy to misread.

Example 5. Procedure — proving a set identity by element-chasing

Input: a claimed identity $X = Y$ between two set expressions.

Steps:

1. Take an arbitrary element x and write "$x \in X$".
2. Unfold the definitions of the operations in X, turning membership into a logical formula in statements like $x \in A$.
3. Transform that formula with the laws of logic until it is the unfolded form of $x \in Y$.
4. Fold it back up: "$x \in Y$". Since every step was an equivalence ($\iff$), you have shown $x \in X \iff x \in Y$ for all x, so $X = Y$.
5. If some step is only an implication, prove the two inclusions $X \subseteq Y$ and $Y \subseteq X$ separately.

Output: a proof of the identity, or a counterexample element found while trying.

21.5.5. Worked example: De Morgan for sets

Prove $\overline{A \cup B} = \overline{A} \cap \overline{B}$. *Try it:* De Morgan on a Venn diagram .

$$
\begin{aligned}
x \in \overline{A \cup B} &\iff \neg\,(x \in A \cup B) && \textbf{(definition of complement)} \\
&\iff \neg\,(x \in A \vee x \in B) && \textbf{(definition of union)} \\
&\iff \neg(x \in A) \wedge \neg(x \in B) && \textbf{(De Morgan for logic)} \\
&\iff x \in \overline{A} \wedge x \in \overline{B} && \textbf{(definition of complement, twice)} \\
&\iff x \in \overline{A} \cap \overline{B} && \textbf{(definition of intersection)}
\end{aligned}
$$

The set identity is literally the logic identity, applied to membership.

21.5.6. Counting: inclusion–exclusion

How many elements are in $A \cup B$? Adding $|A| + |B|$ counts the overlap twice, so subtract it once: *Try it:* Inclusion–exclusion with counts .

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

Example: 120 users enabled two-factor authentication, 85 use a passkey, and 40 do both. Then $|A \cup B| = 120 + 85 - 40 = 165$ users have at least one of the two. This **principle of inclusion–exclusion** grows to three or more sets in Chapter 24, and becomes the addition rule for probabilities in Chapter 29.

21.5.7. Power set and Cartesian product

Two constructions build new, bigger sets from old ones.



Power set

The **power set** $\mathcal{P}(A)$ is the set of all subsets of A, including $\varnothing$ and A itself. For $A = \{a, b, c\}$:

$$\mathcal{P}(A) = \{\varnothing,\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}\}.$$

If $|A| = n$ then $|\mathcal{P}(A)| = 2^n$, because building a subset means making an independent in-or-out decision for each of the n elements. A subset of an n-element set **is** an n-bit string; that is why bitmasks represent sets.



Ordered pair and Cartesian product

An **ordered pair** (a, b) is like a two-element set except that order matters: $(1, 2) \neq (2, 1)$, and $(a, b) = (c, d)$ exactly when $a = c$ and $b = d$. More generally an **n-tuple** is $(a_1, \ldots, a_n)$.

The **Cartesian product** of A and B is the set of all ordered pairs with first entry from A and second from B:

$$A \times B = \{\,(a, b) \mid a \in A,\ b \in B\,\}, \qquad |A \times B| = |A| \cdot |B|.$$

$\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ is the coordinate plane of Volume 1, Chapter 10 , which is where the name (after Descartes) comes from.

Computer-science connection: SQL is set theory

A relational database table is (almost) a set of tuples, and SQL is (almost) set theory. "Almost" because SQL tables can contain duplicate rows unless you use `DISTINCT` or a key.

- `UNION`, `INTERSECT` and `EXCEPT` are $\cup$, $\cap$ and $\setminus$ on result sets (`UNION ALL` keeps duplicates).
- A `WHERE` clause is a predicate; the result is the set-builder $\{\, r \in T \mid P(r) \,\}$. Its `AND`, `OR`, `NOT` follow the laws of this chapter, so De Morgan can rewrite `NOT (a = 1 OR b = 2)` as `a <> 1 AND b <> 2`. (Careful: SQL has a third truth value, `NULL`/unknown, and with it `NOT (x = 1)` does not select rows where `x` is `NULL`.)
- `FROM emp, dept` forms the Cartesian product $\mathbf{emp} \times \mathbf{dept}$. An inner `JOIN … ON` is, by definition, that product filtered by a predicate. Real query planners never build the whole product, but that is the meaning.

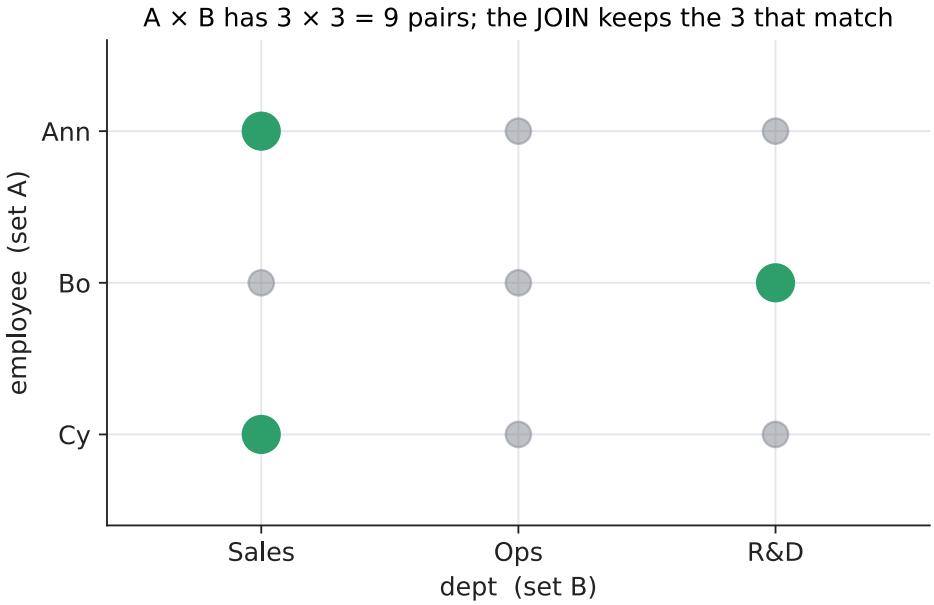


Figure 7. A join is a Cartesian product followed by a filter

You can watch this happen with Python's built-in `sqlite3` module:

```python
import sqlite3

db = sqlite3.connect(":memory:")
db.executescript("""
CREATE TABLE emp(name TEXT, dept_id INT);
CREATE TABLE dept(id INT, dname TEXT);
INSERT INTO emp VALUES ('Ann', 1), ('Bo', 3), ('Cy', 1);
INSERT INTO dept VALUES (1, 'Sales'), (2, 'Ops'), (3, 'R&D');
""")
print(len(db.execute("SELECT * FROM emp, dept").fetchall()))  # A x B
print(db.execute("""SELECT name, dname FROM emp, dept
                    WHERE emp.dept_id = dept.id""").fetchall())
print(db.execute("""SELECT name, dname FROM emp JOIN dept
                    ON emp.dept_id = dept.id""").fetchall())
```

```
9
[('Ann', 'Sales'), ('Bo', 'R&D'), ('Cy', 'Sales')]
[('Ann', 'Sales'), ('Bo', 'R&D'), ('Cy', 'Sales')]
```

A forgotten join condition gives you the full Cartesian product: 9 rows here, but $10^6 \times 10^6 = 10^{12}$ rows for two million-row tables. That is the classic "why is this query taking forever" bug.

Computer-science connection: Python sets

Python's `set` type implements finite sets directly, using a hash table so that membership `x in s` takes constant time on average (versus scanning a list).

PYTHON

```python
A = {1, 2, 3, 4}
B = {3, 4, 5}
print(A | B)         # union
print(A & B)         # intersection
print(A - B)         # difference
print(A ^ B)         # symmetric difference
print(3 in A, 7 in A)
print({1, 2} <= A)   # subset test
evens = {n for n in range(10) if n % 2 == 0}   # set-builder notation
print(evens)
```

```
{1, 2, 3, 4, 5}
{3, 4}
{1, 2}
{1, 2, 5}
True False
True
{0, 2, 4, 6, 8}
```

The operator choices are not accidental: `|` is bitwise or, `&` bitwise and, `^` bitwise xor. If you represent a subset of $\{0, \ldots, 63\}$ as a 64-bit integer whose bit i is 1 when i is in the set, then set union **is** bitwise or. Bitsets, Bloom filters, permission masks and chess engines' bitboards all use this. Note that elements must be hashable and immutable, which is why Python has `frozenset` for sets of sets, and why `{}` is an empty **dict**; the empty set is `set()`.



Russell's paradox

Can a set contain itself? Consider $R = \{\, S \mid S \notin S \,\}$, the set of all sets that are not members of themselves. Is $R \in R$? If it is, then by its own rule it is not; if it is not, then by its rule it is. Bertrand Russell found this contradiction in 1901, in the foundations Gottlob Frege had just finished building. The fix, modern axiomatic set theory (ZFC), forbids building a set from an unrestricted predicate: set-builder notation must always draw from an existing set, $\{\, x \in A \mid P(x) \,\}$. The same self-reference drives Turing's proof that the halting problem is undecidable.



Interactive Lab: Venn region painter

A three-set Venn diagram has eight regions. Type a set expression such as $(A \setminus B) \cup \overline{C}$ and the lab shades the matching regions; click regions to paint a shape and the lab finds an expression for it. A second panel shows the same regions as rows of a three-variable truth table, making the sets-equal-logic correspondence concrete.

Try it: Regions as truth-table rows · Open the Venn region painter

21.5.8. Where you'd use this

Everyday: surveys and inclusion–exclusion. Of 30 people at a meeting, 18 drink coffee, 15 drink tea and 7 drink both. Then $18 + 15 - 7 = 26$ drink at least one, so $30 - 26 = 4$ want neither: order 4 bottles of water, not 0 and not 30 minus 33. *Try it:* Coffee, tea, or neither .

Medicine: blood types are subsets. Red cells carry some subset of three antigens, A, B and the Rhesus antigen D. The eight blood types are exactly the $2^3 = 8$ subsets: O negative is $\varnothing$, AB positive is $\{A, B, D\}$. A transfusion of red cells is compatible when the donor's set is a **subset** of the recipient's, because the recipient's immune system attacks any antigen it lacks. So O negative ($\varnothing \subseteq$ everything) is the universal donor and AB positive the universal recipient; an A-positive patient ($\{A, D\}$) can receive O−, O+, A− or A+, the four subsets of $\{A, D\}$.

Computing: search and similarity. A search engine stores, for each word, the set of documents containing it (an **inverted index**). The query `cat dog` is the intersection of two such sets. Near-duplicate detection compares documents by the **Jaccard similarity** $|A \cap B| / |A \cup B|$ of their word sets: "the cat sat on the mat" and "the cat lay on the mat" share 4 of 6 distinct words, a similarity of $4/6 \approx 0.67$.

21.5.9. Exercises

1. Let $U = \{1, \dots, 10\}$, $A = \{1, 2, 3, 4, 5\}$, $B = \{4, 5, 6, 7\}$. Find $A \cup B$, $A \cap B$, $A \setminus B$, $B \setminus A$, $\overline{A}$, and $\overline{A \cup B}$.
2. True or false: (a) $\varnothing \in \{1, 2\}$; (b) $\varnothing \subseteq \{1, 2\}$; (c) $\{1\} \in \mathcal{P}(\{1, 2\})$; (d) $|\mathcal{P}(\mathcal{P}(\varnothing))| = 2$.
3. Write $\{1, 4, 9, 16, 25\}$ in set-builder notation.
4. List $\{0, 1\} \times \{a, b, c\}$. How many elements does $\{0, 1\}^8 = \{0, 1\} \times \cdots \times \{0, 1\}$ (8 factors) have, and what familiar object is each element?

5. Prove $A \setminus B = A \cap \overline{B}$ by element-chasing.
6. In a class of 30 students, 18 take Spanish, 14 take French, and 5 take neither. How many take both?

▼ **Answers**

1. $A \cup B = \{1,2,3,4,5,6,7\}$; $A \cap B = \{4,5\}$; $A \setminus B = \{1,2,3\}$; $B \setminus A = \{6,7\}$; $\overline{A} = \{6,7,8,9,10\}$; $\overline{A} \cup \overline{B} = \{8,9,10\}$. (Check De Morgan: $\overline{A} \cap \overline{B} = \{6,\dots,10\} \cap \{1,2,3,8,9,10\} = \{8,9,10\}$.)
2. (a) False: the elements are 1 and 2, and $\varnothing$ is neither. (b) True: the empty set is a subset of every set. (c) True: $\{1\} \subseteq \{1,2\}$. (d) True: $\mathcal{P}(\varnothing) = \{\varnothing\}$ has 1 element, so its power set has $2^1 = 2$ elements, namely $\varnothing$ and $\{\varnothing\}$.
3. $\{\, n^2 \mid n \in \mathbb{Z},\ 1 \le n \le 5 \,\}$.
4. $\{(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)\}$, six pairs. $\{0,1\}^8$ has $2^8 = 256$ elements, each an 8-tuple of bits: a byte.
5. $x \in A \setminus B \iff x \in A \wedge x \notin B \iff x \in A \wedge x \in \overline{B} \iff x \in A \cap \overline{B}$.
6. $30 - 5 = 25$ take at least one. $25 = 18 + 14 - |S \cap F|$, so $|S \cap F| = 7$.

> **US curriculum**
>
> Set notation, Venn diagrams and union/intersection appear in middle-school and **Algebra I** (grades 7–9), and again in **Algebra II** or **Statistics** in the probability unit ("compound events"). Power sets, Cartesian products and element-chasing proofs belong to college **Discrete Mathematics**.

21.6. Relations

21.6.1. The problem

"Is less than", "divides", "has the same remainder when divided by 3", "is a prerequisite of", "follows on social media", "is in the same connected component as". Each of these takes two things and answers yes or no. Some behave very differently from others: "same remainder" sorts things into groups, "prerequisite of" puts them in an order, "follows" does neither. To reason about these patterns, and to know which algorithms apply, we need precise names for the properties a relation can have.

21.6.2. Intuition

A relation is just a set of pairs: the pairs that are related. Draw the elements as dots, and draw an arrow from a to b whenever a is related to b. The properties are then visual: a loop at every dot (reflexive), every arrow paired with a reverse arrow (symmetric), every two-step path having a shortcut (transitive). That dots-and-arrows picture is a directed graph, the subject of Chapter 25.

Relation

A **(binary) relation** R from set A to set B is a subset of the Cartesian product: $R \subseteq A \times B$. We write $a \; R \; b$ for $(a, b) \in R$. A relation **on** A is a relation from A to itself. Example on $\{1, 2, 3\}$: "less than" is $\{(1, 2), (1, 3), (2, 3)\}$.

Properties of a relation on a set A

Property	Definition	Example
Reflexive	$\forall a \; (a \; R \; a)$	$\leq$, $=$, "divides"
Symmetric	$\forall a, b \; (a \; R \; b \to b \; R \; a)$	$=$, "is a sibling of"
Antisymmetric	$\forall a, b \; (a \; R \; b \wedge b \; R \; a \to a = b)$	$\leq$, $\subseteq$
Transitive	$\forall a, b, c \; (a \; R \; b \wedge b \; R \; c \to a \; R \; c)$	$<$, $\leq$, "is an ancestor of"

Antisymmetric is not the opposite of symmetric: a relation can be both (equality) or neither ("follows").

21.6.3. Equivalence relations

Equivalence relation and equivalence class

An **equivalence relation** is a relation that is reflexive, symmetric and transitive. It behaves like "is the same as, in some respect". Common notation is $a \sim b$.

The **equivalence class** of a is $[a] = \{ x \in A \mid x \sim a \}$, everything equivalent to a. The classes form a **partition** of A: they are nonempty, pairwise disjoint, and their union is all of A. Every element lives in exactly one class.

The canonical example: on the integers, say $a \sim b$ when a and b leave the same remainder on division by 3. Check the three properties: *Try it:* Same remainder mod 3 .

- Reflexive: a has the same remainder as itself.
- Symmetric: if a has the same remainder as b, then b has the same remainder as a.
- Transitive: if a matches b and b matches c, all three share one remainder.

So the integers split into exactly three classes, $[0] = \{\ldots, -3, 0, 3, 6, \ldots\}$, $[1] = \{\ldots, -2, 1, 4, 7, \ldots\}$ and $[2] = \{\ldots, -1, 2, 5, 8, \ldots\}$. This is the "congruence modulo 3" that Chapter 23 builds a whole arithmetic on.

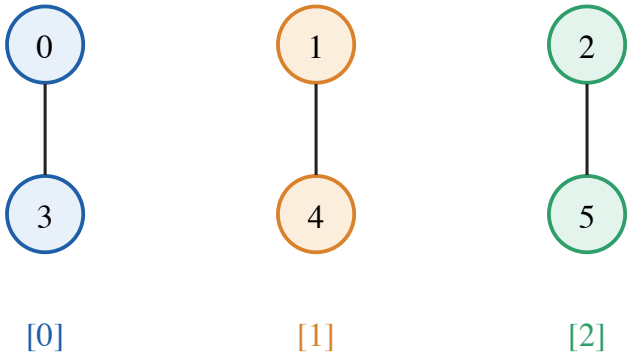


Figure 8. Same remainder mod 3 on $\{0, \ldots, 5\}$: three classes. Lines mean "related"; self-loops (reflexivity) are omitted.

Why does an equivalence relation always partition? Suppose two classes $[a]$ and $[b]$ share some element x. Then $x \sim a$ and $x \sim b$; by symmetry $a \sim x$, and by transitivity $a \sim b$. Now any $y \in [a]$ has $y \sim a \sim b$, so $y \in [b]$, and vice versa. So two classes either coincide or are disjoint. Reflexivity ensures every element is in at least one class (its own).

21.6.4. Partial orders



Partial order

A **partial order** is a relation that is reflexive, antisymmetric and transitive. It behaves like "comes before or equals". A set with a partial order is a **poset**. It is **partial** because two elements may be **incomparable**: neither $a \preceq b$ nor $b \preceq a$. If every pair is comparable it is a **total** (or **linear**) order, like $\leq$ on numbers.

"Divides" on $\{1, 2, 3, 4, 6, 12\}$ is a partial order: every number divides itself; if $a \mid b$ and $b \mid a$ for positive integers then $a = b$; and divisibility chains. But 4 and 6 are incomparable: neither divides the other. A **Hasse diagram** draws a poset economically: put larger elements higher, draw a line only for "immediate" relationships, and let reflexivity and transitivity be implied (12 is above 2 via 4 or 6, so no separate 2–12 line is drawn). *Try it:* Divisors of 12 as a Hasse diagram .

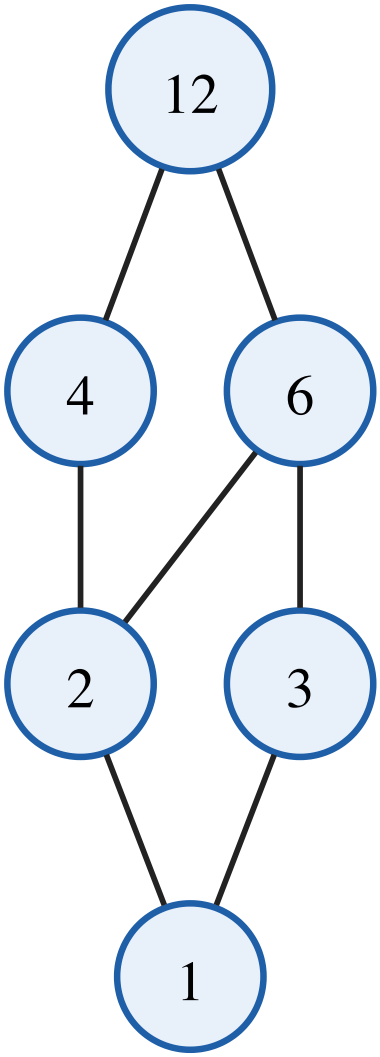


Figure 9. Hasse diagram of the divisors of 12 under "divides"

21.6.5. Worked example: classifying relations

Classify each relation on the given set. *Try it:* Classify “less than” .

(a) "$a \le b$" on the integers. Reflexive ($a \le a$). Not symmetric ($1 \le 2$ but $2 \not\le 1$). Antisymmetric ($a \le b$ and $b \le a$ force $a = b$). Transitive. So: a partial order, in fact a total order.

(b) "$a < b$" on the integers. Not reflexive ($1 \not< 1$). Antisymmetric (vacuously: $a < b \wedge b < a$ never happens). Transitive. Neither an equivalence relation nor a partial order; it is a **strict** order.

(c) "x and y are friends" on users. Symmetric, not reflexive by convention, and not transitive (your friend’s friend may be a stranger). Not an equivalence relation, which is why "friend groups" are not well-defined but "connected components" (friends-of-friends-of-… chains) are: taking the **transitive closure** repairs it.

The checks are mechanical enough to code. Note how each property is a quantified statement turned into `all()`:

PYTHON

```python
from itertools import product

def properties(S, R):
    """R is a set of pairs (a, b) meaning a R b, over the finite set S."""
    reflexive = all((a, a) in R for a in S)
    symmetric = all((b, a) in R for (a, b) in R)
    transitive = all((a, d) in R
                     for (a, b) in R for (c, d) in R if b == c)
    antisymmetric = all(a == b for (a, b) in R if (b, a) in R)
    return dict(reflexive=reflexive, symmetric=symmetric,
                transitive=transitive, antisymmetric=antisymmetric)

S = range(6)
same_mod3 = {(a, b) for a, b in product(S, S) if a % 3 == b % 3}
divides = {(a, b) for a, b in product(range(1, 13), repeat=2) if b % a == 0}
less_than = {(a, b) for a, b in product(S, S) if a < b}
print(properties(S, same_mod3))
print(properties(range(1, 13), divides))
print(properties(S, less_than))
```

```
{'reflexive': True, 'symmetric': True, 'transitive': True, 'antisymmetric': False}
{'reflexive': True, 'symmetric': False, 'transitive': True, 'antisymmetric': True}
{'reflexive': False, 'symmetric': False, 'transitive': True, 'antisymmetric': True}
```



Transitivity is the one people assume

Many natural relations fail transitivity, and code that assumes it breaks. "Approximately equal", `abs(a - b) < 1e-9`, is reflexive and symmetric but not transitive: $0 \approx 0.6 \times 10^{-9}$ and $0.6 \times 10^{-9} \approx 1.2 \times 10^{-9}$, but $0 \not\approx 1.2 \times 10^{-9}$. Using it as an equality in a hash table or a deduplication pass gives results that depend on processing order. Likewise "is preferred to" in voting can be cyclic (Condorcet's paradox), so it is not an order at all, and a comparator that is not transitive can make sort routines misbehave or throw (Java's `TimSort` reports "Comparison method violates its general contract!").

Computer-science connection: equivalence relations and partial orders in code

- **The `equals`/`hash` contract.** Java requires `equals` to be reflexive, symmetric and transitive: an equivalence relation. Hash tables bucket objects by equivalence class, and the rule "equal objects must have equal hashes" says the hash is constant on each class. Break symmetry (a subclass `equals` that compares extra fields) and `HashSet` gives inconsistent answers.
- **Union–find** (disjoint-set) data structures maintain the partition of an equivalence relation as pairs are added, used in Kruskal's algorithm, image segmentation and type unification.
- **Partial orders** describe dependencies: build targets, package versions, course prerequisites, and "happened-before" between events in a distributed system (vector clocks). A **topological sort** (Chapter 25) extends a partial order to a total order, giving a valid build or execution sequence. Two incomparable tasks can run in parallel.

21.6.6. Functions as special relations

A function is a relation with one extra rule. A relation $f \subseteq A \times B$ is a **function from A to B**, written $f : A \to B$, if every $a \in A$ is related to **exactly one** $b \in B$. We then write $f(a) = b$ instead of $(a, b) \in f$. In quantifier language: $\forall a \in A \; \exists! \, b \in B \; (a, b) \in f$.

The relation "x is a square root of y" on real numbers is not a function from y to x (4 has two square roots, -1 has none); the relation $\{(x, x^2)\}$ is a function. A Python `dict` is a finite function: each key maps to exactly one value. The full treatment of domain, codomain, range, injective, surjective and inverse functions is in Volume 1, Chapter 11 .



Function (as a relation)

A **function $f : A \to B$** is a relation from A to B in which each element of the **domain A** is paired with exactly one element of the **codomain B**. "Exactly one" has two halves: **at least one** (defined everywhere on A) and **at most one** (no input has two outputs).



Interactive Lab: relation property explorer

Draw arrows between up to eight dots. The lab lights up which of reflexive, symmetric, antisymmetric and transitive hold, and for any that fail it highlights a witness (a missing loop, an unreturned arrow, a two-step path with no shortcut). Buttons add the reflexive, symmetric or transitive **closure**; when the result is an equivalence relation the dots recolor by class, and when it is a partial order the lab redraws it as a Hasse diagram.

Try it: Open the relation explorer

21.6.7. Where you'd use this

Everyday: getting dressed is a partial order. Underwear before trousers, trousers before shoes, socks before shoes; shirt and socks are incomparable, so either order is fine. Any morning routine that respects the order is a **topological sort** of it, and recipes ("make the sauce while the pasta boils") are the same structure with parallel steps.

Physics: temperature exists because of an equivalence relation. The zeroth law of thermodynamics says that if bodies A and B are each in thermal equilibrium with C, they are in equilibrium with each other. Together with symmetry, that is transitivity, so "in thermal equilibrium with" is an equivalence relation. A thermometer works by naming the equivalence classes: a temperature is a label for a class.

Biology and chemistry: classification. "Is a subgroup of" in taxonomy (species, genus, family, ...) is a partial order whose Hasse diagram is a tree. "Has the same molecular formula" is an equivalence relation whose classes group isomers (ethanol and dimethyl ether are both $\mathrm{C_2H_6O}$).

Computing. Beyond the sidebar above: version constraints in a package manager compare versions by a total order, while "this commit is an ancestor of that one" in Git is a partial order (two branches' tips are incomparable until merged).

21.6.8. Exercises

1. On the set of all people, is "has the same birthday as" an equivalence relation? How many equivalence classes are there?
2. On $\{1, 2, 3\}$, give a relation that is symmetric and transitive but not reflexive.
3. Draw the Hasse diagram of $\mathcal{P}(\{a, b\})$ ordered by $\subseteq$. Which pairs are incomparable? *Try it:* Hasse diagram of P({a, b}) .
4. Which of these relations from $\{1, 2, 3\}$ to $\{x, y\}$ are functions? (a) $\{(1, x), (2, x), (3, y)\}$; (b) $\{(1, x), (1, y), (2, x), (3, x)\}$; (c) $\{(1, x), (3, y)\}$.

▼ **Answers**

1. Yes: reflexive, symmetric and transitive. There are 366 classes, one per calendar day including February 29 (assuming each day has at least one person born on it, which for all humanity is certainly true).
2. $R = \{(1, 1)\}$: symmetric and transitive (trivially), not reflexive because $(2, 2) \notin R$. Another: $R = \varnothing$. (This is why the "proof" that symmetry plus transitivity implies reflexivity, "aRb and bRa give aRa", fails: it assumes some b related to a exists.)
3. $\varnothing$ at the bottom; $\{a\}$ and $\{b\}$ in the middle; $\{a, b\}$ at the top; lines from $\varnothing$ to each singleton and from each singleton to $\{a, b\}$. The only incomparable pair is $\{a\}$ and $\{b\}$. The diagram is a diamond.
4. (a) Yes: each input has exactly one output. (b) No: 1 has two outputs. (c) No: 2 has no output (it is a **partial** function).

US curriculum

Relations appear in **Algebra I** as "a set of ordered pairs", with functions defined as relations where each input has one output (the "vertical line test", grade 8–9). Equivalence relations, partial orders and their properties are college **Discrete Mathematics**. Equivalence classes meet high-school students informally in modular ("clock") arithmetic.

21.7. Coda: satisfiability, the hardest easy question

Given a boolean formula, is there **any** assignment of T/F to its variables that makes it true? This is the **Boolean satisfiability problem**, SAT. It sounds like an exercise from earlier in the chapter, and for three variables it is: try all eight rows.

```python
from itertools import product

# (x or y) and (not x or z) and (not y or not z) and (not z or x)
clauses = [[1, 2], [-1, 3], [-2, -3], [-3, 1]]   # DIMACS-style literals

def satisfies(assign, clauses):
    return all(any(assign[abs(l)] == (l > 0) for l in c) for c in clauses)

def brute_force_sat(clauses, n):
    for bits in product([False, True], repeat=n):
        assign = dict(zip(range(1, n + 1), bits))
        if satisfies(assign, clauses):
            return assign
    return None

print(brute_force_sat(clauses, 3))
```

Try it: Brute-force this SAT instance ; Count every satisfying row .

```
{1: False, 2: True, 3: False}
```

The formula is written in **conjunctive normal form** (CNF): an "and" of **clauses**, each an "or" of **literals** (a variable or its negation). Literal `3` means z; `-3` means $\neg z$. Any formula can be converted to CNF with the laws from Table 2. Check the answer: $x = F, y = T, z = F$ gives $(F \vee T) \wedge (T \vee F) \wedge (F \vee T) \wedge (T \vee F) = T$. *Try it:* Convert a formula to CNF .

Computer-science connection: SAT solvers

Brute force needs 2^n rows. SAT was the first problem proved **NP-complete** (Cook, 1971): if you could solve every SAT instance quickly, you could solve every problem whose solutions can be checked quickly, which would settle the P versus NP question. No one expects a guaranteed-fast algorithm.

And yet modern SAT solvers (MiniSat, CaDiCaL, Kissat) routinely solve industrial instances with millions of variables, using clever search (**conflict-driven clause learning**: when a partial assignment fails, learn a new clause that rules out the reason, and backtrack non-chronologically). They are used to verify chip designs, find bugs in software (bounded model checking), resolve package dependencies (some package managers encode "which versions can coexist" as SAT), schedule, and even settle open problems in mathematics. SMT solvers such as Z3 extend SAT with integers, arrays and bit-vectors, and power many program verifiers. The worst case is exponential; the typical case, it turns out, is often not.

Try it: Package dependencies as SAT .



A 200-terabyte proof

The Boolean Pythagorean triples problem asked: can you color every positive integer red or blue so that no Pythagorean triple $a^2 + b^2 = c^2$ is all one color? In 2016 a SAT solver showed that you can do it up to 7,824 but not to 7,825. The proof certificate was about 200 terabytes, at the time the largest proof ever produced. Every step is a resolution of clauses of the kind in this chapter.

21.8. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Boolean logic
- Sets, relations and quantifiers

21.9. Mastery check

1. Build the truth table of $(p \to q) \wedge (q \to p)$ and identify the connective it equals.
2. State the contrapositive and the converse of "if a function is differentiable, it is continuous". Which one is guaranteed true, given that the original is true?
3. Simplify `if not (x >= 0 and x < len(a)) or a[x] is None:` so that every `not` is gone (change comparisons as needed).
4. Negate $\forall \varepsilon > 0\ \exists N\ \forall n > N\ (|a_n| < \varepsilon)$, pushing the negation all the way in.
5. Let $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$. Find $A \triangle B$, $|A \times B|$, and $|\mathcal{P}(A \cap B)|$.
6. Is "$a - b$ is an even integer" an equivalence relation on $\mathbb{Z}$? If so, describe its classes.
7. Explain in one sentence why `all([])` is `True` in Python and `any([])` is `False`.

Decision rule: All correct without hesitation → move on to Chapter 22, which puts this logic to work. Stuck on 1–3 → reread Section 21.2 and Section 21.3. Stuck on 4 or 7 → reread Section 21.4. Stuck on 5–6 → reread Section 21.5 and Section 21.6, then do their exercises.

▼ **Mastery check answers**

1. Rows TT, TF, FT, FF: $p \to q$ is T, F, T, T; $q \to p$ is T, T, F, T; their conjunction is T, F, F, T. That is the biconditional $p \leftrightarrow q$.
2. Contrapositive: "if a function is not continuous, it is not differentiable" (guaranteed true, equivalent to the original). Converse: "if a function is continuous, it is differentiable" (not guaranteed, and in fact false: $|x|$ at 0, see Volume 2, Chapter 17).
3. De Morgan: $\neg(x \geq 0 \wedge x < n) \equiv x < 0 \vee x \geq n$. So: `if x < 0 or x >= len(a) or a[x] is None:` . Short-circuiting means `a[x]` is only evaluated once the index is known to be in range.
4. $\exists \varepsilon > 0\ \forall N\ \exists n > N\ (|a_n| \geq \varepsilon)$. Each quantifier flips, the domain conditions ($\varepsilon > 0$, $n > N$) stay, and $<$ becomes $\geq$. In words: there is some tolerance that the sequence keeps violating, no matter how far out you go; i.e. a_n does not tend to 0 (see Volume 2, Chapter 16).
5. $A \triangle B = \{1, 4\}$; $|A \times B| = 3 \cdot 3 = 9$; $A \cap B = \{2, 3\}$, so $|\mathcal{P}(A \cap B)| = 2^2 = 4$.
6. Yes. Reflexive: $a - a = 0$ is even. Symmetric: if $a - b = 2k$ then $b - a = 2(-k)$. Transitive: if $a - b = 2k$ and $b - c = 2m$ then $a - c = 2(k + m)$. Two classes: the even integers and the odd integers.
7. `all` of an empty collection is an "and" of no statements, vacuously true because there is no element to violate it; `any` is an "or" of no statements, false because there is no witness.

US curriculum

Overall, this chapter maps to the logic unit of high-school **Geometry**, the sets-and-probability material of **Algebra I/II**, **AP Computer Science A** (boolean logic), and the first third of a typical college **Discrete Mathematics** course (often CS 2xx or MATH 2xx), which many CS majors take in their first year.

Chapter 22. Proof and Induction

A program that passes a thousand tests can still be wrong; a statement that holds for forty cases can fail on the forty-first. This chapter is about the one tool that settles a question for every case at once: proof. We start with why checking examples is not enough, take apart what a proof actually is, and then build a toolkit of techniques: direct proof, contrapositive, contradiction, cases, and counterexamples. The second half is devoted to mathematical induction, the technique that is to proof what recursion is to programming, and ends by using the same ideas to prove that real code (binary search, insertion sort) is correct.

Where this chapter sits

You need: integer arithmetic and divisibility words like even, odd and factor (Volume 1, Chapter 1), confident algebraic manipulation, especially expanding and factoring (Volume 1, Chapter 4 , Volume 1, Chapter 7), and the logic vocabulary of Chapter 21: implication, contrapositive, negation, and the quantifiers $\forall$ and $\exists$. If Chapter 21 is shaky, read its sections on the conditional and on negating quantifiers first; this chapter leans on them constantly.

This chapter unlocks: every chapter that says "we can show that". Most directly, Chapter 23 (where the Euclidean algorithm and RSA need correctness proofs), Chapter 24 (counting arguments and identities proved by induction), Chapter 25 (proofs about trees and paths), and Volume 1, Chapter 14 (where sum formulas reappear). For a programmer it also unlocks a habit: reasoning about code with invariants rather than only with test cases.

22.1. Why prove anything?

22.1.1. The problem

Here is a formula that looks like a prime-number generator:

$$f(n) = n^2 + n + 41.$$

Try it. $f(0) = 41$, prime. $f(1) = 43$, prime. $f(2) = 47$, prime. $f(3) = 53$, prime. $f(4) = 61$, prime. Keep going and you will find prime after prime. Leonhard Euler noticed this in 1772. If you were writing a unit test for the claim "$f(n)$ is always prime", you would probably stop at ten cases and declare victory. A more careful engineer might check a hundred.

Let the computer check:

```python
def is_prime(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:
        if n % d == 0:
            return False
        d += 1
    return True

for n in range(100):
    value = n * n + n + 41
    if not is_prime(value):
        print(f"n = {n}: {value} is composite")
        break
else:
    print("no counterexample below 100")
```

```
n = 40: 1681 is composite
```

Forty consecutive successes, and then $f(40) = 1681 = 41^2$. In hindsight it is obvious: when $n = 40$,

$$\begin{aligned} f(40) &= 40^2 + 40 + 41 \\ &= 40 \cdot 40 + 40 + 41 \\ &= 40(40 + 1) + 41 \qquad \text{(factor 40 out of the first two terms)} \\ &= 40 \cdot 41 + 41 \\ &= 41(40 + 1) = 41^2. \qquad \text{(factor out 41)} \end{aligned}$$

And $f(41) = 41^2 + 41 + 41 = 41 \cdot 43$ is plainly divisible by 41 too.

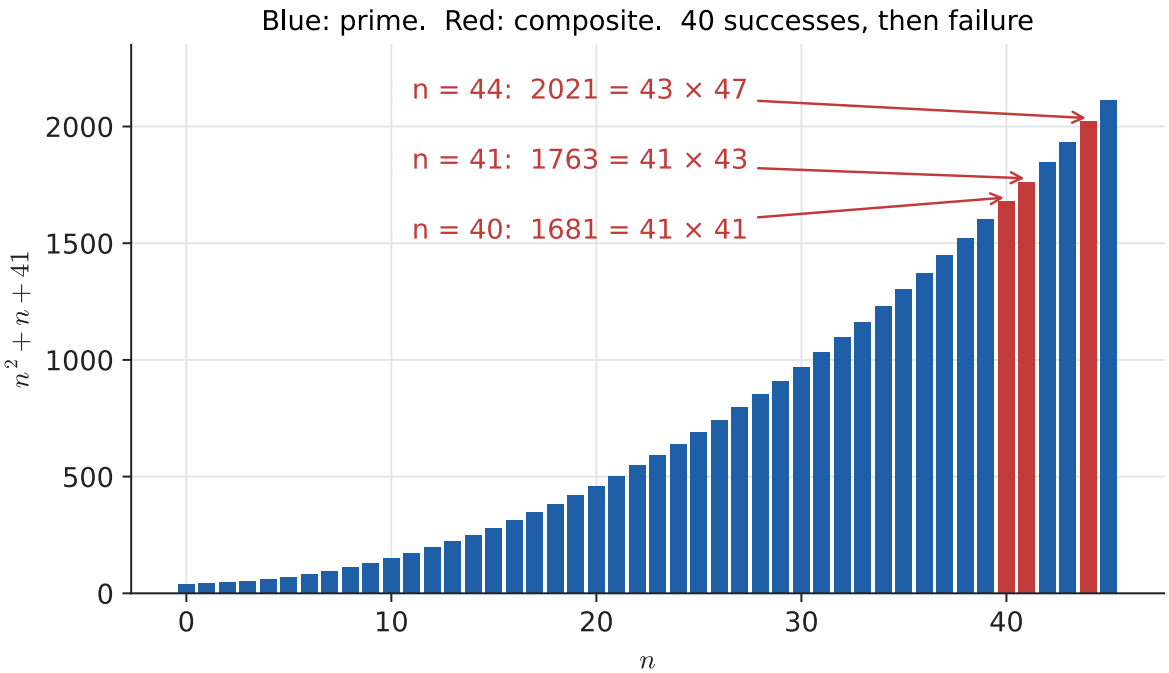


Figure 10. Forty successes, then failure: $n^2 + n + 41$ for $n = 0$ to 45. Blue bars are prime, red bars are composite.

22.1.2. Intuition: evidence versus certainty

Checking examples gives you **evidence**. A proof gives you **certainty**. The difference is the difference between "it works on my machine" and "it works on every input the type system allows".

Mathematics cares about statements of the form "for all n", and "all" is infinitely many. No finite amount of testing covers infinitely many cases. A proof is a finite argument that covers them all at once, because it reasons about a **general** n rather than any particular one.

That is also why a single failing case is decisive. "For all n, $f(n)$ is prime" is destroyed by one n that fails. Proving a universal claim is hard; disproving one needs just one counterexample.



Theorem, lemma, corollary, conjecture

- A **theorem** is a statement that has been proved. It is usually an important result.
- A **lemma** is a smaller proved statement used as a stepping stone towards a theorem (Greek *lemma*, "something taken for granted").
- A **corollary** is a statement that follows quickly from a theorem just proved.
- A **conjecture** is a statement believed to be true but not yet proved, such as "every even number greater than 2 is the sum of two primes" (Goldbach's conjecture).
- An **axiom** is a statement accepted without proof as a starting point, such as "for any integers a, b, $a + b = b + a$".
- A **proof** is a chain of statements, each justified by axioms, definitions, earlier results or logical rules, that ends at the claim being proved.

Testing shows the presence of bugs, not their absence

Edsger Dijkstra's famous line ("program testing can be used to show the presence of bugs, but never to show their absence") is the Euler polynomial in software form. A test suite samples the input space; a proof covers it. In practice engineers combine both: tests catch the bugs you can imagine, proofs (or proof-like reasoning with invariants and types) rule out whole classes of bugs you cannot. Property-based testing tools such as Python's Hypothesis sit in between: they search hard for counterexamples, which is exactly what you should do before attempting a proof.



Conjectures that failed very late

The Euler polynomial fails at 40. Some patterns hold for far longer. In 1919 George Pólya conjectured that, for every $N \geq 2$, at least half of the numbers up to N have an odd number of prime factors (counted with repetition). It was checked extensively by hand and was widely believed. The smallest counterexample, found by computer in 1980, is $N = 906{,}150{,}257$. No amount of hand testing would ever have found it.

22.1.3. Where you'd use this

Proof is not only for mathematicians. Wherever a mistake is expensive and the cases are too many to test, people reach for it.

- **Hardware.** In 1994 Intel's Pentium processor was found to divide slightly wrongly for rare inputs: 4,195,835 / 3,145,727 came out as 1.333739 instead of 1.333820. The flaw sat in five missing entries of a lookup table that random testing had not hit. Replacing the chips cost Intel a $475 million charge, and the company turned heavily to machine-checked proofs for its floating-point hardware afterwards.
- **Distributed systems.** Engineers at Amazon Web Services write precise specifications of protocols (in the language TLA+) and check them exhaustively. In one published case a model checker found a bug in a DynamoDB replication design whose shortest triggering sequence was 35 steps long, far deeper than any test or code review had reached.
- **Engineering and everyday reasoning.** A structural calculation showing that a beam carries its design load is a direct proof from stated assumptions (the material strengths, the load cases); the safety factor exists because the assumptions, not the logic, are uncertain. In daily life the same skill is spotting when an argument jumps from "it worked the last five times" to "it always works".

US curriculum

Most US students first meet formal proof in high-school **Geometry** (typically grade 9 or 10), in the form of "two-column proofs" (statements on the left, reasons on the right). Algebraic proof and induction appear in some **Algebra II** and **Precalculus** courses, especially honors tracks, but are mostly taught in a college **Discrete Mathematics** course, often the first-year course for computer-science majors.

22.2. Anatomy of a proof

22.2.1. The problem

Claim: *the sum of two odd numbers is even.* You "know" this, and you can check that $3 + 5 = 8$ and $7 + 11 = 18$. But how would you **prove** it, for every pair of odd numbers?

The first obstacle is not cleverness; it is precision. To prove anything about odd numbers you need a definition of "odd" that you can compute with. "Not divisible by 2" is true but hard to manipulate. "Ends in 1, 3, 5, 7 or 9" depends on base ten. The definition mathematicians use is algebraic.

Even, odd, divides

- $\mathbb{Z}$ (blackboard-bold Z, from German *Zahlen*, "numbers") is the set of all integers $\ldots, -2, -1, 0, 1, 2, \ldots$. $\mathbb{N}$ is the natural numbers; in this chapter $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- An integer n is **even** if $n = 2k$ for some integer k.
- An integer n is **odd** if $n = 2k + 1$ for some integer k.
- For integers $a \neq 0$ and b, we say a **divides** b, written $a \mid b$, if $b = ak$ for some integer k. So $3 \mid 12$ (because $12 = 3 \cdot 4$) but $3 \nmid 13$. Note that $a \mid b$ is a *statement* (true or false), not a number; do not confuse it with a/b.
- Every integer is either even or odd, and never both. (This follows from division with remainder: dividing by 2 leaves remainder 0 or 1.)

22.2.2. Intuition

A proof is like a program that takes the hypotheses as input and produces the conclusion as output, where each line must "type-check" against a definition, an earlier result or a rule of logic. The definitions are the API. A proof that says "odd numbers are, like, not even, so..." does not compile. A proof that says "let $a = 2j + 1$" does.

Here is the proof of the claim, written in the standard style.

Claim. If a and b are odd integers, then $a + b$ is even.

Proof. Let a and b be odd integers. By the definition of odd, there are integers j and k with $a = 2j + 1$ and $b = 2k + 1$. Then

$$\begin{aligned} a + b &= (2j + 1) + (2k + 1) \\ &= 2j + 2k + 2 \qquad \text{(rearrange)} \\ &= 2(j + k + 1). \qquad \text{(factor out 2)} \end{aligned}$$

Since $j + k + 1$ is an integer, $a + b$ is two times an integer, so $a + b$ is even by definition. ■

22.2.3. The parts, named

Look at the structure:

1. **Statement of the claim**, as an implication "if (hypotheses) then (conclusion)".
2. **Assume the hypotheses** for arbitrary objects: "Let a and b be odd integers." Note the word *arbitrary*: we did not pick $a = 3$. Whatever we conclude about this unnamed a therefore holds for every odd a. This is how a finite argument covers infinitely many cases.
3. **Unpack definitions** into algebra: $a = 2j + 1$.
4. **Manipulate** towards the form the conclusion needs: $2(\text{integer})$.
5. **Repack** into the conclusion's definition: "so $a + b$ is even".

6. **Mark the end**: ■, or "QED".

Notice one subtle point: we used *different* letters j and k. Writing $a = 2k + 1$ and $b = 2k + 1$ would secretly assume $a = b$, and the "proof" would only cover adding a number to itself.



QED and the tombstone

QED abbreviates the Latin *quod erat demonstrandum*, "which was to be demonstrated". Modern texts usually end a proof with a small filled square ■, called a **tombstone** or **Halmos symbol** after the mathematician Paul Halmos who popularised it. Either mark says "the proof is complete; you can stop checking here".



Proof vocabulary you will see

- **"Let ..."** introduces an arbitrary object with stated properties.
- **"Suppose ..."** or **"Assume ..."** introduces a hypothesis, possibly temporary (as in proof by contradiction).
- **"Without loss of generality (WLOG)"** means "the other case is identical by symmetry, so we treat only this one".
- **"It suffices to show ..."** means "if we prove this simpler thing, the original claim follows".
- **"Hence"**, **"thus"**, **"therefore"**, **"it follows that"** mark a step derived from earlier lines.
- **"if and only if"** (iff, ⇔) requires proving two directions: $P \Rightarrow Q$ and $Q \Rightarrow P$.

Two-column proofs in high-school geometry

US **Geometry** courses (grades 9–10) teach the same anatomy in table form: a "Statements" column and a "Reasons" column, where each reason must be a given, a definition, a postulate or a previously proved theorem. It is a good discipline. Mathematicians write proofs in paragraphs, but every sentence should still have an implicit "reason" you could write in the right-hand column.

22.3. Direct proof

22.3.1. The method

The sum-of-odds proof was a **direct proof**: to prove $P \Rightarrow Q$, assume P and derive Q by a chain of valid steps. It is the default technique; reach for it first.

Example 6. Procedure — Direct proof of "if P then Q"

Input: a claim of the form "for all x, if $P(x)$ then $Q(x)$".

Steps:

1. Write "Let x be arbitrary and suppose $P(x)$."
2. Replace every term in $P(x)$ by its definition (introduce new variables for any "there exists").
3. Write down what $Q(x)$ means by definition: this is your target form.
4. Manipulate the information from step 2 until it matches the target form from step 3. If stuck, work backwards from the target on scrap paper, then write the steps forwards.
5. Conclude "therefore $Q(x)$" and mark the end.

Output: a proof that $P(x) \Rightarrow Q(x)$ for every x.

22.3.2. Worked example 1: the square of an odd number is odd

Claim. If n is odd, then n^2 is odd.

Target form: $n^2 = 2(\text{integer}) + 1$.

Proof. Let n be odd, so $n = 2k + 1$ for some integer k. Then

$$\begin{aligned} n^2 &= (2k+1)^2 \\ &= 4k^2 + 4k + 1 && \text{(expand: } (a+b)^2 = a^2 + 2ab + b^2\text{)} \\ &= 2(2k^2 + 2k) + 1. && \text{(factor 2 from the first two terms)} \end{aligned}$$

Since $2k^2 + 2k$ is an integer, n^2 is odd. ■

22.3.3. Worked example 2: divisibility is transitive

Claim. For integers a, b, c with $a, b \neq 0$: if $a \mid b$ and $b \mid c$, then $a \mid c$.

Target form: $c = a \cdot (\text{integer})$.

Proof. Suppose $a \mid b$ and $b \mid c$. By definition there are integers r and s with $b = ar$ and $c = bs$. Substitute the first into the second:

$$\begin{aligned} c &= bs \\ &= (ar)s && \text{(substitute } b = ar\text{)} \\ &= a(rs). && \text{(associativity of multiplication)} \end{aligned}$$

Since rs is an integer, $a \mid c$. ■

22.3.4. Worked example 3: a geometric direct proof of Gauss's sum

Direct proofs need not be algebra. Here is a claim we will prove again by induction later:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}.$$

Draw the sum as a staircase of squares: one square in the first row, two in the second, up to n in the last. Take a second copy, rotate it by 180°, and fit it against the first. The two staircases tile an $n \times (n+1)$ rectangle. So twice the sum is $n(n+1)$, and the sum is $n(n+1)/2$.

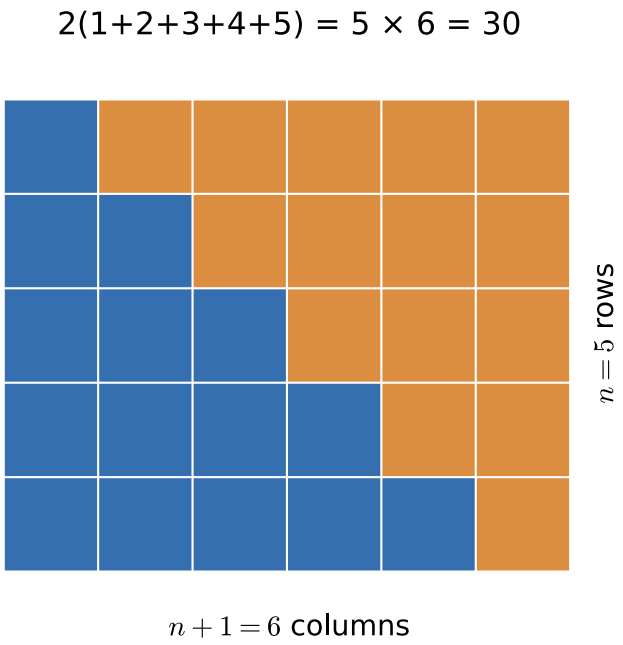


Figure 11. Two copies of the staircase 1 + 2 + … + n form an n by (n + 1) rectangle (shown for n = 5).

The algebraic version of the same idea: write the sum forwards and backwards, and add column by column.

$$\begin{aligned} S &= 1 + 2 + \cdots + (n-1) + n \\ S &= n + (n-1) + \cdots + 2 + 1 \\ 2S &= (n+1) + (n+1) + \cdots + (n+1) + (n+1) \qquad \text{(each column sums to } n+1\text{)} \\ 2S &= n(n+1) \qquad \text{(there are } n \text{ columns)} \\ S &= \frac{n(n+1)}{2}. \end{aligned}$$



Gauss and the schoolroom sum

The story (probably embellished in the retelling) is that the young Carl Friedrich Gauss, set the task of adding 1 to 100 to keep him quiet, wrote 5050 on his slate almost at once, having paired 1 with 100, 2 with 99, and so on: 50 pairs each summing to 101. Whether or not it happened, the pairing trick is exactly the forwards-and-backwards argument above.



Working backwards, writing forwards

A common mistake is to start from the conclusion and manipulate until you reach something true, such as

$$n^2 \text{ is odd} \Rightarrow \ldots \Rightarrow 1 = 1, \text{ which is true, so done.}$$

This proves nothing: you can derive a true statement from a false one ($-1 = 1$ squares to $1 = 1$). Working backwards is a fine way to **discover** a proof, but the written proof must run from hypotheses to conclusion, and every step must be valid in that direction.



Proof by example is not proof

"$3 + 5 = 8$, which is even, so the sum of two odds is even" is not a proof of a universal statement. Examples are excellent for understanding and for finding the right algebra, but they belong on scrap paper. (An example **is** a complete proof of an **existence** claim, such as "there is an even prime": exhibit 2, done.)

22.4. Proof by contrapositive

22.4.1. The problem

Claim: *if n^2 is even, then n is even.*

Try a direct proof. Suppose $n^2 = 2k$. Then $n = \sqrt{2k}$... and now what? Square roots are awkward, and there is no obvious way to show $\sqrt{2k}$ is an even integer. The direct route is stuck.

22.4.2. Intuition

Recall from Chapter 21 that an implication $P \Rightarrow Q$ is logically equivalent to its **contrapositive** $\neg Q \Rightarrow \neg P$. "If it rained, the ground is wet" says exactly the same as "if the ground is dry, it did not rain". Both statements rule out the same single situation: P true and Q false.

So instead of proving "n^2 even $\Rightarrow$ n even" we may prove "n not even $\Rightarrow$ n^2 not even", that is, "n odd $\Rightarrow$ n^2 odd". And we have **already proved that** in the previous section. The hypothesis "n is odd" gives us something concrete to work with ($n = 2k + 1$), whereas "n^2 is even" gave us a square root.



Contrapositive, converse, inverse

For the implication $P \Rightarrow Q$ ("if P then Q"):

- **Contrapositive:** $\neg Q \Rightarrow \neg P$. Always equivalent to the original.
- **Converse:** $Q \Rightarrow P$. **Not** equivalent in general.
- **Inverse:** $\neg P \Rightarrow \neg Q$. **Not** equivalent in general (it is the contrapositive of the converse).

The symbol ¬ means "not". See Chapter 21 for the truth tables.

22.4.3. Worked example 1

Claim. For every integer n, if n^2 is even then n is even.

Proof (by contrapositive). We prove the equivalent statement: if n is odd then n^2 is odd. Let n be odd, so $n = 2k + 1$. Then $n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$, which is odd. Hence, by contraposition, if n^2 is even then n is even. ■

We will use this as a lemma in the proof that $\sqrt{2}$ is irrational.

22.4.4. Worked example 2: a pigeonhole-flavoured claim

Claim. If x and y are real numbers with $x + y \geq 100$, then $x \geq 50$ or $y \geq 50$.

Directly, it is not clear where to start: "or" conclusions are awkward because you do not know which half holds.

The contrapositive negates both sides. By De Morgan's law, $\neg(x \geq 50 \vee y \geq 50)$ is $x < 50 \wedge y < 50$. So the contrapositive is: *if* $x < 50$ *and* $y < 50$, *then* $x + y < 100$.

Proof (by contrapositive). Suppose $x < 50$ and $y < 50$. Adding the two inequalities,

$$x + y < 50 + 50 = 100.$$

So $x + y < 100$, which is the negation of the hypothesis. ■

22.4.5. When to choose the contrapositive

Use the contrapositive when:

- the conclusion Q is a **negative** ("is not divisible", "is irrational") or an **"or"**, so that $\neg Q$ is a positive, concrete fact;
- the hypothesis P is hard to unpack (a square, a square root, an inequality in the wrong direction), while $\neg Q$ is easy.



Proving the converse by mistake

The classic error is to prove $Q \Rightarrow P$ and claim you proved $P \Rightarrow Q$. For example, to show "if n^2 is even then n is even" you might write "let $n = 2k$; then $n^2 = 4k^2$ is even". That proves the **converse**: "if n even then n^2 even". True, but not what was asked. Always write the claim you are proving at the top, and check that your first line assumes **its** hypothesis.

Contrapositives in code review

Guard clauses are contrapositives. A function whose contract is "if the input is valid, return a result" is often written as "if the input is not valid, return early / raise". When you reason "this branch can only be reached if `user is not None`", you are using the contrapositive of "if `user is None` we returned at line 3". Static analysers and type checkers do the same reasoning when they narrow a variable's type after an `if` test.

22.5. Proof by contradiction

22.5.1. The problem

Is $\sqrt{2}$ a fraction? Your calculator says $\sqrt{2} \approx 1.41421356$, and decimals that stop are fractions, but the calculator's display stops only because it ran out of digits. The claim "$\sqrt{2}$ is **irrational**" is a negative: it says there is **no** pair of integers p, q with $\sqrt{2} = p/q$. How can you prove that something does not exist, when you cannot check all the infinitely many pairs?

22.5.2. Intuition

Assume the opposite, and follow the consequences until something impossible happens. If assuming $\neg S$ leads to a statement that is definitely false (such as "this number is both even and odd", or "$1 < 0$"), then $\neg S$ cannot be true, so S is true.

This is how a debugger thinks: "suppose this pointer really were null here; then line 40 would have crashed; but line 40 ran; so the pointer was not null". It is also how a detective thinks.



Rational, irrational, lowest terms

- A real number is **rational** if it equals p/q for some integers p, q with $q \neq 0$. The set of rationals is written $\mathbb{Q}$ (for *quotient*).
- A real number that is not rational is **irrational**. Examples: $\sqrt{2}$, π, e.
- A fraction p/q is in **lowest terms** if p and q have no common factor greater than 1. Every rational number can be written in lowest terms (divide top and bottom by their greatest common divisor; see Chapter 23).
- A **contradiction** is a statement that is false by its form, such as $R \wedge \neg R$ ("R and not R").

22.5.3. The method

Example 7. Procedure — Proof by contradiction

Input: a statement S to prove.

Steps:

1. Write "Suppose, for contradiction, that $\neg S$." State $\neg S$ precisely (use Chapter 21 rules to negate quantifiers and connectives).
2. Unpack definitions in $\neg S$ to get concrete objects and equations.
3. Derive consequences until you reach a contradiction: some statement R together with its negation $\neg R$, or a known falsehood.
4. Conclude "This is a contradiction, so $\neg S$ is false, and therefore S is true."

Output: a proof of S.

22.5.4. Worked example 1: $\sqrt{2}$ is irrational

Claim. $\sqrt{2}$ is irrational.

Proof. Suppose, for contradiction, that $\sqrt{2}$ is rational. Then we can write $\sqrt{2} = p/q$ where p, q are positive integers and the fraction is in lowest terms, so p and q have no common factor greater than 1.

$$\begin{aligned} \sqrt{2} &= \frac{p}{q} \\ 2 &= \frac{p^2}{q^2} && \text{(square both sides)} \\ 2q^2 &= p^2. && \text{(multiply both sides by } q^2\text{)} \end{aligned}$$

So p^2 is two times an integer, i.e., p^2 is even. By the lemma proved by contrapositive in Section 22.4, p is even. Write $p = 2r$ and substitute:

$$\begin{aligned} 2q^2 &= (2r)^2 = 4r^2 \\ q^2 &= 2r^2. && \text{(divide both sides by 2)} \end{aligned}$$

So q^2 is even, and by the same lemma q is even.

Now p and q are both even, so 2 is a common factor greater than 1. This contradicts our choice of p/q in lowest terms. Hence $\sqrt{2}$ is not rational. ∎

Notice how the pieces of the chapter fit together: a direct proof (odd squared is odd) gave a contrapositive lemma (n^2 even implies n even), which powered a contradiction. Real mathematics is built from lemmas like this, much as software is built from tested functions.



The secret of the Pythagoreans

The discovery that the diagonal of a unit square is not a ratio of whole numbers is attributed to the Pythagorean school in ancient Greece, whose philosophy held that "all is number" (meaning whole numbers and their ratios). Legend says Hippasus of Metapontum was drowned at sea for revealing it. The legend is almost certainly fiction, but the shock was real: Greek mathematics afterwards treated geometric magnitudes and numbers as separate things for centuries.

22.5.5. Worked example 2: there are infinitely many primes

This proof, from Euclid's *Elements* (around 300 BCE), is one of the oldest and most admired in mathematics.



Prime

A **prime** is an integer $p > 1$ whose only positive divisors are 1 and p. The first few are $2, 3, 5, 7, 11, 13$. An integer $n > 1$ that is not prime is **composite**. A fact we will prove by strong induction in Section 22.10: every integer $n > 1$ has at least one prime divisor.

Claim. There are infinitely many primes.

Proof. Suppose, for contradiction, that there are only finitely many primes, and list all of them: $p_1, p_2, \dots, p_k$. Consider

$$N = p_1 p_2 \cdots p_k + 1.$$

$N > 1$, so N has some prime divisor p. Since our list contains every prime, $p = p_i$ for some i. Then p divides the product $p_1 p_2 \cdots p_k$, and p divides N. So p divides their difference:

$$N - p_1 p_2 \cdots p_k = 1.$$

But no prime divides 1 (primes are greater than 1). This is a contradiction, so the list could not have contained every prime: there are infinitely many. ■



Euclid's N need not be prime

A very common misreading is "the product of the first k primes, plus one, is always prime". The proof does not say that. It says N has a prime factor **not in the list**. Indeed

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30031 = 59 \cdot 509,$$

which is composite, but its prime factors 59 and 509 are not among 2, 3, 5, 7, 11, 13, exactly as the proof predicts.

22.5.6. Alternative approaches: contradiction versus contrapositive

Every proof by contrapositive can be rewritten as a contradiction ("suppose P and $\neg Q$; ...; then $\neg P$; contradiction"). So which should you use?

- If your argument goes cleanly from $\neg Q$ to $\neg P$, **write it as a contrapositive**. It is clearer, because every intermediate statement is actually true and can be reused.
- Use **contradiction** when the statement is not naturally an implication ("$\sqrt{2}$ is irrational", "there are infinitely many primes"), or when you need to use both P and $\neg Q$ together.
- Try a **direct** proof first when the hypothesis gives you something concrete to compute with.

Contradiction in type systems and solvers

SAT and SMT solvers (see Chapter 21) prove a property by contradiction: to show a formula F always holds, they search for an assignment satisfying $\neg F$. If the search proves none exists ("UNSAT"), F is valid. Tools like Microsoft's Z3 are used in exactly this way to verify that compiler optimisations, cloud access policies or smart contracts cannot reach a bad state. In type theory, the "empty type" (Rust's `!`, TypeScript's `never`, Haskell's `Void`) plays the role of a contradiction: a value of that type can never exist, so code that would have one is unreachable.

22.6. Proof by cases

22.6.1. The problem

Claim: *for every integer n, $n^2 + n$ is even.*

You cannot write $n = 2k$ (that assumes n is even) or $n = 2k + 1$ (that assumes n is odd). But every integer is one or the other. So handle each separately.

22.6.2. The method

Example 8. Procedure — Proof by cases

Input: a claim S about objects that fall into a known finite list of types.

Steps:

1. Choose cases $C_1, C_2, \ldots, C_m$ and justify that they are **exhaustive** (every object falls into at least one). They need not be exclusive, but usually are.
2. For each case C_i, assume C_i and prove S.
3. Conclude that S holds in all cases.

Output: a proof of S.

22.6.3. Worked example 1

Claim. For every integer n, $n^2 + n$ is even.

Proof. Every integer is even or odd.

Case 1: n is even. Write $n = 2k$. Then

$$n^2 + n = 4k^2 + 2k = 2(2k^2 + k),$$

which is even.

Case 2: n is odd. Write $n = 2k + 1$. Then

$$\begin{aligned} n^2 + n &= (4k^2 + 4k + 1) + (2k + 1) \\ &= 4k^2 + 6k + 2 \\ &= 2(2k^2 + 3k + 1), \end{aligned}$$

which is even. In both cases $n^2 + n$ is even. ■

22.6.4. Alternative approach

Factor first: $n^2 + n = n(n + 1)$. Of two consecutive integers, one is even, and an even number times anything is even. This is shorter, but notice that "one of two consecutive integers is even" is itself a proof by cases, just hidden. When you find a factorisation like this, use it: it is clearer and generalises (the product of three consecutive integers is divisible by 3, for the same reason).

22.6.5. Worked example 2: last digits of squares

Claim. No perfect square ends in the digit 2, 3, 7 or 8.

Proof. The last digit of n^2 depends only on the last digit of n (because $n = 10a + d$ gives $n^2 = 100a^2 + 20ad + d^2$, and the first two terms are multiples of 10). So there are ten cases, $d = 0, 1, \dots, 9$:

d	0	1	2	3	4	5	6	7	8	9
d^2	0	1	4	9	16	25	36	49	64	81
last digit	0	1	4	9	6	5	6	9	4	1

The possible last digits are 0, 1, 4, 5, 6, 9. None is 2, 3, 7 or 8. ■

This is a perfectly rigorous proof even though it is just a table, because the ten cases are **exhaustive**. That is the key difference from "proof by example": here the examples provably cover everything.



Non-exhaustive cases

Proofs by cases fail silently when a case is missing. "Case $x > 0$ … Case $x < 0$ …" forgets $x = 0$. "Case n prime … Case n composite" forgets $n = 1$ (which is neither) and $n \leq 0$. Before writing the cases, write the sentence that justifies exhaustiveness.

Cases and exhaustive pattern matching

A `match` statement in Rust, Swift, Scala or Haskell is a proof by cases, and the compiler checks the exhaustiveness condition for you: leave out a variant of an enum and you get an error. Python 3.10's `match` does not enforce this, which is why a final `case _: raise AssertionError` arm is good practice. The bug class "forgot a case" (an unhandled state in a state machine, a missing `else`) is precisely the failure mode of a non-exhaustive proof.



A proof with 1,936 cases

The Four Colour Theorem (any map can be coloured with four colours so that neighbouring regions differ) was proved in 1976 by Kenneth Appel and Wolfgang Haken as a proof by cases: they reduced the problem to 1,936 configurations and checked them by computer. It was the first major theorem whose proof no human could check by hand, and it caused real controversy. In 2005 Georges Gonthier verified the whole proof in the Coq proof assistant, which settled most remaining doubts. You will meet graph colouring in Chapter 25.

22.7. Counterexamples

22.7.1. The problem

Is the following true? *If p is prime, then $2^p - 1$ is prime.* Check: $2^2 - 1 = 3$, $2^3 - 1 = 7$, $2^5 - 1 = 31$, $2^7 - 1 = 127$. All prime. Should you try to prove it?

22.7.2. Intuition

Before investing in a proof, spend a few minutes trying to **break** the claim. Recall from Chapter 21 how a universal statement negates:

$$\neg\big(\forall x\ P(x)\big) \ \equiv\ \exists x\ \neg P(x).$$

"It is not true that every x has property P" means "some x lacks property P". So to disprove a "for all" claim you need exactly one **counterexample**: a specific object for which the claim fails. No general argument is needed.

Try the next prime: $p = 11$ gives $2^{11} - 1 = 2047 = 23 \times 89$. The claim is false. (Numbers of the form $2^p - 1$ that **are** prime are called Mersenne primes; the largest known primes are of this form.)

22.7.3. The method

Example 9. Procedure — Hunting for a counterexample

Input: a universal claim "for all x in D, $P(x)$".

Steps:

1. Try the **edge cases** first: 0, 1, negative numbers, the smallest allowed value, equal inputs, the empty set or empty list.
2. Try **special** values: 2 (the only even prime), perfect squares, numbers just below powers of ten, fractions between 0 and 1 (where squaring makes things smaller).
3. If those pass, **search systematically**, by hand or with a short program.
4. If you find x_0 with $\neg P(x_0)$, verify it carefully and state it: "The claim is false: take $x = x_0$; then ...".
5. If a long search finds nothing, that is evidence (not proof) that the claim is true. Ask **why** it keeps working; the reason is often the proof.

Output: a counterexample, or a well-founded belief that it is time to attempt a proof.

22.7.4. Worked example

Claim. For all real x, $x^2 \geq x$.

Edge cases: $x = 0$ gives $0 \geq 0$, true. $x = 1$ gives $1 \geq 1$, true. $x = -3$ gives $9 \geq -3$, true. Special values: a fraction between 0 and 1. $x = \frac{1}{2}$ gives $\frac{1}{4} \geq \frac{1}{2}$, **false**.

Disproof. The claim is false: for $x = \frac{1}{2}$, $x^2 = \frac{1}{4} < \frac{1}{2} = x$. ■

One clean counterexample is the whole disproof. You do not need to describe all the failures (here, every x with $0 < x < 1$), though doing so can be illuminating.



Counterexamples to existence claims, and to implications

- An **existence** claim ("there is an x with $P(x)$") cannot be disproved by a counterexample. Its negation is "for all x, not $P(x)$", which needs a general proof.
- A counterexample to "if $P(x)$ then $Q(x)$" must satisfy P **and** fail Q. An x for which P is false is not a counterexample: the implication is vacuously true there (see Chapter 21).



Euler wrong again

In 1769 Euler conjectured that you need at least k k-th powers to sum to another k-th power (generalising Fermat's Last Theorem). It stood for nearly two centuries, until in 1966 L. J. Lander and T. R. Parkin found, by computer search, a counterexample whose paper is one of the shortest ever published:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5.$$

Four fifth powers summing to a fifth power. One line disproved a conjecture by one of history's greatest mathematicians.



Counterexample hunter

Type a claim as a Python predicate (for example `lambda n: is_prime(n*n + n + 41)` or `lambda p: not is_prime(p) or is_prime(2**p - 1)`) and a search range. The lab tests every value, plots passes and failures like the Euler-polynomial chart above, and highlights the first counterexample. Try to guess, before running, how far each claim survives.

Try it: Hunt counterexamples in Python

22.7.5. Where you'd use this

Contradiction, on your desk. A4 paper is designed so that halving a sheet gives the same shape, which forces the ratio of the sides to be $\sqrt{2}$. Because $\sqrt{2}$ is irrational, no sheet with whole-millimetre sides can have exactly that ratio. The standard picks 210 mm by 297 mm, and $297/210 = 1.41428\ldots$ against $\sqrt{2} = 1.41421\ldots$: off by about 0.005%, which is the best the proof allows at that precision.

Cases, as a quick check. The last-digit table proves that a number ending in 2, 3, 7 or 8 cannot be a perfect square. So 7,423,898 is not a square, with no square root needed. Fast perfect-square tests in big-number libraries use the same idea with remainders mod 64 or mod 256 (there are only 44 possible remainders mod 256 for squares, out of 256) to reject most non-squares before computing anything expensive.

Infinitely many primes, in your browser. Every HTTPS connection that uses RSA relies on a fresh supply of large primes; Euclid's proof guarantees the supply never runs out, and the prime number theorem (Chapter 23) says how dense it is: about one in $\ln 2^{1024} \approx 710$ numbers near 2^{1024} is prime.

Counterexamples, at work. A single failing case refutes a universal claim, so a bug report with one reproducible input is worth more than a hundred "works for me" reports. Medicine works the same way: one well-documented serious reaction is enough to withdraw the claim that a drug is safe for everyone.

22.8. Exercises: direct, contrapositive, contradiction, cases

1. **Recognition.** For each claim, name the technique you would try first and say why: (a) If $a \mid b$ then $a \mid 3b$. (b) If n^2 is odd then n is odd. (c) $\sqrt{3}$ is irrational. (d) For every integer n, $n^3 - n$ is divisible by 3. (e) Every prime is odd.
2. **Application.** Prove that the product of two odd integers is odd.
3. **Application.** Prove: if n^2 is odd then n is odd.
4. **Application.** Prove that $n^3 - n$ is divisible by 3 for every integer n.
5. **Problem solving.** Prove that $\sqrt{3}$ is irrational. (You will first need a lemma: if $3 \mid p^2$ then $3 \mid p$. Prove it by contrapositive, with cases.)
6. **Problem solving.** Disprove: "for integers a, b, c with $a \neq 0$, if $a \mid bc$ then $a \mid b$ or $a \mid c$."
7. **Problem solving.** Prove that there is no largest even integer.

▼ **Answers**

1. (a) Direct: $b = ak$ so $3b = a(3k)$. (b) Contrapositive: "if n even then n^2 even" is easy to compute. (c) Contradiction: the statement is a negative ("not a fraction"). (d) Cases on the remainder of n modulo 3, or factor $n^3 - n = (n-1)n(n+1)$. (e) It is false: look for a counterexample (2).
2. Let $a = 2j + 1$, $b = 2k + 1$. Then $ab = 4jk + 2j + 2k + 1 = 2(2jk + j + k) + 1$, which is odd.
3. Contrapositive: if n is even, $n = 2k$, then $n^2 = 4k^2 = 2(2k^2)$ is even. So if n^2 is odd, n is odd.
4. $n^3 - n = n(n^2 - 1) = (n-1)n(n+1)$, a product of three consecutive integers. Every integer leaves remainder 0, 1 or 2 when divided by 3. If $n = 3k$, then $3 \mid n$. If $n = 3k + 1$, then $n - 1 = 3k$. If $n = 3k + 2$, then $n + 1 = 3k + 3 = 3(k + 1)$. In every case one factor is divisible by 3, so the product is.
5. **Lemma** (contrapositive): if $3 \nmid p$ then $3 \nmid p^2$. If $3 \nmid p$, then $p = 3m + 1$ or $p = 3m + 2$. Case 1: $p^2 = 9m^2 + 6m + 1 = 3(3m^2 + 2m) + 1$. Case 2: $p^2 = 9m^2 + 12m + 4 = 3(3m^2 + 4m + 1) + 1$. Either way p^2 leaves remainder 1, so $3 \nmid p^2$. **Main proof:** suppose $\sqrt{3} = p/q$ in lowest terms. Then $3q^2 = p^2$, so $3 \mid p^2$, so $3 \mid p$ by the lemma. Write $p = 3r$: $3q^2 = 9r^2$, so $q^2 = 3r^2$, so $3 \mid q^2$ and $3 \mid q$. Then 3 divides both p and q, contradicting lowest terms.
6. Take $a = 4$, $b = 2$, $c = 2$. Then $bc = 4$ and $4 \mid 4$, but $4 \nmid 2$. (The statement **is** true when a is prime; that is Euclid's lemma, see Chapter 23.)
7. Suppose, for contradiction, that M is the largest even integer. Then $M = 2k$, and $M + 2 = 2(k + 1)$ is even and larger than M. Contradiction.

22.9. Mathematical induction

22.9.1. The problem

Here are some sums of odd numbers:

$$1 = 1, \quad 1 + 3 = 4, \quad 1 + 3 + 5 = 9, \quad 1 + 3 + 5 + 7 = 16.$$

They are the perfect squares. It seems that the sum of the first n odd numbers is n^2. We have learned to distrust patterns, so we need a proof for **every** n. A direct algebraic manipulation is not obvious. But there is something we can say: if the pattern holds up to some $n = k$, then adding the next odd number, $2k + 1$, gives $k^2 + 2k + 1 = (k + 1)^2$. So the pattern **propagates**: each case hands its truth to the next.

That idea, made precise, is **mathematical induction**.

22.9.2. Intuition: dominoes

Imagine an infinite row of dominoes, one for each natural number. Two facts guarantee they all fall:

1. The first domino is pushed over. (**Base case.**)
2. Whenever any domino falls, it knocks over the next one. (**Inductive step.**)

You do not need to watch each domino. Fact 2 is a single statement about a **general** domino k; together with fact 1 it reaches every domino in the row.

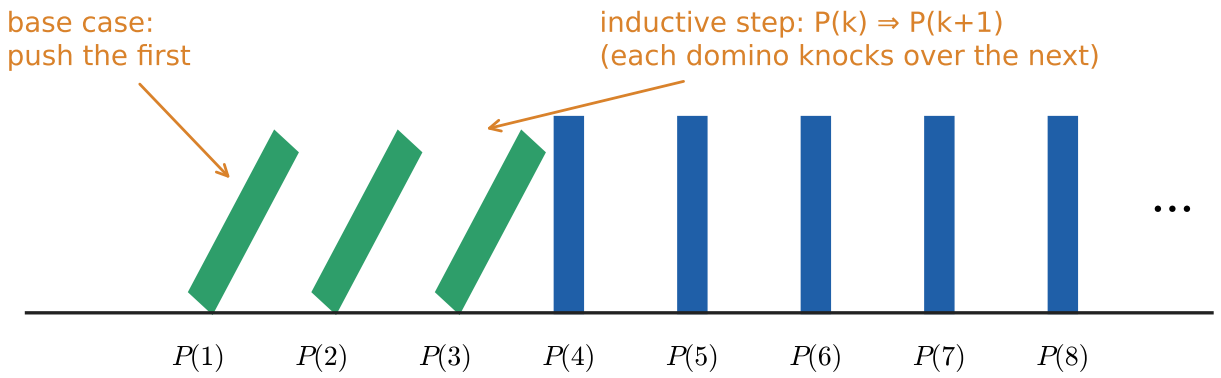


Figure 12. Induction as falling dominoes: the base case starts the chain, and the inductive step makes each domino knock over the next.

Both facts are essential. Without the base case, every domino **would** knock over its neighbour, but nothing starts the chain. Without the inductive step, the first domino falls and nothing follows.

22.9.3. Vocabulary and notation

Sigma notation

The symbol Σ (capital Greek sigma, for "sum") abbreviates long sums:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n.$$

Read it as "the sum, for i from 1 to n, of a_i". The **index** i is a dummy variable (like a loop variable: it has no meaning outside the sum), the numbers below and above Σ are its starting and ending values, and a_i is the term, an expression in i. For example $\sum_{i=1}^{4}(2i-1) = 1+3+5+7 = 16$. In Python: `sum(2*i - 1 for i in range(1, n + 1))`. A key identity for induction is "peel off the last term":

$$\sum_{i=1}^{k+1} a_i = \left(\sum_{i=1}^{k} a_i\right) + a_{k+1}.$$

Sigma notation is covered in more depth in Volume 1, Chapter 14.

Induction vocabulary

- $P(n)$ denotes a **statement** depending on n, true or false for each n; for example $P(n)$: "$\sum_{i=1}^{n}(2i-1) = n^2$". It is a predicate, not a number.
- The **base case** is the proof of $P(n_0)$ for the starting value n_0 (often 0 or 1).
- The **inductive hypothesis** (IH) is the assumption "$P(k)$ is true" for an arbitrary $k \geq n_0$.
- The **inductive step** is the proof that $P(k) \Rightarrow P(k+1)$ for every $k \geq n_0$.
- The **principle of mathematical induction**: if $P(n_0)$ is true and $P(k) \Rightarrow P(k+1)$ for all $k \geq n_0$, then $P(n)$ is true for all integers $n \geq n_0$.

22.9.4. Why it works

Why is induction a valid proof technique, and not just a nice story about dominoes? Suppose the base case and inductive step are proved, but $P(n)$ is false for some $n \geq n_0$. Among all the n for which $P(n)$ fails, there is a **smallest** one; call it m. (Every non-empty set of natural numbers has a least element. This is the **well-ordering principle**, an axiom of the natural numbers.) $m \neq n_0$, because the base case holds. So $m - 1 \geq n_0$, and $P(m-1)$ is true, because m was the smallest failure. But then the inductive step, with $k = m - 1$, gives $P(m)$. Contradiction. So there is no failure.

In other words: induction is a packaged proof by contradiction about the "first bad case". That viewpoint is often useful when you are stuck: ask "what would the smallest counterexample look like?"

22.9.5. The method

Example 10. Procedure — Proof by (ordinary) induction

Input: a claim "for all integers $n \geq n_0$, $P(n)$".

Steps:

1. **State $P(n)$** explicitly as a sentence or equation.
2. **Base case.** Verify $P(n_0)$ by direct computation. Show both sides.
3. **Inductive hypothesis.** Write "Let $k \geq n_0$ and assume $P(k)$." Write out $P(k)$ in full.
4. **Target.** Write out $P(k+1)$ in full by substituting $k+1$ for n. Simplify the target if helpful.
5. **Inductive step.** Starting from one side of $P(k+1)$, find $P(k)$ inside it (for sums: peel off the last term), apply the IH, and simplify until you reach the other side. Label the line where the IH is used.
6. **Conclude:** "By induction, $P(n)$ holds for all $n \geq n_0$."

Output: a proof for every $n \geq n_0$.

22.9.6. Worked example 1: the sum of the first n odd numbers

Claim. For all integers $n \geq 1$, $\displaystyle\sum_{i=1}^{n}(2i-1) = n^2$.

Step 1. $P(n)$: $1 + 3 + \cdots + (2n-1) = n^2$.

Step 2: base case $n = 1$. Left side: $\sum_{i=1}^{1}(2i-1) = 2(1) - 1 = 1$. Right side: $1^2 = 1$. Equal, so $P(1)$ holds.

Step 3: inductive hypothesis. Let $k \geq 1$ and assume

$$\sum_{i=1}^{k}(2i-1) = k^2.$$

Step 4: target. We must show $\sum_{i=1}^{k+1}(2i-1) = (k+1)^2$.

Step 5: inductive step.

$$\begin{aligned}
\sum_{i=1}^{k+1}(2i-1) &= \left(\sum_{i=1}^{k}(2i-1)\right) + \big(2(k+1)-1\big) && \text{(peel off the last term)} \\
&= k^2 + \big(2(k+1)-1\big) && \text{(by the IH)} \\
&= k^2 + 2k + 1 && \text{(simplify)} \\
&= (k+1)^2. && \text{(factor the perfect square)}
\end{aligned}$$

Step 6. By induction, $\sum_{i=1}^{n}(2i-1) = n^2$ for every $n \geq 1$. ■

The inductive step has a picture. A $k \times k$ square of dots grows into a $(k+1) \times (k+1)$ square by adding an L-shaped border of k dots along the top, k down the side and one in the corner: $2k+1$ dots, the next odd number.

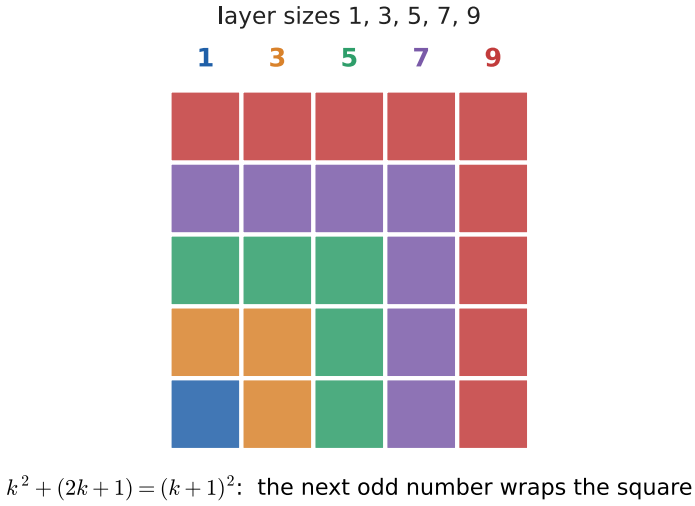


Figure 13. The sum of the first five odd numbers is 5^2: each odd number is an L-shaped layer wrapped around the previous square

22.9.7. Worked example 2: the sum of squares

Claim. For all $n \geq 1$,

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

Base case $n = 1$. Left: $1^2 = 1$. Right: $\frac{1 \cdot 2 \cdot 3}{6} = \frac{6}{6} = 1$. Equal.

Inductive hypothesis. Assume $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$ for some $k \geq 1$.

Target. Substitute $n = k+1$ into the right-hand side:

$$\frac{(k+1)\big((k+1)+1\big)\big(2(k+1)+1\big)}{6} = \frac{(k+1)(k+2)(2k+3)}{6}.$$

Inductive step.

$$
\begin{aligned}
\sum_{i=1}^{k+1} i^2 &= \sum_{i=1}^{k} i^2 + (k+1)^2 && \text{(peel off the last term)} \\
&= \frac{k(k+1)(2k+1)}{6} + (k+1)^2 && \text{(by the IH)} \\
&= \frac{k(k+1)(2k+1)}{6} + \frac{6(k+1)^2}{6} && \text{(common denominator)} \\
&= \frac{(k+1)\bigl[k(2k+1) + 6(k+1)\bigr]}{6} && \text{(factor out } k+1\text{)} \\
&= \frac{(k+1)\bigl[2k^2 + k + 6k + 6\bigr]}{6} && \text{(expand inside the bracket)} \\
&= \frac{(k+1)(2k^2 + 7k + 6)}{6} && \text{(collect like terms)} \\
&= \frac{(k+1)(k+2)(2k+3)}{6}. && \text{(factor: } (k+2)(2k+3) = 2k^2 + 7k + 6\text{)}
\end{aligned}
$$

That is the target, so by induction the formula holds for all $n \geq 1$. ■

The tactical lesson: **factor out the common piece early** (here $k+1$) instead of expanding everything into a cubic. You know what the target looks like, so steer towards it.

22.9.8. Worked example 3: an inequality

Induction is not just for equations.

Claim. $2^n > n^2$ for all integers $n \geq 5$.

Why start at 5? Check small values: $2^1 = 2 > 1$, $2^2 = 4 = 4$ (not greater), $2^3 = 8 < 9$, $2^4 = 16 = 16$, $2^5 = 32 > 25$. The claim is false for $n = 2, 3, 4$, so the base case must be $n_0 = 5$.

Base case $n = 5$: $2^5 = 32 > 25 = 5^2$. True.

Inductive hypothesis. Assume $2^k > k^2$ for some $k \geq 5$.

Target. $2^{k+1} > (k+1)^2$.

Inductive step. With inequalities, the trick is to bridge from what the IH gives you to what the target needs:

$$
\begin{aligned}
2^{k+1} &= 2 \cdot 2^k \\
&> 2k^2 && \text{(by the IH, multiplied by 2).}
\end{aligned}
$$

Now it suffices to show $2k^2 \geq (k+1)^2$, i.e., $2k^2 \geq k^2 + 2k + 1$, i.e., $k^2 - 2k - 1 \geq 0$. Complete the square: $k^2 - 2k - 1 = (k-1)^2 - 2$. For $k \geq 5$, $(k-1)^2 \geq 16$, so $(k-1)^2 - 2 \geq 14 > 0$. Therefore

$$
2^{k+1} > 2k^2 \geq (k+1)^2,
$$

which gives $2^{k+1} > (k+1)^2$. By induction, $2^n > n^2$ for all $n \geq 5$. ■

22.9.9. Worked example 4: divisibility

Claim. $3 \mid 4^n - 1$ for all $n \geq 0$.

Base case $n = 0$: $4^0 - 1 = 0 = 3 \cdot 0$, divisible by 3.

Inductive hypothesis. Assume $4^k - 1 = 3m$ for some integer m.

Inductive step. The trick is to make $4^k - 1$ appear inside $4^{k+1} - 1$:

$$\begin{aligned} 4^{k+1} - 1 &= 4 \cdot 4^k - 1 \\ &= 4 \cdot 4^k - 4 + 3 && \text{(subtract and add 3)} \\ &= 4(4^k - 1) + 3 && \text{(factor 4 from the first two terms)} \\ &= 4(3m) + 3 && \text{(by the IH)} \\ &= 3(4m + 1). \end{aligned}$$

So $3 \mid 4^{k+1} - 1$. By induction the claim holds for all $n \geq 0$. ■

(Chapter 23 gives a one-line alternative: $4 \equiv 1 \pmod 3$, so $4^n \equiv 1^n = 1$.)

22.9.10. Alternative approaches and choosing a method

Induction **verifies** a formula; it does not **discover** it. To prove $\sum i^2 = n(n+1)(2n+1)/6$ you must already know the right-hand side. So the realistic workflow is:

1. **Discover** the formula: compute small cases, guess a pattern, or derive it (pairing, as Gauss did; telescoping sums; fitting a polynomial of the right degree to the first few values).
2. **Check** it numerically for many n (a quick script).
3. **Prove** it, usually by induction.

When a direct derivation exists (the Gauss pairing for $\sum i$, the geometric series formula derived in Volume 1, Chapter 14), it is often more illuminating than induction because it explains **why**. Induction is the reliable fallback that works when you have the answer but not the insight, and it is the natural tool whenever the objects are **defined** step by step (recurrences, recursive functions, data structures).



Forgetting the base case

"Claim: $n = n + 1$ for all n. Inductive step: assume $k = k + 1$. Adding 1 to both sides, $k + 1 = k + 2$. So $P(k) \Rightarrow P(k+1)$." The inductive step is **valid**. The claim is absurd. What is missing is the base case: $0 = 1$ is false, so no domino is ever pushed. Always prove the base case, even when it looks trivial.



Assuming what you want to prove

The inductive hypothesis is "assume $P(k)$ for **one** arbitrary k", not "assume $P(n)$ for all n", which would be assuming the theorem. If your write-up says "assume the formula is true for all n", it is circular.

22.9.11. Where you'd use this

Games: a winning strategy is an induction proof. Twenty coins are on the table. Players take turns removing 1, 2 or 3 coins, and whoever takes the last coin wins. Claim: the player to move loses exactly when the number of coins is a multiple of 4. With 0 coins the player to move has already lost. If the count is a multiple of 4, any move leaves a non-multiple; if it is not, you can take 1, 2 or 3 to leave a multiple of 4. By (strong) induction the claim holds for every count, so with 20 coins you should let your opponent go first and then always make each round total 4.

Finance: why the compound-interest formula is right. If a balance starts at $B_0 = P$ and each year $B_{k+1} = B_k(1 + r)$, induction proves $B_n = P(1 + r)^n$ for every n: the base case is $B_0 = P$, and the step is $B_{k+1} = P(1 + r)^k(1 + r) = P(1 + r)^{k+1}$. So \$1,000 at 5% for 3 years becomes $1000 \times 1.05^3 = 1157.63$, and the formula, not a spreadsheet of 30 rows, is what you trust for a 30-year mortgage.

Medicine: drug accumulation. A patient takes 100 mg of a drug once every half-life, so half of what is in the body is cleared between doses. Just after the n-th dose the amount is $100(1 + \frac{1}{2} + \cdots + \frac{1}{2^{n-1}})$, which induction shows equals $200(1 - 2^{-n})$ mg. After 5 doses that is $200 \times \frac{31}{32} = 193.75$ mg, and the level never exceeds 200 mg however long treatment continues. Pharmacologists use exactly this "steady state" rule of thumb (about five half-lives to get within a few percent).

Computing. Every recursive function and every divide-and-conquer running time (merge sort's $T(n) = 2T(n/2) + n$ gives $n \log_2 n$ comparisons, roughly) is justified by induction; Section 22.11 makes the link explicit.



Proof by induction, checked by a script

An "induction workbench": enter a conjectured closed form such as `n*(n+1)(2*n+1)//6` and a recursive definition such as `S(n) = S(n-1) + n *2`. The lab tabulates both for $n = 1 \ldots 50$ (discovery and checking), then animates the dominoes: the base-case domino turns green when you confirm $P(n_0)$, and the chain falls only after you enter the algebra of the inductive step.



Induction everywhere

The first clear use of induction is usually credited to Francesco Maurolico (1575) and Blaise Pascal, who used it around 1654 in his work on the arithmetic triangle. The name "mathematical induction" came much later, from Augustus De Morgan in 1838. It is unrelated to **inductive reasoning** in science (generalising from observations), which is exactly the kind of reasoning the Euler polynomial warns against. Mathematical induction is fully deductive.

US curriculum

Induction appears in some **Algebra II** or **Precalculus** courses (often with sequences and series) and occasionally in honors high-school classes, but most US students meet it properly in college **Discrete Mathematics**. The AP Calculus syllabi do not include it. A student comfortable with algebraic manipulation from **Algebra I–II** has everything needed.

22.10. Strong induction

22.10.1. The problem

Claim: *every integer $n \geq 2$ is a product of one or more primes.* (A prime by itself counts as a product of one prime.)

Try ordinary induction. Assume k is a product of primes; show $k+1$ is. But k and $k+1$ have no useful relationship here: knowing $14 = 2 \cdot 7$ tells you nothing about 15. What **does** help: if $k+1$ is composite, say $k+1 = ab$, then a and b are **smaller** numbers, and we would like to know they are products of primes. We need the hypothesis for all smaller numbers, not just the one immediately before.

22.10.2. Intuition

Strong induction (also called complete induction) lets the inductive step use **every** earlier case:

$$\big(P(n_0) \wedge P(n_0+1) \wedge \cdots \wedge P(k)\big) \Rightarrow P(k+1).$$

In domino terms: each domino is knocked over by the combined weight of all the dominoes before it. It is no more powerful than ordinary induction (you can convert one to the other by using the statement "P holds for all values up to n"), but it is far more convenient whenever $P(k+1)$ depends on cases further back.

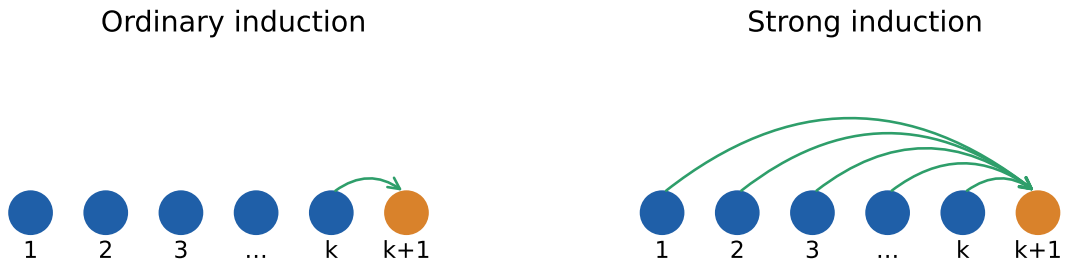


Figure 14. Ordinary induction uses only P(k) to reach P(k + 1); strong induction may use all of P(1), ..., P(k).

22.10.3. The method

Example 11. Procedure — Proof by strong induction

Input: a claim "for all integers $n \geq n_0$, $P(n)$".

Steps:

1. **Base case(s).** Prove $P(n_0)$, and as many further cases as the inductive step needs (if the step reaches back j places, prove j base cases).
2. **Strong inductive hypothesis.** Let k be at least the last base case and assume $P(i)$ for **all** i with $n_0 \leq i \leq k$.
3. **Inductive step.** Prove $P(k+1)$, using any of the $P(i)$ that you need. Check that every i you use is in the range $n_0 \leq i \leq k$.
4. **Conclude** by strong induction.

Output: a proof for all $n \geq n_0$.

22.10.4. Worked example 1: every integer $n \geq 2$ is a product of primes

Base case $n = 2$: 2 is prime, so it is a product of one prime.

Strong IH. Let $k \geq 2$ and assume every integer i with $2 \leq i \leq k$ is a product of primes.

Inductive step. Consider $k+1$. There are two cases.

Case 1: $k+1$ is prime. Then it is a product of one prime. Done.

Case 2: $k+1$ is composite. Then $k+1 = ab$ for integers with $2 \leq a, b \leq k$. (Composite means it has a divisor a other than 1 and itself; then $b = (k+1)/a$ is also strictly between 1 and $k+1$.) By the strong IH, $a = p_1 \cdots p_r$ and $b = q_1 \cdots q_s$ are products of primes. So $k+1 = p_1 \cdots p_r \, q_1 \cdots q_s$ is a product of primes.

By strong induction, every integer $n \geq 2$ is a product of primes. ■

In particular every $n \geq 2$ has at least one prime divisor, which is the fact Euclid's proof used. This is the existence half of the **Fundamental Theorem of Arithmetic**; uniqueness of the factorisation is proved in Chapter 23.

22.10.5. Worked example 2: postage stamps

Claim. Every amount of postage of 12 cents or more can be made using only 4-cent and 5-cent stamps.

Here $P(n)$: "there are non-negative integers a, b with $n = 4a + 5b$".

The idea: if we can make $n - 4$ cents, add one 4-cent stamp to make n. That reaches back **four** places, so we need four base cases.

Base cases.

$$\begin{aligned} 12 &= 4 \cdot 3 + 5 \cdot 0 \\ 13 &= 4 \cdot 2 + 5 \cdot 1 \\ 14 &= 4 \cdot 1 + 5 \cdot 2 \\ 15 &= 4 \cdot 0 + 5 \cdot 3 \end{aligned}$$

Strong IH. Let $k \geq 15$ and assume $P(i)$ for all $12 \leq i \leq k$.

Inductive step. Since $k \geq 15$, $k + 1 \geq 16$, so $12 \leq (k+1) - 4 \leq k$. By the IH, $(k+1) - 4 = 4a + 5b$. Then $k + 1 = 4(a+1) + 5b$. Done. $\blacksquare$

Why 12? Because 11 cannot be made (11 is not of the form $4a + 5b$: try $b = 0, 1, 2$, giving remainders 11, 6, 1, none a multiple of 4). This is the **Frobenius coin problem**; for two coin values x, y with no common factor the largest unpayable amount is $xy - x - y = 20 - 4 - 5 = 11$.

22.10.6. Worked example 3: a bound on Fibonacci numbers

The Fibonacci numbers are defined by $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$: $1, 1, 2, 3, 5, 8, 13, \ldots$

Claim. $F_n < 2^n$ for all $n \geq 1$.

The recurrence reaches back two places, so two base cases. $F_1 = 1 < 2 = 2^1$ and $F_2 = 1 < 4 = 2^2$.

Strong IH. Let $k \geq 2$, and assume $F_i < 2^i$ for all $1 \leq i \leq k$.

Inductive step.

$$\begin{aligned} F_{k+1} &= F_k + F_{k-1} && \text{(definition, valid since } k+1 \geq 3) \\ &< 2^k + 2^{k-1} && \text{(IH for } i = k \text{ and } i = k-1, \text{ both in } [1, k]) \\ &< 2^k + 2^k && \text{(since } 2^{k-1} < 2^k) \\ &= 2^{k+1}. \end{aligned}$$

$\blacksquare$



Too few base cases

If the inductive step for $P(k+1)$ uses $P(k-1)$, then with only one base case $P(1)$ the step for $k+1=2$ would need $P(0)$, which you never proved (and F_0 may not even be defined). The rule: if the step reaches back j places, you need j consecutive base cases. Check the smallest k for which you invoke the step and make sure everything it uses is covered.

22.11. Structural induction and recursion

22.11.1. The problem

Programmers rarely work with "for all integers n". They work with lists, trees, expressions and JSON documents. How do you prove something about **every** binary tree, or **every** well-formed expression?

22.11.2. Intuition

These objects are defined **recursively**: a list is either empty or an element followed by a list. A full binary tree is either a single leaf, or a root node with two full binary subtrees. Every object is built from smaller objects of the same kind, and the building process starts from base objects.

Structural induction follows the definition:

- **Base case:** prove the property for each base object (empty list, single leaf).
- **Inductive step:** for each way of building a bigger object from smaller ones, assume the property for the smaller pieces and prove it for the result.

This is ordinary strong induction in disguise (induct on the size of the object), but it is much more natural to write.



Recursive definition, full binary tree

- A **recursive definition** (or inductive definition) defines a set of objects by (1) some base objects and (2) rules that build new objects from existing ones, with nothing else included.
- A **full binary tree** is defined by: (base) a single node is a full binary tree; (rule) if T_1 and T_2 are full binary trees, then a new root node with left subtree T_1 and right subtree T_2 is a full binary tree. In a full binary tree every node has 0 or 2 children. A node with 0 children is a **leaf**; one with 2 is an **internal node**.

22.11.3. Worked example: leaves versus internal nodes

Claim. In every full binary tree, the number of leaves L equals the number of internal nodes I plus one: $L = I + 1$.

Base case. A single node is a leaf and has no children: $L = 1$, $I = 0$, and $1 = 0 + 1$.

Inductive step. Let T be built from a new root with subtrees T_1 and T_2, and assume (structural IH) that $L_1 = I_1 + 1$ and $L_2 = I_2 + 1$. The leaves of T are exactly the leaves of T_1 and T_2 (the root is not a leaf). The internal nodes of T are those of T_1 and T_2 plus the root. So

$$\begin{aligned} L &= L_1 + L_2 \\ &= (I_1 + 1) + (I_2 + 1) \qquad \text{(by the structural IH)} \\ &= (I_1 + I_2 + 1) + 1 \\ &= I + 1. \qquad \text{(since } I = I_1 + I_2 + 1\text{)} \end{aligned}$$

By structural induction, $L = I + 1$ for every full binary tree. ■

Application: a knockout tournament is a full binary tree whose leaves are the L players and whose internal nodes are the matches. So a tournament with 64 players has exactly 63 matches, regardless of how the bracket is arranged. Another: a Huffman code tree (used in compression formats such as DEFLATE) with L symbols has $L - 1$ merge steps.

22.11.4. Induction and recursion are the same idea

Compare a recursive function with an induction proof:

```python
def sum_to(n):
    """1 + 2 + ... + n, defined recursively."""
    if n == 1:                  # base case
        return 1
    return sum_to(n - 1) + n    # "inductive step": trust sum_to(n-1)

assert all(sum_to(n) == n * (n + 1) // 2 for n in range(1, 500))
print(sum_to(100))
```

```
5050
```

To convince yourself `sum_to` is correct you reason exactly by induction: it is correct for $n = 1$ (base case); and **if** `sum_to(n - 1)` returns $(n-1)n/2$ (inductive hypothesis), **then** `sum_to(n)` returns

$$\frac{(n-1)n}{2} + n = \frac{(n-1)n + 2n}{2} = \frac{n(n+1)}{2}.$$

The `assert` line is a test, valuable but not a proof. The algebra is the proof.

The recursive leap of faith

Teachers of recursion tell students to take the "recursive leap of faith": when writing `f(n)`, assume `f` already works on smaller inputs and concentrate on combining the results. That leap is not faith at all; it is the inductive hypothesis, and the reasoning is valid exactly when (1) the base cases are correct and (2) every recursive call is on a **strictly smaller** input that eventually reaches a base case. Condition (2) is the structural-induction version of "the dominoes are in a line": it is what makes the recursion terminate. Functional languages (Haskell, OCaml, and especially proof-oriented ones like Lean and Agda) make the correspondence explicit: a function defined by structural recursion over an algebraic data type **is** a proof by structural induction.



Recursion tree and proof tree

Build a full binary tree by clicking leaves to split them. The lab keeps live counts of leaves and internal nodes (watch $L - I$ stay at 1), and shows the structural-induction proof annotating each split: "this split turns one leaf into an internal node and adds two leaves: L goes up by 1, I goes up by 1".

22.12. Exercises: induction

1. **Recognition.** In the proof that $F_n < 2^n$, why are two base cases needed? What would go wrong with only $F_1 < 2$?
2. **Application.** Prove by induction that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all $n \geq 1$.
3. **Application.** Prove that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ for all $n \geq 1$. (Interpret this in binary.)
4. **Application.** Prove that $n! > 2^n$ for all $n \geq 4$. ($n! = 1 \cdot 2 \cdots n$, "n factorial".)
5. **Problem solving.** Prove by induction that $6 \mid n^3 - n$ for all $n \geq 0$. (Use the result of Section 22.6.)
6. **Problem solving.** Prove that every integer $n \geq 8$ can be written as $3a + 5b$ with non-negative integers a, b.
7. **Problem solving.** A **perfect** binary tree of height h is a single node if $h = 0$, and otherwise a root with two perfect subtrees of height $h - 1$. Prove it has $2^{h+1} - 1$ nodes.

▼ **Answers**

1. The step proves $P(k+1)$ from $P(k)$ and $P(k-1)$, and is first used at $k+1 = 3$, which needs $P(2)$ and $P(1)$. With only $P(1)$ proved, the step for $k+1 = 2$ would need $P(0)$, and the recurrence $F_2 = F_1 + F_0$ is not part of the definition. Without $P(2)$ the chain never starts.
2. Base: $1 = 1 \cdot 2/2$. Step: $\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$.

3. Base $n = 1$: $2^0 = 1 = 2^1 - 1$. Step: $\sum_{i=0}^{k} 2^i = (2^k - 1) + 2^k = 2 \cdot 2^k - 1 = 2^{k+1} - 1$. In binary, $111\ldots1$ (n ones) plus 1 is $1000\ldots0$ (a 1 followed by n zeros), which is 2^n. This is why the largest unsigned n-bit integer is $2^n - 1$.

4. Base $n = 4$: $4! = 24 > 16 = 2^4$. Step: assume $k! > 2^k$ with $k \geq 4$. Then $(k+1)! = (k+1)\,k! > (k+1)\,2^k \geq 2 \cdot 2^k = 2^{k+1}$, using $k + 1 \geq 2$.

5. Base $n = 0$: $0 - 0 = 0 = 6 \cdot 0$. Step: $(k+1)^3 - (k+1) = k^3 + 3k^2 + 3k + 1 - k - 1 = (k^3 - k) + 3k(k+1)$. By the IH, $6 \mid k^3 - k$. By Section 22.6, $k(k+1)$ is even, so $k(k+1) = 2t$ and $3k(k+1) = 6t$. A sum of two multiples of 6 is a multiple of 6.

6. Strong induction with three base cases: $8 = 3 + 5$, $9 = 3 \cdot 3$, $10 = 5 \cdot 2$. For $k \geq 10$, $k + 1 \geq 11$, so $8 \leq (k+1) - 3 \leq k$; by the IH $(k+1) - 3 = 3a + 5b$, hence $k + 1 = 3(a+1) + 5b$.

7. Structural induction on h. Base $h = 0$: one node, and $2^1 - 1 = 1$. Step: a tree of height $h + 1$ is a root plus two perfect trees of height h, each with $2^{h+1} - 1$ nodes by the IH. Total: $1 + 2(2^{h+1} - 1) = 2^{h+2} - 1 = 2^{(h+1)+1} - 1$.

22.13. Proving programs correct: loop invariants

22.13.1. The problem

Binary search is ten lines long and notoriously easy to get wrong: off-by-one errors in `lo`, `hi` and `mid`, infinite loops, wrong results on empty arrays. Jon Bentley reported that when he set it as an exercise for professional programmers, around 90% of their solutions contained a bug. Tests catch some of these errors. How do we **know** a correct version is correct?

22.13.2. Intuition

A loop is induction in disguise. Each iteration is an inductive step; the state before the first iteration is the base case. The statement being proved is a **loop invariant**: a condition on the program's variables that is true before the loop starts and stays true after every iteration. When the loop stops, the invariant plus the stopping condition should give you exactly what you wanted.



Loop invariant, variant, pre- and postcondition

- A **precondition** is what must be true when a piece of code starts (e.g., "`a` is sorted").
- A **postcondition** is what the code guarantees when it ends (e.g., "the return value is an index of `target`, or -1 if `target` is absent").
- A **loop invariant** is a statement about the variables that holds every time execution reaches the top of the loop (before the test).
- A **loop variant** (or **ranking function**, or **termination measure**) is an integer quantity that is always non-negative while the loop runs and strictly decreases with every iteration. Its existence proves the loop terminates.
- A program is **partially correct** if, whenever it terminates, the postcondition holds; **totally correct** if, in addition, it always terminates.

22.13.3. The method

Example 12. Procedure — Proving a loop correct

Input: a loop, its precondition, and the desired postcondition.

Steps:

1. **Choose an invariant I**: typically "the work done so far is correct" or "the answer, if it exists, is still in the region we have not excluded". A good invariant mentions every variable the loop changes.
2. **Initialisation (base case).** Show I holds just before the first iteration, using the precondition and the initial assignments.
3. **Maintenance (inductive step).** Assume I holds at the top of an iteration and the loop condition is true. Show I holds again after the body runs. Do this for every branch of the body.
4. **Termination.** Exhibit a variant: a non-negative integer that strictly decreases on every iteration. Then the loop must stop.
5. **Exit (conclusion).** Show that I together with the **negation** of the loop condition implies the postcondition. Also check any early `return`.

Output: a proof of total correctness.

22.13.4. Worked example: binary search

```python
def binary_search(a, target):
    """Return an index i with a[i] == target, or -1. `a` must be sorted."""
    lo, hi = 0, len(a) - 1
    while lo <= hi:
        # Invariant: if target is in a, then it is in a[lo..hi].
        mid = (lo + hi) // 2
        if a[mid] == target:
            return mid
        elif a[mid] < target:
            lo = mid + 1      # a[0..mid] are all < target: discard them
        else:
            hi = mid - 1      # a[mid..] are all > target: discard them
    return -1                 # a[lo..hi] is empty, so target is not in a

a = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
print(binary_search(a, 23), binary_search(a, 7))
```

```
5 -1
```

Here `a[lo..hi]` means the elements with indices from `lo` to `hi` **inclusive**; it is empty when `lo > hi`.

Precondition: `a` is sorted in non-decreasing order. **Postcondition:** the result `r` satisfies `a[r] == target`, or `r == -1` and `target` does not occur in `a`.

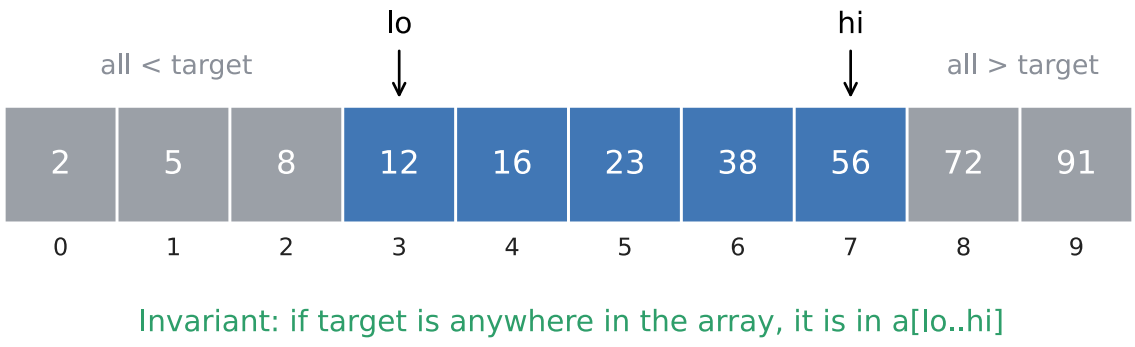


Figure 15. The binary-search invariant: everything to the left of lo is smaller than the target and everything to the right of hi is larger, so if the target is anywhere, it is between lo and hi.

Step 1: invariant. I: "if `target` occurs in `a`, then it occurs in `a[lo..hi]`."

Step 2: initialisation. Before the loop, `lo = 0` and `hi = len(a) - 1`, so `a[lo..hi]` is the whole array. "If `target` is in `a` then it is in `a`" is trivially true.

Step 3: maintenance. Assume I and `lo ≤ hi`. Then `mid = (lo + hi) // 2` satisfies `lo ≤ mid ≤ hi`, so `a[mid]` is a valid element of the window. Three branches:

- `a[mid] == target`: we return, so there is no next iteration to worry about (it is handled in step 5).

- `a[mid] < target`: since `a` is sorted, every element `a[j]` with `j ⇐ mid` satisfies `a[j] ⇐ a[mid] < target`, so none of them equals `target`. If `target` is in `a[lo..hi]`, it must therefore be in `a[mid+1..hi]`. Setting `lo = mid + 1` keeps I true.
- `a[mid] > target`: symmetric. Every `a[j]` with `j >= mid` is at least `a[mid] > target`, so `target` (if present in the window) is in `a[lo..mid-1]`. Setting `hi = mid - 1` keeps I true.

Step 4: termination. Use the variant $V = \texttt{hi} - \texttt{lo} + 1$, the number of elements in the window. While the loop runs (`lo ⇐ hi`), $V \geq 1 > 0$. In the second branch the new window is `a[mid+1..hi]`, which excludes at least `a[mid]`; in the third it is `a[lo..mid-1]`, also excluding `a[mid]`. So V decreases by at least 1 on every iteration that does not return. A positive integer cannot decrease for ever, so the loop terminates.

Step 5: exit. There are two ways out.

- Early return in the first branch: `a[mid] == target`, so the postcondition holds with `r = mid`.
- The loop condition fails: `lo > hi`, so `a[lo..hi]` is empty. By I, if `target` were in `a` it would be in this empty window, which is impossible. So `target` is not in `a`, and returning `-1` is correct.

Both exits satisfy the postcondition, so binary search is totally correct. ■

Trace for `target = 23` on the array above (indices 0–9):

lo	hi	mid	a[mid]	**action**
0	9	4	16	$\mathbf{16 < 23}$: `lo = 5`
5	9	7	56	$\mathbf{56 > 23}$: `hi = 6`
5	6	5	23	found: return 5

And for `target = 7`: (0, 9, mid 4, 16 > 7, `hi = 3`), (0, 3, mid 1, 5 < 7, `lo = 2`), (2, 3, mid 2, 8 > 7, `hi = 1`), and now $\texttt{lo} = 2 > 1 = \texttt{hi}$, so return -1. The window size went $10 \to 4 \to 2 \to 0$: the variant at work.

22.13.5. A second example: insertion sort

For insertion sort the natural invariant is "the prefix `a[0:i]` is sorted". You can even turn the invariant into a runtime check:

```python
def insertion_sort(a):
    a = list(a)
    for i in range(1, len(a)):
        # Invariant: a[0:i] is sorted (and is a permutation of the original a[0:i]).
        assert all(a[j] <= a[j + 1] for j in range(i - 1))
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]   # shift larger element right
            j -= 1
        a[j + 1] = key
    return a

print(insertion_sort([5, 2, 9, 1, 5, 6]))
```

```
[1, 2, 5, 5, 6, 9]
```

- **Initialisation:** when $i = 1$, `a[0:1]` has one element, and a one-element list is sorted.
- **Maintenance:** the inner loop shifts every element of the sorted prefix that is larger than `key` one place right and drops `key` into the gap, so `a[0:i+1]` is sorted. (A full proof gives the inner loop its own invariant: "`a[j+2..i]` are all greater than `key`, and the prefix with the gap removed is sorted".)
- **Termination:** a `for` loop over a finite range always terminates; the inner `while` has variant $j + 1$, which decreases by 1 each time and stays non-negative while the loop runs.
- **Exit:** the outer loop ends with $i = n$ (the length), so the invariant says `a[0:n]`, the whole list, is sorted.

`assert` statements like the one above turn your proof sketch into a check that runs during testing. They do not prove anything, but a failing invariant assertion pinpoints **where** the reasoning breaks, which is far more useful than a wrong final answer.



Termination is a separate obligation

An invariant proves the answer is right **if** the loop stops. It says nothing about whether it stops. Write `lo = mid` instead of `lo = mid + 1` in binary search and the invariant is still maintained, but with `lo = 0`, `hi = 1` and `a[0] < target`, `mid` is 0 and `lo` stays 0 for ever: the variant does not decrease. Always check both.



> *The proof assumed mathematical integers*
>
> In 2006 Joshua Bloch reported that the binary search in Java's standard library, which had shipped for about nine years, had a bug. The line `mid = (low + high) / 2` overflows when `low + high` exceeds $2^{31} - 1$, which happens for arrays of more than about a billion elements, giving a negative `mid` and an exception. The code followed the version in Jon Bentley's *Programming Pearls*, which Bentley had **proved** correct, but the proof silently assumed that `int` behaves like the mathematical integers $\mathbb{Z}$. The fix is `mid = low + (high - low) / 2`. Python's integers are unbounded, so the version above is safe; in C, C++, Java or Go it is not. A proof is only as good as its model of the machine.

22.13.6. Termination arguments

Every termination proof is the same argument: find a quantity in a **well-ordered** set (usually the non-negative integers) that strictly decreases. Because there is no infinite decreasing sequence of non-negative integers, the process must stop. That is the well-ordering principle again, the same fact that justified induction.

Euclid's algorithm for the greatest common divisor (studied in Chapter 23) is a good example:

```python
def gcd(a, b):
    while b != 0:
        # Variant: b is a non-negative integer that strictly decreases.
        a, b = b, a % b
    return a

print(gcd(252, 105))
```

```
21
```

For non-negative inputs, the new b is $a \bmod b$, the remainder of dividing a by b, which always satisfies $0 \le a \bmod b < b$. So b strictly decreases and stays non-negative: the loop must reach $b = 0$.

Sometimes no variant is known:

```python
def collatz_steps(n):
    steps = 0
    while n != 1:
        n = n // 2 if n % 2 == 0 else 3 * n + 1
        steps += 1
    return steps

print([collatz_steps(n) for n in range(1, 11)])
print(collatz_steps(27))
```

```
[0, 1, 7, 2, 5, 8, 16, 3, 19, 6]
111
```



A four-line loop nobody can prove terminates

The Collatz conjecture says the loop above terminates for every positive integer n. It has been checked by computer for every n up to beyond 2^{68}, yet nobody has a proof, and the value of n does not decrease steadily (starting from 27, it takes 111 steps and climbs as high as 9232 before falling to 1). Paul Erdős reportedly said "mathematics may not be ready for such problems". It is a sobering reminder that "does this loop terminate?" can be beyond the reach of current mathematics, and, by Turing's halting theorem, no algorithm can answer it for all programs.

22.13.7. Where you'd use this

Invariants are not a programming invention. **Double-entry bookkeeping** maintains the invariant "total debits = total credits" through every transaction; when the trial balance disagrees, an entry broke the invariant, and you know to look for it. **Physics simulations** check conservation laws the same way: in a game-physics or orbital simulation, total energy and momentum should stay constant (up to rounding), so a drifting total is the first sign of a bad time step. And a **bank transfer** between two accounts must preserve the sum of the balances at every observable moment, which is the invariant that database transactions exist to protect.



Invariant tracer

Step through binary search on an array you choose. At each iteration the lab shades the excluded regions grey (as in the figure), displays the invariant and the variant $\texttt{hi} - \texttt{lo} + 1$, and turns red if you edit the code (for example to `lo = mid`) in a way that breaks maintenance or termination. Try to find an edit that keeps the invariant but loops forever.

Try it: Trace binary search in Python

Formal verification and proof assistants

Everything in this section can be done by machine. **Proof assistants** such as Lean, Coq (now renamed Rocq), Isabelle and Agda check every step of a proof against the rules of logic; nothing is accepted on trust. In Lean, a theorem is a type and its proof is a program of that type; the system accepts it only if it type-checks. A tiny example in Lean 4:

```
theorem add_zero_right (n : Nat) : n + 0 = n := rfl

theorem zero_add_left (n : Nat) : 0 + n = n := by
  induction n with
  | zero => rfl
  | succ k ih => exact congrArg Nat.succ ih
```

The second proof is literally an induction: a `zero` (base) case and a `succ k` ($k+1$) case that uses the hypothesis `ih` for k.

Large systems have been verified this way: the CompCert C compiler (Coq) is proved to preserve the meaning of the programs it compiles, and the seL4 operating-system microkernel (Isabelle) is proved to implement its specification. Tools such as Dafny, Why3 and Frama-C let you write loop invariants and variants as annotations in ordinary code and discharge the proof obligations automatically with SMT solvers. And mathematicians now formalise research mathematics in Lean's `mathlib` library. The habits of this section (pre/postconditions, invariants, variants) are exactly the inputs these tools need.

US curriculum

Loop invariants and program correctness are not part of the US high-school mathematics curriculum. They appear in AP Computer Science A only informally, and properly in college **Data Structures and Algorithms** and **Discrete Mathematics** courses (the standard algorithms textbook by Cormen, Leiserson, Rivest and Stein uses the initialisation/maintenance/termination format above).

22.14. Exercises: programs

1. **Recognition.** What is a suitable invariant for this loop, which finds the largest element of a non-empty list `a`?

   ```python
   best = a[0]
   for i in range(1, len(a)):
       if a[i] > best:
           best = a[i]
   ```

2. **Application.** After the following loop has completed i iterations, what is the value of `s`? State and prove an invariant.

```python
s = 0
for i in range(n):
    s += 2 * i + 1
```

3. **Application.** Prove that `while x > 0: x = x // 2` terminates for every integer `x`.
4. **Problem solving.** A colleague writes binary search with `while lo < hi:` (instead of `<=`) but otherwise identical code. Find an input where it gives the wrong answer, and say which proof step fails.

▼ **Answers**

1. Invariant: "at the top of the iteration with index i, `best` is the maximum of `a[0:i]`". Initialisation: `\(i = 1\), `best = a[0]` is the maximum of the one-element prefix. Maintenance: the new `best` is the larger of the old maximum and `a[i]`, which is the maximum of `a[0:i+1]`. Exit: $i = n$, so `best` is the maximum of the whole list.
2. Invariant: after i iterations, $s = i^2$. Initially $s = 0 = 0^2$. If $s = i^2$ before the iteration with loop variable i, it becomes $i^2 + 2i + 1 = (i+1)^2$, and $i + 1$ iterations have now completed. At the end, $s = n^2$. (This is the sum-of-odd-numbers theorem as a program.)
3. If $x \le 0$ the loop does not run. Otherwise the variant x is a positive integer, and $x // 2 < x$ for $x \ge 1$ (it is at most $x/2$), and $x // 2 \ge 0$. A strictly decreasing sequence of non-negative integers is finite, so the loop ends.
4. Take `a = [5]`, `target = 5`. Then `lo = hi = 0`, the test `0 < 0` is false, the loop never runs, and the function returns -1 although 5 is present. The **exit** step fails: `lo < hi` being false means `lo >= hi`, and when `lo == hi` the window `a[lo..hi]` has one element, not zero, so the invariant does not imply that `target` is absent.

22.15. Common proof errors

Most wrong proofs fail in a handful of recognisable ways. Learning the patterns is the fastest way to check your own work and to read other people's.

22.15.1. Dividing by zero in disguise

Let $a = b$, with $b \neq 0$. Then:

$$\begin{aligned} a^2 &= ab && \text{(multiply both sides by } a\text{)} \\ a^2 - b^2 &= ab - b^2 && \text{(subtract } b^2\text{)} \\ (a+b)(a-b) &= b(a-b) && \text{(factor both sides)} \\ a + b &= b && \text{(divide by } a-b\text{: INVALID, since } a-b=0\text{)} \\ 2b &= b && \text{(substitute } a = b\text{)} \\ 2 &= 1. && \text{(divide by } b\text{)} \end{aligned}$$

Every line is correct except the fourth. Since $a = b$, $a - b = 0$, and cancelling a zero factor is not allowed. Whenever you cancel or divide, ask: could this be zero?

22.15.2. The broken bridge: all horses are the same colour

"**Claim:** in any set of $n \geq 1$ horses, all horses are the same colour. **Base case:** one horse is the same colour as itself. **Step:** given $k+1$ horses, remove the first: the remaining k are the same colour by the IH. Remove the last instead: those k are also the same colour. The two groups overlap, so all $k+1$ share a colour."

The base case is right, and the step is right for $k \geq 2$. It fails at $k = 1 \to 2$: with two horses, removing the first leaves {horse 2}, removing the last leaves {horse 1}, and these groups do **not** overlap. One broken link and the whole chain collapses. Lesson: check that the inductive step works for the **smallest** k, not just "typical" large k.

22.15.3. A catalogue of errors



The usual suspects

- **Proof by example:** checking cases for a "for all" claim (the Euler polynomial).
- **Circular reasoning:** assuming the conclusion, or something equivalent to it, along the way.
- **Proving the converse:** showing $Q \Rightarrow P$ when asked for $P \Rightarrow Q$.
- **Reusing a variable:** "let $a = 2k+1$ and $b = 2k+1$" silently makes $a = b$.
- **Hidden division by zero** or square roots of negatives; taking square roots of both sides without $\pm$.
- **Non-exhaustive cases:** forgetting zero, negatives, the empty set, or $n = 1$.
- **Missing or wrong base case;** too few base cases for strong induction.
- **Inductive step that fails for small k** (the horses).
- **Swapped quantifiers:** "for every ε there is a δ" is not "there is a δ that works for every ε" (see Chapter 21).
- **"Clearly", "obviously", "similarly"** covering a gap. If a step is clear, it costs one line to write it down.
- **Proof about the model, not the machine:** the binary-search overflow.



Spot the bug

A gallery of short "proofs", each with exactly one flaw (division by zero, missing base case, non-exhaustive cases, proving the converse, a broken small case). Click the line you think is wrong; the lab explains the error and shows a counterexample or a corrected proof. A good self-test before the mastery check.

22.16. Choosing a proof technique

You now have six tools. Which should you pick? The flowchart summarises the heuristics developed in this chapter. It is a starting point, not an algorithm: real proofs often combine techniques (the $\sqrt{2}$ proof used a direct proof inside a contrapositive inside a contradiction).

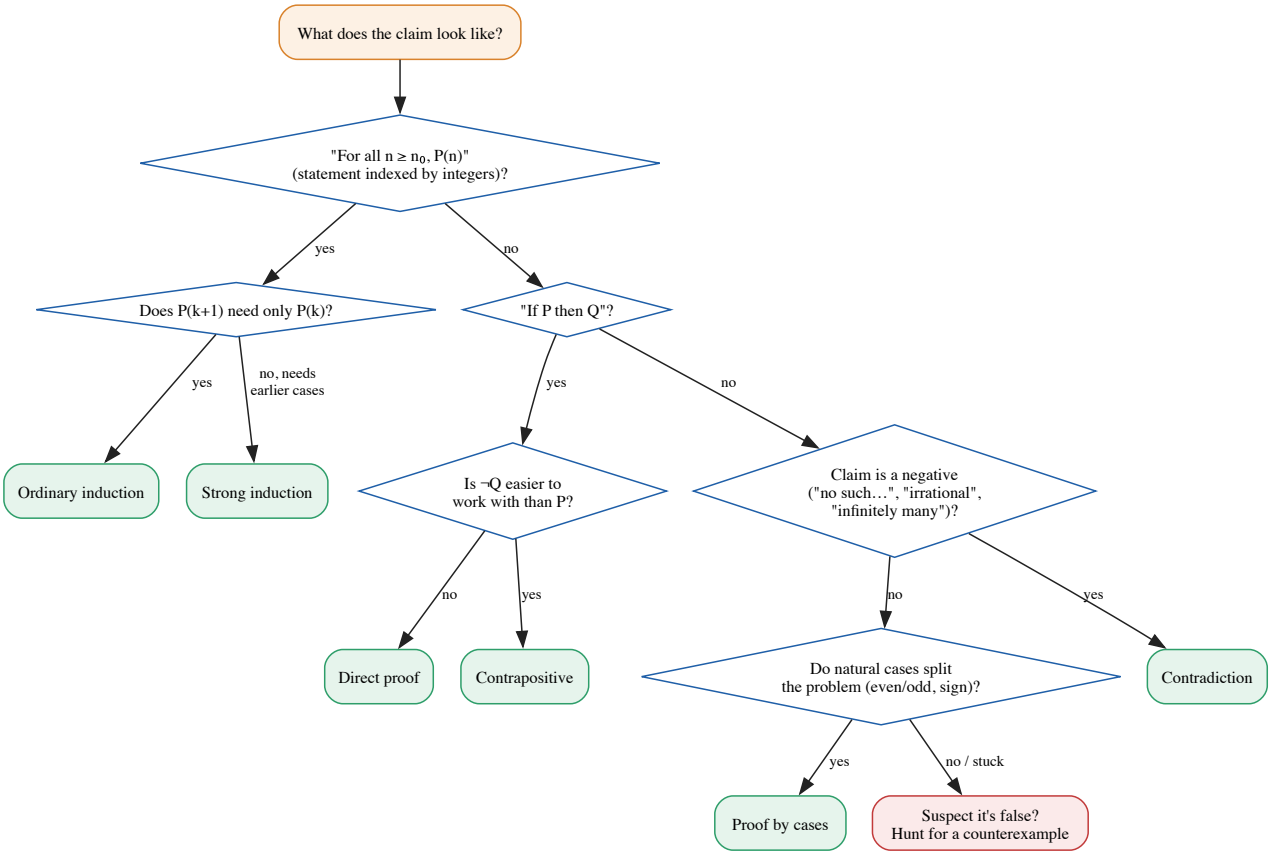


Figure 16. A first guess at which technique to try. Whatever the shape, spend a few minutes hunting for a counterexample before investing in a proof.

In words:

- **Is it false?** Try edge cases and a quick script first.
- **Indexed by integers or defined recursively?** Induction; strong induction if the step needs more than the previous case; structural induction for trees, lists and expressions.
- **"If P then Q" with a concrete P?** Direct proof.
- **Conclusion is negative, or an "or", or P is awkward to unpack?** Contrapositive.
- **Claim is a bare negative or about non-existence or infinitude?** Contradiction.
- **Objects split naturally into types?** Cases, with an exhaustiveness check.
- **Is it code?** Invariant + variant, which is induction plus well-ordering.

22.17. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Language sandbox

22.18. Mastery check

1. Prove that the sum of any three consecutive integers is divisible by 3.
2. Prove: for every integer n, if $5n + 4$ is odd then n is odd. Name the technique.
3. Prove that $\log_2 3$ is irrational.
4. Disprove: "for every real number x, $x^2 > x$."
5. Prove by induction that $\sum_{i=1}^{n} i(i+1) = \frac{n(n+1)(n+2)}{3}$ for all $n \geq 1$.
6. Prove by induction that $3^n \geq 2n + 1$ for all $n \geq 0$.
7. State a loop invariant and a variant for the following function (assume `e >= 0`), and use them to prove it returns b^e.

```python
def power(b, e):
    result, i = 1, 0
    while i < e:
        result *= b
        i += 1
    return result
```

8. Find the error: "Claim: every positive integer is less than 100. Inductive step: assume $k < 100$. Then $k + 1 < 101$... and so on, so $k + 1 < 100$."

Decision rule: All correct without hesitation → move on to Chapter 23, where you will use these techniques constantly. Stuck on 1–4 → reread Section 22.3, Section 22.4 and Section 22.5. Stuck on 5–6 → reread Section 22.9 and redo its exercises. Stuck on 7 → reread Section 22.13. Stuck on 8 → reread Section 22.15.

▼ **Mastery check answers**

1. **Direct.** Three consecutive integers are $n - 1, n, n + 1$. Their sum is $(n - 1) + n + (n + 1) = 3n$, which is divisible by 3.
2. **Contrapositive:** if n is even then $5n + 4$ is even. Let $n = 2k$; then $5n + 4 = 10k + 4 = 2(5k + 2)$, which is even. Hence if $5n + 4$ is odd, n is odd.
3. **Contradiction.** Suppose $\log_2 3 = p/q$ with integers $p, q, q > 0$. Since $\log_2 3 > 0$ (because $3 > 1$), also $p > 0$. Then $2^{p/q} = 3$, and raising both sides to the power q gives $2^p = 3^q$. The left side is even ($p \geq 1$) and the right side is odd (a product of odd numbers). Contradiction.
4. **Counterexample:** $x = 1$ gives $1 > 1$, false. (Any x with $0 \leq x \leq 1$ works, e.g. $x = \frac{1}{2}$.)
5. Base $n = 1$: left $1 \cdot 2 = 2$, right $\frac{1 \cdot 2 \cdot 3}{3} = 2$. Step:

$$\begin{aligned}\sum_{i=1}^{k+1} i(i+1) &= \frac{k(k+1)(k+2)}{3} + (k+1)(k+2) && \text{(IH)}\\ &= \frac{k(k+1)(k+2) + 3(k+1)(k+2)}{3} \\ &= \frac{(k+1)(k+2)(k+3)}{3}. && \text{(factor out } (k+1)(k+2))\end{aligned}$$

6. Base $n = 0$: $3^0 = 1 \geq 1 = 2 \cdot 0 + 1$. Step: assume $3^k \geq 2k + 1$. Then $3^{k+1} = 3 \cdot 3^k \geq 3(2k+1) = 6k + 3 \geq 2k + 3 = 2(k+1) + 1$, using $6k \geq 2k$ for $k \geq 0$.

7. Invariant: $\texttt{result} = b^{\,i}$ and $0 \leq i \leq e$. Initialisation: $1 = b^0$, $i = 0$. Maintenance: if $\texttt{result} = b^i$ and $i < e$, then after the body $\texttt{result} = b^{i+1}$ and $i + 1 \leq e$. Variant: $e - i$, a non-negative integer that decreases by 1 each iteration, so the loop terminates. Exit: $i \geq e$ and $i \leq e$ give $i = e$, so $\texttt{result} = b^e$.

8. The base case is missing ($1 < 100$ is true, so that part is easy to add), but the real error is in the inductive step, which does not follow: $k < 100$ gives only $k + 1 < 101$, which allows $k + 1 = 100$; "and so on" hides the gap. Indeed $k = 99$ satisfies the hypothesis while $k + 1 = 100$ fails the conclusion, so $P(k) \Rightarrow P(k+1)$ is false. (The claim is false: 100 is a counterexample.)

Chapter 23. Integers, Divisibility, and Modular Arithmetic

Chapter map: we go back to the whole numbers and ask questions that ordinary arithmetic skips. When does one number divide another? What are the indivisible building blocks? How do you find the largest common divisor of two 300-digit numbers without factoring them? Then we wrap the number line into a circle. That "clock arithmetic" turns out to be the arithmetic your CPU already does. It is also the arithmetic behind hash tables, check digits, random-number generators, and the public-key cryptography that protects nearly every web connection you make. The chapter ends by running RSA by hand, with nothing hidden.

Where this chapter sits

Prerequisites: whole numbers, integers, and the order of operations (Volume 1, Chapter 1); factors, multiples and common denominators (Volume 1, Chapter 2); exponents and their laws (Volume 1, Chapter 3); using letters for numbers (Volume 1, Chapter 4). No algebra beyond that is assumed. The occasional "proof" here is short and informal. Chapter 22 treats proof technique in its own right, and you can read the two chapters in either order.

Unlocks: counting arguments in Chapter 24; randomness and simulation in Chapter 29; and every later mention of hashing, checksums, or cryptography. Modular arithmetic also gives you a useful way to look at the overflow behaviour of fixed-width integers, which Volume 2, Chapter 32 picks up again.

Number theory has a reputation as the "purest" corner of mathematics. G. H. Hardy boasted in 1940 that it had no practical use whatsoever. Thirty-seven years later RSA made it one of the most practically important subjects there is. Keep that irony in mind. Everything in this chapter is elementary, meaning it needs nothing past arithmetic, but none of it is trivial.

23.1. Divisibility and the division algorithm

23.1.1. The problem

You have 1,071 identical tiles and want to lay them in a complete rectangle with no gaps and none left over. Which rectangle shapes work? You have 7 days of logs and want them bucketed by weekday: which bucket does day 1,000 fall in? A loop has to process items in batches of 64. How many full batches are there, and how many items are left over?

All three questions ask the same thing: what happens when one whole number is divided by another *and we refuse to leave the integers*. There are no fractions and no decimals. There is a whole-number quotient and a whole-number remainder.

23.1.2. Intuition

Think of dividing 23 by 5 as dealing 23 cards into piles of 5. You make 4 complete piles and 3 cards are left over:

$$23 = 4 \cdot 5 + 3.$$

The leftover is always smaller than the pile size. If it weren't, you could make one more pile. That simple fact is what everything in this chapter rests on.

23.1.3. Vocabulary and notation

Dividend, divisor, quotient, remainder

In $23 = 4 \cdot 5 + 3$ the number being divided (23) is the **dividend**. The number you divide by (5) is the **divisor**. The number of whole times it fits (4) is the **quotient**, and what is left (3) is the **remainder**. "Quotient" comes from the Latin *quotiens*, "how many times".

Divides, factor, multiple

For integers a and b, we say a **divides** b, written $a \mid b$, if $b = ka$ for some integer k. Equivalently, dividing b by a leaves remainder 0. Then a is a **divisor** (or **factor**) of b, and b is a **multiple** of a.

So $3 \mid 12$ because $12 = 4 \cdot 3$, and $5 \nmid 12$ ("5 does not divide 12"). The vertical bar is a *statement* (true or false), not an operation: $3 \mid 12$ is not the number 4. Don't confuse it with the fraction $3/12$.

Even and odd, parity

An integer is **even** if $2 \mid n$ and **odd** otherwise. Whether a number is even or odd is its **parity**. Every odd number can be written as $2k + 1$ for some integer k.

23.1.4. The division algorithm

The card-dealing picture becomes a theorem. (The name is old. It is a statement that a quotient and remainder exist, not a step-by-step algorithm.)

Division algorithm (quotient–remainder theorem)

For any integer a and any positive integer b, there are **unique** integers q (the quotient) and r (the remainder) with

$$a = qb + r, \qquad 0 \le r < b.$$

We write $r = a \bmod b$, read "a mod b".

The condition $0 \le r < b$ is what makes the answer unique. Without it, $23 = 3 \cdot 5 + 8$ would also be a valid split. With it, there is only one choice: take q to be the largest integer with $qb \le a$. That works for negative a too:

$$-23 = (-5) \cdot 5 + 2, \qquad \text{so } -23 \bmod 5 = 2.$$

Note that the quotient is -5, not -4. Since $(-4)\cdot 5 = -20$, that choice would give remainder -3, which breaks $0 \le r$. Programming languages disagree about this case. Section 23.7 sorts out who does what.

23.1.5. Why divisibility behaves nicely

Three facts follow directly from the definition. Each one is a one-line argument, so here they are:

1. **If $d \mid a$ and $d \mid b$, then $d \mid (a+b)$ and $d \mid (a-b)$.** Write $a = kd$ and $b = ld$. Then $a \pm b = (k \pm l)d$.
2. **If $d \mid a$, then $d \mid ca$ for every integer c.** $ca = (ck)d$.
3. **Divisibility is transitive: if $a \mid b$ and $b \mid c$, then $a \mid c$.** $c = mb = m(ka) = (mk)a$.

Combining the first two: if d divides both a and b, then d divides every **integer combination** $xa + yb$. We'll lean on this sentence repeatedly. It is the engine behind both the Euclidean algorithm and Bézout's identity.

23.1.6. Worked example: tiles and weekdays

Which rectangles can 1,071 tiles make? A rectangle with sides $w \times h$ uses wh tiles, so we need the divisors of 1,071. Divide by small numbers:

$$\begin{aligned} 1071 &= 3 \cdot 357 && \text{(digit sum } 1+0+7+1 = 9 \text{ is divisible by 3)} \\ 357 &= 3 \cdot 119 && \text{(digit sum 15)} \\ 119 &= 7 \cdot 17 \\ \text{so } 1071 &= 3^2 \cdot 7 \cdot 17. \end{aligned}$$

Each divisor picks 0, 1 or 2 threes, 0 or 1 seven, and 0 or 1 seventeen, so there are $3 \cdot 2 \cdot 2 = 12$ divisors: 1, 3, 7, 9, 17, 21, 51, 63, 119, 153, 357, 1071. That gives six rectangle shapes up to rotation: 1×1071, 3×357, 7×153, 9×119, 17×63 and 21×51.

Day 1,000. If day 0 is a Monday, then day 1,000 lands on weekday $1000 \bmod 7$:

$$1000 = 142 \cdot 7 + 6 \quad \Rightarrow \quad 1000 \bmod 7 = 6,$$

six days after Monday, a Sunday.



> *Divisibility tests are modular arithmetic in disguise*
>
> The schoolroom test "a number is divisible by 3 (or 9) exactly when its digit sum is" isn't a coincidence. We'll prove it in two lines in Section 23.6, along with the stranger test for 11: alternate the signs of the digits. For 1,071 that gives $1 - 7 + 0 - 1 = -7$, which is not a multiple of 11, so 11 does not divide 1,071.

US curriculum

Factors, multiples, and divisibility rules come in **grades 4–6** (Common Core 4.OA.B.4: "find all factor pairs for a whole number in the range 1–100"). Remainders appear with long division in grades 4–5. The formal division algorithm and the $a \mid b$ notation usually wait until a college **Discrete Mathematics** course or a competition-math class (MATHCOUNTS, AMC).

23.2. Primes and the fundamental theorem of arithmetic

23.2.1. The problem

Some numbers break into smaller factors: $1071 = 3^2 \cdot 7 \cdot 17$. Some don't: 17 has no divisors except 1 and itself. Is there a finite list of these "atoms", or do they go on forever? And when you break a number into atoms, can two different people end up with two different answers?

23.2.2. Vocabulary



Prime, composite, unit

A **prime** is an integer $p > 1$ whose only positive divisors are 1 and p. An integer $n > 1$ that is not prime is **composite**: it can be written $n = ab$ with $1 < a, b < n$. The number 1 is **neither**. It is called a **unit**.

The primes start 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … and 2 is the only even one.



Prime factorization

A **prime factorization** writes a number as a product of primes, usually grouped with exponents in increasing order: $360 = 2^3 \cdot 3^2 \cdot 5$. The exponent of p is called the **multiplicity** of p.



Why isn't 1 prime?

If 1 counted as prime, factorizations would stop being unique: $6 = 2 \cdot 3 = 1 \cdot 2 \cdot 3 = 1 \cdot 1 \cdot 2 \cdot 3 = \cdots$. Mathematicians *chose* to exclude 1 so that the theorem below could be stated cleanly. Definitions are design decisions, just as they are in an API.

23.2.3. The fundamental theorem of arithmetic



Fundamental theorem of arithmetic

Every integer $n > 1$ can be written as a product of primes, and this factorization is **unique** apart from the order of the factors.

It has two halves, and they deserve different amounts of attention.

Existence is the easy half. Take n. If it is prime, you're done. If not, split it as $n = ab$ with both factors smaller, and keep splitting. The numbers shrink every time, so the process must stop, and when it stops every piece is prime. If you sketch the splits you get a **factor tree**:

$$360 \to 36 \cdot 10 \to (6 \cdot 6)(2 \cdot 5) \to (2 \cdot 3)(2 \cdot 3)(2 \cdot 5) = 2^3 \cdot 3^2 \cdot 5.$$

Uniqueness is the deep half. If you'd started with $360 = 8 \cdot 45$, could you have finished with different primes? The answer is no, but the reason needs a fact we don't have yet, *Euclid's lemma* (if a prime divides a product, it divides one of the factors). We'll prove that lemma in Section 23.5 using the Euclidean algorithm. Uniqueness then follows quickly. Two factorizations of the same number share any prime p that appears in the first, because p divides the product on the second side and so must divide (that is, equal) one of its primes. Cancel it from both sides and repeat.



Uniqueness can fail

Uniqueness isn't automatic. In the number system made of all $a + b\sqrt{-5}$ with integer a, b, the number 6 has two genuinely different factorizations into "atoms": $6 = 2 \cdot 3 = (1 + \sqrt{-5})(1 - \sqrt{-5})$. In the 1840s, mathematicians attacking Fermat's Last Theorem had assumed unique factorization in systems like this one. When Kummer noticed it can fail there, the repair led to much of modern algebra.

23.2.4. Finding a factorization: trial division

To factor n, try dividing by 2, 3, 4, 5, … in turn. Whenever a trial divisor d works, divide it out as many times as it goes and carry on with what's left. Two observations make this efficient enough for small numbers.

- **You only need to try d up to $\sqrt{n}$.** If $n = ab$ with $a \le b$, then $a \cdot a \le ab = n$, so $a \le \sqrt{n}$. Any composite number has a factor no larger than its square root. If nothing up to $\sqrt{n}$ divides n, then n is prime.
- **Composite trial divisors never succeed**, because their prime factors were already divided out. Trying 4 after removing every 2 is wasted work but harmless.

Example 13. Procedure — Prime factorization by trial division

Input: an integer $n > 1$.

Steps:

1. Set $d = 2$ and start with an empty list of factors.
2. While $d \cdot d \leq n$:
   a. While $d \mid n$: record d as a factor and replace n by n/d.
   b. Increase d by 1.
3. If the remaining $n > 1$, it is prime: record it.

Output: the prime factors of the original n, with multiplicity.

PYTHON

```
def factorize(n):
    """Prime factorization by trial division: returns {prime: exponent}."""
    factors = {}
    d = 2
    while d * d <= n:
        while n % d == 0:
            factors[d] = factors.get(d, 0) + 1
            n //= d
        d += 1
    if n > 1:
        factors[n] = factors.get(n, 0) + 1
    return factors

print(factorize(360), factorize(97), factorize(1001), factorize(2**10 * 3**3))
```

```
{2: 3, 3: 2, 5: 1} {97: 1} {7: 1, 11: 1, 13: 1} {2: 10, 3: 3}
```

Worked example: is 391 prime? $\sqrt{391} \approx 19.8$, so try primes up to 19. It isn't even. Its digit sum is 13, so 3 fails. It doesn't end in 0 or 5. $391 = 7 \cdot 55 + 6$, so 7 fails. For 11, the alternating sum $1 - 9 + 3 = -5$ is not a multiple of 11. $391 = 13 \cdot 30 + 1$, so 13 fails. Then $391 = 17 \cdot 23$. Composite. Numbers like this, a product of two primes of similar size, are exactly the kind RSA uses, just millions of times larger. *Try it:* Is 391 prime? Step through ; Factorise 391 .



Trial division doesn't scale

Trial division takes about $\sqrt{n}$ steps. For a 20-digit number that is 10^{10} steps, which is a few seconds. For a 600-digit RSA modulus it is 10^{300} steps, far more than the age of the universe on every computer ever built. Better factoring algorithms exist (the quadratic sieve and the general number field sieve), but none of them is remotely fast enough at that size. **Multiplying is easy and factoring is hard.** That asymmetry is the foundation of RSA.

23.2.5. How many primes are there?

Infinitely many. Euclid's proof, about 300 BCE, is one of the first proofs anyone should see, and it appears in full in Chapter 22. In brief: given any finite list of primes, multiply them all together and add 1. The result leaves remainder 1 when divided by each listed prime, so its prime factors are all new. No finite list can be complete.

The primes also thin out. Among the first 100 integers, 25 are prime. Among the first million, 78,498 are. The **prime number theorem** (1896) says the number of primes up to n is roughly $n / \ln n$. For $n = 10^6$ that estimate gives about 72,382, which is low by 8% but captures the trend. Near a 300-digit number, roughly one integer in 690 is prime. That is plenty for RSA, which needs to *find* big primes quickly.



> *The largest known primes*
>
> The record primes are almost always **Mersenne primes** of the form $2^p - 1$, because a fast special-purpose test (Lucas–Lehmer) exists for that shape. As of late 2024 the largest known was $2^{136\,279\,841} - 1$, which has over 41 million digits. It was found by the volunteer GIMPS project running on cloud GPUs. In binary it is a string of 136,279,841 ones.

23.2.6. Where you'd use this

Everyday: common denominators. To add $\frac{5}{12} + \frac{7}{18}$, factor: $12 = 2^2 \cdot 3$ and $18 = 2 \cdot 3^2$. The least common denominator takes the highest power of each prime, $2^2 \cdot 3^2 = 36$, so the sum is $\frac{15}{36} + \frac{14}{36} = \frac{29}{36}$. Uniqueness of factorization is what makes "highest power of each prime" a well-defined recipe.

Biology: prime-numbered life cycles. Periodical cicadas in North America spend 13 or 17 years underground and then emerge together in their billions. One leading explanation is arithmetic. A predator whose population peaks every 2, 3, 4, 5 or 6 years coincides with a 17-year brood only every lcm of the two periods: 34, 51, 68, 85 or 102 years. A 12-year cicada would meet the 2-, 3-, 4- and 6-year predators every single emergence.

Engineering: hunting-tooth gears. A car's final drive often pairs an 11-tooth pinion with a 41-tooth ring gear (ratio $41/11 \approx 3.73$). Because 41 is prime, the two counts share no factor, so a given pinion tooth meets every one of the 41 ring-gear teeth before any pairing repeats, after $11 \times 41 = 451$ tooth engagements. Any manufacturing flaw on one tooth is spread over the whole gear instead of grinding the same spot.

Computing. Public-key cryptography rests on the asymmetry between multiplying two large primes (microseconds) and factoring the product (beyond any known classical computer for 2,048-bit numbers); Section 23.12 runs the numbers.

US curriculum

Primes, composites and prime factorization ("factor trees") belong to **grades 4–6**. Common Core 6.NS.B.4 uses factorization to find the greatest common factor and least common multiple. Infinitely many primes and the uniqueness of factorization are usually first *proved* in a college **Discrete Mathematics** or **Number Theory** course.

23.3. The sieve of Eratosthenes

23.3.1. The problem

Trial division tests one number at a time. Often you want *all* the primes up to some bound: to precompute a lookup table, to factor many small numbers, or to generate test data. Testing each of a million numbers separately repeats a lot of work. Can we share it?

23.3.2. Intuition

Turn the question around. Instead of asking "what divides this number?", ask "which numbers does this prime knock out?" Write down 2 through n. The first number, 2, is prime, so cross out every multiple of 2 after it. The next number not crossed out, 3, must be prime, because nothing smaller divides it. Cross out its multiples. Then 5, then 7, and so on. Whatever survives is prime.

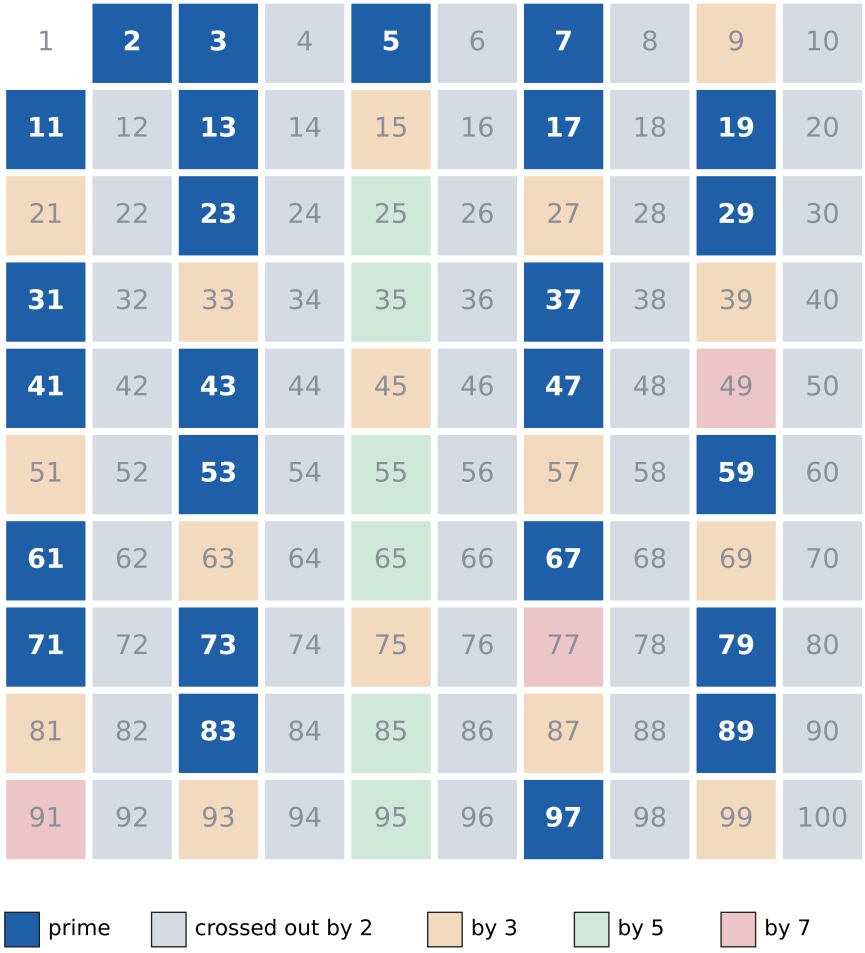


Figure 17. The sieve of Eratosthenes on 1–100. Every composite is crossed out by its smallest prime factor; only 2, 3, 5 and 7 are needed, since $11^2 = 121 > 100$.

23.3.3. Why it works, and two shortcuts

A number survives exactly when no smaller prime divides it, and that is the definition of prime. Two refinements matter. *Try it:* Animate the sieve to 100 .

- **Start crossing at p^2.** Any smaller multiple of p, say kp with $k < p$, has a prime factor smaller than p and was already crossed out.
- **Stop once $p^2 > n$.** By the square-root argument from trial division, every composite up to n has a prime factor at most $\sqrt{n}$.

Example 14. Procedure — Sieve of Eratosthenes

Input: an integer $n \geq 2$.

Steps:

1. Make a table `is_prime[0..n]`, all `True`. Set entries 0 and 1 to `False`.
2. For $p = 2, 3, 4, \ldots$ while $p^2 \leq n$:
   a. If `is_prime[p]` is still `True`, set `is_prime[m] = False` for $m = p^2, p^2 + p, p^2 + 2p, \ldots \leq n$.
3. Collect every k with `is_prime[k]` true.

Output: the list of primes up to n.

23.3.4. Worked example: primes up to 30

Start with 2 through 30. Since $\sqrt{30} \approx 5.48$, only $p = 2, 3, 5$ are needed. *Try it:* Sieve to 30 by rounds; The primes up to 30.

Step	Crossed out
$p = 2$ (start at 4)	4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30
$p = 3$ (start at 9)	9, 15, 21, 27 *(12, 18, 24, 30 were already gone)*
$p = 4$	already crossed out, skip
$p = 5$ (start at 25)	25 *(30 already gone)*
$p = 6$	$36 > 30$, stop

Survivors: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. Ten primes.

PYTHON

```python
def sieve(n):
    """Return the list of primes <= n (sieve of Eratosthenes)."""
    if n < 2:
        return []
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    p = 2
    while p * p <= n:
        if is_prime[p]:
            for multiple in range(p * p, n + 1, p):
                is_prime[multiple] = False
        p += 1
    return [k for k in range(n + 1) if is_prime[k]]

print(sieve(50))
print(len(sieve(1_000_000)))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
78498
```

23.3.5. Choosing a method

- **One number, small (up to about 10^{12}):** trial division up to $\sqrt{n}$.
- **All primes up to n, with n up to about 10^{8}:** the sieve. It runs in about $n \log \log n$ operations, which is nearly linear, but it needs n bits of memory. Segmented sieves process the range in cache-sized blocks.
- **One huge number (hundreds of digits):** neither. Use a probabilistic test such as Miller–Rabin, which is built on Fermat's little theorem (Section 23.9).



Sieve memory

A Python list of a billion booleans takes about 8 GB, because each entry is a pointer. Use `bytearray(n + 1)` (1 byte per entry) or a NumPy boolean array if you need big sieves. The algorithm is the same, but you get 8 times the reach.

Sieve as a pattern

The sieve is the first example many programmers meet of trading memory for time and of *inverting a loop*. Instead of "for each candidate, search for a divisor", it does "for each divisor, mark its multiples". The same inversion shows up in computing all divisor sums, in Euler's totient sieve, in the smallest-prime-factor table that lets you factor any number up to n in $O(\log n)$ steps, and in database-style "push" versus "pull" evaluation.

Watch the sieve

Animate the sieve on a 20 × 20 grid. Each prime colours its multiples in its own colour, starting at p^2. Let the reader change the grid width. With width 6, the primes after 3 line up in just two columns (the numbers of the form $6k \pm 1$). With width 10, they fall in the columns ending 1, 3, 7, 9. Ask why: the other columns are multiples of 2 or 3, or of 2 or 5.

Try it: Watch the sieve, width 6 · The primes up to 400

23.3.6. Exercises

1. Which of these are prime: 91, 97, 119, 127? Explain how you know when to stop testing.
2. Factorize 1001, 2024 and 5040 by trial division, showing each division.
3. Run the sieve by hand up to 60. Which primes do you need to sieve with?
4. Prove that every prime greater than 3 is of the form $6k + 1$ or $6k + 5$. (Consider the six possible remainders mod 6.)

▼ **Answers**

1. $91 = 7 \cdot 13$ (composite). For 97, $\sqrt{97} < 10$, and 2, 3, 5, 7 all fail ($97 = 13 \cdot 7 + 6$), so it is prime. $119 = 7 \cdot 17$ (composite). For 127, $\sqrt{127} < 12$, and 2, 3, 5, 7, 11 all fail ($127 = 18 \cdot 7 + 1 = 11 \cdot 11 + 6$), so it is prime. Stop once the trial divisor exceeds $\sqrt{n}$.
2. $1001 = 7 \cdot 143 = 7 \cdot 11 \cdot 13$. $2024 = 2 \cdot 1012 = 2^2 \cdot 506 = 2^3 \cdot 253 = 2^3 \cdot 11 \cdot 23$. $5040 = 2^4 \cdot 315 = 2^4 \cdot 3^2 \cdot 35 = 2^4 \cdot 3^2 \cdot 5 \cdot 7$ (this is $7!$).
3. $\sqrt{60} < 8$, so sieve with 2, 3, 5 and 7 (7 first crosses out 49). Primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, which makes 17 of them.
4. Any integer has remainder 0–5 mod 6. Remainders 0, 2, 4 make it even. Remainder 3 makes it a multiple of 3 ($6k + 3 = 3(2k + 1)$). A prime greater than 3 is neither, so its remainder is 1 or 5.

23.4. Greatest common divisor and the Euclidean algorithm

23.4.1. The problem

To simplify the fraction $\frac{462}{1071}$ you need the largest number dividing both. To tile a 48×18 floor with identical square tiles and no cutting, you need the largest square side that divides both lengths. To find the period of two interlocking gears, or to normalise an aspect ratio such as $1920 \times 1080 \to 16 : 9$, you need the same thing again: the **greatest common divisor**.

You could factor both numbers and compare. For $462 = 2 \cdot 3 \cdot 7 \cdot 11$ and $1071 = 3^2 \cdot 7 \cdot 17$, the shared part is $3 \cdot 7 = 21$. But factoring is the hard problem we just met. For 300-digit numbers it is hopeless. Remarkably, there is an algorithm that finds the GCD *without factoring anything*, and it is about 2,300 years old.

GCD, coprime

The **greatest common divisor** $\gcd(a, b)$ of two integers, not both zero, is the largest positive integer dividing both. US schools call it the **greatest common factor (GCF)**, and UK schools the **highest common factor (HCF)**. If $\gcd(a, b) = 1$, then a and b are **coprime** (or **relatively prime**). They need not be prime themselves: 8 and 15 are coprime. By convention $\gcd(a, 0) = |a|$.

23.4.2. Intuition: tiles

Take the 48×18 rectangle. The biggest square you can cut from it is 18×18, and you can fit two of them, leaving a 12×18 strip. Any square tile that fits the original rectangle exactly also fits the squares (its side divides 18) and therefore fits the leftover strip. Conversely, a tile that fits the 18×12 strip fits everything. So the problem has shrunk from $(48, 18)$ to $(18, 12)$ with the same answer. Repeat: one 12×12 square leaves 12×6. Then two 6×6 squares fill it exactly. The answer is 6. *Try it:* Tile 48 × 18 with squares .

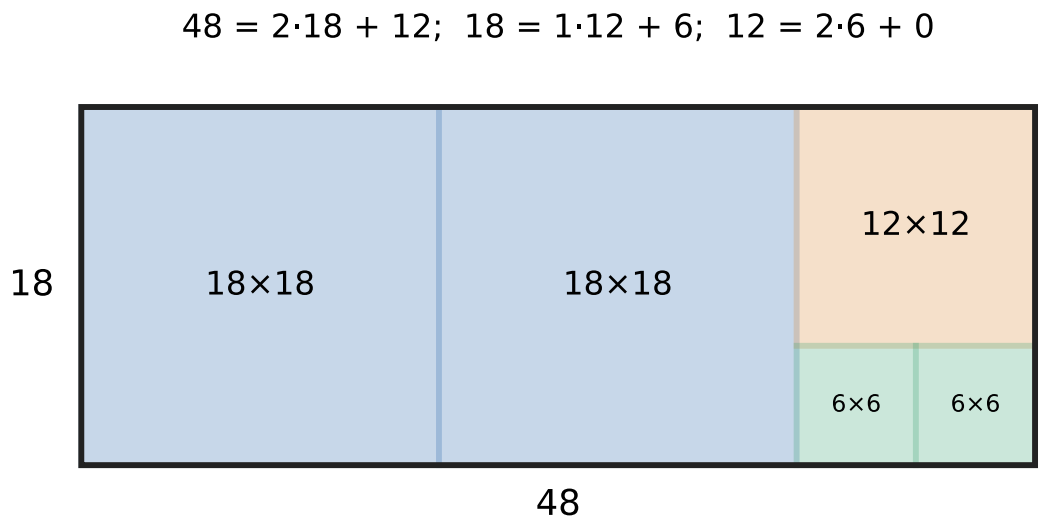


Figure 18. Euclid's algorithm as tiling. Each step cuts off as many largest squares as fit; the last square size that tiles the leftover exactly is the GCD.

23.4.3. Why it works

The tiling picture is the following fact, written as algebra.

$$\gcd(a, b) = \gcd(b,\ a \bmod b) \qquad (b > 0).$$

Proof. Write $a = qb + r$ with $r = a \bmod b$. If d divides a and b, it divides $r = a - qb$, because that is an integer combination of a and b. Conversely, if d divides b and r, it divides $a = qb + r$. So the pairs (a, b) and (b, r) have *exactly the same common divisors*, and in particular the same greatest one. □

Each step replaces the pair with a strictly smaller remainder, and remainders can't go below 0, so the process must stop. It stops when the remainder is 0, and then $\gcd(g, 0) = g$.

Example 15. Procedure — Euclidean algorithm

Input: integers $a, b \geq 0$, not both zero.

Steps:

1. If $b = 0$, return a.
2. Otherwise compute $r = a \bmod b$.
3. Replace (a, b) with (b, r) and go to step 1.

Output: $\gcd(a, b)$, the last nonzero remainder.

23.4.4. Worked example: $\gcd(1071, 462)$

Every division is written out in full: *Try it:* gcd(1071, 462) as a trace ; gcd and Bézout for 1071, 462 .

$$\begin{aligned} 1071 &= 2 \cdot 462 + 147 && (2 \cdot 462 = 924,\ 1071 - 924 = 147) \\ 462 &= 3 \cdot 147 + 21 && (3 \cdot 147 = 441,\ 462 - 441 = 21) \\ 147 &= 7 \cdot 21 + 0 && (7 \cdot 21 = 147) \end{aligned}$$

The last nonzero remainder is 21, so $\gcd(1071, 462) = 21$. It took three divisions, and nothing was factored. Check: $1071 = 21 \cdot 51$ and $462 = 21 \cdot 22$, and $\gcd(51, 22) = 1$. So

$$\frac{462}{1071} = \frac{22}{51}.$$

As a trace table, which is how you'd debug it:

a	b	$q = \lfloor a/b \rfloor$	$r = a \bmod b$
1071	462	2	147
462	147	3	21
147	21	7	0
21	0	—	(return 21)

```python
def gcd(a, b):
    """Greatest common divisor of integers a, b (not both zero)."""
    a, b = abs(a), abs(b)
    while b != 0:
        a, b = b, a % b      # replace (a, b) with (b, remainder)
    return a

print(gcd(1071, 462), gcd(462, 1071), gcd(17, 5), gcd(0, 9))
```

```
21 21 1 9
```

Notice that `gcd(462, 1071)` works too. The first step computes $462 \bmod 1071 = 462$ and simply swaps the pair. In real code, use `math.gcd`.

23.4.5. How fast is it?

Every two steps, the remainder at least halves. (If $b \le a/2$, then $r < b \le a/2$. If $b > a/2$, then $q = 1$ and $r = a - b < a/2$.) So the number of steps is at most about $2 \log_2 b$: around 2,000 divisions for 1,000-bit numbers, which takes microseconds. Compare that with the astronomically slow factoring approach.



The first analysed algorithm

The algorithm appears in Euclid's *Elements* (Book VII, around 300 BCE), stated with lengths of line segments, just like our tiles. In 1844 Gabriel Lamé proved that the worst case is consecutive Fibonacci numbers. For example, $\gcd(89, 55)$ takes 9 division steps because every quotient except the last is 1. His bound, at most five times the number of decimal digits, is often called the first worst-case analysis of an algorithm's running time.

Try it: Lamé's worst case, 89 and 55.

23.4.6. Alternative approaches and choosing one

- **Compare factorizations.** $\gcd$ takes each shared prime to the *smaller* exponent. This is fine when you already have the factorizations or the numbers are tiny, and it explains *what* the GCD is. Hopeless for big numbers.
- **Subtraction-only Euclid.** Replace the larger number by the difference until the two are equal. It is correct, but slow when one number is much bigger: $\gcd(10^9, 1)$ takes a billion steps.
- **Binary GCD (Stein's algorithm).** It uses only shifts and subtraction, exploiting $\gcd(2a, 2b) = 2\gcd(a, b)$, and was attractive on hardware with slow division.
- **Euclid with remainders** is the default for everything else.

23.4.7. The least common multiple



LCM

The **least common multiple** $\operatorname{lcm}(a, b)$ is the smallest positive integer that both a and b divide. It is the "lowest common denominator" you used when adding fractions in Volume 1, Chapter 2.

Comparing factorizations shows what's going on. GCD takes the **minimum** exponent of each prime and LCM takes the **maximum**. Since $\min(x, y) + \max(x, y) = x + y$, multiplying the two gives back ab:

$$\gcd(a, b) \cdot \operatorname{lcm}(a, b) = a \cdot b, \qquad \text{so} \qquad \operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}.$$

Worked example. For $a = 252 = 2^2 \cdot 3^2 \cdot 7$ and $b = 198 = 2 \cdot 3^2 \cdot 11$: *Try it:* gcd and lcm of 252, 198 ; gcd and lcm of 252, 198 .

$$\begin{aligned} 252 &= 1 \cdot 198 + 54 \\ 198 &= 3 \cdot 54 + 36 \\ 54 &= 1 \cdot 36 + 18 \\ 36 &= 2 \cdot 18 + 0 \qquad\qquad \Rightarrow \gcd = 18 \end{aligned}$$

$$\operatorname{lcm}(252, 198) = \frac{252 \cdot 198}{18} = 252 \cdot 11 = 2772.$$

The factorizations agree: the GCD is $2^1 \cdot 3^2 = 18$ and the LCM is $2^2 \cdot 3^2 \cdot 7 \cdot 11 = 2772$.



Overflow in lcm

In fixed-width integer languages, compute `a / gcd(a, b) * b`, not `a * b / gcd(a, b)`. The product ab can overflow even when the LCM itself fits.

Applications: two cron jobs running every 6 and every 10 minutes coincide every $\operatorname{lcm}(6, 10) = 30$ minutes. Gears with 12 and 18 teeth return to their starting alignment after $\operatorname{lcm}(12, 18) = 36$ tooth-passes. Planetary conjunctions and traffic-light cycles work the same way.

Computer-science connection: GCD everywhere

- **Fractions and rationals:** Python's `fractions.Fraction` calls `gcd` on every operation to keep results in lowest terms.
- **Aspect ratios and pixel art:** 1920×1080 reduces by $\gcd = 120$ to 16×9.
- **Rotating an array in place:** the "juggling" algorithm rotates an array of length n by k in $\gcd(n, k)$ independent cycles.
- **RSA:** key generation needs $\gcd(e, \varphi) = 1$ and a modular inverse. That is the next section.

US curriculum

GCF and LCM by listing or factor trees: **grade 6** (Common Core 6.NS.B.4). The Euclidean algorithm itself is rarely taught in US schools. It turns up in **Discrete Mathematics** (college, or high-school electives), in competition math, and in AP Computer Science as a recursion example.

23.5. Extended Euclid and Bézout's identity

23.5.1. The problem

Here is a puzzle with a long history. You have a 1,071 ml jug and a 462 ml jug and unlimited water. Can you measure exactly 21 ml? Measuring means pouring whole jugfuls in and out of a tank, so the amounts you can reach are $1071x + 462y$ for integers x, y, where negative means "poured out". The real question is which numbers can be written as $1071x + 462y$, and with which x and y.

This isn't just a puzzle. Finding such x, y is *the* way to compute modular inverses (Section 23.8), and modular inverses are how RSA keys get made.

23.5.2. Intuition

Every number of the form $1071x + 462y$ is a multiple of 21, because 21 divides both terms. So you can't measure 10 ml or 20 ml. The surprise is the converse: *21 itself* is reachable, so every multiple of 21 is too. The Euclidean algorithm's divisions, read backwards, tell you exactly how.



Bézout's identity

For integers a, b, not both zero, there exist integers x, y with

$$ax + by = \gcd(a, b).$$

The x, y are **Bézout coefficients**. They are not unique. A **linear combination** of a and b is any expression $ax + by$. The integer combinations of a and b are exactly the multiples of $\gcd(a, b)$.

23.5.3. The method: back-substitution

Take the divisions from the $\gcd(1071, 462)$ example and solve each one for its remainder: *Try it:* Back-substitution, line by line ; Check the Bézout coefficients .

$$\begin{aligned} 147 &= 1071 - 2 \cdot 462 \\ 21 &= 462 - 3 \cdot 147 \end{aligned}$$

Now start from the last line, which already expresses the GCD, and substitute the line above for 147:

$$\begin{aligned} 21 &= 462 - 3 \cdot 147 \\ &= 462 - 3 \cdot (1071 - 2 \cdot 462) && \text{(substitute for 147)} \\ &= 462 - 3 \cdot 1071 + 6 \cdot 462 && \text{(expand)} \\ &= 7 \cdot 462 - 3 \cdot 1071 && \text{(collect the 462s)} \end{aligned}$$

So $x = -3$ and $y = 7$. **Check:** $7 \cdot 462 = 3234$, $3 \cdot 1071 = 3213$, and $3234 - 3213 = 21$. With the jugs: fill the small jug 7 times into the tank and empty out the large jug's worth 3 times.

23.5.4. A forward version for programs

Back-substitution means storing every line. Code usually keeps a running pair instead. It tracks how each remainder is written in terms of the original a and b, and updates that alongside the remainders. Each row satisfies $r_i = ax_i + by_i$. *Try it:* The forward extended-Euclid table .

q	r	x	y	Check $1071x + 462y$
	1071	1	0	1071
	462	0	1	462
2	147	$1 - 2 \cdot 0 = 1$	$0 - 2 \cdot 1 = -2$	$1071 - 924 = 147$
3	21	$0 - 3 \cdot 1 = -3$	$1 - 3 \cdot (-2) = 7$	$-3213 + 3234 = 21$
7	0	$1 - 7 \cdot (-3) = 22$	$-2 - 7 \cdot 7 = -51$	$23562 - 23562 = 0$

Each new row equals (the row two above) minus q times (the row above), with q being the quotient of the two remainders above it. The row just before the 0 gives the GCD and its coefficients.

Example 16. Procedure — Extended Euclidean algorithm

Input: integers $a, b \geq 0$, not both zero.

Steps:

1. Start with two rows: $(r, x, y) = (a, 1, 0)$ and $(b, 0, 1)$.
2. While the latest $r \neq 0$:
   a. Let $q = \lfloor r_{\text{prev}} / r_{\text{latest}} \rfloor$.
   b. New row = previous row $- q \times$ latest row (applied to each of r, x, y).
3. The last row with $r \neq 0$ is (g, x, y).

Output: $g = \gcd(a, b)$ and x, y with $ax + by = g$.

```python
def extended_gcd(a, b):
    """Return (g, x, y) with g = gcd(a, b) and a*x + b*y == g."""
    old_r, r = a, b
    old_x, x = 1, 0
    old_y, y = 0, 1
    while r != 0:
        q = old_r // r
        old_r, r = r, old_r - q * r
        old_x, x = x, old_x - q * x
        old_y, y = y, old_y - q * y
    return old_r, old_x, old_y

print(extended_gcd(1071, 462))
```

```
(21, -3, 7)
```

23.5.5. Why it matters: Euclid's lemma

We promised a proof of unique factorization. Here is the missing piece.



Euclid's lemma

If a prime p divides a product ab, then $p \mid a$ or $p \mid b$.

Proof. Suppose $p \nmid a$. The only positive divisors of p are 1 and p, so $\gcd(p, a) = 1$. By Bézout, $px + ay = 1$ for some integers x, y. Multiply through by b:

$$pbx + aby = b.$$

p divides pbx obviously, and p divides aby because $p \mid ab$. So p divides their sum, which is b. □

The lemma needs p prime: $6 \mid 4 \cdot 9$ but $6 \nmid 4$ and $6 \nmid 9$. That one fact is why primes behave like atoms.



Bézout coefficients are not unique

If (x, y) works, so does $\left(x + k\frac{b}{g},\ y - k\frac{a}{g}\right)$ for every integer k. From the table, $(-3, 7)$ and $(-3 + 22,\ 7 - 51) = (19, -44)$ both give 21. Different libraries can return different (equally correct) pairs. When you need a canonical answer, such as a modular inverse in the range $0..m - 1$, reduce at the end.



Jug puzzles

Build a two-jug simulator: choose capacities a and b and allow fill, empty and pour. Highlight the reachable amounts. The reader should discover that exactly the multiples of $\gcd(a, b)$ (up to the larger capacity) are reachable. This is the *Die Hard 3* puzzle: measure 4 gallons with 3- and 5-gallon jugs. Show the Bézout coefficients the path corresponds to.

Try it: Play the jug puzzle · Bézout for jugs of 3 and 5

23.5.6. Exercises

1. Use the Euclidean algorithm to find $\gcd(84, 120)$ and $\gcd(89, 55)$. How many division steps does each take?
2. Find integers x, y with $240x + 46y = \gcd(240, 46)$, using back-substitution, and check your answer. *Try it:* Check 240x + 46y = 2 .
3. Compute $\operatorname{lcm}(84, 120)$ two ways.
4. Can a 6-litre and a 15-litre jug measure exactly 4 litres? 9 litres? Explain with Bézout. *Try it:* Try the 6- and 15-litre jugs .
5. Show that consecutive integers n and $n + 1$ are always coprime.

▼ **Answers**

1. $120 = 1 \cdot 84 + 36$, $84 = 2 \cdot 36 + 12$, $36 = 3 \cdot 12 + 0$, so the GCD is 12 in 3 steps. For $(89, 55)$, every quotient except the last is 1: the remainders run 34, 21, 13, 8, 5, 3, 2, 1, 0, so the GCD is 1 in 9 steps. This is Lamé's Fibonacci worst case.
2. $240 = 5 \cdot 46 + 10$, $46 = 4 \cdot 10 + 6$, $10 = 1 \cdot 6 + 4$, $6 = 1 \cdot 4 + 2$, $4 = 2 \cdot 2$, so the GCD is 2. Back-substitute:
   $2 = 6 - 4 = 6 - (10 - 6) = 2 \cdot 6 - 10 = 2(46 - 4 \cdot 10) - 10 = 2 \cdot 46 - 9 \cdot 10 = 2 \cdot 46 - 9(240 - 5 \cdot 46) = 47 \cdot 46 - 9 \cdot 240$. So $x = -9$, $y = 47$. Check: $47 \cdot 46 = 2162$ and $9 \cdot 240 = 2160$, difference 2.
3. $84 \cdot 120 / 12 = 840$. By factorization: $84 = 2^2 \cdot 3 \cdot 7$ and $120 = 2^3 \cdot 3 \cdot 5$. Taking maximum exponents gives $2^3 \cdot 3 \cdot 5 \cdot 7 = 840$.
4. $\gcd(6, 15) = 3$. Reachable amounts are multiples of 3, so 4 litres is impossible and 9 litres is possible. For example, $9 = 6 \cdot (-1) + 15 \cdot 1$: fill the 15 and pour off 6.
5. Any common divisor d divides $(n + 1) - n = 1$, so $d = 1$. In Bézout form, $(n + 1) \cdot 1 + n \cdot (-1) = 1$.

23.6. Modular arithmetic: clock arithmetic made precise

23.6.1. The problem

It is 9 o'clock. What time will it be in 5 hours? Nobody says "14 o'clock" on a 12-hour clock. You say 2. In 100 hours? You don't count 100 ticks. You throw away whole 12-hour laps. A 32-bit counter at $2^{32} - 1$ that gets incremented becomes 0. The day of the week, the angle of a rotating wheel, the slot in a ring buffer: these are all quantities where only the *remainder* matters.

Modular arithmetic is the algebra of "only the remainder matters". The surprise is how much ordinary algebra still works there, and the one rule that doesn't.

23.6.2. Intuition: wrapping the number line

Wrap the number line around a circle of circumference 12. Then 2, 14, 26 and -10 all land on the same spot. Adding means walking clockwise. Subtracting means walking anticlockwise. Multiplying is repeated addition. *Try it:* 9 + 5 on a clock .

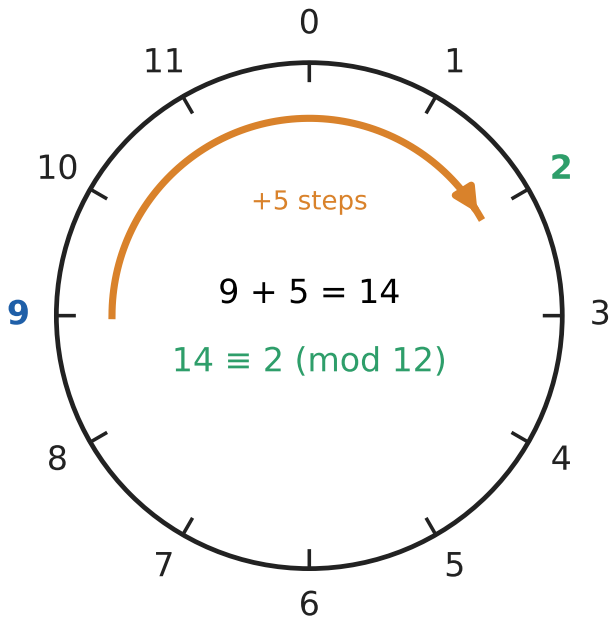


Figure 19. Arithmetic modulo 12: starting at 9 and walking 5 steps lands on 2, because 14 and 2 differ by a full lap of 12.

23.6.3. Vocabulary and notation

Congruence modulo m

Fix a positive integer m, the **modulus**. Integers a and b are **congruent modulo m**, written

$$a \equiv b \pmod{m},$$

if $m \mid (a - b)$. Equivalently, a and b leave the same remainder when divided by m. Read it as "a is congruent to b mod m". For example, $14 \equiv 2 \pmod{12}$, $-10 \equiv 2 \pmod{12}$, and $100 \equiv 4 \pmod{12}$.

The three-bar symbol $\equiv$ was Gauss's choice (1801). It looks like "=" on purpose.

mod as an operator vs. mod as a context

The same word does two jobs. $17 \bmod 5 = 2$ uses **mod as an operator**: a function that returns a number, like `%` in code. $17 \equiv 2 \pmod{5}$ uses **mod as a context**: a statement about two numbers, and the "(mod 5)" applies to the whole line. So $17 \equiv 7 \pmod{5}$ is true, but $17 \bmod 5 = 7$ is false.

Residue, residue class

The **residue** of a mod m is the remainder $a \bmod m$, one of $0, 1, \ldots, m-1$. The **residue class** $[a]_m$ is the set of all integers congruent to a. For example, $[2]_{12} = \{\ldots, -10, 2, 14, 26, \ldots\}$. There are exactly m classes, and the set of them is written $\mathbb{Z}_m$ or $\mathbb{Z}/m\mathbb{Z}$.

remainder 0 [0] (mod 3)	remainder 1 [1] (mod 3)	remainder 2 [2] (mod 3)
-6	-5	-4
-3	-2	-1
0	1	2
3	4	5
6	7	8
9	10	11

a ≡ b (mod 3) ⇔ a and b in the same column

Figure 20. The integers sorted into residue classes mod 3. Congruent numbers sit in the same column; arithmetic mod 3 is arithmetic on columns.

Congruence is an **equivalence relation** (reflexive, symmetric and transitive; see Chapter 21). That is the formal reason the "columns" picture is legitimate. Each integer belongs to exactly one column.

23.6.4. Why it works: congruences respect + − ×

Here is the key theorem. Suppose $a \equiv a'$ and $b \equiv b' \pmod{m}$. Then

$$a+b \equiv a'+b', \qquad a-b \equiv a'-b', \qquad ab \equiv a'b' \pmod{m}.$$

Why. Write $a' = a + km$ and $b' = b + lm$. Then

$$\begin{aligned} a'+b' &= (a+b) + (k+l)m && \text{(differs from } a+b \text{ by a multiple of } m\text{)} \\ a'b' &= (a+km)(b+lm) \\ &= ab + alm + bkm + klm^2 && \text{(expand)} \\ &= ab + m(al + bk + klm) && \text{(factor out } m\text{).} \end{aligned}$$

In practice this means **you may reduce at any point**. Replace any number by its remainder whenever you like, and the final remainder comes out the same. That is what makes huge computations tractable: intermediate values never need to grow past m^2.

Worked example. What is the last digit of $7 \cdot 13 \cdot 29 \cdot 48$? "Last digit" means "mod 10".

$$\begin{aligned} 7 \cdot 13 &= 91 \equiv 1 \pmod{10} \\ 1 \cdot 29 &= 29 \equiv 9 \\ 9 \cdot 48 &\equiv 9 \cdot 8 = 72 \equiv 2 \pmod{10}. \end{aligned}$$

The last digit is 2. You never needed the full product, which is 126,672.

23.6.5. Divisibility tests, explained

Since $10 \equiv 1 \pmod{9}$, every power $10^k \equiv 1^k = 1 \pmod{9}$. A number with digits $d_k \ldots d_1 d_0$ is $\sum d_i 10^i$. (Here $\sum$ means "add up the following terms over all values of i".) So

$$d_k 10^k + \cdots + d_1 10 + d_0 \equiv d_k + \cdots + d_1 + d_0 \pmod 9.$$

A number and its digit sum leave the same remainder mod 9, and therefore also mod 3. Similarly, $10 \equiv -1 \pmod{11}$, so $10^k \equiv (-1)^k$ and the *alternating* digit sum gives the remainder mod 11. Old-school "casting out nines" to check hand multiplication is the same idea.

23.6.6. The one rule that breaks: division

In ordinary arithmetic, $2x = 2y$ lets you cancel the 2. Mod 6, though, *Try it:* Zero divisors mod 6 .

$$2 \cdot 1 \equiv 2 \cdot 4 \pmod 6 \quad (2 \equiv 8), \qquad \text{but} \qquad 1 \not\equiv 4 \pmod 6.$$

You can't cancel 2, because 2 shares a factor with 6. Worse, $2 \cdot 3 = 6 \equiv 0 \pmod 6$: two nonzero numbers multiply to zero. These are called **zero divisors**. Division mod m works only for numbers coprime to m, and modular inverses (Section 23.8) make that precise.

23.6.7. Where you'd use this

Time and calendars. A flight leaves at 21:00 and the journey, including a stopover, takes 50 hours. Arrival is at $21 + 50 = 71 \equiv 23 \pmod{24}$, so 23:00, and $\lfloor 71/24 \rfloor = 2$ days later. Days of the week work mod 7: since $365 = 52 \cdot 7 + 1 \equiv 1 \pmod 7$, a date moves one weekday later each ordinary year (two after a leap day). Christmas 2026 is a Friday, so Christmas 2027 is a Saturday.

Navigation and physics. Compass bearings live mod 360: a ship on heading 350° that turns 30° to starboard is on heading $380 \bmod 360 = 20°$. Phase angles of waves and alternating currents live mod 2π (see Volume 1, Chapter 13), which is why a phase of 370° and one of 10° describe the same signal.

Music. Western pitch classes are residues mod 12: an octave is 12 semitones, a perfect fifth is 7. Stepping up by fifths gives $0, 7, 14 \equiv 2, 9, 16 \equiv 4, \ldots$; because $\gcd(7, 12) = 1$ the walk visits all 12 pitch classes before returning to the start. That walk is the circle of fifths. Stepping by 4 semitones (a major third) instead, with $\gcd(4, 12) = 4$, cycles through only 3 notes.

Computer-science connection: your CPU is a modular arithmetic machine

An unsigned 32-bit integer *is* a residue mod 2^{32}. Addition, subtraction and multiplication in C (for unsigned types), Java, Rust's `wrapping_*` operations and GPUs are all exactly arithmetic in $\mathbb{Z}_{2^{32}}$. The congruence rules above guarantee that the low 32 bits of a result depend only on the low 32 bits of the inputs. Two's complement signed integers are the same residues with a different choice of representatives: $-2^{31} \dots 2^{31} - 1$ instead of $0 \dots 2^{32} - 1$. That is why the CPU can use one adder circuit for both.

Practical consequences:

- `x % 2k == x & (2k - 1)` for nonnegative `x`, which is why hash tables love power-of-two sizes.
- A ring buffer's index is `(head + i) % capacity`.
- Overflowing multiplication in a hash function doesn't "go wrong". It just works mod 2^{64}.



Any modulus clock

A clock face with an adjustable modulus m. The user picks a and b and sees $a + b$ and $a \cdot b$ as walks around the dial. A second mode draws the "times k" map as chords: connect each x to $kx \bmod m$. With $m = 200, k = 2$ you get a cardioid, and other k give nephroids and stranger curves. It is a striking picture of multiplication as rotation.

Try it: Open the any-modulus clock

23.7. What `%` means for negative numbers

23.7.1. The problem

Ask Python and C the same question: *Try it:* % on negatives, language by language .

Expression	Python	C, C++, Java, JavaScript, Go, Rust (`%`)
`7 / 2` (integer) and `7 % 2`	`3`, `1`	`3`, `1`
`-7 / 2` and `-7 % 2`	`-4`, `1`	`-3`, `-1`
`7 / -2` and `7 % -2`	`-4`, `-1`	`-3`, `1`

(In Python, integer division is `//`.) Both camps are internally consistent. They disagree on which quotient to pick.

23.7.2. Why: two rounding rules

Every language preserves the identity

$$a = b \cdot q + r,$$

so choosing q determines r. For $-7/2 = -3.5$ there are two natural choices:

- **Truncate toward zero** (C99 and later, Java, JavaScript, Go, Rust): $q = -3$. Then $r = -7 - (2)(-3) = -1$. The remainder **takes the sign of the dividend**.
- **Floor, toward $-\infty$** (Python, Ruby, Haskell's `mod`, mathematics): $q = \lfloor -3.5 \rfloor = -4$. Then $r = -7 - (2)(-4) = 1$. The remainder **takes the sign of the divisor**, so for positive m it always lies in $0 \ldots m - 1$, matching the division algorithm.



Floor and truncation

The **floor** $\lfloor x \rfloor$ is the greatest integer $\leq x$: $\lfloor 3.5 \rfloor = 3$ and $\lfloor -3.5 \rfloor = -4$. **Truncation** drops the fractional part, rounding toward 0: $-3.5 \to -3$. They agree for nonnegative numbers and differ for negative non-integers.

PYTHON

```python
print(7 // 2, 7 % 2, -7 // 2, -7 % 2, 7 // -2, 7 % -2)
import math
print(math.fmod(-7, 2), int(-7 / 2))
```

```
3 1 -4 1 -4 -1
-1.0 -3
```

Python's `math.fmod` and `int(a / b)` give you the C-style behaviour when you need it. Be careful with `int(a / b)` for huge integers, though, since it goes through floating point.



The negative-index bug

In C, Java or JavaScript, `arr[(i - 1) % n]` with `i = 0` computes `-1 % n == -1`. That is an out-of-bounds index, or worse, a silent read from the end of the array in languages with negative indexing. The portable fix is

```
((i - 1) % n + n) % n
```

or `Math.floorMod(i - 1, n)` in Java, or `i.rem_euclid(n)` in Rust. The same bug breaks "is odd" tests: `x % 2 == 1` is **false** for `x = -3` in C and Java. Write `x % 2 != 0` instead.

Choosing a convention

Use **floor (Python-style)** whenever the remainder is used as an index, a bucket, a day of the week or a clock position. You want $0 \ldots m-1$ every time. **Truncation** survives mainly because early hardware divide instructions worked that way and C standardised it. Wherever correctness matters, write a `mod(a, m)` helper and use it everywhere rather than trusting `%`.

23.8. Modular inverses and linear congruences

23.8.1. The problem

Solve $7x \equiv 3 \pmod{26}$. In ordinary arithmetic you'd divide by 7. Mod 26 there are no fractions, so what does "divide by 7" even mean? We need a whole number that *acts like* $\frac{1}{7}$: something that, multiplied by 7, gives 1.



Modular inverse

An **inverse of a modulo m** is an integer a^{-1} with

$$a \cdot a^{-1} \equiv 1 \pmod{m}.$$

It exists **if and only if $\gcd(a, m) = 1$**, and it is unique mod m. For example, $3^{-1} \equiv 4 \pmod{11}$ because $3 \cdot 4 = 12 \equiv 1$. The notation a^{-1} here never means the fraction $1/a$.

23.8.2. Why gcd 1 is exactly the condition

- **If $\gcd(a, m) = 1$:** Bézout gives $ax + my = 1$. Read it mod m. The my term is a multiple of m, so $ax \equiv 1$. The Bézout coefficient x *is* the inverse.
- **If $\gcd(a, m) = g > 1$:** every $ax + my$ is a multiple of g, so it can never equal 1, and no inverse exists.

The two multiplication tables below show this. Mod 7 (a prime) every nonzero row contains exactly one 1. Mod 8, the rows for 2, 4 and 6, which share a factor with 8, never reach 1 and even hit 0.

× mod 7

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

every row contains a 1: every nonzero class is invertible

× mod 8

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

rows 2, 4, 6 have no 1 (and hit 0): no inverse

Figure 21. Multiplication tables mod 7 and mod 8. A 1 in row a marks its inverse. Mod a prime, every nonzero element has one.

Example 17. Procedure — Modular inverse

Input: integers a and $m > 1$.

Steps:

1. Run the extended Euclidean algorithm on (m, a) to get g and coefficients with $m \cdot s + a \cdot t = g$.
2. If $g \neq 1$, stop: a has no inverse mod m.
3. Otherwise return $t \bmod m$, reduced into $0 \dots m - 1$.

Output: $a^{-1} \bmod m$.

Try it: Explore tables mod 7 and 8 .

23.8.3. Worked example: $7^{-1} \pmod{26}$

Euclid on $(26, 7)$: *Try it:* 7^{-1} mod 26, step by step ; Compute 7^{-1} mod 26 .

$$\begin{aligned} 26 &= 3 \cdot 7 + 5 \\ 7 &= 1 \cdot 5 + 2 \\ 5 &= 2 \cdot 2 + 1 \\ 2 &= 2 \cdot 1 + 0 \qquad \Rightarrow \gcd = 1, \text{ an inverse exists.} \end{aligned}$$

Back-substitute, starting from the line that produced 1:

$$
\begin{aligned}
1 &= 5 - 2 \cdot 2 \\
&= 5 - 2 \cdot (7 - 5) && (2 = 7 - 5) \\
&= 3 \cdot 5 - 2 \cdot 7 \\
&= 3 \cdot (26 - 3 \cdot 7) - 2 \cdot 7 && (5 = 26 - 3 \cdot 7) \\
&= 3 \cdot 26 - 11 \cdot 7.
\end{aligned}
$$

Mod 26 the $3 \cdot 26$ term vanishes, leaving $-11 \cdot 7 \equiv 1$. So $7^{-1} \equiv -11 \equiv 15 \pmod{26}$. **Check:** $7 \cdot 15 = 105 = 4 \cdot 26 + 1$. Correct.

Now solve $7x \equiv 3 \pmod{26}$ by multiplying both sides by 15:

$$
x \equiv 15 \cdot 3 = 45 \equiv 19 \pmod{26}.
$$

Check: $7 \cdot 19 = 133 = 5 \cdot 26 + 3$. Correct.

```python
def mod_inverse(a, m):
    g, x, _ = extended_gcd(a, m)
    if g != 1:
        raise ValueError(f"{a} has no inverse mod {m} (gcd is {g})")
    return x % m

print(mod_inverse(3, 40), mod_inverse(17, 3120), mod_inverse(7, 26))
print(pow(17, -1, 3120))      # Python 3.8+ does it for you
```

```
27 2753 15
2753
```

(`mod_inverse` reuses `extended_gcd` from Section 23.5. Here x is the coefficient of a, which is the one we want.)

23.8.4. When $\gcd > 1$: linear congruences in general

$ax \equiv b \pmod{m}$ has solutions exactly when $g = \gcd(a, m)$ divides b. In that case, divide everything by g and solve the smaller congruence, which gives g solutions mod m. **Example:** $4x \equiv 6 \pmod{10}$. Here $g = 2$ divides 6, so dividing by 2 gives $2x \equiv 3 \pmod{5}$. Since $2^{-1} \equiv 3 \pmod{5}$, we get $x \equiv 9 \equiv 4 \pmod{5}$. Mod 10 that means $x \in \{4, 9\}$. Check: $16 \equiv 6$ and $36 \equiv 6 \pmod{10}$. In contrast, $4x \equiv 5 \pmod{10}$ has no solutions, since $4x$ is always even and so is 10: the difference would be odd. *Try it:* Solve 4x ≡ 6 (mod 10) .



> *Primes make perfect number systems*
>
> When p is prime, every nonzero element of $\mathbb{Z}_p$ has an inverse, so you can add, subtract, multiply and divide, just as with the rationals. Mathematicians call that a **field**, and $\mathbb{Z}_p$ is the finite field $\mathbb{F}_p$. Reed–Solomon codes (QR codes, CDs, deep-space probes), elliptic-curve cryptography and AES all do their arithmetic in finite fields.

Computer-science connection: the Chinese remainder theorem

If m_1, m_2 are coprime, a number mod $m_1 m_2$ is completely determined by its two residues mod m_1 and mod m_2. Inverses let you reconstruct it. This **Chinese remainder theorem** (Sunzi, 3rd–5th century CE) is how RSA implementations speed up decryption about fourfold: they work mod p and mod q separately. It is also how "residue number systems" do wide-integer arithmetic in parallel channels.

Try it: Solve a CRT system .

23.9. Fermat's little theorem

23.9.1. The problem

Compute the powers of 3 mod 7: $3, 2, 6, 4, 5, 1, 3, 2, 6, \ldots$ The sequence repeats with period 6. Powers of 2 mod 7 go $2, 4, 1, 2, 4, 1, \ldots$ with period 3. Both periods divide 6, which is one less than the prime 7. Coincidence?



Fermat's little theorem

If p is prime and $p \nmid a$, then

$$a^{p-1} \equiv 1 \pmod{p}.$$

Equivalently, $a^p \equiv a \pmod{p}$ for every integer a. ("Little" distinguishes it from Fermat's Last Theorem.)

23.9.2. Why it works: shuffling the nonzero residues

Take $p = 7$, $a = 3$, and multiply each of $1, 2, 3, 4, 5, 6$ by 3, mod 7:

$$3,\ 6,\ 9 \equiv 2,\ 12 \equiv 5,\ 15 \equiv 1,\ 18 \equiv 4.$$

You get $3, 6, 2, 5, 1, 4$, which is the same six numbers in a different order. That always happens. If $ai \equiv aj$, multiplying by a^{-1} gives $i \equiv j$, so no two products collide, and none is 0. Now multiply all six products together. They're the same numbers, so they have the same product:

$$\begin{aligned} (3\cdot 1)(3\cdot 2)(3\cdot 3)(3\cdot 4)(3\cdot 5)(3\cdot 6) &\equiv 1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6 \pmod{7} \\ 3^6 \cdot 6! &\equiv 6! \pmod{7} \\ 3^6 &\equiv 1 \pmod{7} \qquad \text{(cancel 6!, which is coprime to 7)} \end{aligned}$$

Replace 7 by p and 3 by a and the argument is the general proof.

23.9.3. Using it: shrinking exponents

Fermat lets you reduce an exponent mod $p - 1$.

Worked example: $3^{100} \bmod 7$. Since $3^6 \equiv 1$ and $100 = 16 \cdot 6 + 4$: *Try it:* Compute 3^{100} mod 7 .

$$3^{100} = (3^6)^{16} \cdot 3^4 \equiv 1^{16} \cdot 81 \equiv 81 - 77 = 4 \pmod 7.$$

Worked example: $7^{222} \bmod 11$. $7^{10} \equiv 1$ and $222 = 22 \cdot 10 + 2$, so $7^{222} \equiv 7^2 = 49 \equiv 5 \pmod{11}$.

23.9.4. Euler's generalisation

For a composite modulus, replace $p - 1$ with **Euler's totient** $\varphi(n)$, the count of numbers in $1 \ldots n$ that are coprime to n. **Euler's theorem**: if $\gcd(a, n) = 1$, then $a^{\varphi(n)} \equiv 1 \pmod n$. For a prime, $\varphi(p) = p - 1$. For a product of two distinct primes, $\varphi(pq) = (p-1)(q-1)$: of the pq numbers, the q multiples of p and the p multiples of q are excluded, and pq itself was counted twice, so $pq - p - q + 1 = (p-1)(q-1)$. RSA is built on this formula.

23.9.5. Primality testing

Turn the theorem around. If $a^{n-1} \not\equiv 1 \pmod n$ for some a, then n is *definitely composite*, and we learned that without finding a factor. For example, $2^{14} = 16384 = 1092 \cdot 15 + 4$, so $2^{14} \equiv 4 \not\equiv 1 \pmod{15}$ and 15 is composite.

```python
def fermat_probably_prime(n, bases=(2, 3, 5, 7)):
    return all(pow(a, n - 1, n) == 1 for a in bases if a % n != 0)

print([k for k in range(2, 40) if fermat_probably_prime(k)])
print(pow(2, 560, 561), 561 == 3 * 11 * 17)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
1 True
```



Passing the Fermat test does not prove primality

The theorem only goes one way: prime implies $a^{p-1} \equiv 1$. The converse is false. $561 = 3 \cdot 11 \cdot 17$ satisfies $a^{560} \equiv 1 \pmod{561}$ for *every* a coprime to 561. Numbers like this are **Carmichael numbers**, and there are infinitely many of them. Real libraries use the **Miller–Rabin** test, a strengthened Fermat test with no such blind spots. Each random base catches a composite with probability at least $3/4$, so 40 rounds leave an error chance below 2^{-80}. (Confusing a theorem with its converse is a classic logic error; see Chapter 21.)

23.10. Fast modular exponentiation

23.10.1. The problem

RSA needs things like $m^{65537} \bmod n$ with n around 600 digits long. Computing m^{65537} first is absurd: the number would have about 40 million digits before you reduced it. Multiplying by m 65,536 times, reducing as you go, keeps numbers small but takes too many steps once exponents are themselves 600 digits long. We need something exponentially better.

23.10.2. Intuition: square, don't multiply

Squaring doubles the exponent: $a \to a^2 \to a^4 \to a^8 \to \cdots$. After 10 squarings you have a^{1024}. Any exponent is a sum of powers of 2, its binary expansion, so multiply together the squarings you need:

$$13 = 8 + 4 + 1 = 1101_2 \quad \Rightarrow \quad a^{13} = a^8 \cdot a^4 \cdot a^1.$$

An n-bit exponent needs at most n squarings and n multiplications. For a 2048-bit exponent that is about 4,000 operations instead of 2^{2048}. Reducing mod m after each step (allowed by Section 23.6) keeps every number below m^2.

Example 18. Procedure — Square-and-multiply (right-to-left binary exponentiation)

Input: base a, exponent $e \geq 0$, modulus m.

Steps:

1. Set $\mathbf{result} = 1$ and $\mathbf{base} = a \bmod m$.
2. While $e > 0$:
   a. If the lowest bit of e is 1 (e odd): $\mathbf{result} \leftarrow \mathbf{result} \cdot \mathbf{base} \bmod m$.
   b. $\mathbf{base} \leftarrow \mathbf{base}^2 \bmod m$.
   c. $e \leftarrow \lfloor e/2 \rfloor$ (shift right one bit).

Output: $a^e \bmod m$.

23.10.3. Worked example: $7^{13} \bmod 11$

First tabulate the repeated squares of 7 mod 11: *Try it:* 7^{13} mod 11, bit by bit ; Compute 7^{13} mod 11 .

$$\begin{aligned} 7^1 &\equiv 7 \\ 7^2 &= 49 = 4 \cdot 11 + 5 \equiv 5 \\ 7^4 &\equiv 5^2 = 25 = 2 \cdot 11 + 3 \equiv 3 \\ 7^8 &\equiv 3^2 = 9. \end{aligned}$$

Since $13 = 8 + 4 + 1$, multiply the matching rows and reduce at each step:

$$\begin{aligned} 7^{13} &= 7^8 \cdot 7^4 \cdot 7^1 \\ &\equiv 9 \cdot 3 \cdot 7 \\ &= 27 \cdot 7 \equiv 5 \cdot 7 && (27 = 2 \cdot 11 + 5) \\ &= 35 \equiv 2 \pmod{11} && (35 = 3 \cdot 11 + 2). \end{aligned}$$

The procedure does the same thing bit by bit ($13 = 1101_2$, read from the right):

e	lowest bit	base before	result after	base after squaring
13	1	7	$1 \cdot 7 = 7$	$49 \equiv 5$
6	0	5	7 (unchanged)	$25 \equiv 3$
3	1	3	$7 \cdot 3 = 21 \equiv 10$	9
1	1	9	$10 \cdot 9 = 90 \equiv 2$	$81 \equiv 4$
0	—		2	

Cross-check with Fermat: $7^{10} \equiv 1$, so $7^{13} \equiv 7^3 = 343 = 31 \cdot 11 + 2 \equiv 2$. Correct.

```python
def power_mod(base, exp, m):
    """Compute base**exp % m by repeated squaring."""
    result = 1
    base %= m
    while exp > 0:
        if exp & 1:                  # lowest bit of exp is 1
            result = result * base % m
        base = base * base % m       # square for the next bit
        exp >>= 1                    # shift to the next bit
    return result

print(power_mod(7, 13, 11), power_mod(13, 27, 55), power_mod(2790, 2753, 3233))
print(pow(7, 13, 11))   # the built-in three-argument pow agrees
```

```
2 7 65
2
```

23.10.4. Choosing a method

- **Small exponent, prime modulus, base not a multiple of it:** reduce the exponent with Fermat first (mod $p - 1$), then finish by hand.
- **Anything in code:** the built-in `pow(a, e, m)` (Python), `BigInteger.modPow` (Java), or a constant-time library routine for cryptography.
- **Never** compute `a ** e % m` with a large `e`. It builds the gigantic integer first.



Timing leaks

The textbook procedure does extra work exactly when a bit of the exponent is 1. If the exponent is a *secret key*, an attacker who can time or power-measure your device can read the bits off one by one. Paul Kocher demonstrated this in 1996. Cryptographic libraries use constant-time variants such as the "Montgomery ladder", which do the same operations whatever the bits are. Use your own `power_mod` for learning, never for real keys.

Computer-science connection: the same trick everywhere

Square-and-multiply works for any associative operation, not just modular multiplication. It computes Fibonacci numbers in $O(\log n)$ steps by raising a 2×2 matrix to the nth power (Volume 2, Chapter 27). It counts paths of length k in a graph using adjacency-matrix powers (Chapter 25), and it composes a permutation with itself k times. Recognising "this is exponentiation in disguise" is a classic algorithm-interview move.



Exponentiation race

Let the user enter a, e and m. Show the exponent in binary, then animate the square-and-multiply table, lighting each bit as it is consumed. Beside it, run naive repeated multiplication and keep a step counter. Push e to 10^6 and then 10^{18} to feel the difference between linear and logarithmic.

Try it: Run the exponentiation race · Modular powers, any size

23.10.5. Exercises

1. Compute $2^{100} \bmod 7$ using Fermat's little theorem.
2. What is the last digit of 3^{2026}? (Find the cycle of powers of 3 mod 10.)
3. Compute $5^{23} \bmod 13$ with square-and-multiply, showing the table.
4. Find $11^{-1} \pmod{26}$, and use it to solve $11x \equiv 4 \pmod{26}$.
5. Solve $4x \equiv 6 \pmod{10}$ and explain why $6x \equiv 4 \pmod{9}$ has no solution.
6. In C, what does `-17 % 5` give, and in Python? Write an expression that gives Python's answer in C. *Try it:* –17 % 5 in each language .
7. Show that the square of any integer is congruent to 0 or 1 mod 4. Deduce that no integer of the form $4k + 3$ is a sum of two squares.

▼ **Answers**

1. $2^3 = 8 \equiv 1 \pmod{7}$, which is even better than Fermat's 2^6. $100 = 33 \cdot 3 + 1$, so $2^{100} \equiv 2^1 = 2$. Using Fermat ($2^6 \equiv 1$, $100 = 16 \cdot 6 + 4$): $2^4 = 16 \equiv 2$. Same answer.

2. Powers of 3 mod 10 cycle $3, 9, 7, 1$ with period 4. $2026 = 506 \cdot 4 + 2$, so the last digit is the second in the cycle, 9.

3. $23 = 10111_2 = 16 + 4 + 2 + 1$. Squares mod 13: $5^1 = 5$, $5^2 = 25 \equiv 12$, $5^4 \equiv 144 \equiv 1$, $5^8 \equiv 1$, $5^{16} \equiv 1$. Product: $1 \cdot 1 \cdot 12 \cdot 5 = 60 \equiv 8$. So $5^{23} \equiv 8 \pmod{13}$. (Check with Fermat: $5^{12} \equiv 1$, so $5^{23} \equiv 5^{11} = 5^8 \cdot 5^2 \cdot 5 \equiv 1 \cdot 12 \cdot 5 \equiv 8$.)

4. $26 = 2 \cdot 11 + 4$, $11 = 2 \cdot 4 + 3$, $4 = 1 \cdot 3 + 1$. Back-substitute: $1 = 4 - 3 = 4 - (11 - 2 \cdot 4) = 3 \cdot 4 - 11 = 3(26 - 2 \cdot 11) - 11 = 3 \cdot 26 - 7 \cdot 11$. So $11^{-1} \equiv -7 \equiv 19$. Check: $11 \cdot 19 = 209 = 8 \cdot 26 + 1$. Then $x \equiv 19 \cdot 4 = 76 \equiv 76 - 52 = 24$. Check: $11 \cdot 24 = 264 = 10 \cdot 26 + 4$.

5. As in the text, $x \equiv 4 \pmod{5}$, i.e. $x \in \{4, 9\} \pmod{10}$. For $6x \equiv 4 \pmod{9}$, $\gcd(6, 9) = 3$ does not divide 4. Every $6x - 9y$ is a multiple of 3, so it can never equal 4.

6. In C, `-17 / 5 == -3` and `-17 % 5 == -2`. In Python, `-17 // 5 == -4` and `-17 % 5 == 3`. Portable C: `((-17 % 5) + 5) % 5`, which gives 3.

7. Every integer is $\equiv 0, 1, 2, 3 \pmod{4}$, and the squares are $0, 1, 4 \equiv 0, 9 \equiv 1$. So a sum of two squares is $\equiv 0 + 0, 0 + 1$ or $1 + 1$, i.e. 0, 1 or 2 mod 4, and never 3.

23.11. Applications: hashing, check digits, and pseudo-random numbers

23.11.1. Hashing: mod as a bucket chooser

A hash table with m buckets stores key k in bucket $h(k) \bmod m$, where h scrambles the key into a big integer. The remainder lands in $0 \ldots m - 1$ (if you use floor-mod!) and spreads keys around, *provided the keys don't share structure with* m.

PYTHON
```
keys = [0, 8, 16, 24, 32, 40, 48, 56]      # e.g. aligned addresses
for m in (8, 7):
    print(m, sorted(k % m for k in keys))
```

```
8 [0, 0, 0, 0, 0, 0, 0, 0]
7 [0, 0, 1, 2, 3, 4, 5, 6]
```

Keys that are all multiples of 8 collide completely in 8 buckets, but spread out across 7. In general, the keys $k, k + s, k + 2s, \ldots$ hit only $m / \gcd(s, m)$ distinct buckets. A prime m shares no factor with any stride smaller than itself. That is the classical case for prime table sizes. Modern tables (Python's `dict`, Java's `HashMap`, Rust's `HashMap`) use power-of-two sizes for speed (`& (m - 1)` instead of division) and compensate by mixing the hash bits thoroughly first. Java's `HashMap` XORs the high half of the hash into the low half before masking.

Hash functions themselves are modular arithmetic. Java's `String.hashCode` is

$$h = s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \cdots + s_{n-1} \pmod{2^{32}},$$

computed by Horner's rule ($h \leftarrow 31h + s_i$), with the mod coming free from 32-bit overflow. For `"hi"` it gives $104 \cdot 31 + 105 = 3329$. The **Rabin–Karp** substring search uses the same polynomial hash mod a large prime. Sliding the window by one character updates the hash in constant time: subtract the old leading character's term, multiply by the base, add the new character.

23.11.2. Check digits: catching typos with a remainder

A check digit is one extra digit chosen so that some modular sum comes out to 0. Typing errors break the sum.

ISBN-10 uses weights 10 down to 1 and requires

$$10d_1 + 9d_2 + 8d_3 + \cdots + 2d_9 + 1 \cdot d_{10} \equiv 0 \pmod{11},$$

where a check value of 10 is written X. **Worked example: 0-306-40615-2.** *Try it:* Check the ISBN-10 example .

$$\begin{aligned} &10 \cdot 0 + 9 \cdot 3 + 8 \cdot 0 + 7 \cdot 6 + 6 \cdot 4 + 5 \cdot 0 + 4 \cdot 6 + 3 \cdot 1 + 2 \cdot 5 + 1 \cdot 2 \\ &= 0 + 27 + 0 + 42 + 24 + 0 + 24 + 3 + 10 + 2 = 132 = 12 \cdot 11 \equiv 0 \pmod{11}. \quad \checkmark \end{aligned}$$

Why 11 and not 10? Because 11 is prime. A single wrong digit changes the sum by $w \cdot (\text{error})$, where both factors are nonzero and less than 11, so by Euclid's lemma 11 can't divide the change. Swapping adjacent digits x, y changes the sum by $\pm(x - y)$, again not a multiple of 11 unless $x = y$. So **every** single-digit error and **every** adjacent swap is caught. (ISBN-13 switched to weights 1, 3, 1, 3, … mod 10, which misses a few swaps, for compatibility with retail barcodes.)

The **Luhn algorithm** protects credit-card and IMEI numbers. From the right, double every second digit, and if a doubled value exceeds 9, subtract 9 (which is the same as adding its two digits). The total must be $\equiv 0 \pmod{10}$.

Example 19. Procedure — Luhn check

> **Input:** a string of digits.
>
> **Steps:**
>
> 1. Starting from the rightmost digit (position 0), go leftwards.
> 2. Keep digits in even positions as they are. Double digits in odd positions, and subtract 9 from any result above 9.
> 3. Add everything up.
>
> **Output:** valid if the total is a multiple of 10.

Worked example: 7992 7398 713. Reading from the right, the digits are 3, 1, 7, 8, 9, 3, 7, 2, 9, 9, 7. The odd positions (1, 8, 3, 2, 9) are doubled to 2, 16, 6, 4, 18, which become 2, 7, 6, 4, 9 after subtracting 9 where needed. Even positions (3, 7, 9, 7, 9, 7) stay as they are. *Try it:* Check the Luhn example .

$$(3+7+9+7+9+7)+(2+7+6+4+9) = 42+28 = 70 \equiv 0 \pmod{10}. \quad \checkmark$$

```python
def luhn_valid(number: str) -> bool:
    digits = [int(c) for c in number if c.isdigit()]
    total = 0
    for i, d in enumerate(reversed(digits)):
        if i % 2 == 1:          # every second digit from the right
            d *= 2
            if d > 9:
                d -= 9          # same as adding the two digits of d
        total += d
    return total % 10 == 0

def isbn10_valid(isbn: str) -> bool:
    chars = [c for c in isbn if c.isdigit() or c in "Xx"]
    values = [10 if c in "Xx" else int(c) for c in chars]
    return sum((10 - i) * v for i, v in enumerate(values)) % 11 == 0

print(luhn_valid("7992 7398 713"), luhn_valid("7992 7398 731"))
print(isbn10_valid("0-306-40615-2"), isbn10_valid("0-306-46015-2"))
```

```
True False
True False
```



Check digits are not security

Check digits detect *accidents*, not *attacks*. Anyone can compute a valid Luhn digit for a made-up number. Luhn also misses the adjacent swap $09 \leftrightarrow 90$, since both pairs contribute 9. For tamper resistance you need a cryptographic MAC or signature. For bulk data (network packets, files) you need stronger codes such as CRCs, which are polynomial remainders mod 2, or checksums like Adler-32, which is two running sums mod 65521, a prime.

23.11.3. Pseudo-random numbers: the linear congruential generator

A computer is deterministic, yet simulations, games and randomised algorithms need numbers that *look* random. The oldest workable trick is the **linear congruential generator (LCG)**:

$$x_{n+1} = (a\,x_n + c) \bmod m.$$

Pick a **seed** x_0 and iterate. There are only m possible states, so the sequence must eventually cycle. A well-chosen LCG cycles through *all* m values before repeating.

Worked example: $a = 5$, $c = 3$, $m = 16$, seed $x_0 = 7$. *Try it:* Trace the LCG's 16 states .

$$\begin{aligned} x_1 &= (5\cdot 7 + 3) \bmod 16 = 38 \bmod 16 = 6 \\ x_2 &= (5\cdot 6 + 3) \bmod 16 = 33 \bmod 16 = 1 \\ x_3 &= (5\cdot 1 + 3) \bmod 16 = 8 \\ x_4 &= (5\cdot 8 + 3) \bmod 16 = 43 \bmod 16 = 11 \\ &\vdots \end{aligned}$$

```python
def lcg(seed, a=5, c=3, m=16):
    x = seed
    while True:
        x = (a * x + c) % m
        yield x

gen = lcg(7)
print([next(gen) for _ in range(17)])
```

```
[6, 1, 8, 11, 10, 5, 12, 15, 14, 9, 0, 3, 2, 13, 4, 7, 6]
```

All 16 values appear before 7 recurs and the cycle restarts at 6. This is no accident. The **Hull–Dobell theorem** says an LCG has full period m exactly when $\gcd(c, m) = 1$, $a - 1$ is divisible by every prime factor of m, and $a - 1$ is divisible by 4 if m is. Here $c = 3$ is coprime to 16 and $a - 1 = 4$ is divisible by both 2 and 4.

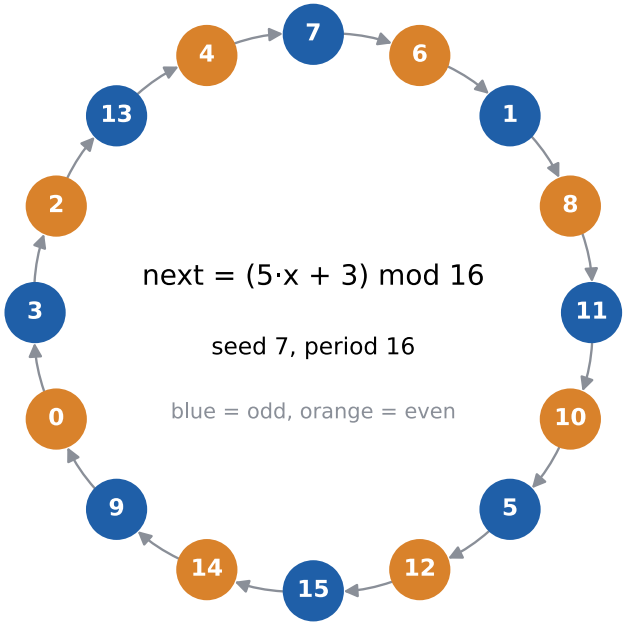


Figure 22. The LCG $x \mapsto (5x + 3) \bmod 16$ visits all 16 states in one cycle. Notice the colours: odd and even simply alternate.



LCG low bits are not random

Look at the diagram: the outputs alternate odd, even, odd, even. With $m = 2^k$, the lowest bit of an LCG has period 2, the lowest two bits have period 4, and so on. Write `rand() % 2` for a coin flip on such a generator and you get perfectly alternating "random" coins. Old C library `rand()` implementations did exactly this. Use the high bits, or better, a modern generator. Python's `random` uses the Mersenne Twister. NumPy defaults to PCG64, which is an LCG whose output is scrambled before use. **Never** use any of these for passwords, tokens or keys. Use `secrets` (Python) or the OS's cryptographic generator. An attacker who sees a few LCG outputs can recover the parameters and predict everything that follows.



RANDU

IBM's RANDU, $x_{n+1} = 65539\, x_n \bmod 2^{31}$, was used throughout the 1960s and 70s. Since $65539 = 2^{16} + 3$, every output satisfies $x_{n+2} = 6x_{n+1} - 9x_n \pmod{2^{31}}$. Plot consecutive triples in 3D and they all fall on just 15 parallel planes. Donald Knuth called it "really horrible". Many published simulation results from that era had to be re-examined.

Period explorer

Sliders for a, c, m and the seed. Draw the state graph as in the figure, so that cycles and tails become visible, and report the period. Let the reader test the Hull–Dobell conditions by breaking them one at a time (for example, set $c = 4$ with $m = 16$). A second panel plots consecutive pairs (x_n, x_{n+1}) so the lattice structure of LCGs becomes visible.

Try it: Open the period explorer

23.12. Cryptography: Diffie–Hellman and RSA

23.12.1. The problem

You want to send your card number to a shop you've never contacted, over a network where anyone can read every byte. Classical ciphers need a *shared secret key*. How do you agree on one when every message is public? Before 1976 almost everyone assumed this was impossible. Number theory says otherwise.

23.12.2. Diffie–Hellman: agreeing on a secret in public

The idea needs only modular powers and one observation: $(g^a)^b = g^{ab} = (g^b)^a$.

1. Alice and Bob publicly agree on a prime p and a base g.
2. Alice picks a secret a and sends $A = g^a \bmod p$. Bob picks a secret b and sends $B = g^b \bmod p$.
3. Alice computes $B^a \bmod p$ and Bob computes $A^b \bmod p$. Both equal $g^{ab} \bmod p$, which is their shared secret.

Worked example with $p = 23$, $g = 5$, Alice's $a = 6$, Bob's $b = 15$. The repeated squares of 5 mod 23 are *Try it:* Diffie–Hellman with an eavesdropper ; Compute 5^6 mod 23 .

$$5^1 \equiv 5, \quad 5^2 = 25 \equiv 2, \quad 5^4 \equiv 2^2 = 4, \quad 5^8 \equiv 4^2 = 16.$$

$$A = 5^6 = 5^4 \cdot 5^2 \equiv 4 \cdot 2 = 8 \pmod{23}$$
$$B = 5^{15} = 5^8 \cdot 5^4 \cdot 5^2 \cdot 5^1 \equiv 16 \cdot 4 \cdot 2 \cdot 5 = 640 = 27 \cdot 23 + 19 \equiv 19$$

Alice: $B^a = 19^6 \equiv (-4)^6 = 4096 = 178 \cdot 23 + 2 \equiv 2$

Bob: $A^b = 8^{15} = 2^{45} = (2^{11})^4 \cdot 2 \equiv 1^4 \cdot 2 = 2$ $\qquad (2^{11} = 2048 = 89 \cdot 23 + 1)$

Both arrive at the shared secret 2. An eavesdropper sees $p = 23$, $g = 5$, $A = 8$ and $B = 19$. To get the secret they must recover a from $5^a \equiv 8$. That is the **discrete logarithm problem**. It's trivial for $p = 23$ (try all 22 exponents) but believed infeasible when p has 600+ digits, even though computing g^a is fast. Modern TLS usually does the same dance on an elliptic curve instead of with integers mod p.

```python
p, g = 23, 5                # public: prime modulus and generator
a, b = 6, 15                # private: Alice's and Bob's secrets
A = pow(g, a, p)            # Alice sends A
B = pow(g, b, p)            # Bob sends B
print(A, B, pow(B, a, p), pow(A, b, p))
```

```
8 19 2 2
```



Paint-mixing intuition

The usual analogy: mixing paint is easy, but un-mixing it is practically impossible. Both parties start with a public colour and add their own secret colour. They swap the mixtures in public, and each adds their secret colour again. They end up with the same three-colour mix, which nobody watching can reproduce. Diffie and Hellman published in 1976. It later emerged that GCHQ's Malcolm Williamson had found the same scheme around 1974, and that Clifford Cocks had found RSA in 1973, but both were classified.

23.12.3. RSA: a lock anyone can close but only you can open

Diffie–Hellman agrees on a key. **RSA** (Rivest, Shamir, Adleman, 1977) goes further: a **public key** that anyone can use to encrypt a message to you, and a **private key** that only you can use to decrypt. It rests on everything in this chapter:

- **Primes and factoring** (Section 23.2): the public modulus is $n = pq$, and nobody can factor it.
- **Euler's theorem** (Section 23.9): raising to the power $1 + k\varphi(n)$ gets you back where you started.
- **Modular inverses via extended Euclid** (Section 23.8): to build the private key.
- **Fast exponentiation** (Section 23.10): to encrypt and decrypt efficiently.

Example 20. Procedure — RSA

Key generation (done once by the receiver):

1. Choose two distinct primes p and q. Compute $n = pq$ and $\varphi = (p-1)(q-1)$.
2. Choose a public exponent e with $1 < e < \varphi$ and $\gcd(e, \varphi) = 1$.
3. Compute the private exponent $d = e^{-1} \bmod \varphi$ by extended Euclid.
4. Publish (n, e). Keep d secret, and destroy p, q and φ.

Encryption (anyone): the message is a number $0 \le M < n$. Send $C = M^e \bmod n$.

Decryption (receiver): compute $M = C^d \bmod n$.

23.12.4. Worked example: RSA with tiny numbers, every step shown

Step 1: primes. $p = 5$ and $q = 11$. Then *Try it:* Tiny RSA, every step ; Find d = 3^{-1} mod 40 .

$$n = 5 \cdot 11 = 55, \qquad \varphi = (5-1)(11-1) = 4 \cdot 10 = 40.$$

Step 2: public exponent. Try $e = 3$. Since $40 = 13 \cdot 3 + 1$, we have $\gcd(40, 3) = \gcd(3, 1) = 1$, so $e = 3$ is allowed.

Step 3: private exponent. We need $3d \equiv 1 \pmod{40}$. The Euclid line $40 = 13 \cdot 3 + 1$ rearranges to

$$1 = 40 - 13 \cdot 3 \quad \Rightarrow \quad 3 \cdot (-13) \equiv 1 \pmod{40} \quad \Rightarrow \quad d \equiv -13 \equiv 27 \pmod{40}.$$

Check: $3 \cdot 27 = 81 = 2 \cdot 40 + 1$. Correct. The public key is $(n, e) = (55, 3)$ and the private key is $d = 27$.

Step 4: encrypt the message $M = 7$:

$$C = 7^3 \bmod 55 = 343 \bmod 55 = 343 - 6 \cdot 55 = 343 - 330 = 13.$$

Step 5: decrypt $C = 13$ by computing $13^{27} \bmod 55$ with square-and-multiply. In binary $27 = 11011_2 = 16 + 8 + 2 + 1$. The squares are *Try it:* Decrypt 13^{27} mod 55 .

$$\begin{aligned} 13^1 &\equiv 13 \\ 13^2 &= 169 = 3 \cdot 55 + 4 \equiv 4 \\ 13^4 &\equiv 4^2 = 16 \\ 13^8 &\equiv 16^2 = 256 = 4 \cdot 55 + 36 \equiv 36 \\ 13^{16} &\equiv 36^2 = 1296 = 23 \cdot 55 + 31 \equiv 31 \end{aligned}$$

and multiplying the ones we need gives

$$\begin{aligned} 13^{27} &= 13^{16} \cdot 13^8 \cdot 13^2 \cdot 13^1 \\ &\equiv 31 \cdot 36 \cdot 4 \cdot 13 \\ &= 1116 \cdot 4 \cdot 13 \equiv 16 \cdot 4 \cdot 13 && (1116 = 20 \cdot 55 + 16) \\ &= 64 \cdot 13 \equiv 9 \cdot 13 && (64 = 55 + 9) \\ &= 117 \equiv 7 \pmod{55} && (117 = 2 \cdot 55 + 7). \end{aligned}$$

The receiver recovers $M = 7$.

23.12.5. Why RSA works

Because $ed \equiv 1 \pmod{\varphi}$, we have $ed = 1 + k\varphi$ for some integer k. In the example, $3 \cdot 27 = 81 = 1 + 2 \cdot 40$. If $\gcd(M, n) = 1$, Euler's theorem gives

$$\begin{aligned} C^d &\equiv (M^e)^d = M^{ed} && (\text{definition of } C) \\ &= M^{1+k\varphi} = M \cdot (M^\varphi)^k && (\text{exponent laws}) \\ &\equiv M \cdot 1^k = M \pmod{n} && (\text{Euler: } M^{\varphi(n)} \equiv 1). \end{aligned}$$

(If M happens to share a factor with n, the result still holds. Applying Fermat separately mod p and mod q and combining with the Chinese remainder theorem shows it.)

Why it's secure. An attacker knows n and e. To find d they need $\varphi = (p-1)(q-1)$, and getting that from n is as hard as factoring n. With $n = 55$, factoring takes a second. Real keys use primes of 1,024+ bits each, giving a 617-digit n, and the best known algorithms would take far longer than the age of the universe.

The same arithmetic at the size of a classic textbook example: *Try it:* RSA with 61 and 53 .

PYTHON

```python
p, q = 61, 53
n = p * q                       # public modulus
phi = (p - 1) * (q - 1)         # Euler's totient of n
e = 17                          # public exponent, gcd(e, phi) == 1
d = pow(e, -1, phi)             # private exponent: e*d = 1 (mod phi)
print(f"n={n} phi={phi} e={e} d={d} check={(e * d) % phi}")

m = 65                          # the message, 0 <= m < n
c = pow(m, e, n)                # encrypt with the public key (n, e)
back = pow(c, d, n)             # decrypt with the private key d
print(f"m={m} c={c} decrypted={back}")
```

```
n=3233 phi=3120 e=17 d=2753 check=1
m=65 c=2790 decrypted=65
```



Textbook RSA is not real RSA

What we just did, "textbook RSA", is **insecure** as written. It is deterministic: the same message always gives the same ciphertext, so an attacker can encrypt guesses and compare. Small messages with small e can be recovered by an ordinary integer root: if $M^3 < n$, then $C = M^3$ exactly. Ciphertexts can also be multiplied to forge related ones. Real systems pad the message with randomness (OAEP), use $e = 65537$, generate primes with vetted libraries, and today mostly use RSA only for signatures or not at all, preferring elliptic curves. Learn the mathematics here, and use a library such as `cryptography` or libsodium for anything real.

Computer-science connection: what a quantum computer would break

Shor's algorithm (1994) would let a large, error-corrected quantum computer factor n and compute discrete logarithms efficiently. That would break both RSA and Diffie–Hellman (including the elliptic-curve versions). In 2024 NIST standardised **post-quantum** replacements (ML-KEM, from Kyber, and ML-DSA, from Dilithium). They are built on lattice problems: linear algebra mod q with deliberately added noise. The modulus never goes away.

RSA playground

Let the reader pick small primes p, q and an e. Show $\gcd(e, \varphi)$ by Euclid, flag an invalid e, and derive d with the extended-Euclid table on screen. Encrypt a short word letter by letter (and show why that is a terrible idea: identical letters give identical ciphertexts). Finally, an "attacker" pane factors n by trial division and shows the time growing as p and q get bigger.

Try it: Open the RSA playground · Inverses and powers mod n

US curriculum

Modular arithmetic is **not** a standard topic in the US high-school sequence (Algebra I, Geometry, Algebra II, Precalculus). Students meet it through the `%` operator in **AP Computer Science A** / **Principles**, through math competitions (MATHCOUNTS, AMC 10/12, where it is essential), and in some **Discrete Mathematics** or cryptography electives. The rigorous treatment (congruences, Fermat, RSA) is standard in a first-year college **Discrete Mathematics** course and in **Elementary Number Theory**.

23.13. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Number theory step by step
- Number theory

23.14. Mastery check

1. Use the Euclidean algorithm to find $\gcd(391, 299)$, showing each division. Then express it as $391x + 299y$.
2. Find $\operatorname{lcm}(391, 299)$.
3. Find $11^{-1} \pmod{26}$ and verify it. Does 13 have an inverse mod 26? Why or why not?
4. Compute $5^{100} \bmod 13$ using Fermat's little theorem.
5. Compute $3^{45} \bmod 17$ using square-and-multiply.
6. In Python, `-17 // 5` and `-17 % 5` give what? In C or Java? Which one is consistent with $0 \le r < m$?
7. Toy RSA: $p = 3$, $q = 11$, $e = 3$. Find n, φ and d. Encrypt $M = 4$, then decrypt your ciphertext and confirm you get 4 back.
8. Is 0-19-853453-1 a valid ISBN-10? Show the weighted sum.

Decision rule: All correct without hesitation → move on to Chapter 24. Stuck on 1–3 → reread Section 23.4, Section 23.5 and Section 23.8. Stuck on 4, 5 or 7 → reread Section 23.9, Section 23.10 and Section 23.12. Stuck on more than three → work the section exercises before continuing.

Try it: Check gcd(391, 299) ; Check 3^{45} mod 17 ; Check the toy RSA ; Check 0-19-853453-1 .

▼ **Mastery check answers**

1. $391 = 1 \cdot 299 + 92$, $299 = 3 \cdot 92 + 23$, $92 = 4 \cdot 23 + 0$. So the GCD is 23. Back-substitute: $23 = 299 - 3 \cdot 92 = 299 - 3(391 - 299) = 4 \cdot 299 - 3 \cdot 391$, giving $x = -3, y = 4$. Check: $1196 - 1173 = 23$.
2. $391 \cdot 299 / 23 = 17 \cdot 299 = 5083$. (Indeed $391 = 17 \cdot 23$ and $299 = 13 \cdot 23$, so the LCM is $13 \cdot 17 \cdot 23 = 5083$.)
3. From $1 = 3 \cdot 26 - 7 \cdot 11$ (see the exercises), $11^{-1} \equiv -7 \equiv 19$. Check: $11 \cdot 19 = 209 = 8 \cdot 26 + 1$. 13 has no inverse mod 26 because $\gcd(13, 26) = 13 \neq 1$. Every multiple of 13 is $\equiv 0$ or $13 \pmod{26}$, never 1.
4. $5^{12} \equiv 1 \pmod{13}$ and $100 = 8 \cdot 12 + 4$, so $5^{100} \equiv 5^4$. $5^2 = 25 \equiv -1$, so $5^4 \equiv 1$. The answer is 1.
5. $45 = 101101_2 = 32 + 8 + 4 + 1$. Squares mod 17: $3^1 = 3$, $3^2 = 9$, $3^4 = 81 \equiv 13$, $3^8 \equiv 169 \equiv 16$, $3^{16} \equiv 256 \equiv 1$, $3^{32} \equiv 1$. Product: $1 \cdot 16 \cdot 13 \cdot 3 = 624 = 36 \cdot 17 + 12$. The answer is 12. (Fermat check: $3^{16} \equiv 1$, so $3^{45} \equiv 3^{13} = 3^8 \cdot 3^4 \cdot 3 \equiv 16 \cdot 13 \cdot 3 \equiv 12$.)
6. Python: -4 and 3 (floor division). C/Java: -3 and -2 (truncation). Python's remainder 3 lies in $0 \le r < 5$. Both satisfy $-17 = 5q + r$.

7. $n = 33$ and $\varphi = 2 \cdot 10 = 20$. $\gcd(3, 20) = 1$. $3d \equiv 1 \pmod{20}$ gives $d = 7$ ($21 = 20 + 1$). Encrypt: $4^3 = 64 = 33 + 31$, so $C = 31$. Decrypt: $31 \equiv -2 \pmod{33}$, so $31^7 \equiv (-2)^7 = -128 \equiv -128 + 4 \cdot 33 = 4$. Correct.

8. $10 \cdot 0 + 9 \cdot 1 + 8 \cdot 9 + 7 \cdot 8 + 6 \cdot 5 + 5 \cdot 3 + 4 \cdot 4 + 3 \cdot 5 + 2 \cdot 3 + 1 \cdot 1 = 0 + 9 + 72 + 56 + 30 + 15 + 16 + 15 + 6 + 1 = 220 = 20 \cdot 11$. That is $\equiv 0 \pmod{11}$, so the ISBN is **valid**.

Chapter 24. Counting and Combinatorics

How many passwords of length 12 are there? How many five-card poker hands? How many ways can a test suite order 10 cases, and how many subsets of 40 features could a model use? Each question asks you to count a set far too large to list. This chapter builds a small toolkit of counting principles (the sum and product rules, permutations, combinations, stars and bars, inclusion–exclusion and the pigeonhole principle) and, just as important, a way of deciding which tool a problem needs. Along the way you will see why brute-forcing a long password is hopeless, why algorithms that try every subset hit a wall, and why no compression program can shrink every file.

Where this chapter sits

You need: multiplication and powers (Volume 1, Chapter 1 , Volume 1, Chapter 3), algebraic expansion (Volume 1, Chapter 4 , Volume 1, Chapter 7), sets and logic (Chapter 21), and some comfort with proof by induction and by cases (Chapter 22).

This chapter unlocks: probability, where "favourable outcomes over total outcomes" needs exactly these counts (Chapter 29, Chapter 30); graph theory, where you will count edges, paths and spanning trees (Chapter 25); and algorithm analysis in general. The binomial coefficients you meet here reappear in the binomial distribution, in Taylor series (Volume 1, Chapter 14) and in error-correcting codes.

24.1. Why counting needs principles

24.1.1. The problem

"Counting" sounds like the first thing you learned in mathematics. One, two, three. But try these:

- How many different 4-digit PINs have no repeated digit?
- How many ways can you choose 6 numbers from 49 in a lottery?
- How many ways can 7 identical scoops of ice cream be split among 3 flavours?
- Of the numbers 1 to 1000, how many are divisible by 2, 3 or 5?

You could, in principle, list every case and count. In practice, the lists contain thousands or millions of items, and the classic mistakes are counting something twice or missing a whole family of cases. *Combinatorics* is the branch of mathematics that counts without listing. Its tools let you reduce "how many?" to a short calculation that you can trust.

24.1.2. Intuition

Every counting technique in this chapter does one of three things:

1. **Break a choice into stages** and multiply (the product rule) or into separate cases and add (the sum rule).

2. **Count something easier** and correct for over-counting, by dividing (combinations) or subtracting (inclusion–exclusion).
3. **Find a one-to-one correspondence** (a *bijection*) between the thing you want to count and something you already know how to count (stars and bars, subsets as binary strings).

If you remember those three moves, the formulas follow.



Combinatorics, enumeration, outcome

Combinatorics is the mathematics of counting arrangements and selections, and of deciding whether arrangements with given properties exist. To **enumerate** a set is to list or count its elements. An **outcome** is one specific result of a choice process (for example, one particular poker hand). For a finite set S, the notation $|S|$ means the number of elements of S, called its **cardinality**.

24.2. The sum and product rules

24.2.1. The problem

A café offers 3 mains (pasta, curry, salad) and 2 drinks (tea, water). How many different main-plus-drink meals are there? And if instead you may order *either* a main *or* one of 4 desserts (but not both), how many single-item orders are there?

24.2.2. Intuition and the tree diagram

For the meal, imagine making the choice in two stages. First pick a main: 3 branches. From *each* main, pick a drink: 2 branches. Every complete path from the root to a leaf is one meal, so the number of meals is the number of leaves.

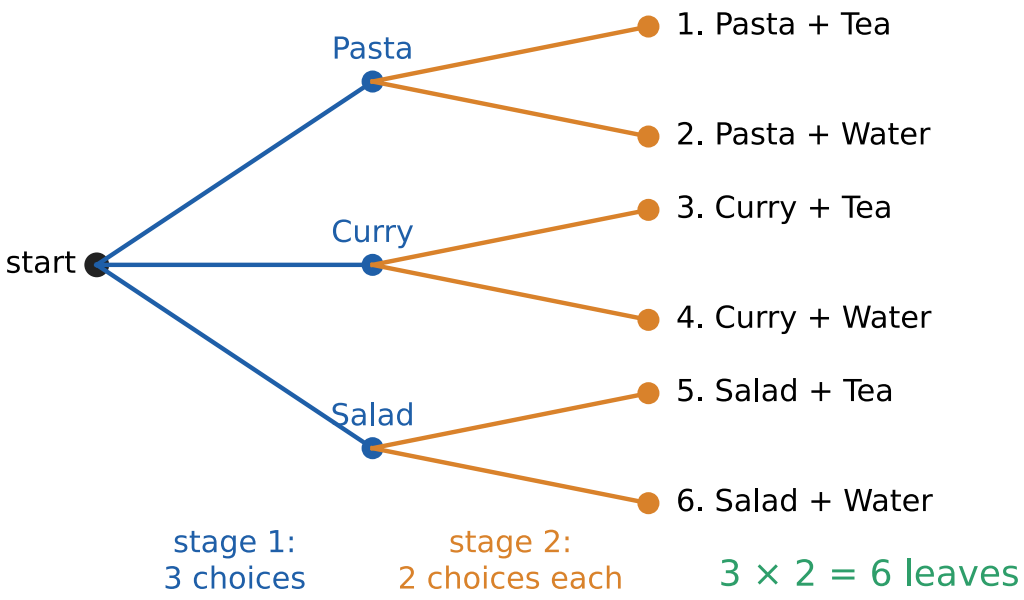


Figure 23. A tree diagram for "a main, then a drink". Each leaf is one outcome; there are 3 × 2 of them.

Each of the 3 first-stage branches sprouts the same number (2) of second-stage branches, so there are $\mathbf{3 \times 2 = 6}$ leaves. That is the product rule.

For "a main *or* a dessert", there is no sequence of stages, just two disjoint pools of options. You pick from one pool or the other, so there are $\mathbf{3 + 4 = 7}$ options. That is the sum rule.



Sum rule and product rule

Sum rule. If a choice can be made from pool A *or* from pool B, and no option is in both (the pools are **disjoint**, $A \cap B = \varnothing$), the number of choices is $|A| + |B|$.

Product rule. If a process has two stages, the first can be done in m ways, and *for each of those* the second can be done in n ways, the whole process can be done in $m \times n$ ways. It extends to any number of stages: $n_1 \times n_2 \times \cdots \times n_k$.

Rule of thumb: **"or" between disjoint cases → add; "and then" between stages → multiply.**

24.2.3. Why it works

The product rule is really the area of a rectangle. Lay the first-stage options along one side of a grid and the second-stage options along the other. Each cell is one pair, so there are m rows of n cells: mn cells. In set language, the outcomes form the *Cartesian product* $A \times B = \{(a, b) : a \in A,\ b \in B\}$, and $|A \times B| = |A|\,|B|$. With more stages you get a box, or a higher-dimensional box, and the count is still the product of the side lengths.

The important fine print is "for each". The second stage need not offer the *same* options each time, only the *same number* of options. When you choose a gold, then a silver medallist from 8 runners, the silver choices depend on who won gold, but there are always 7 of them.

Example 21. Procedure — counting by stages

Input: a description of the objects to count.

Steps:

1. Describe how to build one object as a sequence of decisions (stages).
2. Check that every object is produced by *exactly one* sequence of decisions (no object built twice, none missed).
3. For each stage, count the options available, given any earlier decisions. Check that this number does not depend on *which* earlier options were chosen.
4. Multiply the stage counts. If the objects split into disjoint cases with different stage structures, count each case this way and add.

Output: the number of objects.

24.2.4. Worked example: number plates

A (hypothetical) number plate is 3 letters followed by 4 digits, such as `KTX 3071`. How many plates are possible? How many if no letter and no digit may repeat? *Try it:* Count number plates in stages.

Stages: letter 1, letter 2, letter 3, digit 1, …, digit 4. Seven stages.

With repetition allowed, every letter stage has 26 options and every digit stage has 10:

$$\begin{aligned} N &= 26 \times 26 \times 26 \times 10 \times 10 \times 10 \times 10 \\ &= 26^3 \times 10^4 \\ &= 17\,576 \times 10\,000 \\ &= 175\,760\,000. \end{aligned}$$

Without repetition, each stage loses one option for each earlier choice of the same kind: *Try it:* Plates without repeats.

$$\begin{aligned} N &= (26 \times 25 \times 24) \times (10 \times 9 \times 8 \times 7) \\ &= 15\,600 \times 5\,040 \\ &= 78\,624\,000. \end{aligned}$$

24.2.5. Worked example: the sum rule with cases

How many positive integers less than 1000 have digits that are all odd?

Split by number of digits (disjoint cases, so add):

$$\begin{aligned} \text{1 digit: } & 5 && (1, 3, 5, 7, 9) \\ \text{2 digits: } & 5 \times 5 = 25 && (\text{product rule per case}) \\ \text{3 digits: } & 5 \times 5 \times 5 = 125 && \\ \text{total: } & 5 + 25 + 125 = 155. && (\text{sum rule across cases}) \end{aligned}$$

24.2.6. A powerful shortcut: count the complement

How many 4-digit PINs (0000 to 9999) contain *at least one* repeated digit? "At least one" questions split into many messy cases (exactly one pair, two pairs, a triple …). Instead, count what you *don't* want and subtract from the total: *Try it:* PINs with no repeated digit .

$$\begin{aligned}\textbf{all PINs} &= 10^4 = 10\,000\\ \textbf{PINs with no repeat} &= 10 \times 9 \times 8 \times 7 = 5\,040\\ \textbf{PINs with some repeat} &= 10\,000 - 5\,040 = 4\,960.\end{aligned}$$

Almost half of all PINs repeat a digit. Counting the complement is the single most useful trick in elementary counting, and it is exactly the move behind the birthday paradox later in this chapter.



Adding overlapping cases; multiplying dependent stages

- **The sum rule needs disjoint cases.** "How many cards are red or a king?" is not $26 + 4 = 30$: the two red kings were counted twice. The correct count is $26 + 4 - 2 = 28$. Fixing this properly is the job of inclusion–exclusion (Section 24.9).
- **The product rule needs a constant number of options per stage.** "Choose a card, then a second card of the same suit" works (always 12 options), but "choose a number from 1–10, then a larger number" does not: the second stage has 9 options after choosing 1 but 0 after choosing 10. Split into cases instead.
- **Each object must be built by exactly one sequence of choices.** If you count two-person teams by "pick the first person, then the second", the team {Ann, Bob} is built twice. That over-count is the whole story of combinations (Section 24.4).

Computer-science connection: nested loops are the product rule

A loop nest is a product rule you can execute:

```python
mains = ["pasta", "curry", "salad"]
drinks = ["tea", "water"]
for main in mains:          # 3 iterations
    for drink in drinks:    # 2 iterations each
        print(main, drink)  # runs 3 * 2 = 6 times
```

This is the root of complexity analysis. Two nested loops over n items do n^2 units of work; k nested loops do n^k. Python's `itertools.product(mains, drinks)` is the Cartesian product $A \times B$ packaged as an iterator. Configuration spaces (every combination of OS × browser × locale for a test matrix) grow multiplicatively for the same reason: 4 operating systems × 5 browsers × 30 locales is already 600 test configurations.

24.2.7. Where you'd use this

Biology: why the genetic code uses triplets. DNA has 4 bases and must encode 20 amino acids. Words of 2 bases give only $4^2 = 16$ possibilities, too few; words of 3 give $4^3 = 64$, enough with room to spare. The product rule predicts the three-letter codon, and the spare capacity shows up as redundancy: most amino acids have several codons.

Engineering: test matrices. A web app that must work on 3 browsers, 4 operating systems and 2 languages has $3 \times 4 \times 2 = 24$ configurations. Add 5 screen sizes and it is 120. Product-rule growth is why testers use **pairwise** testing, which covers every pair of settings with far fewer runs.

Everyday. A combination bike lock with 4 dials of 10 digits has $10^4 = 10\,000$ settings; at one try every 2 seconds, trying them all takes under 6 hours. A 3-dial lock (1,000 settings) falls in about half an hour.

US curriculum

The **Fundamental Counting Principle** (the product rule) and tree diagrams appear in **Grade 7** probability (compound events) and again in **Algebra II** or **Precalculus**. The phrase "Fundamental Counting Principle" is what a US student will recognise.

24.3. Permutations: arranging things in order

24.3.1. The problem

Eight runners contest a race. In how many ways can gold, silver and bronze be awarded? In how many ways can all eight finish, in order?

24.3.2. Intuition

Fill the podium one step at a time. Any of the 8 runners can win gold. Whoever wins, 7 remain for silver, then 6 for bronze. The product rule gives $8 \times 7 \times 6 = 336$. For a complete finishing order, keep going all the way down: $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$.



Factorial

For a positive integer n, **n factorial**, written $n!$, is the product of all integers from 1 to n: $n! = n \times (n-1) \times \cdots \times 2 \times 1$. So $3! = 6$, $5! = 120$, $10! = 3\,628\,800$. By convention $0! = 1$.

$n!$ counts the ways to arrange n distinct objects in a row. The useful identity $n! = n \times (n-1)!$ is the recursive definition you would write in code.

Permutation and r-permutation

A **permutation** of a set is an arrangement of all its elements in an order. An **r-permutation** of n distinct objects is an ordered selection of r of them (no repeats). The number of r-permutations is written $P(n, r)$, ${}_nP_r$ or $n^{\underline{r}}$ (read "n to the r falling"):

$$P(n, r) = n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!}.$$

24.3.3. Why it works: deriving $P(n, r)$

There are r stages. Stage 1 has n options. Each earlier stage has used up one object, so stage i has $n-(i-1)$ options. The last stage, stage r, has $n-r+1$. By the product rule,

$$P(n, r) = \underbrace{n(n-1)\cdots(n-r+1)}_{r \text{ factors}}.$$

To see the factorial form, multiply and divide by the "missing tail" $(n-r)!$:

$$\begin{aligned} P(n, r) &= n(n-1)\cdots(n-r+1) \cdot \frac{(n-r)(n-r-1)\cdots 1}{(n-r)(n-r-1)\cdots 1} \\ &= \frac{n!}{(n-r)!}. \end{aligned}$$

The product form is better for hand calculation ($P(8, 3) = 8 \cdot 7 \cdot 6$ is immediate, while $8!/5!$ invites you to compute two large numbers and divide). The factorial form is better for algebra.

24.3.4. Why $0! = 1$

With $r = n$, the formula says $P(n, n) = n!/0!$. But $P(n, n)$ is the number of orderings of all n objects, which is $n!$. So $0!$ has to be 1 for the formula to work. There is also a direct reason: there is exactly one way to arrange nothing (the empty arrangement). And the recursion $n! = n \cdot (n-1)!$ at $n = 1$ gives $1 = 1 \cdot 0!$. Three different arguments, one answer: this is a definition chosen because it makes everything consistent, not an arbitrary rule.

24.3.5. Worked examples

Podium. $P(8, 3) = 8 \times 7 \times 6 = 336$.

Full finishing order. $8! = 40\,320$.

Seating with a constraint. Five people, including Ann and Bob, sit in a row. In how many arrangements are Ann and Bob next to each other?

Glue Ann and Bob into a single block. Now arrange 4 units (the block and three others), then arrange the two people inside the block:

$$\begin{aligned}\text{arrange 4 units}&: 4! = 24\\ \text{order inside block}&: 2! = 2\\ \text{total}&: 24 \times 2 = 48.\end{aligned}$$

As a check, the arrangements where they are *not* adjacent number $5! - 48 = 120 - 48 = 72$.

Round table. In how many ways can 5 people sit round a circular table, if only who sits next to whom matters (rotations are the same seating)? Each circular seating corresponds to 5 different linear ones (start reading at any of the 5 seats), so the count is $5!/5 = 4! = 24$. In general, $(n-1)!$ circular arrangements. This is your first glimpse of "count something easier, then divide out the over-count". *Try it:* Round-table seatings counted five times .



Permutations in the wild

A standard deck has $52! \approx 8.07 \times 10^{67}$ orderings. That is so large that every thoroughly shuffled deck in history has almost certainly been in an order never seen before. If every person on Earth had shuffled a deck once a second since the Big Bang, the total number of shuffles would be roughly 3×10^{27}, a fraction of about 4×10^{-41} of the possible orders.

Computer-science connection: sorting and n!

A comparison sort must distinguish all $n!$ possible input orders, and each comparison has two outcomes. A decision tree with $n!$ leaves needs depth at least $\log_2(n!)$, and $\log_2(n!) \approx n \log_2 n$ (see Volume 1, Chapter 12). That is the counting argument behind the $O(n \log n)$ lower bound for comparison sorting: no amount of cleverness can beat it.

24.3.6. Where you'd use this

Sport. With 8 sprinters in a final, the number of possible podiums (gold, silver, bronze) is $P(8,3) = 8 \cdot 7 \cdot 6 = 336$. Order matters, so it is a permutation; the number of possible **sets** of medallists is only $336/3! = 56$.

Logistics. A delivery driver with 10 drops can visit them in $10! = 3\,628\,800$ orders. With 20 drops it is about 2.4×10^{18}, which is why route planners use heuristics instead of checking every order (the travelling salesman problem, Chapter 25).

History of cryptography. The German Army's Enigma machine placed 3 of its 5 rotors into 3 slots, and order mattered: $P(5,3) = 60$ wheel orders. Each of the 60 had to be tested separately, which is one reason the Bletchley Park codebreakers built banks of electromechanical "bombes" to run in parallel.

24.4. Combinations: choosing when order doesn't matter

24.4.1. The problem

A club of 10 people must choose a committee of 3. How many different committees are possible?

24.4.2. Intuition: count ordered, then divide

You already know how to count *ordered* choices: $P(10, 3) = 10 \cdot 9 \cdot 8 = 720$. But a committee is a *set* of people; {Ann, Bob, Cat} is the same committee however you list them. Each committee shows up in the ordered count once for each of its orderings, and three people can be ordered in $3! = 6$ ways: *Try it:* Each committee counted 3! times .

```
ordered lists that all describe the committee {A, B, C}:
  ABC  ACB  BAC  BCA  CAB  CBA      (3! = 6 of them)
```

Every committee is counted exactly 6 times, so the number of committees is $720/6 = 120$.



Combination and binomial coefficient

A **combination** (or r-combination) of n distinct objects is an *unordered* selection of r of them, that is, a subset of size r. The number of them is the **binomial coefficient**

$$\binom{n}{r} = \frac{n!}{r!\,(n-r)!},$$

read "n choose r". Other notations: $C(n, r)$, ${}_nC_r$, ${}^nC_r$ (common in UK textbooks). By convention $\binom{n}{r} = 0$ if $r < 0$ or $r > n$.

24.4.3. Why it works: the division principle

Here is the derivation made precise. Consider all ordered selections of r objects from n. Build each one in two stages:

1. choose *which* r objects (a combination): $\binom{n}{r}$ ways;
2. choose *the order* of those r objects: $r!$ ways.

Every ordered selection arises exactly once this way, so by the product rule

$$P(n, r) = \binom{n}{r} \cdot r!$$

$$\binom{n}{r} = \frac{P(n, r)}{r!} = \frac{n!}{(n-r)!\,r!}.$$

This is the *division principle*: if a count treats every object you care about exactly k times, divide by k.

For hand calculation, use the cancelled form: $\binom{n}{r} = \dfrac{n(n-1)\cdots(n-r+1)}{r!}$, with r factors on top and $r!$ below.

24.4.4. Symmetry

Choosing the 3 people who *are* on the committee is the same as choosing the 7 who *are not*. So

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

The formula confirms it (swap r and $n-r$ in the denominator), but the counting argument says *why*. It also saves work: $\binom{100}{98} = \binom{100}{2} = \dfrac{100 \cdot 99}{2} = 4950$.

24.4.5. Worked example: committees with structure

From 10 people (4 women and 6 men), how many 4-person committees have exactly 2 women?

Stage 1, choose the women; stage 2, choose the men. These are independent, so multiply:

$$\begin{aligned}\binom{4}{2}\binom{6}{2} &= \frac{4 \cdot 3}{2 \cdot 1} \cdot \frac{6 \cdot 5}{2 \cdot 1} \\ &= 6 \times 15 \\ &= 90.\end{aligned}$$

How many have *at least one* woman? Use the complement:

$$\begin{aligned}\textbf{all committees} &= \binom{10}{4} = \frac{10 \cdot 9 \cdot 8 \cdot 7}{4 \cdot 3 \cdot 2 \cdot 1} = \frac{5040}{24} = 210 \\ \textbf{no women} &= \binom{6}{4} = \binom{6}{2} = 15 \\ \textbf{at least one woman} &= 210 - 15 = 195.\end{aligned}$$

24.4.6. Worked example: a committee with a chair

Choose a 3-person committee from 10 *and* name one of the three as chair. Two methods: *Try it:* Committee with a chair, both ways .

- **Committee first.** $\binom{10}{3} = 120$ committees, then 3 choices of chair: $120 \times 3 = 360$.
- **Chair first.** 10 choices of chair, then choose the other 2 from the remaining 9: $10 \times \binom{9}{2} = 10 \times 36 = 360$.

Both agree, which is the point. When you can count the same set two ways, you have not just a check but an identity: $3\binom{10}{3} = 10\binom{9}{2}$, and in general $r\binom{n}{r} = n\binom{n-1}{r-1}$. Proofs of this kind, called *double counting* or *combinatorial proofs*, are the subject's favourite technique, and you will use one next for Pascal's rule.



Forgetting whether order matters

The most common counting error is using P where C belongs or the reverse. Ask: *if I swapped two of the chosen items, would I have a different outcome?* Swapping two committee members gives the same committee (use C). Swapping the gold and silver medallists gives a different podium (use P). Swapping two digits of a PIN gives a different PIN (order matters, and repetition is allowed: n^r, see Section 24.11).

A second trap: mixing ordered and unordered stages. "Choose 2 women, then 2 men" is fine. But "choose one woman to guarantee at least one, then choose 3 more from the other 9" gives $4 \times \binom{9}{3} = 336$, not 195, because a committee with two women is built twice (once for each woman as "the guaranteed one"). When an answer to an "at least" question comes out too big, suspect this.

Try it: See the 336 over-count .

Computer-science connection: math.comb and itertools

Python's standard library knows all of this:

```python
import math

def perm(n, r):
    """Ordered selections of r from n distinct items: n!/(n-r)!"""
    return math.factorial(n) // math.factorial(n - r)

def comb(n, r):
    """Unordered selections: divide out the r! orderings of each selection."""
    return perm(n, r) // math.factorial(r)

print(perm(10, 3), math.perm(10, 3))
print(comb(10, 3), math.comb(10, 3))
print(math.comb(52, 5))
print(math.comb(49, 6))
```

```
720 720
120 120
2598960
13983816
```

`math.comb` and `math.perm` (Python 3.8+) use exact integer arithmetic, so they never overflow. When you need the objects themselves rather than the count, `itertools` generates them lazily:

```python
from itertools import permutations, combinations, product, combinations_with_replacement

letters = "ABC"
print(list(permutations(letters, 2)))
print(list(combinations(letters, 2)))
print(list(product(letters, repeat=2)))
print(list(combinations_with_replacement(letters, 2)))
```

```
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
[('A', 'B'), ('A', 'C'), ('B', 'C')]
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'), ('C', 'A'), ('C',
'B'), ('C', 'C')]
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

The four lines have 6, 3, 9 and 6 items: $P(3,2)$, $\binom{3}{2}$, 3^2 and $\binom{4}{2}$. Those are exactly the four cells of the decision table in Section 24.11.

24.4.7. Where you'd use this

Sport. A 20-team league in which every pair of teams meets twice (home and away) plays $2\binom{20}{2} = 380$ matches, which is exactly the length of an English Premier League season.

Engineering: links in a network. Connecting every pair of 50 offices directly needs $\binom{50}{2} = 1225$ links; routing everything through one hub needs 49. That quadratic-versus-linear gap is why telephone exchanges, Ethernet switches and airline hubs exist.

Science: screening combinations. A lab with 20 candidate drugs that wants to test every pair needs $\binom{20}{2} = 190$ experiments; every triple needs $\binom{20}{3} = 1140$. Counting first tells you whether an exhaustive screen is affordable or whether you need a statistical design that samples combinations.

Everyday. Choosing 3 pizza toppings from 12 gives $\binom{12}{3} = 220$ different pizzas, and the menu that boasts "over 200 combinations" is quoting exactly this number.

> ### US curriculum
>
> Permutations and combinations (usually written ${}_nP_r$ and ${}_nC_r$, and computed on a TI-84 graphing calculator) appear in **Algebra II** and **Precalculus**, and heavily in **AP Statistics** as preparation for probability. Combinatorial proof and double counting are usually first met in a college **Discrete Mathematics** course or in competition preparation (AMC 10/12).

24.5. Binomial coefficients and Pascal's triangle

24.5.1. The problem

Computing $\binom{n}{r}$ from factorials is fine for a calculator, but it hides a beautiful structure. Is there a way to build all the binomial coefficients from simpler ones, without any multiplication or division?

24.5.2. Pascal's triangle

Write $\binom{n}{0}, \binom{n}{1}, \dots, \binom{n}{n}$ as row n of a triangle, rows centred:

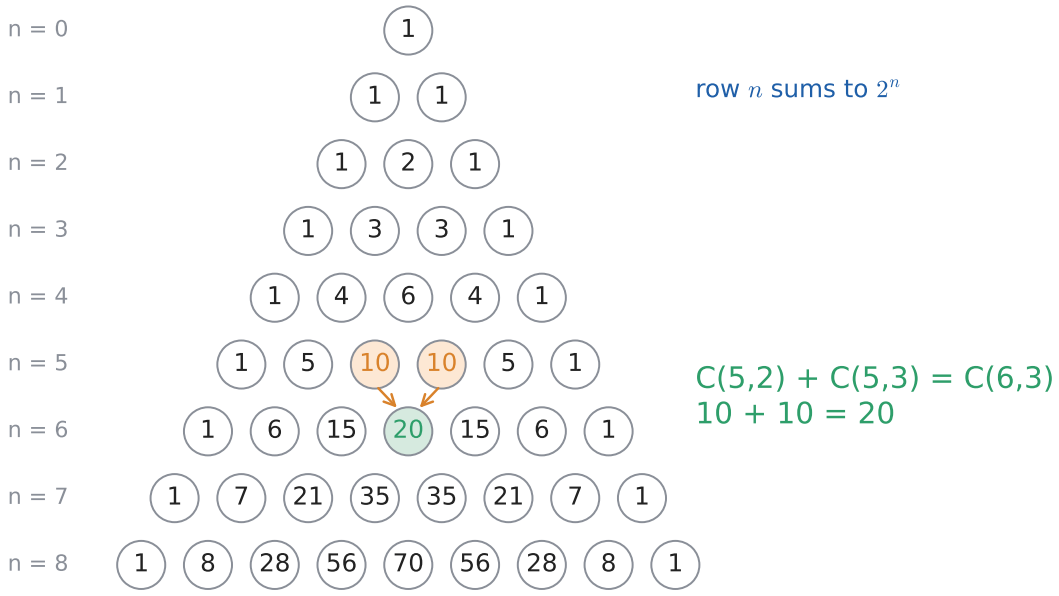


Figure 24. Pascal's triangle, rows 0 to 8. Each entry is the sum of the two entries above it.

Each row begins and ends with 1, and every interior entry is the sum of the two diagonally above it. In symbols, that is Pascal's rule. *Try it:* Explore Pascal's triangle .

24.5.3. Pascal's rule and its counting proof

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r} \qquad (1 \le r \le n-1).$$

You could prove this by adding fractions of factorials. The counting proof is shorter and tells you *why*.

Proof. Count the r-element subsets of a set of n people, one of whom is called Zoe. The left side, $\binom{n}{r}$, counts all of them directly. Now sort the subsets into two disjoint cases:

- **Subsets containing Zoe.** Zoe is in; choose the other $r-1$ members from the remaining $n-1$ people: $\binom{n-1}{r-1}$ ways.
- **Subsets not containing Zoe.** Choose all r members from the other $n-1$ people: $\binom{n-1}{r}$ ways.

Every subset is in exactly one case, so by the sum rule the total is $\binom{n-1}{r-1} + \binom{n-1}{r}$. Both sides count the same set, so they are equal. ■

This "single out one element and ask whether it's in" move is also the recursion behind many algorithms (the 0/1 knapsack dynamic programme makes exactly this decision for each item).

24.5.4. More patterns in the triangle

- **Row sums are powers of 2:** $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = 2^n$. Counting proof: the left side counts subsets of an n-element set by size; the right side counts them by deciding, for each element, "in or out", 2 ways each, n times.
- **Symmetry** $\binom{n}{r} = \binom{n}{n-r}$ is the left–right mirror symmetry.

- **The hockey stick.** Summing down a diagonal gives the entry just below and to the side: $\binom{2}{2}+\binom{3}{2}+\binom{4}{2}+\binom{5}{2}=1+3+6+10=20=\binom{6}{3}$. *Try it:* Highlight the hockey stick .
- **Diagonals:** the second diagonal (1, 2, 3, 4, …) is the counting numbers; the third (1, 3, 6, 10, 15, …) is the triangular numbers $\binom{n}{2}=n(n-1)/2$, the number of handshakes among n people or the number of pairs to compare in a naive all-pairs algorithm.

Example 22. Procedure — computing binomial coefficients

Input: non-negative integers n and $r \le n$.

Steps:

1. If $r > n-r$, replace r by $n-r$ (symmetry; fewer factors).
2. Set $c=1$.
3. For $i=1,2,\dots,r$: set $c \leftarrow c\cdot(n-r+i)/i$. (After step i, $c=\binom{n-r+i}{i}$, which is an integer, so the division is always exact.)

Output: $c=\binom{n}{r}$.

For example, $\binom{8}{3}$: $c=1\cdot 6/1=6$, then $6\cdot 7/2=21$, then $21\cdot 8/3=56$. The intermediate values 6, 21, 56 are themselves binomial coefficients ($\binom{6}{1},\binom{7}{2},\binom{8}{3}$). If instead you need a whole table, build Pascal's triangle by addition, which is what a dynamic-programming solution does.



Not Pascal's triangle

The triangle is named after Blaise Pascal's 1654 treatise, but it was known centuries earlier: to Jia Xian in 11th-century China (it is called *Yang Hui's triangle* in China after the 13th-century mathematician who popularised it), to Omar Khayyam in Persia (*Khayyam's triangle* in Iran), and in India, where Pingala's work on poetic metres (c. 200 BCE) implicitly counted combinations. Colour the odd entries black and the even ones white, and Sierpiński's fractal triangle appears.

Try it: Sierpiński in 64 rows .



Pascal's triangle explorer

Display Pascal's triangle up to row 64. Let the reader colour entries by their remainder modulo 2, 3 or 5, and watch fractal patterns emerge. Clicking an entry highlights the two entries that sum to it, the hockey-stick diagonal ending at it, and its value as $n!/(r!(n-r)!)$.

Try it: Pascal's triangle mod 3 · Pascal's triangle mod 5 · Pascal's triangle explorer (mod 2)

24.6. The binomial theorem

24.6.1. The problem

Expand $(x+y)^5$ without multiplying out five brackets by hand. More generally, what is the coefficient of x^3y^2 in $(x+y)^5$, and why are binomial coefficients called *binomial*?

24.6.2. Intuition: expansion is a counting process

Write $(x+y)^3 = (x+y)(x+y)(x+y)$. Fully expanding means: from each of the three brackets pick either x or y, multiply your three picks, and add up the results over all $2^3 = 8$ ways of picking:

```
xxx  xxy  xyx  xyy  yxx  yxy  yyx  yyy
```

Try it: Pick x or y, visually .

The term x^2y appears once for each way of choosing *which one* bracket supplies the y: $\binom{3}{1} = 3$ ways (xxy, xyx, yxx). So

$$(x+y)^3 = \binom{3}{0}x^3 + \binom{3}{1}x^2y + \binom{3}{2}xy^2 + \binom{3}{3}y^3 = x^3 + 3x^2y + 3xy^2 + y^3.$$

The coefficients 1, 3, 3, 1 are row 3 of Pascal's triangle. That is why they're called binomial coefficients: they are the coefficients in the expansion of a power of a *binomial* (a two-term expression).



Binomial, the Σ notation

A **binomial** is a polynomial with two terms, such as $x+y$ or $2x-1$.

The **summation sign** $\sum$ (capital Greek sigma, for "sum") packages a long sum. $\sum_{k=0}^{n} a_k$ means "let the **index** k run through the integers from the lower limit 0 to the upper limit n, and add up the terms a_k": $a_0 + a_1 + \cdots + a_n$.

24.6.3. The theorem

For any numbers x, y and non-negative integer n,

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

Why. Each term of the expansion comes from choosing x or y from each of the n brackets. A choice that takes y from exactly k brackets (and so x from the other $n-k$) contributes $x^{n-k}y^k$. The number of such choices is the number of ways to pick which k brackets supply y, namely $\binom{n}{k}$. Collect like terms and you get the theorem. (A proof by induction using Pascal's rule also works; see Chapter 22. The counting proof is the one that explains it.)

Example 23. Procedure — expanding $(a+b)^n$

Input: the two terms a, b (each may include a coefficient and a sign), and n.

Steps:

1. Write row n of Pascal's triangle: $\binom{n}{0}, \dots, \binom{n}{n}$.
2. For $k = 0, \dots, n$, form the term $\binom{n}{k}\, a^{n-k}\, b^{k}$: powers of a count down from n, powers of b count up from 0.
3. Simplify each term, keeping track of the signs and coefficients inside a and b.

Output: the expanded polynomial, with $n+1$ terms.

24.6.4. Worked example: $(x+2)^4$

Here $a = x$, $b = 2$, $n = 4$; row 4 is 1, 4, 6, 4, 1. *Try it:* Expand (x + 2)⁴ .

$$\begin{aligned}(x+2)^4 &= 1\cdot x^4\cdot 2^0 + 4\cdot x^3\cdot 2^1 + 6\cdot x^2\cdot 2^2 + 4\cdot x\cdot 2^3 + 1\cdot x^0\cdot 2^4\\ &= x^4 + 4\cdot 2\,x^3 + 6\cdot 4\,x^2 + 4\cdot 8\,x + 16\\ &= x^4 + 8x^3 + 24x^2 + 32x + 16.\end{aligned}$$

Check by substituting $x = 1$: the left side is $3^4 = 81$; the right side is $1 + 8 + 24 + 32 + 16 = 81$. Substituting a convenient value is a quick, reliable test of any expansion.

24.6.5. Worked example: a single coefficient

Find the coefficient of x^3 in $(2x-1)^5$. You do not need the whole expansion. Here $a = 2x$, $b = -1$, $n = 5$. A term has the form $\binom{5}{k}(2x)^{5-k}(-1)^k$, which contains x^{5-k}. For x^3 you need $5-k = 3$, so $k = 2$: *Try it:* Find the x³ coefficient .

$$\begin{aligned}\binom{5}{2}(2x)^3(-1)^2 &= 10\cdot 8x^3\cdot 1\\ &= 80x^3.\end{aligned}$$

The coefficient is 80.

24.6.6. Consequences you get for free

- Put $x = y = 1$: $2^n = \sum_k \binom{n}{k}$, the row-sum identity again.
- Put $x = 1, y = -1$: $0 = \binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \cdots$ for $n \geq 1$. A set has as many even-sized subsets as odd-sized ones.
- For small y, $(1+y)^n \approx 1 + ny + \binom{n}{2}y^2$. With $y = 0.05$ and $n = 10$, compound growth $1.05^{10} \approx 1 + 0.5 + 45 \times 0.0025 = 1.6125$; the true value is 1.6289. This truncated expansion is the seed of Taylor series (Volume 1, Chapter 14).



Dropping the coefficients inside the terms

In $(2x-1)^5$, the term is $\binom{5}{k}(2x)^{5-k}(-1)^k$, not $\binom{5}{k}\,2x^{5-k}\ldots$. The 2 is raised to the power too, and so is the minus sign. Bracket each of a and b *before* raising them to powers, and the signs take care of themselves (they alternate when b is negative).

Computer-science connection: binomial coefficients at work

- **Probability.** The probability of exactly k heads in n tosses of a coin with heads-probability p is $\binom{n}{k}p^k(1-p)^{n-k}$, one term of $(p+(1-p))^n=1$. This is the binomial distribution (Chapter 30).
- **Error-correcting codes.** The number of bit strings of length n within Hamming distance t of a codeword is $\sum_{k=0}^{t}\binom{n}{k}$, which bounds how many codewords can fit (the Hamming bound).
- **Big-number arithmetic.** $(a+b)^p \equiv a^p + b^p \pmod p$ for a prime p, because every middle coefficient $\binom{p}{k}$ is divisible by p. This "freshman's dream" underlies Fermat's little theorem and primality tests (Chapter 23).

24.6.7. Where you'd use this

Genetics: how likely is "exactly one"? Two parents who both carry a recessive gene each pass it on with probability $\frac{1}{2}$, so each child is affected with probability $\frac{1}{4}$. In a family of 4 children, the chance that exactly one is affected is one term of the binomial expansion of $(\frac{1}{4}+\frac{3}{4})^4$: *Try it:* Exactly one of four children .

$$\binom{4}{1}\left(\tfrac{1}{4}\right)^1\left(\tfrac{3}{4}\right)^3 = 4\cdot\frac{1}{4}\cdot\frac{27}{64} = \frac{27}{64} \approx 0.42.$$

The coefficient 4 counts **which** child it is; the powers give the probability of one particular pattern. Chapter 30 turns this into the binomial distribution.

Finance and physics: quick approximations. The truncated expansion above, $(1+y)^n \approx 1+ny$, is how people estimate small compound changes in their heads (2% a year for 5 years is roughly 10%; the exact figure is 10.4%), and how physicists simplify formulas such as $(1-v^2/c^2)^{-1/2} \approx 1+\frac{1}{2}v^2/c^2$ at everyday speeds (the binomial series for non-integer powers, Volume 1, Chapter 14).

US curriculum

The binomial theorem and its link to Pascal's triangle are taught in **Algebra II** or **Precalculus** (Common Core standard A-APR.C.5). Sigma notation is usually introduced alongside sequences and series in Precalculus.

24.7. Repetition: words, multisets and anagrams

24.7.1. The problem

How many 4-letter "words" (any strings, not necessarily English) use the letters A–Z? How many distinct rearrangements are there of the letters of MISSISSIPPI?

24.7.2. Ordered selection with repetition

When repetition is allowed and order matters, each of the r positions independently has n options:

$$\underbrace{n \times n \times \cdots \times n}_{r \text{ times}} = n^r.$$

So there are $26^4 = 456\,976$ four-letter strings, 10^4 four-digit PINs, and 2^n bit strings of length n. This is the count behind every keyspace calculation in security.

24.7.3. Arrangements of a multiset

MISSISSIPPI has 11 letters, but they are not all distinct: M once, I four times, S four times, P twice. If the letters were all different (label them I_1, I_2, I_3, I_4 and so on), there would be $11!$ arrangements. But swapping the labelled I's among themselves gives the same word; each word is counted $4!$ times on account of the I's, and likewise $4!$ for the S's and $2!$ for the P's. Divide out each over-count: *Try it:* Step through MISSISSIPPI anagrams .

$$\begin{aligned} \frac{11!}{1!\,4!\,4!\,2!} &= \frac{39\,916\,800}{1 \cdot 24 \cdot 24 \cdot 2} \\ &= \frac{39\,916\,800}{1152} \\ &= 34\,650. \end{aligned}$$



Multiset and multinomial coefficient

A **multiset** is like a set but elements may appear more than once: $\{M, I, I, I, I, S, S, S, S, P, P\}$. The number of distinct orderings of a multiset with n items, of which n_1 are of type 1, n_2 of type 2, ..., n_k of type k, is the **multinomial coefficient**

$$\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{n_1!\, n_2! \cdots n_k!}, \qquad n_1 + \cdots + n_k = n.$$

With two types it is just $\binom{n}{n_1}$: arranging r identical H's and $n - r$ identical T's is choosing which r positions get an H.

Another way to see it. Place the letters into 11 slots in stages: choose 1 slot for M, then 4 of the remaining 10 for the I's, then 4 of the remaining 6 for the S's, and the last 2 go to the P's:

$$\binom{11}{1}\binom{10}{4}\binom{6}{4}\binom{2}{2} = 11 \times 210 \times 15 \times 1 = 34\,650.$$

Same answer, by a method that never divides. Both are worth having: the first is faster to write, the second is easier to justify step by step.

Grid paths. How many shortest routes are there on a city grid from one corner to a point 5 blocks east and 3 blocks north? Every shortest route is a string of 5 E's and 3 N's in some order, so the count is $\binom{8}{3} = 56$. Recognising a problem as "arrangements of a multiset" in disguise is a recurring skill. *Try it:* Walk the 56 grid routes .

24.8. Stars and bars: distributing identical objects

24.8.1. The problem

An ice-cream shop has 3 flavours: vanilla, chocolate and mint. You order a tub of 7 scoops. Scoops of the same flavour are identical, and only how many of each flavour you get matters. How many different tubs are there?

In algebraic form: how many solutions are there to

$$x_1 + x_2 + x_3 = 7, \qquad x_i \text{ non-negative integers?}$$

24.8.2. Intuition: a bijection with strings

Draw each scoop as a star ★ and separate the flavours with 2 bars:

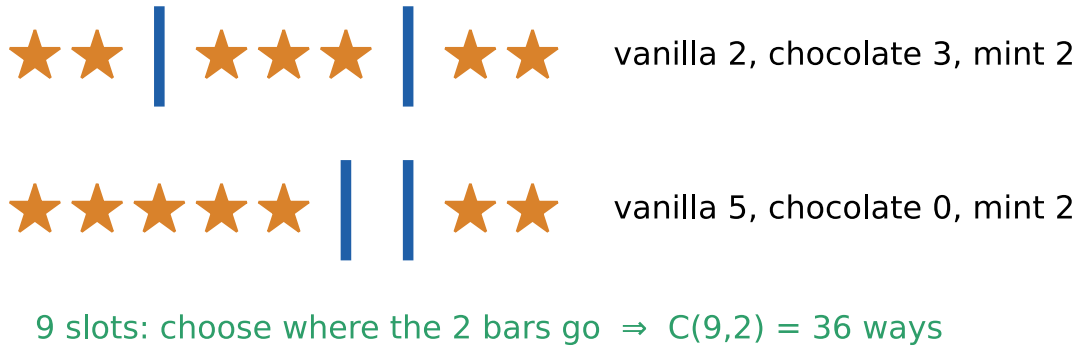


Figure 25. Two tubs of 7 scoops encoded as stars and bars. Every arrangement of 7 stars and 2 bars is a different tub, and vice versa.

Stars before the first bar are vanilla, between the bars chocolate, after the second bar mint. Two bars together (`||*`) means zero chocolate. Every tub gives exactly one such string, and every string of 7 stars and 2 bars gives exactly one tub. So count the strings: 9 positions, choose which 2 hold bars: $\binom{9}{2} = 36$. *Try it:* Drag the stars and bars .

24.8.3. The general formula

Distributing n identical objects into k distinct boxes (empty boxes allowed) uses n stars and $k-1$ bars, so

$$\#\text{ways} = \binom{n+k-1}{k-1} = \binom{n+k-1}{n}.$$

Equivalently, this is the number of *multisets* of size n drawn from k types: unordered selection *with* repetition. That fills the last cell of the order/repetition table.

24.8.4. Worked example: at least one of each

How many tubs of 7 scoops have at least one scoop of each flavour ($x_i \geq 1$)? *Try it:* At least one of each .

Hand out one scoop of each flavour first. That leaves 4 scoops to distribute freely among 3 flavours: $y_1 + y_2 + y_3 = 4$ with $y_i = x_i - 1 \geq 0$:

$$\binom{4+3-1}{3-1} = \binom{6}{2} = 15.$$

Alternatively, think of the 7 stars in a row with 6 gaps between them; placing 2 bars in 2 *different* gaps (no empty flavour) is $\binom{6}{2} = 15$.

24.8.5. Worked example: a dozen doughnuts

A shop sells 5 kinds of doughnut. How many different boxes of 12 can you buy? Here $n = 12$ objects into $k = 5$ kinds: *Try it:* List the 1820 doughnut boxes .

$$\binom{12+5-1}{5-1} = \binom{16}{4} = \frac{16 \cdot 15 \cdot 14 \cdot 13}{4 \cdot 3 \cdot 2 \cdot 1} = \frac{43\,680}{24} = 1820.$$

Example 24. Procedure — distributing identical objects

Input: n identical objects, k distinct boxes, lower bounds $m_1, \ldots, m_k$ (use 0 if none).

Steps:

1. Pre-load each box i with its minimum m_i. Set $n' = n - (m_1 + \cdots + m_k)$. If $n' < 0$, the answer is 0.
2. Distribute the remaining n' objects freely: $\binom{n'+k-1}{k-1}$.
3. If there are also upper bounds, count the solutions that violate them and remove those with inclusion–exclusion (Section 24.9).

Output: the number of distributions.



Distinct objects are a different problem

Stars and bars needs **identical objects**. Seven *different* people choosing among 3 rooms is $3^7 = 2187$ (each person independently picks a room), not $\binom{9}{2}$. Ask: if I swapped two of the objects, would the outcome change? If yes, they are distinct and you want the product rule.

Computer-science connection: stars and bars in the wild

Stars and bars counts the monomials of degree n in k variables ($x^2y^3z^2$ is "7 scoops: 2 of x, 3 of y, 2 of z"), so it tells you how many features a degree-d polynomial model creates; for 10 input features and degree 3, there are $\binom{12}{3} = 220$ monomials of degree exactly 3. It also counts ways to split a budget of n identical units (CPU shares, replicas) among k services, the size of the search space for such allocation problems.

24.8.6. Where you'd use this

Physics: counting microstates. In the simplest model of a solid (Einstein's), energy comes in identical packets, **quanta**, shared among atomic oscillators. The number of ways to share q quanta among N oscillators is a stars-and-bars count, $\binom{q+N-1}{q}$. Three quanta among three oscillators: $\binom{5}{3} = 10$ microstates (3-0-0 in 3 ways, 2-1-0 in 6 ways, 1-1-1 in 1 way). Statistical mechanics defines entropy from exactly this count, and it is why heat flows from hot to cold: that direction has overwhelmingly more microstates.

Everyday. Splitting \$10 of pocket money, in whole dollars, between saving, spending and giving: $\binom{12}{2} = 66$ possible plans. Allocating 6 identical seats among 3 parties: $\binom{8}{2} = 28$ possible results.



Stars and bars sliders

Choose n (scoops) and k (flavours). The lab shows a random distribution as stars and bars and as a vector $(x_1, \ldots, x_k)$; dragging a bar moves stars between flavours. A counter enumerates all distributions for small n, k and confirms the formula $\binom{n+k-1}{k-1}$.

Try it: Stars and bars sliders

24.9. Inclusion–exclusion

24.9.1. The problem

Of the integers from 1 to 1000, how many are divisible by 2, by 3 or by 5?

24.9.2. Intuition: correct the double counting

For two sets, adding $|A| + |B|$ counts the overlap twice, so subtract it once:

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

For three sets, adding the three sizes counts each two-way overlap twice and the triple overlap three times. Subtracting the three pairwise intersections fixes the two-way regions, but now the triple region has been added 3 times and subtracted 3 times, so it isn't counted at all. Add it back once:

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

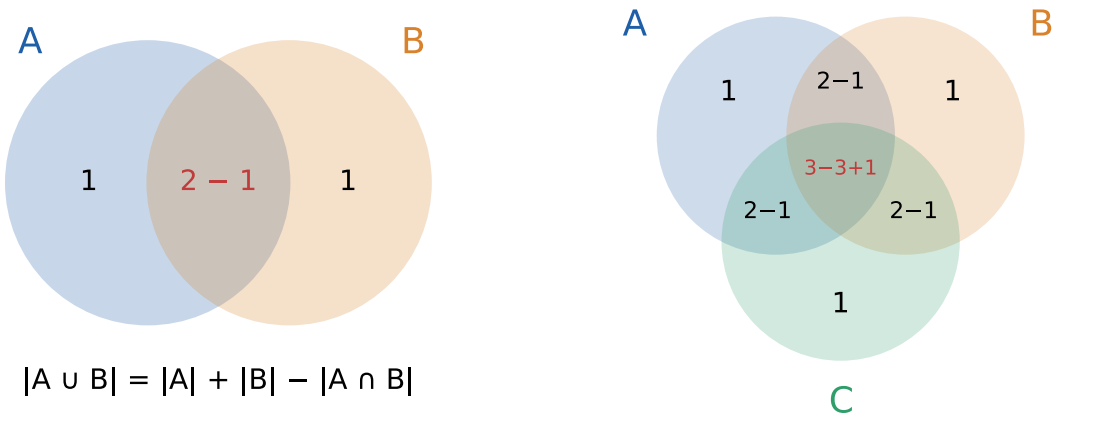


Figure 26. Inclusion–exclusion for two and three sets: the bookkeeping in each region of the Venn diagram.



Union, intersection, Venn diagram

The **union** $A \cup B$ is the set of elements in A or B (or both). The **intersection** $A \cap B$ is the set of elements in both. A **Venn diagram** draws sets as overlapping regions so that each region represents one combination of "in" and "out". See Chapter 21 for the full set vocabulary.

24.9.3. Why it works in general

For k sets the pattern continues: add the single sets, subtract all pairwise intersections, add all triple intersections, subtract quadruples, and so on, with alternating signs. Why does every element end up counted exactly once? Take an element that lies in exactly $m \geq 1$ of the sets. It is counted $\binom{m}{1}$ times in the singles, $\binom{m}{2}$ times in the pairs, and so on, so its total count is

$$\binom{m}{1} - \binom{m}{2} + \binom{m}{3} - \cdots \pm \binom{m}{m} = 1 - \left(\binom{m}{0} - \binom{m}{1} + \binom{m}{2} - \cdots \right) = 1 - 0 = 1,$$

using the alternating-sum identity from the binomial theorem (put $x = 1, y = -1$ in $(x+y)^m$). The two big ideas of the chapter meet here.

24.9.4. Worked example: divisibility

Let A_2, A_3, A_5 be the multiples of 2, 3, 5 in $\{1, \ldots, 1000\}$. The number of multiples of d up to N is $\lfloor N/d \rfloor$ (the **floor**, or integer part, of N/d). A number divisible by both 2 and 3 is divisible by 6, and so on (because 2, 3, 5 are pairwise coprime; see Chapter 23). *Try it:* Multiples of 2, 3, 5, counted .

$$\begin{aligned} |A_2| + |A_3| + |A_5| &= 500 + 333 + 200 = 1033 \\ |A_2 \cap A_3| + |A_2 \cap A_5| + |A_3 \cap A_5| &= \lfloor 1000/6 \rfloor + \lfloor 1000/10 \rfloor + \lfloor 1000/15 \rfloor = 166 + 100 + 66 = 332 \\ |A_2 \cap A_3 \cap A_5| &= \lfloor 1000/30 \rfloor = 33 \\ |A_2 \cup A_3 \cup A_5| &= 1033 - 332 + 33 = 734. \end{aligned}$$

So 734 numbers are divisible by at least one of 2, 3, 5, and $1000 - 734 = 266$ are divisible by none. A one-line brute-force check agrees:

```python
brute = sum(1 for k in range(1, 1001) if k % 2 == 0 or k % 3 == 0 or k % 5 == 0)
formula = (500 + 333 + 200) - (166 + 100 + 66) + 33
print(brute, formula)
```

```
734 734
```

Example 25. Procedure — inclusion–exclusion

Input: sets $A_1, \dots, A_k$ whose intersections are easy to count.

Steps:

1. Compute $S_1 = \sum |A_i|$ (all singles).
2. Compute $S_2 = \sum |A_i \cap A_j|$ over all pairs $i < j$.
3. Continue: S_t is the sum over all $\binom{k}{t}$ intersections of t sets.
4. Combine with alternating signs: $|A_1 \cup \cdots \cup A_k| = S_1 - S_2 + S_3 - \cdots + (-1)^{k+1} S_k$.
5. If you want the elements in *none* of the sets, subtract from the total.

Output: the size of the union (or of its complement).



Hats, letters and 1/e

n people check their hats; the hats are handed back at random. What is the chance nobody gets their own hat? A permutation with no element in its original place is a **derangement**. Let A_i be the permutations that fix person i. Any t specified people can all get their own hats in $(n-t)!$ ways, so inclusion–exclusion gives

$$D_n = n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \cdots = n! \sum_{t=0}^{n} \frac{(-1)^t}{t!}.$$

For $n = 4$: $D_4 = 24(1 - 1 + \tfrac{1}{2} - \tfrac{1}{6} + \tfrac{1}{24}) = 12 - 4 + 1 = 9$. The probability $D_n/n!$ is a partial sum of the series for e^{-1}, so it approaches $1/e \approx 0.368$ astonishingly fast, and it is essentially the same whether 10 or 10 million people check hats. A Secret Santa draw that must be redrawn whenever someone picks themselves will need, on average, about $e \approx 2.7$ attempts.

Try it: Derangements racing to 1/e ; Count the 9 derangements .



When inclusion–exclusion becomes impractical

With k sets there are $2^k - 1$ intersection terms. That's fine for 3 or 4 sets, or when symmetry makes all t-fold intersections the same size (as with derangements). Without symmetry and with many sets, the formula is exponential work, and other methods (direct construction, complement, or a program) are better. Also check that the intersections really are what you think: multiples of 4 and of 6 intersect in the multiples of 12, not 24.

24.9.5. Where you'd use this

Everyday: who takes which class. A school has 120 students; 50 take French, 40 Spanish and 30 German; 15 take French and Spanish, 10 French and German, 8 Spanish and German, and 3 take all three. Then $50 + 40 + 30 - 15 - 10 - 8 + 3 = 90$ take at least one language, so 30 take none. Checking region by region (28 French only, 20 Spanish only, 15 German only, 12, 7 and 5 in exactly two, 3 in all) gives the same 90.

Security: password policies. How many 8-character passwords over the 62 letters and digits contain at least one digit, one lowercase and one uppercase letter? Count the complement with inclusion–exclusion. Passwords missing digits number 52^8; missing lowercase, 36^8; missing uppercase, 36^8. Missing two classes: $26^8 + 26^8 + 10^8$. Missing all three: 0. So the policy allows *Try it:* Password policy by inclusion–exclusion .

$$62^8 - (52^8 + 2 \cdot 36^8) + (2 \cdot 26^8 + 10^8) = 159\,655\,911\,367\,680 \approx 1.6 \times 10^{14},$$

about 73% of the 2.18×10^{14} unrestricted strings. Composition rules **shrink** the keyspace slightly; it is length that grows it.

Science: overlapping risk factors. Epidemiologists use the same bookkeeping on patient records: the number with **any** of several conditions is the sum of the counts, minus the pairwise overlaps, plus the triple overlap. Forgetting the overlaps overstates prevalence, which is exactly the double counting the formula corrects.

US curriculum

The two-set version, $P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$, is the **addition rule** of probability in **Algebra II**, **Geometry** (Common Core S-CP.7) and **AP Statistics**. The general principle is a college **Discrete Mathematics** topic.

24.10. The pigeonhole principle

24.10.1. The problem

Some counting questions ask not "how many?" but "must something happen?" In a drawer of black, blue and grey socks, how many must you pull out in the dark to be sure of a matching pair? In any group of 367 people, must two share a birthday?

24.10.2. The principle

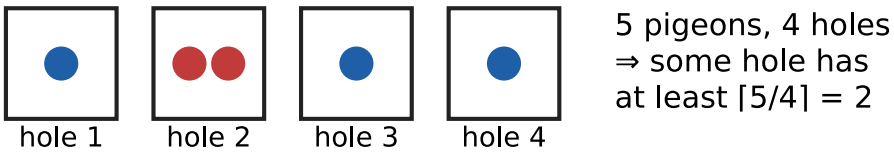


Figure 27. Five pigeons, four holes: some hole must hold at least two.

If more than k objects are placed into k boxes, some box contains at least two objects.

Proof. If every box held at most one object, there would be at most k objects. ■ (A one-line proof by contradiction; see Chapter 22.)

Generalised form. If N objects go into k boxes, some box contains at least $\lceil N/k \rceil$ objects. (Otherwise every box holds at most $\lceil N/k \rceil - 1 < N/k$ objects, for fewer than $k \cdot N/k = N$ in total.)



Ceiling and floor

The **ceiling** $\lceil x \rceil$ is the smallest integer $\geq x$: $\lceil 2.1 \rceil = 3$, $\lceil 5 \rceil = 5$. The **floor** $\lfloor x \rfloor$ is the largest integer $\leq x$: $\lfloor 2.9 \rfloor = 2$. In code, `math.ceil` and `math.floor`; for positive integers, $\lceil N/k \rceil$ is `(N + k - 1) // k` and $\lfloor N/k \rfloor$ is `N // k`.

The principle is almost embarrassingly obvious. Its power lies in choosing the pigeons and holes cleverly.

24.10.3. Applications, from easy to surprising

- **Socks.** Three colours are three holes. Four socks guarantee a matching pair; three do not (one of each colour).
- **Birthdays.** There are 366 possible birthdays (including 29 February). So 367 people guarantee a shared birthday. (The *probability* of a shared birthday is high much earlier, at just 23 people; see Section 24.13.)
- **Hair.** Greater London has about 9 million people. Nobody has more than about 200,000 hairs on their head. So the "holes" are the hair counts 0 to 200,000 (200,001 holes), and some hole contains at least $\lceil 9\,000\,000 / 200\,001 \rceil = 45$ people. At least 45 Londoners have *exactly* the same number of hairs, and you can say so with certainty without counting anyone's hair.
- **Handshakes.** At any party with $n \geq 2$ people, two people shook the same number of hands. Each person shook between 0 and $n - 1$ hands, which looks like n holes for n people. But 0 and $n - 1$ cannot both occur (if someone shook everyone's hand, nobody shook no hands). So only $n - 1$ holes are actually available for n people. You'll see this again in graph language in Chapter 25.
- **Numbers.** Choose any 6 numbers from 1 to 10. Two of them are consecutive. Pair up the holes $\{1,2\}, \{3,4\}, \{5,6\}, \{7,8\}, \{9,10\}$: five holes, six numbers, so some pair is fully chosen. More generally, any $n + 1$ numbers from $1, \ldots, 2n$ include two consecutive ones (and therefore two that share no common factor).

- **Points in a square.** Place 5 points in a square of side 2. Some two are within distance $\sqrt{2}$ of each other. Divide the square into four unit squares (holes); two points share a unit square, and the farthest apart two points in a unit square can be is its diagonal, $\sqrt{2}$.

Example 26. Procedure — applying the pigeonhole principle

Input: a claim that "some two objects must share a property" (or "some group of at least m must").

Steps:

1. Identify the **pigeons**: the objects you are given (people, numbers, points, files).
2. Design the **holes**: categories such that two pigeons in the same hole have the property you want. This is the creative step.
3. Count: if pigeons $N >$ holes k, some hole has two; in general some hole has $\lceil N/k \rceil$.
4. If the count falls short, look for holes that are impossible to fill simultaneously (as with handshakes) or refine the holes.

Output: a guarantee of existence, usually with no way of telling *which* hole.



Pigeonhole is non-constructive

The principle proves that a collision *exists*; it does not tell you where. That's both its strength (you get a guarantee without searching) and a limitation (if you need the actual pair, you still need an algorithm, such as sorting or hashing to find the duplicate). Also, the bound is only a guarantee for the worst case: with 4 socks you're *sure* of a pair, but you will usually get one sooner.

Computer-science connection: why no compressor can shrink everything

A **lossless** compressor must map different inputs to different outputs, or decompression couldn't recover the original. Suppose it claims to make *every* file of exactly n bits shorter. There are 2^n such files (pigeons). The shorter files, of lengths $0, 1, \ldots, n-1$, number

$$2^0 + 2^1 + \cdots + 2^{n-1} = 2^n - 1$$

(holes). Since $2^n > 2^n - 1$, two inputs must share an output, and the compressor cannot be lossless. So every lossless compressor makes *some* inputs longer (or at best the same length). Compression works in practice because real files (text, images, logs) are a tiny, highly patterned subset of all possible bit strings, and compressors are tuned to shrink that subset at the expense of random-looking data. Try zipping an already-compressed or encrypted file: it typically grows by a few bytes.

The same argument says every hash function from long inputs to b-bit digests has collisions: there are more inputs than the 2^b possible digests. Cryptographic hashes do not avoid collisions; they make them infeasible to *find*.



Where did the pigeons come from?

The principle is often credited to Peter Gustav Lejeune Dirichlet, who used it in 1834 in number theory under the name *Schubfachprinzip* ("drawer principle"); French and Russian mathematicians still call it the drawer or box principle. The pigeons are a 20th-century English invention, possibly after pigeonholes in a writing desk rather than birds.

US curriculum

The pigeonhole principle is not part of the standard US high-school sequence. Students meet it in math competitions (MATHCOUNTS, AMC, AIME) and in a college **Discrete Mathematics** course, typically in the first semester of a CS degree.

24.11. Choosing a counting method

24.11.1. The two questions

Most "choose r things from n types" problems are settled by two yes/no questions:

1. **Does order matter?** Would rearranging the selection give a different outcome?
2. **Is repetition allowed?** Can the same item (or type) be chosen more than once?

	Order matters (arrangements, sequences)	Order doesn't matter (selections, sets)
Repetition allowed	n^r PINs, passwords, bit strings `itertools.product(items, repeat=r)`	$\binom{n+r-1}{r}$ doughnut boxes, scoops, multisets `itertools.combinations_with_replacement`
No repetition	$P(n,r) = \frac{n!}{(n-r)!}$ podiums, seatings, ordered rankings `itertools.permutations(items, r)`	$\binom{n}{r} = \frac{n!}{r!(n-r)!}$ committees, lottery tickets, poker hands `itertools.combinations(items, r)`

Notice how the table is held together by the earlier ideas. Moving down the left column, n^r becomes $n(n-1)\cdots$ because each stage loses an option. Moving right along the bottom row, you divide by the $r!$ orderings. The top-right cell is stars and bars.

Example 27. Procedure — choosing a counting method

Input: a counting question.

Steps:

1. **Is it an existence question** ("must two …?", "is there always …?")? Try the pigeonhole principle.
2. **Does it say "at least one", "not all" or "some repeat"?** Count the complement and subtract from the total.
3. **Is it a union of overlapping conditions** ("divisible by 2 or 3", "contains A or B")? Use inclusion–exclusion.
4. **Does it fit the table?** Decide whether order matters and whether repetition is allowed, and use that cell. For arrangements of items with repeated types (MISSISSIPPI), use the multinomial coefficient. For identical objects into distinct boxes, use stars and bars.
5. **Otherwise,** break the construction into stages and cases (product and sum rules), checking that each outcome is built exactly once. If a natural construction builds each outcome exactly k times, divide by k.
6. **Check** the answer: by a second method, by a small case you can list by hand, or by a brute-force program.

Output: a count with a method you can justify.

24.11.2. Worked example: one scenario, four answers

From the digits 1–5, form a selection of 3.

- Ordered, repetition allowed (3-digit codes): $5^3 = 125$.
- Ordered, no repetition (3-digit codes with distinct digits): $5 \cdot 4 \cdot 3 = 60$.
- Unordered, no repetition (3-element subsets): $\binom{5}{3} = 10$.
- Unordered, repetition allowed (multisets such as $\{1, 1, 4\}$): $\binom{5+3-1}{3} = \binom{7}{3} = 35$.

As a sanity check, the counts must satisfy $10 \le 35 \le 125$ and $10 \le 60 \le 125$, since allowing more freedom never reduces the count.



Sanity checks that catch most mistakes

- **Try a tiny case.** Shrink the problem until you can list everything ($n = 3, r = 2$) and compare with your formula.
- **Compare with neighbouring cells.** An unordered count can never exceed the ordered one; a count with restrictions can never exceed the unrestricted one.
- **Check divisibility.** If you divided to remove over-counting and the answer isn't an integer, your over-count factor was wrong.
- **Brute-force it.** For small sizes, `itertools` plus `sum(1 for … if …)` is a two-line check that you should use freely.

24.12. Applications: lotteries, passwords and poker

24.12.1. Lotteries

A 6/49 lottery draws 6 distinct numbers from 1–49; order doesn't matter. The number of tickets is

$$\binom{49}{6} = \frac{49 \cdot 48 \cdot 47 \cdot 46 \cdot 45 \cdot 44}{6!} = \frac{10\,068\,347\,520}{720} = 13\,983\,816.$$

One ticket wins the jackpot with probability about 1 in 14 million. US Powerball draws 5 white balls from 69 and 1 red ball from 26: $\binom{69}{5} \times 26 = 11\,238\,513 \times 26 = 292\,201\,338$ combinations, so the jackpot odds are 1 in about 292 million. A second prize such as "match exactly 5 of 6" in the 6/49 game is a product of combinations: choose 5 of the 6 winning numbers and 1 of the 43 losing numbers, $\binom{6}{5}\binom{43}{1} = 6 \times 43 = 258$ tickets.

24.12.2. Passwords

A password of length L over an alphabet of a symbols has a^L possibilities (ordered, repetition allowed). Length matters far more than alphabet size, because length is the exponent:

- 8 lowercase letters: $26^8 \approx 2.09 \times 10^{11}$.
- 8 characters from letters and digits: $62^8 \approx 2.18 \times 10^{14}$.

- 12 characters from letters and digits: $62^{12} \approx 3.23 \times 10^{21}$.

Going from 26 to 62 symbols at length 8 multiplies the space by about 1000. Going from length 8 to 12 at 62 symbols multiplies it by $62^4 \approx 1.5 \times 10^7$. The CS section below turns these counts into cracking times.

24.12.3. Poker hands

A five-card hand from a 52-card deck is an unordered selection without repetition: $\binom{52}{5} = 2\,598\,960$ hands. Counting particular hands is a perfect exercise in staged construction. The rule: *build the hand by deciding the ranks, then the suits, making sure each hand is built once.*

Four of a kind (e.g. four 7s and a king). Choose the rank of the four: 13. All four suits are forced: $\binom{4}{4} = 1$. Choose the fifth card from the 48 others: 48.

$$13 \times 1 \times 48 = 624.$$

Full house (three of one rank, two of another). Choose the rank of the triple, then its suits; choose a *different* rank for the pair, then its suits:

$$\begin{aligned} 13 \times \binom{4}{3} \times 12 \times \binom{4}{2} &= 13 \times 4 \times 12 \times 6 \\ &= 3\,744. \end{aligned}$$

Here the two ranks are chosen in order (13 then 12) because their roles differ: "three kings and two 7s" is a different hand from "three 7s and two kings".

Two pair (two of one rank, two of another, and a fifth card of a third rank). Now the two pair ranks play the *same* role, so choose them as a set:

$$\begin{aligned} \binom{13}{2} \times \binom{4}{2}^2 \times 44 &= 78 \times 36 \times 44 \\ &= 123\,552. \end{aligned}$$

(The fifth card can be any of the $52 - 8 = 44$ cards not of either pair rank.) If you'd used 13×12 for the pair ranks, you'd have built each hand twice and got 247,104.

One pair. Choose the pair rank and suits, then 3 different other ranks (as a set) and a suit for each:

$$13 \times \binom{4}{2} \times \binom{12}{3} \times 4^3 = 13 \times 6 \times 220 \times 64 = 1\,098\,240.$$

Flush (all five the same suit, excluding straight flushes): $4 \times \binom{13}{5} - 40 = 4 \times 1287 - 40 = 5108$, where 40 is the number of straight flushes (4 suits × 10 possible starting ranks, ace low to ten high).

Dividing by 2,598,960 gives the probabilities: a full house is about 0.144 %, four of a kind about 0.024 %. Rarer hands beat commoner ones, which is exactly the ranking used in the game. That is not a coincidence: the ranking was settled by counting.



Poker-hand counter

Deal random five-card hands and tally the hand types over thousands of deals, side by side with the exact counts above. A "build a hand" mode walks through the stages (rank choices, suit choices) and lets the reader deliberately make the 13×12 two-pair mistake and see the over-count.

Try it: Poker-hand counter and two-pair mistake · Exact poker hand counts

24.13. Counting in computer science

24.13.1. Keyspaces and brute force

An attacker who has stolen a database of password hashes can guess offline as fast as their hardware allows. For a fast, unsalted hash, a single modern GPU rig can try on the order of 10^{10} guesses per second. The expected time to search the whole keyspace is (keyspace) / (rate); on average the right password is found halfway through. *Try it:* Password keyspace calculator .

PYTHON

```python
def human(seconds):
    for unit, size in [("years", 365.25 * 86400), ("days", 86400),
                       ("hours", 3600), ("minutes", 60)]:
        if seconds >= size:
            v = seconds / size
            return f"{v:,.1f} {unit}" if v < 1e6 else f"{v:.2e} {unit}"
    return f"{seconds:.2f} seconds"

RATE = 1e10   # guesses per second: a fast offline GPU rig on a weak hash

for name, alphabet, length in [("8 lowercase", 26, 8),
                               ("8 letters+digits", 62, 8),
                               ("12 letters+digits", 62, 12),
                               ("16 printable", 95, 16)]:
    space = alphabet ** length               # product rule
    print(f"{name:18} {space:10.3e} keys   {human(space / RATE)}")
```

```
8 lowercase        2.088e+11 keys   20.88 seconds
8 letters+digits   2.183e+14 keys   6.1 hours
12 letters+digits  3.226e+21 keys   10,223.4 years
16 printable       4.401e+31 keys   1.39e+14 years
```

Try it: Time to crack 12 characters .

Two lessons follow. First, exponential growth in length beats everything: every extra character multiplies the work by the alphabet size. Second, these are *upper* bounds on safety: human-chosen passwords are not uniformly random, so attackers try dictionaries and patterns first, and the effective keyspace is far smaller. That's why systems use deliberately slow, salted hash functions (bcrypt, scrypt, Argon2) to cut the guess rate by many orders of magnitude, and why a passphrase of 5 random dictionary words (from a list of 7776, as in Diceware) gives $7776^5 \approx 2.8 \times 10^{19}$ possibilities while staying memorable.

Security engineers express keyspaces in **bits**: a keyspace of size N has $\log_2 N$ bits of entropy (if every key is equally likely). $62^{12} \approx 2^{71.5}$, so a random 12-character alphanumeric password is a 71-bit secret. A 128-bit AES key gives $2^{128} \approx 3.4 \times 10^{38}$ keys.

24.13.2. Subsets, 2^n and exponential algorithms

A set of n elements has 2^n subsets: for each element, "in" or "out". There's a perfect correspondence between subsets and n-bit binary numbers (bit i is 1 if element i is in), which gives the simplest way to enumerate them:

```python
items = ["a", "b", "c"]
n = len(items)
for mask in range(2 ** n):                       # 0 .. 2^n - 1
    subset = [items[i] for i in range(n) if mask >> i & 1]
    print(f"{mask:03b}", subset)
```

```
000 []
001 ['a']
010 ['b']
011 ['a', 'b']
100 ['c']
101 ['a', 'c']
110 ['b', 'c']
111 ['a', 'b', 'c']
```

Any algorithm that tries every subset (the brute-force approach to subset-sum, knapsack, feature selection or set cover) does at least 2^n units of work, and any algorithm that tries every ordering (brute-force travelling salesman) does $n!$.

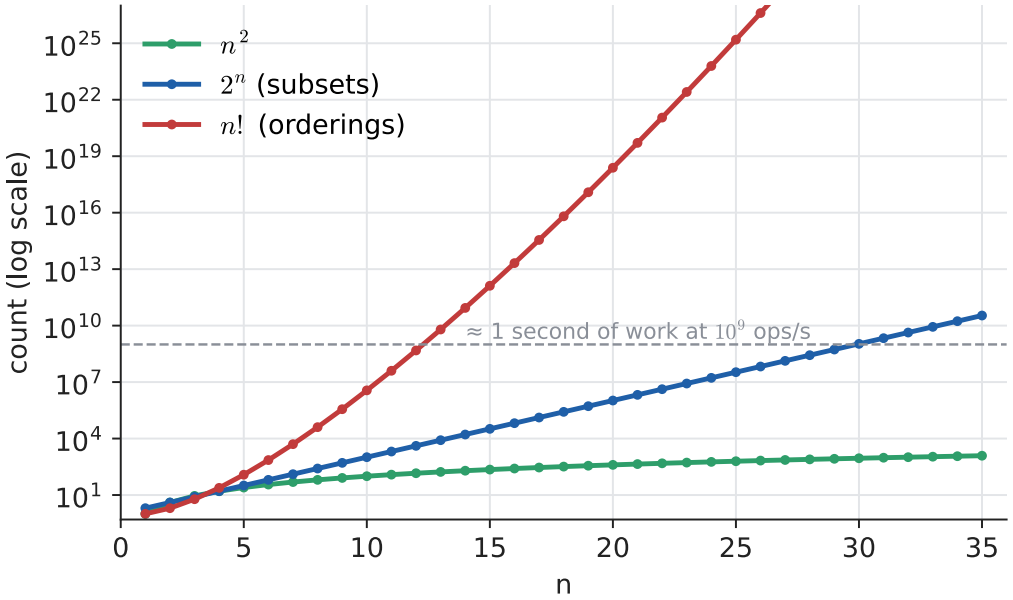


Figure 28. Polynomial, exponential and factorial growth on a logarithmic scale. At 10^9 operations per second, 2^n passes one second near $n = 30$ and $n!$ near $n = 13$.

At a billion operations a second, $2^{30} \approx 10^9$ subsets take a second, 2^{40} take about 18 minutes, 2^{50} take 13 days, and 2^{60} take 36 years. $20! \approx 2.4 \times 10^{18}$ orderings take 77 years. Faster hardware barely helps: a computer 1000 times faster raises the feasible n for a 2^n algorithm by only about 10. This is why the distinction between polynomial and exponential running time is the central dividing line of complexity theory, and why, for problems like these, practical algorithms use pruning, dynamic programming, or approximation instead of enumeration.

24.13.3. Generating permutations

Sometimes you really do need to visit every arrangement: testing all orderings of a small set of operations for race conditions, solving a puzzle exhaustively, or generating test cases. Two standard algorithms do this.

Next permutation (lexicographic order). Given an arrangement, produce the next one in dictionary order. This is what C++'s `std::next_permutation` does, and it handles repeated items gracefully (it produces each distinct arrangement once).

The idea: to get the next larger arrangement, change as far to the right as possible. Find the longest suffix that is already in descending order (it is the largest possible arrangement of those items, so it can't be increased by itself). The element just before it, the *pivot*, must increase, by the smallest possible amount, and then the suffix must be reset to its smallest arrangement (ascending).

Example 28. Procedure — next permutation in lexicographic order

Input: a list $a_0, a_1, \ldots, a_{n-1}$.

Steps:

1. Find the largest index i with $a_i < a_{i+1}$. If there is none, the list is in descending order: it is the last permutation. Stop (optionally reverse it to wrap around to the first).
2. Find the largest index $j > i$ with $a_j > a_i$. (It exists, since $a_{i+1} > a_i$.)
3. Swap a_i and a_j.
4. Reverse the suffix $a_{i+1}, \ldots, a_{n-1}$ (it was descending; now it's ascending).

Output: the next permutation in lexicographic order. Starting from the sorted list and repeating until step 1 stops visits every distinct permutation exactly once.

Worked trace on $(1, 3, 5, 4, 2)$: *Try it:* Step through next permutation .

Step	Result
1. Pivot	Scanning from the right: $4 > 2$, $5 > 4$, but $3 < 5$. Descending suffix is $(5, 4, 2)$; pivot $a_1 = 3$.
2. Successor	Rightmost element of the suffix greater than 3: $a_3 = 4$.
3. Swap	$(1, 4, 5, 3, 2)$
4. Reverse suffix	$(1, 4, 2, 3, 5)$

Check: among arrangements starting $1, 3, \ldots$, $(1, 3, 5, 4, 2)$ is the largest; the next must start $1, 4$, followed by the remaining $\{2, 3, 5\}$ in ascending order. That's $(1, 4, 2, 3, 5)$.

PYTHON

```python
def next_permutation(a):
    """Rearrange list a into the next permutation in lexicographic order.
    Return False (and leave a sorted ascending) if a was the last one."""
    # 1. find the rightmost i with a[i] < a[i+1]  (the "pivot")
    i = len(a) - 2
    while i >= 0 and a[i] >= a[i + 1]:
        i -= 1
    if i < 0:                       # whole list is descending: last permutation
        a.reverse()
        return False
    # 2. find the rightmost j with a[j] > a[i]
    j = len(a) - 1
    while a[j] <= a[i]:
        j -= 1
    # 3. swap pivot with its successor
    a[i], a[j] = a[j], a[i]
    # 4. reverse the suffix so it is as small as possible
    a[i + 1:] = reversed(a[i + 1:])
    return True

a = [1, 2, 3]
out = [a.copy()]
while next_permutation(a):
    out.append(a.copy())
print(out)

b = [1, 3, 5, 4, 2]
next_permutation(b)
print(b)

c = [1, 1, 2]          # repeated items: only distinct arrangements appear
out = [c.copy()]
while next_permutation(c):
    out.append(c.copy())
print(out)
```

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
[1, 4, 2, 3, 5]
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

Try it: All permutations in order ; Permutations with repeated items .

The last line lists $3!/2! = 3$ arrangements, the multinomial count for $\{1, 1, 2\}$, not $3! = 6$.

Backtracking. The other approach builds arrangements position by position, mirroring the product rule's stages: choose an unused item for the next position, recurse, then undo the choice and try the next item. It is the template for a huge family of search algorithms (sudoku, N-queens, constraint solvers), because you can *prune*: abandon a partial arrangement as soon as it violates a constraint, skipping every completion of it at once. *Try it:* Backtracking with pruning (4 queens) .

Example 29. Procedure — generate permutations by backtracking

> **Input:** a list of n distinct items.
>
> **Steps:**
>
> 1. Keep a partial arrangement (initially empty) and a record of which items are used.
> 2. If the partial arrangement has length n, output it and return.
> 3. Otherwise, for each unused item x: mark x used and append it (**choose**); recurse from step 2 (**explore**); remove x and mark it unused (**un-choose**).
>
> **Output:** all $n!$ permutations, each exactly once.

Try it: Watch the recursion tree .

PYTHON

```
def permutations_bt(items):
    """Generate all orderings of items by backtracking."""
    result, current = [], []
    used = [False] * len(items)

    def extend():
        if len(current) == len(items):        # a complete ordering
            result.append(tuple(current))
            return
        for i, x in enumerate(items):
            if not used[i]:
                used[i] = True                 # choose
                current.append(x)
                extend()                       # explore
                current.pop()                  # un-choose
                used[i] = False

    extend()
    return result

print(permutations_bt("abc"))
print(len(permutations_bt("abcde")))
```

```
[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'), ('b', 'c', 'a'), ('c', 'a', 'b'), ('c',
'b', 'a')]
120
```

Choosing between them. In production Python, just use `itertools.permutations`; it is implemented in C. Use next-permutation when you need to step through arrangements one at a time with constant extra memory, or when items repeat and you want each distinct arrangement once. Use backtracking when you want to prune, which is almost always the case in real search problems.

The recursion tree of the backtracking algorithm *is* the product-rule tree diagram from Section 24.2: n branches at the root, $n - 1$ at the next level, and so on, with $n!$ leaves.

24.13.4. The birthday paradox and hash collisions

How many people must be in a room for there to be a better than even chance that two share a birthday? The pigeonhole principle says 367 people make it *certain*; the surprise is how few make it *likely*.

Count the complement. Assume 365 equally likely birthdays (ignoring 29 February). The number of ways n people can have *all different* birthdays is $P(365, n)$ (ordered, no repetition) out of 365^n total assignments, so

$$P(\text{all distinct}) = \frac{365 \cdot 364 \cdots (365 - n + 1)}{365^n} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{365}\right),$$

$$P(\text{some shared birthday}) = 1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{365}\right).$$

(The symbol $\prod$ is the product version of $\sum$: multiply the terms instead of adding them.) The probability language is developed properly in Chapter 29; here you need only "favourable over total", with every assignment equally likely.

```python
import math

def p_collision(n, days=365):
    """Probability that at least two of n people share a birthday."""
    p_distinct = 1.0
    for i in range(n):
        p_distinct *= (days - i) / days    # person i+1 avoids the i taken days
    return 1 - p_distinct

for n in (10, 23, 30, 50, 70):
    print(n, round(p_collision(n), 4))

# Hash collisions: with a b-bit hash, about how many items for a 50% chance?
for bits in (32, 64, 128):
    print(bits, f"{math.sqrt(2 * math.log(2) * 2**bits):.3e}")
```

Try it: Simulate the birthday problem .

```
10 0.1169
23 0.5073
30 0.7063
50 0.9704
70 0.9992
32 7.716e+04
64 5.057e+09
128 2.172e+19
```

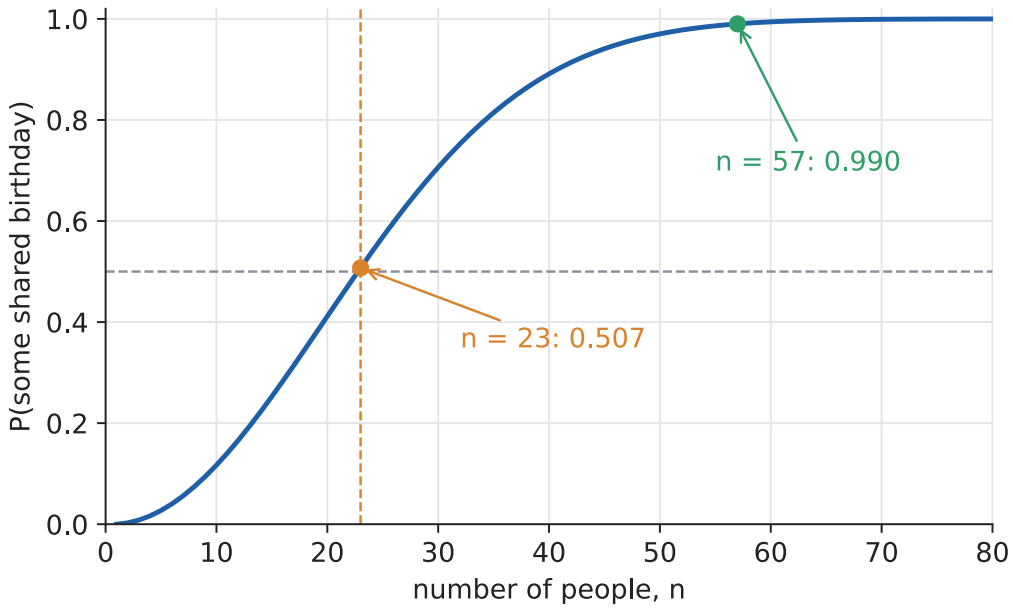


Figure 29. The birthday problem: the chance of a shared birthday passes 50 % at just 23 people and 99 % at 57.

Why so few? Because what matters is the number of *pairs*, not the number of people. 23 people form $\binom{23}{2} = 253$ pairs, and each pair has a 1-in-365 chance of matching. Using $1 - x \approx e^{-x}$ for small x,

$$P(\text{all distinct}) \approx \prod_{i=0}^{n-1} e^{-i/N} = e^{-\binom{n}{2}/N} \approx e^{-n^2/(2N)},$$

where N is the number of possible values. Setting this to $\frac{1}{2}$ and solving gives $n \approx \sqrt{2 \ln 2 \cdot N} \approx 1.18\sqrt{N}$. For $N = 365$, that's about 22.5. The rule of thumb: **collisions become likely after about $\sqrt{N}$ random items.**

Hash collisions. Replace birthdays with hash values. A hash table, a content-addressed store, or a database of short IDs with N possible values will see its first collision after about $\sqrt{N}$ items, not N. A 32-bit hash ($N \approx 4.3 \times 10^9$) has a 50 % chance of a collision after only about 77,000 items, which is why 32-bit checksums are useless as unique identifiers for large collections. A 64-bit hash reaches 50 % at about 5 billion items. Cryptographic hashes use 256 bits so that the "birthday bound" 2^{128} is far beyond any attacker; a hash with b-bit output offers only about $b/2$ bits of collision resistance. The same arithmetic tells you how long random UUIDs (122 random bits) can be generated before a duplicate is plausible: around $2^{61} \approx 2.3 \times 10^{18}$.

Birthday simulator

Add people one at a time to a virtual room, each with a random birthday, and stop at the first collision; repeat thousands of times and plot the histogram of stopping points against the exact curve. Then switch "days" to 2^{16} or 2^{24} hash buckets and see the $\sqrt{N}$ rule hold. A password-keyspace calculator sits alongside: choose alphabet, length and guess rate, and watch the time-to-crack move by orders of magnitude as length grows.

Try it: 32-bit hash collisions · Birthday simulator and √N rule · Birthday bound for hashes

Computer-science connection: where counting shows up

- **Algorithm analysis.** Counting loop iterations, comparisons or recursive calls is how running time is derived. The number of pairs $\binom{n}{2}$ is why naive all-pairs algorithms (duplicate detection, collision detection in games, nearest-neighbour search) are quadratic.
- **Testing.** Exhaustive testing of k boolean flags means 2^k configurations. *Pairwise* (combinatorial) testing instead covers every pair of settings with far fewer test cases, because covering pairs is a much smaller counting target than covering all combinations.
- **Probability and randomised algorithms.** Most discrete probability is counting in disguise: favourable outcomes over total outcomes (Chapter 29).
- **Information theory.** Specifying one item out of N equally likely possibilities needs $\log_2 N$ bits. That's the link between counting, entropy and compression.

24.14. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Counting, visually
- Sets, relations and quantifiers
- Probability: dice, cards, Bayes

24.15. Exercises

1. **(Recognition)** For each scenario, say whether order matters, whether repetition is allowed, and which formula applies: (a) a 5-character licence key using A–Z and 0–9; (b) choosing 4 toppings from 12 for a pizza, no double toppings; (c) assigning first, second and third prizes among 20 entrants; (d) buying 10 bagels from 6 varieties.
2. **(Recognition)** Compute $\binom{7}{3}$, $\binom{7}{4}$, $P(7,3)$ and $\binom{20}{18}$, and explain why the first two are equal.

3. **(Application)** How many 4-digit PINs have digits that are all different and in strictly increasing order (like 1479)?
4. **(Application)** Expand $(a-2b)^4$ using the binomial theorem, and check your answer by substituting $a=b=1$.
5. **(Application)** How many distinct arrangements are there of the letters of BANANA? How many of them begin with B?
6. **(Application)** How many non-negative integer solutions has $x_1+x_2+x_3+x_4=10$? How many with every $x_i \geq 2$?
7. **(Application)** How many integers from 1 to 600 are divisible by none of 3, 4 and 5?
8. **(Problem solving)** A 5-card poker hand is a *straight* if its ranks are consecutive (ace may be low, A-2-3-4-5, or high, 10-J-Q-K-A; there are 10 possible rank sequences). Count straights, including straight flushes, and then excluding them.
9. **(Problem solving)** Prove that among any 11 integers, two have a difference divisible by 10.
10. **(Problem solving)** Give a counting proof that $\binom{2n}{2} = 2\binom{n}{2} + n^2$. (Hint: a group of n women and n men; choose 2 people.)
11. **(Problem solving, CS)** A system issues random 6-character IDs from the 36 symbols A–Z, 0–9. Roughly how many IDs can it issue before a collision becomes more likely than not? What about 10-character IDs?

▼ **Answers**

1. (a) Order matters, repetition allowed: $36^5 = 60\,466\,176$. (b) Order doesn't matter, no repetition: $\binom{12}{4} = 495$. (c) Order matters, no repetition: $P(20,3) = 20\cdot 19\cdot 18 = 6840$. (d) Order doesn't matter, repetition allowed (stars and bars): $\binom{10+6-1}{10} = \binom{15}{5} = 3003$.
2. $\binom{7}{3} = \frac{7\cdot 6\cdot 5}{6} = 35$; $\binom{7}{4} = 35$ by symmetry (choosing 3 to include is choosing 4 to exclude); $P(7,3) = 7\cdot 6\cdot 5 = 210 = 35\times 3!$; $\binom{20}{18} = \binom{20}{2} = 190$.
3. Any set of 4 distinct digits can be arranged in increasing order in exactly one way, so count sets: $\binom{10}{4} = 210$.
4. With $x=a$, $y=-2b$, row 4 = 1, 4, 6, 4, 1: $a^4 + 4a^3(-2b) + 6a^2(4b^2) + 4a(-8b^3) + 16b^4 = a^4 - 8a^3b + 24a^2b^2 - 32ab^3 + 16b^4$. Check: $(1-2)^4 = 1$ and $1-8+24-32+16 = 1$. ✓
5. BANANA: 6 letters with A×3, N×2, B×1: $\frac{6!}{3!\,2!\,1!} = \frac{720}{12} = 60$. Beginning with B: arrange AAANN in the other 5 positions: $\frac{5!}{3!\,2!} = 10$.
6. Free: $\binom{10+4-1}{3} = \binom{13}{3} = 286$. With every $x_i \geq 2$: pre-load 2 each (8 used), distribute the remaining 2: $\binom{2+3}{3} = \binom{5}{3} = 10$.
7. Multiples: $|A_3| = 200$, $|A_4| = 150$, $|A_5| = 120$; sum 470. Pairs: lcm(3,4)=12 → 50, lcm(3,5)=15 → 40, lcm(4,5)=20 → 30; sum 120. Triple: lcm = 60 → 10. Union: $470 - 120 + 10 = 360$. None: $600 - 360 = 240$.

8. Choose the rank sequence (10 ways), then a suit for each of 5 cards ($4^5 = 1024$): $10 \times 1024 = 10\,240$ straights including straight flushes. Straight flushes: 10 sequences × 4 suits = 40. Excluding them: $10\,240 - 40 = 10\,200$.

9. The holes are the 10 possible last digits (remainders 0–9 on division by 10); for negative integers use the remainder in $\{0, \ldots, 9\}$. Eleven integers, ten holes, so two share a remainder r: $a = 10p + r$, $b = 10q + r$, and $a - b = 10(p - q)$ is divisible by 10.

10. The left side counts pairs from all $2n$ people. Split by case: both women $\binom{n}{2}$, both men $\binom{n}{2}$, one of each $n \times n$. The cases are disjoint and cover everything, so $\binom{2n}{2} = 2\binom{n}{2} + n^2$. (Algebra check: $\frac{2n(2n-1)}{2} = 2n^2 - n$ and $n(n-1) + n^2 = 2n^2 - n$. ✓)

11. $N = 36^6 \approx 2.18 \times 10^9$; 50 % collision after about $1.18\sqrt{N} \approx 1.18 \times 46\,656 \approx 55\,000$ IDs. Only fifty-five thousand, despite two billion possible IDs. With 10 characters, $N = 36^{10} \approx 3.66 \times 10^{15}$, $\sqrt{N} = 36^5 \approx 6.05 \times 10^7$, so about 71 million IDs.

24.16. Mastery check

1. In how many ways can 6 books be arranged on a shelf if two particular books must *not* be next to each other?
2. A pizza place offers 10 toppings. How many pizzas have exactly 3 different toppings? How many have any number of different toppings (including none)?
3. Find the coefficient of x^2 in $(3 - x)^6$.
4. How many ways can 15 identical sweets be shared among 4 children if every child gets at least 2?
5. A survey of 100 programmers: 60 use Python, 45 use Java, 25 use both. How many use neither?
6. Show that in any set of 13 people, at least two were born in the same month. How many people guarantee that *three* share a birth month?
7. A 4-character code uses digits 0–9 and must contain at least one 7. How many codes are there?
8. Why can no lossless compression algorithm shrink every 1 MB file?

Decision rule: All correct without hesitation → move on to Chapter 25. Stuck on 1–2 → reread the relevant section (Section 24.11 is the usual culprit). Stuck on more → work the chapter exercises first, checking each answer by brute force in Python.

▼ **Mastery check answers**

1. All arrangements: $6! = 720$. Arrangements with the two books together: glue them, $5! \times 2! = 240$. Not together: $720 - 240 = 480$.

2. Exactly 3: $\binom{10}{3} = 120$. Any number: each topping in or out, $2^{10} = 1024$ (the row sum of Pascal's triangle).
3. Term $\binom{6}{k} 3^{6-k}(-x)^k$ with $k = 2$: $15 \times 81 \times x^2 = 1215x^2$. Coefficient 1215.
4. Give each child 2 first (8 sweets); distribute the remaining 7 freely among 4: $\binom{7+3}{3} = \binom{10}{3} = 120$.
5. Inclusion–exclusion: $|P \cup J| = 60 + 45 - 25 = 80$. Neither: $100 - 80 = 20$.
6. 13 people, 12 months (holes): some month has $\lceil 13/12 \rceil = 2$. For three: need N with $\lceil N/12 \rceil \geq 3$, so $N = 25$ (with 24 people it's possible to have exactly 2 per month).
7. Complement: all codes 10^4, codes with no 7 $9^4 = 6561$; with at least one 7: $10\,000 - 6561 = 3439$.
8. Pigeonhole: there are 2^n files of exactly n bits ($n = 8 \times 2^{20}$) but only $2^n - 1$ strictly shorter bit strings. A map from all n-bit files into shorter strings must send two files to the same output, so it can't be inverted. Every lossless compressor makes some files longer.

Chapter 25. Graph Theory

A road map, a social network, the web, a build system, a compiler's view of your program, and the dependency chart at the front of this book are all the same kind of object: things, and connections between them. Graph theory strips away everything else and asks what can be said about connections alone. This chapter starts with an 18th-century puzzle about bridges, builds the vocabulary (vertices, edges, degree, paths, cycles, trees), and then turns to the algorithms that make graphs useful in computing: breadth-first and depth-first search, Dijkstra's shortest paths, topological sorting, minimum spanning trees and colouring. It ends with planar graphs and Euler's formula, and with a glimpse of how Google ranked the web.

Where this chapter sits

You need: sets and relations (Chapter 21), proof by induction and contradiction (Chapter 22), and counting, especially $\binom{n}{2}$ and the pigeonhole principle (Chapter 24).

This chapter unlocks: adjacency matrices and their powers (Volume 2, Chapter 27), PageRank as an eigenvector problem (Volume 2, Chapter 28), random walks and Markov chains (Chapter 29), and almost every data structure and algorithm course you will ever read. It is the mathematical core of networking, compilers, databases, package managers and distributed systems.

25.1. The bridges of Königsberg

25.1.1. The problem

In the 1730s the Prussian city of Königsberg (now Kaliningrad, Russia) sat on both banks of the river Pregel and on an island, Kneiphof, with a fourth piece of land to the east where the river forked. Seven bridges joined the four land masses. The townspeople asked: *can you take a walk that crosses every bridge exactly once?* Nobody could find one, and nobody could prove it impossible.

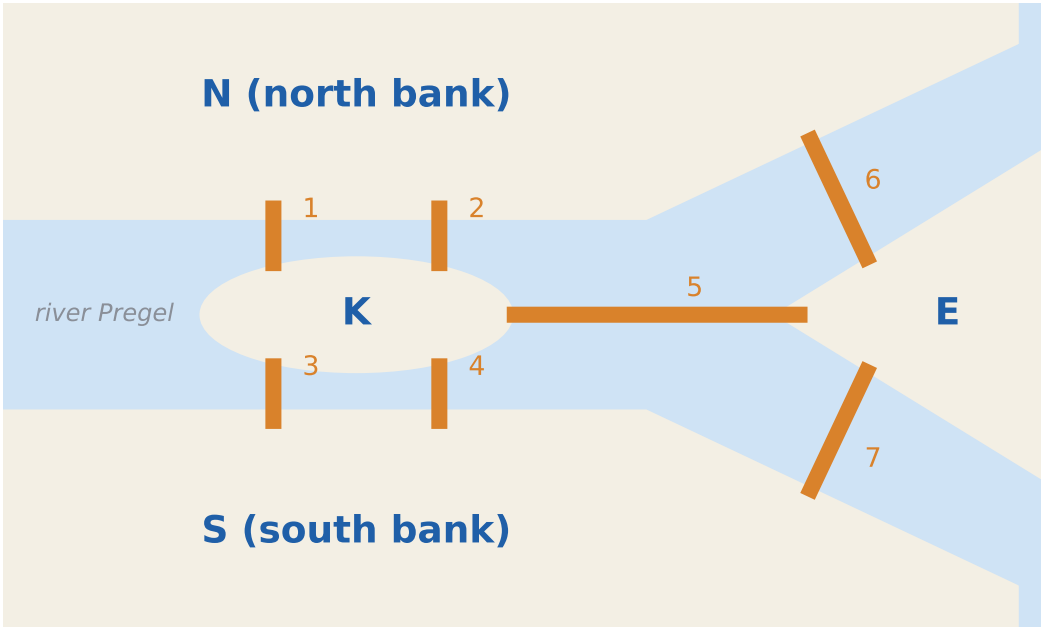


Figure 30. A stylised map of Königsberg: four land masses (N, S, the island K, and E) joined by seven bridges.

25.1.2. Intuition: throw away the geography

In 1736 Leonhard Euler saw that almost nothing on the map matters. The size and shape of the land, the lengths of the bridges, the route you take while on dry land: all irrelevant. Only two things matter: *which land masses exist* and *which pairs each bridge connects*. So shrink each land mass to a dot and draw each bridge as a line between dots:

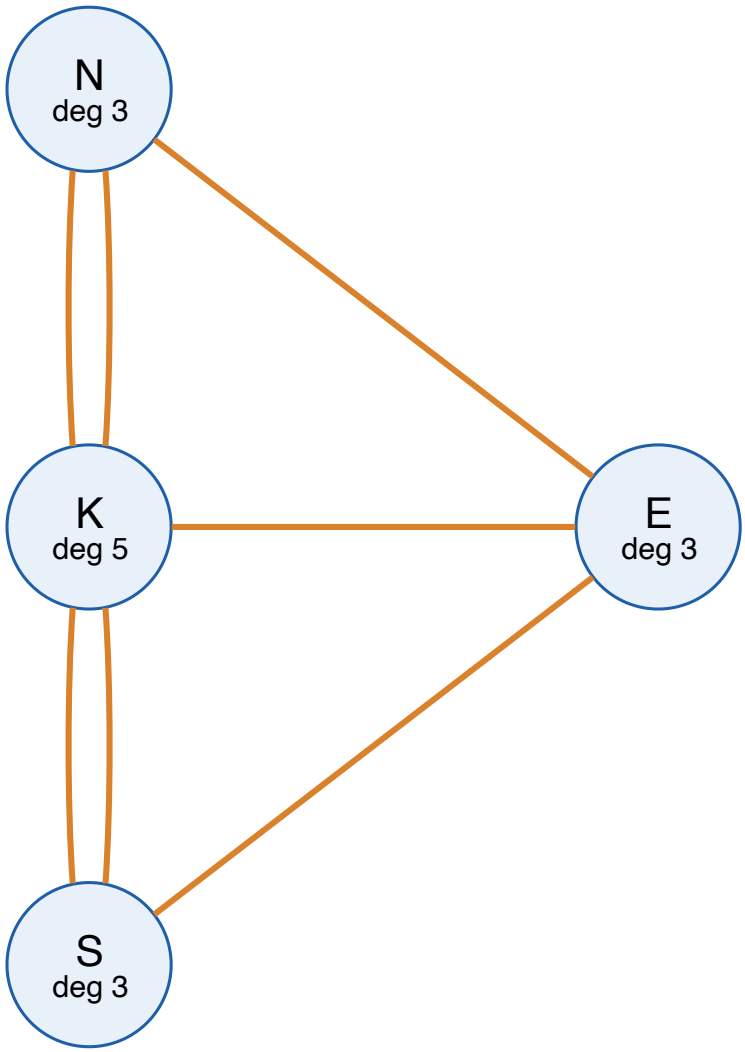


Figure 31. The same problem as a graph: four vertices, seven edges. The number in each vertex is its degree (how many bridges touch it).

Now think about any walk that crosses each bridge once. Every time the walk passes *through* a land mass, it arrives on one bridge and leaves on another, using up bridges in pairs. So every land mass that is not the start or the end of the walk must touch an *even* number of bridges. At most two land masses (the start and the end) can have an odd number. In Königsberg, *all four* have an odd number (5, 3, 3, 3). So no such walk exists. No amount of trying would ever have found one. *Try it:* Königsberg as a graph .

That argument is the birth of graph theory, and it illustrates the subject's power: by finding the right abstraction, a question about a city becomes a question about counting, and the answer comes with a proof. We will make it precise in Section 25.7.



Euler and the "geometry of position"

Euler described the problem as belonging to the *geometria situs* ("geometry of position") that Leibniz had imagined: a geometry where distances don't matter, only arrangement. That idea grew into both graph theory and topology. Several of the original seven bridges were destroyed in the Second World War or demolished afterwards. Today only five bridges join these land masses, and with that layout exactly two land masses have odd degree, so an Euler path (though not a circuit) finally exists.

25.2. Vertices, edges and the kinds of graph

25.2.1. The idea

A graph is the simplest possible model of "things and connections": a set of objects, and a set of pairs of objects that are linked.



Graph, vertex, edge

A **graph** $G = (V, E)$ consists of a set V of **vertices** (also called **nodes**) and a set E of **edges**, each edge joining two vertices. An edge between u and v is written $\{u, v\}$ or simply uv. u and v are its **endpoints**; they are **adjacent** (neighbours), and the edge is **incident** to each of them. The number of vertices $|V|$ is often written n and the number of edges $|E|$ is written m.

The singular of vertices is **vertex**. "Graph" here has nothing to do with the graph of a function (Volume 1, Chapter 11); the word was coined by J. J. Sylvester in 1878 from "chemicograph", a diagram of molecular bonds.



Kinds of graph

- **Undirected graph:** edges have no direction; $\{u, v\}$ is the same as $\{v, u\}$. (Friendship, two-way roads, cables.)
- **Directed graph (digraph):** each edge is an ordered pair (u, v), drawn as an arrow from u (the **tail**) to v (the **head**). (Following on social media, hyperlinks, one-way streets, "depends on".)
- **Weighted graph:** each edge carries a number, its **weight** or **cost**: a distance, a latency, a price, a capacity.
- **Simple graph:** undirected, with no **loops** (edges from a vertex to itself) and no **multiple edges** (two edges joining the same pair). A graph that allows multiple edges, like Königsberg, is a **multigraph**.
- **Complete graph** K_n: a simple graph on n vertices in which every pair is adjacent. It has $\binom{n}{2} = n(n-1)/2$ edges (Chapter 24).

Unless stated otherwise, "graph" in this chapter means a simple undirected graph.

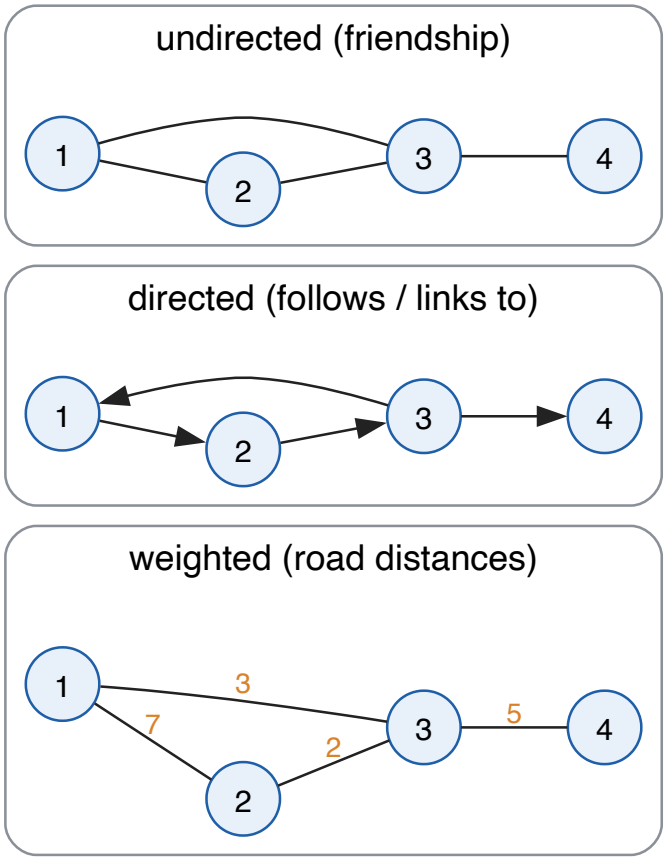


Figure 32. Three kinds of graph. The same four vertices model friendships, one-way links, or roads with distances.

25.2.2. Representing relationships as graphs

Modelling is the first skill. Ask: *what are the things (vertices), and what is the relationship (edges)? Is the relationship symmetric (undirected) or not (directed)? Does it have a size (weighted)?*

- Airports and direct flights: vertices = airports, directed edges = flights, weights = fares or durations.
- A codebase: vertices = modules, directed edge $A \to B$ if A imports B.
- A sudoku: vertices = the 81 cells, edges between cells that may not hold the same digit.
- A chess knight: vertices = squares, edges = legal knight moves. "Can a knight visit every square once?" is then a graph question.

A **directed** edge is a pair, which is exactly a relation in the sense of Chapter 21. A directed graph *is* a relation on V, drawn as a picture; an undirected graph is a symmetric relation.

25.3. Degree and the handshake lemma

25.3.1. The problem

At a meeting, people shake hands; you ask everyone how many hands they shook and add up the answers. Can the total be odd?

25.3.2. Degree



Degree

The **degree** of a vertex v, written $\deg(v)$, is the number of edges incident to it (a loop counts twice). In a directed graph a vertex has an **in-degree** (arrows arriving) and an **out-degree** (arrows leaving). A vertex of degree 0 is **isolated**; a vertex of degree 1 is a **leaf** (or pendant vertex). A graph in which every vertex has the same degree k is **k-regular**.

25.3.3. The handshake lemma

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Proof (double counting). Count the pairs (vertex, edge) where the edge is incident to the vertex. Counting vertex by vertex, each vertex v contributes $\deg(v)$ pairs, for a total of $\sum \deg(v)$. Counting edge by edge, each edge has exactly two endpoints and contributes exactly 2 pairs, for a total of $2|E|$. Both count the same set, so they are equal. ■

In Königsberg: $3 + 5 + 3 + 3 = 14 = 2 \times 7$. ✓

Corollary: the number of odd-degree vertices is even. Split the sum into even-degree and odd-degree vertices. The total $2|E|$ is even and the even-degree part is even, so the odd-degree part is even. A sum of odd numbers is even only if there is an even number of them. So the answer to the handshake question is *no*: the total is always even, and the number of people who shook an odd number of hands is always even.

For a directed graph the analogue is $\sum \deg^+(v) = \sum \deg^-(v) = |E|$: every arrow leaves exactly one vertex and arrives at exactly one.

25.3.4. Worked example: is the graph possible?

Is there a simple graph with degrees $3, 3, 3, 3, 3$? The sum is 15, which is odd, but it must equal $2|E|$. No.

Is there a simple graph with degrees $4, 4, 4, 1, 1$? The sum is 14, so it would have 7 edges, and the parity test passes. But look harder: three vertices of degree 4 in a 5-vertex simple graph must each be adjacent to *every* other vertex, including both degree-1 vertices. Then each degree-1 vertex would have at least 3 neighbours. Contradiction. No. The parity test is necessary, not sufficient.

Degrees in the wild

- **Social graphs.** Degree is "number of friends". The handshake lemma explains the *friendship paradox*: on average, your friends have more friends than you do. Pick a random edge endpoint rather than a random person, and high-degree people are over-represented in proportion to their degree.
- **Sizing data structures.** An adjacency list stores $\sum \deg(v) = 2|E|$ neighbour entries for an undirected graph, so its memory is $O(|V| + |E|)$.
- **Degree distributions.** Real networks (the web, citation graphs, the internet's router graph) have a few vertices of enormous degree ("hubs") and very many of small degree, which matters for caching, sharding and the spread of failures.

25.4. Walks, paths, cycles and connectivity

25.4.1. Vocabulary

Movement through a graph has several flavours, and the words are used carefully.



Walk, trail, path, cycle

- A **walk** is a sequence of vertices $v_0, v_1, \ldots, v_k$ in which each consecutive pair is joined by an edge. Its **length** is the number of edges, k. Vertices and edges may repeat.
- A **trail** is a walk with no repeated edge.
- A **path** is a walk with no repeated vertex (hence no repeated edge).
- A **closed** walk ends where it starts ($v_k = v_0$). A **circuit** is a closed trail. A **cycle** is a closed walk of length at least 3 with no repeated vertex other than $v_0 = v_k$.
- The **distance** $d(u, v)$ is the length of a shortest path from u to v (or, in a weighted graph, the smallest total weight).

25.4.2. Connectivity



Connected, component

An undirected graph is **connected** if there is a path between every pair of vertices. Otherwise it splits into **connected components**: maximal connected pieces. Being "joined by a path" is an equivalence relation (reflexive, symmetric, transitive; see Chapter 21), and the components are its equivalence classes.

A directed graph is **strongly connected** if there is a directed path from every vertex to every other, and **weakly connected** if it is connected when you ignore the arrows.

A useful fact: *every walk from u to v contains a path from u to v*. If a walk revisits a vertex, cut out the loop between the two visits; repeat until no vertex repeats. So "reachable by a walk" and "reachable by a path" mean the same thing, and the shortest walk is always a path.



Interactive graph sandbox

A canvas where the reader adds vertices and drags edges between them. Live panels show each vertex's degree, the degree sum and $2|E|$ side by side, the connected components in different colours, and whether the current graph has an Euler path, an Euler circuit, or neither, with the odd-degree vertices highlighted.

Try it: Open the graph sandbox

25.5. Trees

25.5.1. The problem

You need to connect n offices with network cable so that every office can reach every other. What is the smallest number of links that can do it, and what do minimal networks look like?

25.5.2. Intuition

Start with n isolated offices: n components. Each new link can merge at most two components into one, reducing the count by at most 1. To get down to a single component you need at least $n - 1$ links. And a network that uses exactly $n - 1$ has no spare links: every link is essential, so there are no cycles (a link in a cycle could be removed without disconnecting anything). Such minimal connected graphs are *trees*.



Tree, forest, leaf, rooted tree

A **tree** is a connected graph with no cycles. A **forest** is a graph with no cycles (a disjoint union of trees). A **leaf** is a vertex of degree 1. A **rooted tree** singles out one vertex as the **root**; then every other vertex has a unique **parent** (its neighbour on the path to the root) and possibly **children**. The **depth** of a vertex is its distance from the root. A **spanning tree** of a graph G is a tree that includes every vertex of G and uses only edges of G.

25.5.3. Why a tree on n vertices has $n - 1$ edges

Lemma. Every tree with $n \geq 2$ vertices has at least two leaves.

Proof. Take a longest path $v_0, v_1, \ldots, v_k$ in the tree. Suppose v_0 had a neighbour other than v_1. If that neighbour were on the path, it would close a cycle; if it were off the path, it would extend the path, contradicting "longest". So $\deg(v_0) = 1$, and by symmetry $\deg(v_k) = 1$. ■

Theorem. A tree with n vertices has exactly $n - 1$ edges.

Proof by induction on n (Chapter 22). For $n = 1$, a single vertex has 0 edges. ✓ Suppose every tree with $n - 1$ vertices has $n - 2$ edges, and let T have $n \geq 2$ vertices. By the lemma, T has a leaf ℓ. Removing ℓ and its one edge leaves a graph that is still connected (no path between other vertices passed through a leaf) and still acyclic: a tree with $n - 1$ vertices, hence $n - 2$ edges. Putting ℓ back adds one edge: $n - 1$. ■

The following are all equivalent for a graph with n vertices; any two of the first three imply the third:

1. it is connected;
2. it has no cycles;
3. it has $n - 1$ edges;
4. between every pair of vertices there is *exactly one* path.

Property 4 is why trees are the natural shape for hierarchies: file systems, the DOM, organisational charts, parse trees, and the "parent pointer" structures that BFS and DFS produce.



Watch the edge count

$n - 1$ edges is *not* enough by itself to make a tree: a triangle plus an isolated vertex has 4 vertices and 3 edges but is neither connected nor acyclic. You need $n - 1$ edges *and* one of "connected" or "acyclic". Similarly, in code, checking `len(edges) == n - 1` is a quick filter, not a proof that the input is a tree.



Counting trees

How many different trees can be built on n labelled vertices? Cayley's formula (1889) gives the strikingly simple answer n^{n-2}: 3 trees on 3 vertices, 16 on 4, 125 on 5. One elegant proof (Prüfer, 1918) encodes each tree as a sequence of $n - 2$ vertex labels, a bijection in exactly the style of stars and bars.

25.5.4. Where you'd use this

Chemistry: alkanes are trees. Draw a molecule with atoms as vertices and bonds as edges. An alkane $\mathrm{C}_n\mathrm{H}_{2n+2}$ has $3n + 2$ atoms, and its degree sum is $4n$ (carbon) $+ (2n + 2)$ (hydrogen) $= 6n + 2$, so by the handshake lemma it has $3n + 1$ bonds: exactly one fewer than its atoms. A connected graph with that many edges is a tree, so alkanes have no rings. The different **isomers** are the different trees the carbons can form: butane ($n = 4$) has 2 (a chain and a branched one), pentane has 3. Cayley's interest in counting trees came from exactly this question.

Engineering: radial networks. Local electricity distribution networks are usually **operated** as trees, even when extra cables exist: with n substations and exactly $n - 1$ closed links, there is one path from the supply to each customer, so a fault is easy to locate and isolate. Close one more link and you create a loop, which is what engineers do deliberately (and briefly) to reroute power during repairs.

Everyday and science. Family trees (going back one generation per level), company org charts, folder hierarchies and the "tree of life" in evolutionary biology are all rooted trees; a phylogenetic tree of n species, with branching ancestors as extra vertices, is built by the same "no cycles, everything connected" logic.

25.6. Bipartite graphs

25.6.1. The problem

Four developers and four services: each developer can maintain certain services. Can every service get a distinct maintainer? Problems like this pair up things of two *different* kinds, and edges only ever run between the kinds.

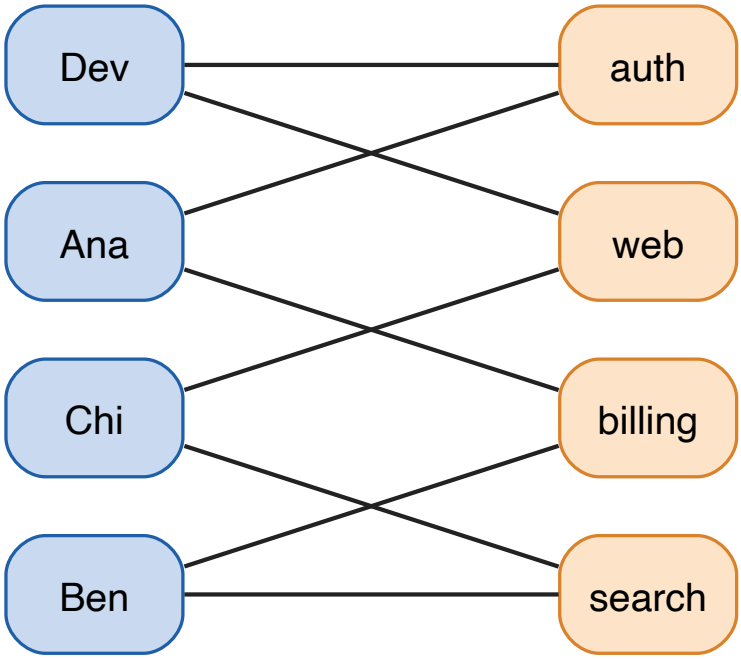


Figure 33. A bipartite graph: edges join a developer to a service, never two developers or two services.



Bipartite graph, matching

A graph is **bipartite** if its vertices can be split into two sets L and R (the **parts**) so that every edge joins a vertex of L to a vertex of R. $K_{a,b}$ is the **complete bipartite graph** with $|L| = a$, $|R| = b$ and all ab edges between them. A **matching** is a set of edges no two of which share an endpoint; a **perfect matching** covers every vertex.

Try it: Two-colour the developer graph .

25.6.2. Recognising bipartite graphs

Theorem. A graph is bipartite if and only if it has no cycle of odd length. *Try it:* Watch two-colouring find odd cycles .

Why: walking round a cycle in a bipartite graph alternates sides, L, R, L, R, ..., and you can only return to your starting side after an even number of steps. Conversely, if there are no odd cycles, colour a start vertex 0, its neighbours 1, their neighbours 0, and so on outward (breadth-first); a clash would mean

two vertices at the same parity of distance are adjacent, which closes an odd cycle.

That proof is an algorithm. Here it is, using breadth-first search from Section 25.10:

```python
from collections import deque

def two_colour(graph):
    """Return a 2-colouring {vertex: 0/1} if the graph is bipartite, else None."""
    side = {}
    for s in graph:                          # handle every component
        if s in side:
            continue
        side[s] = 0
        queue = deque([s])
        while queue:
            u = queue.popleft()
            for v in graph[u]:
                if v not in side:
                    side[v] = 1 - side[u]  # neighbours go on the other side
                    queue.append(v)
                elif side[v] == side[u]:   # edge inside one side: odd cycle
                    return None
    return side

square = {1: [2, 4], 2: [1, 3], 3: [2, 4], 4: [3, 1]}
triangle = {1: [2, 3], 2: [1, 3], 3: [1, 2]}
print(two_colour(square))
print(two_colour(triangle))
```

```
{1: 0, 2: 1, 4: 1, 3: 0}
None
```

In the developer graph above, a perfect matching exists (Ana–billing, Ben–search, Chi–web, Dev–auth, for instance). Finding maximum matchings efficiently is a classic algorithmic problem (Hopcroft–Karp, network flow) behind job assignment, ad allocation, organ-donor exchanges and medical residency placement.

25.6.3. Where you'd use this

The residency match is the famous large-scale example. Each year the US National Resident Matching Program pairs tens of thousands of newly qualified doctors with hospital training posts. The algorithm behind it finds a **stable** matching in a bipartite graph (no doctor and hospital would both prefer each other to what they got). Its theory, by David Gale and Lloyd Shapley, and its practice, by Alvin Roth, won the 2012 Nobel prize in economics. School-place allocation in cities such as New York and Boston uses the same algorithm. Closer to home, assigning 4 people to 4 chores they are each willing to do is a perfect-matching question, and the developer example above is the workplace version.

25.7. Euler paths and circuits

25.7.1. The problem

Return to Königsberg, generalised. Given a connected graph, can you traverse every edge exactly once? A snowplough wants to clear every street once; a pen-plotter wants to draw a figure without lifting the pen; a DNA-sequencing algorithm wants to chain overlapping fragments.



Euler path and Euler circuit

An **Euler path** (or Euler trail) is a trail that uses every edge of the graph exactly once. An **Euler circuit** is an Euler path that starts and ends at the same vertex. A graph with an Euler circuit is called **Eulerian**.

25.7.2. The criterion

Theorem (Euler 1736, Hierholzer 1873). Let G be a connected graph (ignoring isolated vertices).

- G has an Euler circuit if and only if every vertex has even degree.
- G has an Euler path that is not a circuit if and only if exactly two vertices have odd degree. The path must start at one of them and end at the other.

25.7.3. Why it works

Necessity is the Königsberg argument. Each time a trail passes through a vertex, it uses one edge in and one edge out: a pair. For a circuit, the start vertex is also paired (the first edge out with the last edge in), so every degree is even. For an open path, only the two endpoints have an unpaired edge, so exactly those two have odd degree.

Sufficiency (that the condition is enough) is constructive. Start anywhere and keep walking along unused edges. With all degrees even, whenever you enter a vertex other than the start, an unused edge remains to leave by, so you can only get stuck back at the start: you have a closed trail. If it hasn't used every edge, some vertex on it still has unused edges (the graph is connected); start a new closed trail there, using only unused edges, and splice it into the first. Repeat until every edge is used. This is *Hierholzer's algorithm*, which runs in $O(|E|)$ time. For the two-odd-vertex case, add an imaginary edge between the odd vertices, find a circuit, then delete the imaginary edge.

25.7.4. Worked example: the house

The "house" or "open envelope" puzzle asks you to draw this figure without lifting your pen or retracing a line. *Try it:* Trace the house's Euler path .

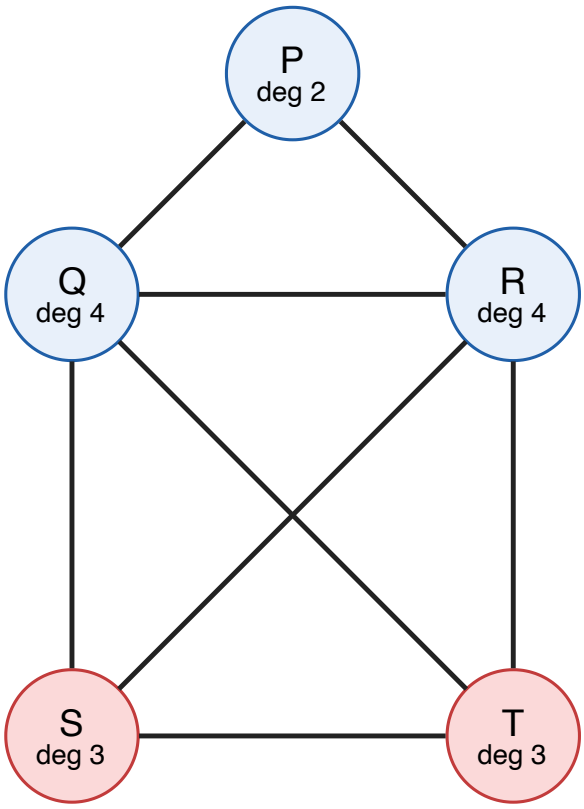


Figure 34. The house: degrees 2, 4, 4, 3, 3. Exactly two odd vertices (S and T), so an Euler path exists and must run from S to T (or T to S).

Degrees: $P = 2, Q = 4, R = 4, S = 3, T = 3$. Sum $16 = 2 \times 8$. ✓ Two odd vertices, so an Euler path exists and must start at S or T. One such path:

$$S \to T \to R \to S \to Q \to R \to P \to Q \to T.$$

Check: 8 steps, using the edges ST, TR, RS, SQ, QR, RP, PQ, QT, which are all 8 edges, each once. Start at P instead and you will always get stuck, which is now a theorem rather than a frustration.

PYTHON

```python
from collections import defaultdict

def adjacency_list(edges):
    """Build an undirected adjacency list (multi-edges allowed)."""
    adj = defaultdict(list)
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)
    return adj

def euler_status(edges):
    adj = adjacency_list(edges)
    odd = sorted(v for v in adj if len(adj[v]) % 2 == 1)
    # (assumes the graph is connected)
    if not odd:
        return "Euler circuit exists"
    if len(odd) == 2:
        return f"Euler path exists, from {odd[0]} to {odd[1]}"
    return f"neither: {len(odd)} odd vertices {odd}"

konigsberg = [("N", "K"), ("N", "K"), ("S", "K"), ("S", "K"),
              ("N", "E"), ("K", "E"), ("S", "E")]
house = [("P", "Q"), ("P", "R"), ("Q", "R"), ("Q", "S"),
         ("Q", "T"), ("R", "S"), ("R", "T"), ("S", "T")]

adj = adjacency_list(konigsberg)
print({v: len(adj[v]) for v in adj}, sum(len(n) for n in adj.values()))
print(euler_status(konigsberg))
print(euler_status(house))
```

```
{'N': 3, 'K': 5, 'S': 3, 'E': 3} 14
neither: 4 odd vertices ['E', 'K', 'N', 'S']
Euler path exists, from S to T
```



Connectivity is part of the criterion

"All degrees even" is not enough if the graph is in two pieces: two separate triangles have all degrees 2, but no single trail covers both. Always check connectivity (for example with BFS) before applying the degree test. For a directed graph, the criterion becomes: every vertex has in-degree equal to out-degree (for a circuit), and the graph is connected when directions are ignored.

Try it: Even degrees, two components .

25.7.5. Where you'd use this

Route planning for every street. Gritting lorries, street sweepers, postal rounds and meter readers want a closed route that covers every street, and every repeated street costs fuel. If every junction has even degree, an Euler circuit does it with no repeats. Otherwise the odd-degree junctions must be paired up and a shortest path between each pair driven twice: the **Chinese postman problem**, first posed by the Chinese mathematician Meigu Guan in 1960. In the house above, a sweeper that must return to its depot has to repeat one street: the S–T edge, giving 9 street-traversals for 8 streets.

Drawing machines. A pen plotter, laser cutter or CNC router drawing a connected outline wastes time on "pen-up" moves. An outline whose graph has $2k$ odd vertices needs at least k separate strokes, so a figure with all even degrees can be cut in one continuous pass.

Computer-science connection: Euler paths in genome assembly

DNA sequencers read millions of short overlapping fragments. In the *de Bruijn graph* approach, each short substring of length $k-1$ is a vertex and each fragment of length k is a directed edge from its prefix to its suffix. Reassembling the genome becomes finding an Euler path, which is fast (linear time). The older formulation used a Hamiltonian path instead, which, as you will see next, is believed to be intractable. Choosing the right graph model turned an impossible computation into an easy one.

25.8. Hamiltonian cycles, and why they are hard

25.8.1. The problem

Change one word. Instead of using every *edge* once, visit every *vertex* once. A delivery van must call at every depot; a tour must visit every city.



Hamiltonian path and cycle

A **Hamiltonian path** visits every vertex exactly once. A **Hamiltonian cycle** is a Hamiltonian path that returns to its start along an edge. The weighted version, "find the cheapest Hamiltonian cycle in a complete weighted graph", is the **travelling salesman problem** (TSP).

25.8.2. The contrast

The two questions look like twins, but they behave completely differently.

- For Euler, there is a simple local test (count degrees) and a linear-time algorithm.
- For Hamilton, no simple test is known, and none is expected. Deciding whether a graph has a Hamiltonian cycle is **NP-complete**: it is one of thousands of problems that are all equally hard, for which no polynomial-time algorithm is known, and for which finding one would also solve the famous P versus NP problem.

Why is the vertex version so much harder? The Euler condition is *local*: each vertex just needs the right parity, and the pieces can always be spliced together. Whether a vertex can be fitted into a Hamiltonian cycle depends on the *global* structure of the whole graph. The obvious algorithm is to try every ordering of the vertices, and there are $(n-1)!/2$ distinct cycles on n labelled vertices of a complete graph (Chapter 24). For 20 cities that is about 6×10^{16}. *Try it:* Brute-force Hamiltonian cycles (cube) .

There are useful *sufficient* conditions. **Dirac's theorem** (1952): if a simple graph has $n \geq 3$ vertices and every vertex has degree at least $n/2$, then it has a Hamiltonian cycle. But many graphs with low degrees are Hamiltonian too (any cycle graph, for instance), so the condition is far from necessary.

25.8.3. Choosing a method in practice

For small graphs (up to about 20 vertices), exact dynamic programming over subsets (the Held–Karp algorithm, $O(n^2 2^n)$) is practical. For larger TSP instances, practitioners use heuristics (nearest neighbour, 2-opt improvement) and branch-and-bound solvers, which routinely solve real instances with tens of thousands of cities to optimality or near it, even though no guarantee of speed exists. "NP-hard" means "no known algorithm is fast on *every* input", not "hopeless in practice". *Try it:* Brute force vs nearest neighbour .



Hamilton's puzzle

William Rowan Hamilton (who also invented quaternions, used today for 3D rotations in games) marketed the *Icosian Game* in 1857: find a cycle through all 20 vertices of a dodecahedron along its edges. It was a commercial flop, being rather easy, but it gave its name to one of the central hard problems of computer science.

US curriculum

Graph theory is not part of the core US high-school sequence (Algebra I, Geometry, Algebra II, Precalculus). Euler circuits, Hamiltonian circuits and the travelling salesman problem appear in **Discrete Mathematics** or **Quantitative Reasoning** electives offered by some high schools, in "finite math" and "math for liberal arts" college courses, and in competitive programming (USACO). A CS major meets them in **Discrete Mathematics** and **Data Structures and Algorithms**.

25.9. Representing graphs in a computer

25.9.1. The problem

To run algorithms you need a data structure. The two standard choices make different operations fast, and the right choice depends on how dense the graph is.

25.9.2. Adjacency matrix

Number the vertices $0, \dots, n-1$. The **adjacency matrix** A is the $n \times n$ table with $A_{ij} = 1$ if there is an edge from i to j and 0 otherwise (or the edge weight, for a weighted graph). For an undirected graph A is symmetric: $A_{ij} = A_{ji}$.

For the directed graph with edges $0 \to 1, 0 \to 2, 1 \to 2, 2 \to 0, 2 \to 3, 3 \to 0$:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

row i lists where i points;
row sums are out-degrees;
column sums are in-degrees.

25.9.3. Adjacency list

The **adjacency list** stores, for each vertex, a list of its neighbours. In Python this is naturally a dictionary of lists:

```python
graph = {0: [1, 2], 1: [2], 2: [0, 3], 3: [0]}
```

25.9.4. Trade-offs

Operation	Adjacency matrix	Adjacency list
Memory	$O(n^2)$	$O(n+m)$
Is uv an edge?	$O(1)$: look up A_{uv}	$O(\deg u)$: scan the list ($O(1)$ with a set)
List neighbours of u	$O(n)$: scan a whole row	$O(\deg u)$
Visit every edge (BFS, DFS)	$O(n^2)$	$O(n+m)$
Add an edge	$O(1)$	$O(1)$
Linear algebra (powers, eigenvectors)	natural	must convert

Choosing. Most real graphs are **sparse**: m is much smaller than the maximum $\binom{n}{2}$. A social network with a billion users and a few hundred friends each has around 10^{11} edges, while its adjacency matrix would have 10^{18} cells, nearly all zero. Use adjacency lists by default. Use a matrix when the graph is small or **dense** (most pairs connected), when you need constant-time edge queries, or when you want to apply linear algebra. In numerical work, *sparse matrix* formats give you the memory of a list with the algebra of a matrix.

25.9.5. Powers of the adjacency matrix count walks

Here is a first taste of how matrices and graphs reinforce each other. Matrix multiplication is defined in Volume 2, Chapter 27 ; for now you need only the rule for one entry: $(A^2)_{ij} = \sum_k A_{ik}A_{kj}$. *Try it:* Count walks with powers of A .

Each term $A_{ik}A_{kj}$ is 1 exactly when there is an edge $i \to k$ *and* an edge $k \to j$, that is, a walk $i \to k \to j$ of length 2. The sum over all intermediate vertices k therefore counts the walks of length 2 from i to j. By induction, **$(A^r)_{ij}$ is the number of walks of length r from i to j.**

```python
import numpy as np

# directed graph on vertices 0..3:  0->1, 0->2, 1->2, 2->0, 2->3, 3->0
A = np.array([[0, 1, 1, 0],
              [0, 0, 1, 0],
              [1, 0, 0, 1],
              [1, 0, 0, 0]])
print(A @ A)
print(np.linalg.matrix_power(A, 3))
```

```
[[1 0 1 1]
 [1 0 0 1]
 [1 1 1 0]
 [0 1 1 0]]
[[2 1 1 1]
 [1 1 1 0]
 [1 1 2 1]
 [1 0 1 1]]
```

Check by hand. $(A^2)_{03} = 1$: the only length-2 walk from 0 to 3 is $0 \to 2 \to 3$. $(A^3)_{00} = 2$: the two closed walks of length 3 from 0 are $0 \to 1 \to 2 \to 0$ and $0 \to 2 \to 3 \to 0$. The diagonal of A^3 counts closed 3-walks through each vertex; in an undirected graph each triangle through a vertex gives two of them (one per direction), so the total number of triangles is $\mathbf{trace}(A^3)/6$, a formula used in network analysis. *Try it:* Closed 3-walks versus triangles .

Computer-science connection: graphs as data

- **Networks.** Routers exchange adjacency information and each computes shortest paths (OSPF uses Dijkstra's algorithm; BGP is a path-vector protocol over the graph of autonomous systems).
- **Social graphs.** "People you may know" ranks friends-of-friends, which are exactly the nonzero entries of A^2. "Six degrees of separation" is a statement about distances in the friendship graph.
- **Graph databases** (Neo4j, Amazon Neptune) store adjacency lists natively, so that "follow this relationship" is a pointer hop rather than a table join.
- **Distributed systems.** Cluster membership, gossip protocols and consensus all run over a communication graph; whether the system tolerates failures is a question about the graph's connectivity after removing vertices or edges.

25.10. Breadth-first search

25.10.1. The problem

Given a graph and a start vertex, which vertices can you reach, and what is the fewest number of edges needed to reach each one? This is "degrees of separation", "minimum number of hops", "fewest moves to solve the puzzle".

25.10.2. Intuition: ripples in a pond

Explore in rings. First the start vertex (distance 0). Then all its neighbours (distance 1). Then all *their* unvisited neighbours (distance 2). And so on. A **queue** (first in, first out) enforces this order automatically: vertices discovered earlier are processed earlier, so you finish each ring before starting the next.



Queue and stack

A **queue** is a list where items are added at the back and removed from the front (FIFO, "first in, first out"), like people waiting in line. A **stack** adds and removes at the same end (LIFO, "last in, first out"), like a pile of plates. In Python, `collections.deque` is an efficient queue (`append`, `popleft`); a plain list is an efficient stack (`append`, `pop`).

Example 30. Procedure — breadth-first search (BFS)

Input: a graph (adjacency lists) and a start vertex s.

Steps:

1. Set $\mathbf{dist}(s) = 0$, $\mathbf{parent}(s) = \mathbf{none}$. Put s in an empty queue. (s is now **discovered**.)
2. While the queue is not empty:
   a. Remove the vertex u at the front of the queue.
   b. For each neighbour v of u that has *not* been discovered: set $\mathbf{dist}(v) = \mathbf{dist}(u) + 1$, set $\mathbf{parent}(v) = u$, and add v to the back of the queue.

Output: every vertex reachable from s, its distance in edges from s, and a parent pointer. Following parent pointers back from any vertex gives a shortest path to it. The parent edges form a spanning tree of s's component, the **BFS tree**.

25.10.3. Worked example

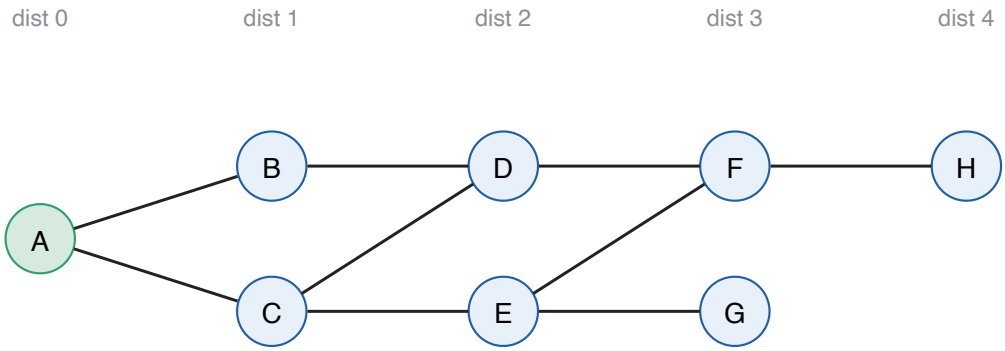


Figure 35. The example graph for BFS and DFS, laid out by distance from A.

Adjacency lists (neighbours in alphabetical order): A: B, C; B: A, D; C: A, D, E; D: B, C, F; E: C, F, G; F: D, E, H; G: E; H: F. *Try it:* Step BFS and DFS .

Remove	Neighbours	Newly discovered (dist, parent)	Queue afterwards
A	B, C	B (1, A), C (1, A)	B, C
B	A, D	D (2, B)	C, D
C	A, D, E	E (2, C) (A and D already discovered)	D, E
D	B, C, F	F (3, D)	E, F
E	C, F, G	G (3, E)	F, G
F	D, E, H	H (4, F)	G, H
G	E	none	H
H	F	none	(empty)

Visit order: A, B, C, D, E, F, G, H. Distances: $A{:}0$, $B, C{:}1$, $D, E{:}2$, $F, G{:}3$, $H{:}4$. A shortest path to H, read from the parent pointers backwards: $H \leftarrow F \leftarrow D \leftarrow B \leftarrow A$, i.e. A, B, D, F, H.

PYTHON

```python
from collections import deque

graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "D", "E"],
    "D": ["B", "C", "F"],
    "E": ["C", "F", "G"],
    "F": ["D", "E", "H"],
    "G": ["E"],
    "H": ["F"],
}

def bfs(graph, start):
    """Return (visit order, distance in edges, parent) from start."""
    dist = {start: 0}
    parent = {start: None}
    order = []
    queue = deque([start])
    while queue:
        u = queue.popleft()                # oldest discovered vertex
        order.append(u)
        for v in graph[u]:
            if v not in dist:              # first time we see v
                dist[v] = dist[u] + 1
                parent[v] = u
                queue.append(v)
    return order, dist, parent

def path_to(parent, target):
    path = []
    while target is not None:
        path.append(target)
        target = parent[target]
    return path[::-1]

order, dist, parent = bfs(graph, "A")
print(order)
print(dist)
print(path_to(parent, "H"))
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
{'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 2, 'F': 3, 'G': 3, 'H': 4}
['A', 'B', 'D', 'F', 'H']
```

25.10.4. Why it works, and how fast

The queue always holds vertices of at most two consecutive distances, d then $d+1$, in that order. So every distance-d vertex is removed before any distance-$(d+1)$ vertex, and the first time a vertex is discovered, it is discovered from a vertex as close to s as possible. Each vertex enters the queue once and each adjacency list is scanned once, so BFS takes $O(n+m)$ time: linear in the size of the graph.



Mark when discovered, not when removed

Mark a vertex as discovered *when you add it to the queue*, not when you take it out. Otherwise the same vertex can be added many times (once per neighbour that finds it), which wastes time and, in dense graphs, memory. And remember that BFS finds fewest *edges*; if edges have different weights, the path with fewest edges may not be the cheapest. That needs Dijkstra's algorithm.

25.11. Depth-first search

25.11.1. Intuition: explore a maze

Depth-first search is how you explore a maze with a ball of string: always go deeper down an unexplored passage; when you reach a dead end, back up to the most recent junction with an unexplored passage, and try that. Replace BFS's queue with a stack, or use recursion (which keeps a stack for you), and you have DFS.

Example 31. Procedure — depth-first search (DFS), recursive

Input: a graph (adjacency lists) and a start vertex s.

Steps:

1. Mark every vertex unvisited. Call **visit**(s).
2. **visit**(u): mark u visited and record it. For each neighbour v of u, in order: if v is unvisited, call **visit**(v). When the loop ends, u is **finished**; return to the caller (backtrack).

Output: the vertices reachable from s in DFS discovery order, and (if recorded) the order in which they finish.

25.11.2. Worked trace

On the same graph, from A: *Try it:* DFS trace, step by step .

```
visit A        A's first unvisited neighbour is B
  visit B      B: A visited; go to D
    visit D    D: B visited; go to C
      visit C  C: A, D visited; go to E
        visit E      E: C visited; go to F
          visit F    F: D, E visited; go to H
            visit H  H: F visited; H finished
          F finished
          (back at E) next neighbour G
          visit G    G finished
        E finished
      C finished
    D finished
  B finished
A finished
```

Discovery order: A, B, D, C, E, F, H, G. Compare BFS (A, B, C, D, E, F, G, H): DFS plunged to H before ever touching G, a neighbour of E.

```python
def dfs(graph, start):
    """Recursive depth-first search; returns vertices in discovery order."""
    visited, order = set(), []

    def visit(u):
        visited.add(u)
        order.append(u)
        for v in graph[u]:
            if v not in visited:
                visit(v)

    visit(start)
    return order

def dfs_iterative(graph, start):
    """Same idea with an explicit stack (no recursion limit)."""
    visited, order = set(), []
    stack = [start]
    while stack:
        u = stack.pop()                    # newest vertex first
        if u in visited:
            continue
        visited.add(u)
        order.append(u)
        for v in reversed(graph[u]):       # reversed: visit neighbours in listed order
            if v not in visited:
                stack.append(v)
    return order

print(dfs(graph, "A"))
print(dfs_iterative(graph, "A"))
```

```
['A', 'B', 'D', 'C', 'E', 'F', 'H', 'G']
['A', 'B', 'D', 'C', 'E', 'F', 'H', 'G']
```

DFS also runs in $O(n + m)$ time.

25.11.3. Choosing between BFS and DFS

Need	Use	Why
Fewest-edge paths, "degrees of separation"	BFS	explores in order of distance
Just reachability, or connected components	either	both visit every reachable vertex once
Detect a cycle; topological order; strongly connected components	DFS	finishing times expose the graph's structure
Explore a huge or infinite space (puzzles, game trees) with limited memory	DFS (or iterative deepening)	stores only the current path, not a whole frontier
Solution likely close to the start	BFS	stops at the first ring that contains it



Recursion depth

Python's default recursion limit is about 1000 frames. A recursive DFS on a long path-like graph (a linked list of 100,000 nodes, say) crashes with `RecursionError`. Use the iterative, explicit-stack version for large inputs. Note that the iterative version pushes neighbours in reverse so that they are popped in the listed order; without the reversal you get a valid DFS, just a different one.



BFS versus DFS race

Draw or load a graph (or a grid maze). Step BFS and DFS side by side from the same start, colouring vertices as discovered, in the queue/stack, and finished, with the queue or stack contents shown live. Toggle "maze mode" to see BFS flood outward in rings while DFS snakes down corridors, and compare the paths each finds to the exit.

Try it: BFS versus DFS race

25.12. Shortest paths: Dijkstra's algorithm

25.12.1. The problem

Edges now have lengths: road distances, network latencies, costs. Find the cheapest route from a source S to every other vertex. BFS no longer works: the path with fewest edges need not be the cheapest.

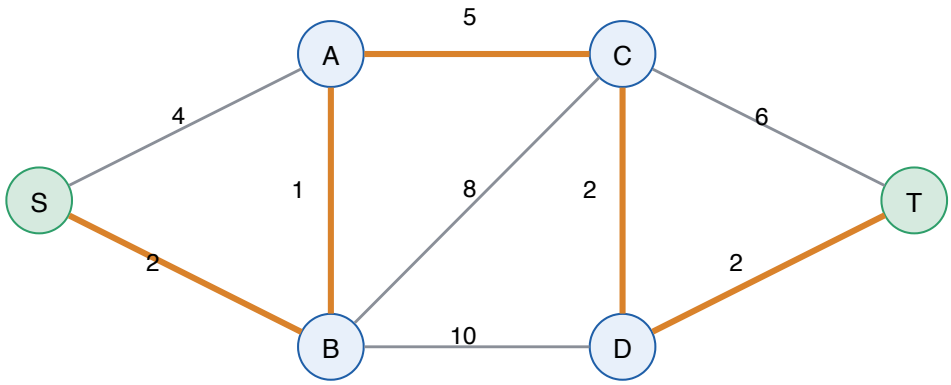


Figure 36. A weighted graph. The shortest S–T route (thick) has 5 edges and total length 12; the 3-edge route S–B–D–T has length 14.

25.12.2. Intuition: settle the nearest vertex first

Imagine pouring water in at S, flowing along edges at constant speed. The water reaches vertices in order of their true distance. Dijkstra's algorithm simulates this. It keeps a **tentative distance** for each vertex: the length of the best route found so far. At each step it takes the unfinished vertex with the *smallest* tentative distance and declares that distance final. Then it checks whether going *through* this newly finished vertex gives a shorter route to any of its neighbours, an update called **relaxing** an edge.



Relaxation

Relaxing an edge $u \to v$ of weight w means: if $\mathbf{dist}(u) + w < \mathbf{dist}(v)$, set $\mathbf{dist}(v) = \mathbf{dist}(u) + w$ and $\mathbf{parent}(v) = u$. It asks "is going via u better than the best route to v known so far?" The symbol ∞ (infinity) stands for "no route found yet".

25.12.3. Why it works

Suppose u is the unfinished vertex with smallest tentative distance d. Any other route to u must leave the finished region somewhere, through some unfinished vertex x with tentative distance at least d, and then continue along edges of *non-negative* weight. So it costs at least d. Hence d is the true distance, and finalising u is safe. This is a **greedy** algorithm (it makes the locally best choice and never revisits it) whose correctness rests entirely on weights being non-negative.

Example 32. Procedure — Dijkstra's algorithm

Input: a graph with non-negative edge weights and a source s.

Steps:

1. Set $\mathbf{dist}(s) = 0$ and $\mathbf{dist}(v) = \infty$ for every other v. No vertex is finished.
2. Repeat until every reachable vertex is finished:
   a. Choose the unfinished vertex u with the smallest $\mathbf{dist}(u)$. Mark it finished.
   b. For each edge $u \to v$ with weight w where v is unfinished: relax it (if $\mathbf{dist}(u) + w < \mathbf{dist}(v)$, update $\mathbf{dist}(v)$ and $\mathbf{parent}(v)$).

Output: $\mathbf{dist}(v)$, the length of a shortest path from s to each v, and parent pointers from which the paths can be read backwards.

25.12.4. Fully worked trace

Edges (undirected): SA 4, SB 2, AB 1, AC 5, BC 8, BD 10, CD 2, CT 6, DT 2. Each cell shows the tentative distance and, in brackets, the parent. A bold entry is finalised in that step; a dash means the vertex was already finished. *Try it:* Step through this trace .

Step	Finish	S	A	B	C	D	T
0	(start)	0	∞	∞	∞	∞	∞
1	S (0)	**0**	4 (S)	2 (S)	∞	∞	∞
2	B (2)	–	3 (B)	**2**	10 (B)	12 (B)	∞
3	A (3)	–	**3**	–	8 (A)	12 (B)	∞
4	C (8)	–	–	–	**8**	10 ©	14 ©
5	D (10)	–	–	–	–	**10**	12 (D)
6	T (12)	–	–	–	–	–	**12**

Step by step:

1. **Finish S** (distance 0). Relax SA: $0 + 4 = 4 < \infty$, so A becomes 4 via S. Relax SB: $0 + 2 = 2$, B becomes 2 via S.
2. **Finish B** (smallest unfinished: B at 2). Relax BA: $2 + 1 = 3 < 4$, so A improves to 3 via B. Relax BC: $2 + 8 = 10$, C becomes 10 via B. Relax BD: $2 + 10 = 12$, D becomes 12 via B.
3. **Finish A** (3). Relax AC: $3 + 5 = 8 < 10$, C improves to 8 via A.

4. **Finish C** (8). Relax CD: $8 + 2 = 10 < 12$, D improves to 10 via C. Relax CT: $8 + 6 = 14$, T becomes 14 via C.
5. **Finish D** (10). Relax DT: $10 + 2 = 12 < 14$, T improves to 12 via D.
6. **Finish T** (12). Done.

Read the path backwards from T using parents: $T \leftarrow D \leftarrow C \leftarrow A \leftarrow B \leftarrow S$. The shortest route is $S \to B \to A \to C \to D \to T$ with length $2 + 1 + 5 + 2 + 2 = 12$. Notice that the direct-looking edge SA (4) is *not* on any shortest path, and that the fewest-edge route S–B–D–T (length $2 + 10 + 2 = 14$) loses.

PYTHON

```python
import heapq

wgraph = {
    "S": {"A": 4, "B": 2},
    "A": {"S": 4, "B": 1, "C": 5},
    "B": {"S": 2, "A": 1, "C": 8, "D": 10},
    "C": {"A": 5, "B": 8, "D": 2, "T": 6},
    "D": {"B": 10, "C": 2, "T": 2},
    "T": {"C": 6, "D": 2},
}

def dijkstra(graph, source):
    """Shortest-path distances and parents from source (weights >= 0)."""
    dist = {source: 0}
    parent = {source: None}
    done = set()
    heap = [(0, source)]                      # (tentative distance, vertex)
    while heap:
        d, u = heapq.heappop(heap)            # closest unfinished vertex
        if u in done:
            continue                          # stale entry; skip it
        done.add(u)
        for v, w in graph[u].items():
            if v in done:
                continue                      # finished vertices are never revisited
            new = d + w
            if v not in dist or new < dist[v]:   # relax edge u -> v
                dist[v] = new
                parent[v] = u
                heapq.heappush(heap, (new, v))
    return dist, parent

dist, parent = dijkstra(wgraph, "S")
print(dist)
path, v = [], "T"
while v is not None:
    path.append(v)
    v = parent[v]
print(path[::-1])
```

```
{'S': 0, 'A': 3, 'B': 2, 'C': 8, 'D': 10, 'T': 12}
['S', 'B', 'A', 'C', 'D', 'T']
```

The **priority queue** (`heapq` , a binary heap) finds the smallest tentative distance in $O(\log n)$ time instead of scanning every vertex. Rather than updating entries inside the heap, the code pushes a new entry on every improvement and skips "stale" entries when they surface. The total running time is $O((n+m)\log n)$.

25.12.5. Failure mode: negative weights

Take vertices S, A, B with edges $S \to A$ (weight 2), $S \to B$ (weight 3) and $B \to A$ (weight -2). Dijkstra finishes A first at distance 2, because it is the nearest unfinished vertex, and never reconsiders it. But $S \to B \to A$ costs $3 - 2 = 1$. The greedy argument broke because a later edge could *reduce* a total.
Try it: Watch the negative edge fool Dijkstra .

25.12.6. Alternative approaches and choosing a method

Situation	Algorithm	Time
Unweighted (all edges cost 1)	BFS	$O(n+m)$
Non-negative weights, one source	Dijkstra	$O((n+m)\log n)$
One source and one target, with a good distance estimate (maps, games)	A* search (Dijkstra guided by a heuristic)	often far less than Dijkstra
Negative weights allowed; detect negative cycles	Bellman–Ford	$O(nm)$
Shortest paths between all pairs, small dense graph	Floyd–Warshall	$O(n^3)$
Directed acyclic graph	relax edges in topological order	$O(n+m)$



> *Dijkstra and the shortest-path problem*
> Edsger Dijkstra designed the algorithm in 1956 in about twenty minutes, sitting at a café terrace in Amsterdam with his fiancée, without pencil or paper. He wanted a demonstration problem for a new computer, the ARMAC, that non-mathematicians could understand: the shortest route between two Dutch cities. He later remarked that designing without pencil and paper was one reason the algorithm turned out so simple: it forces you to avoid all avoidable complexities.



Dijkstra step-through

Load the example graph or draw your own. Step through Dijkstra's algorithm, with finished vertices shaded, tentative distances shown on each vertex, and each relaxation flashing the edge involved. The trace table above fills in row by row. A "negative edge" toggle lets the reader set one weight below zero and watch the algorithm confidently report a wrong answer.

Try it: Dijkstra step-through

25.12.7. Where you'd use this

Navigation. A phone's route planner runs a refined Dijkstra (A* search plus precomputed shortcuts called **contraction hierarchies**) on road graphs with tens of millions of junctions. The weights are travel times, not distances, which is why "fastest" and "shortest" routes differ. Public-transport journey planners do the same on a graph whose vertices are (stop, time) pairs.

Finance: spotting arbitrage. Exchange rates multiply along a path, but shortest-path algorithms add. Taking logarithms fixes that: give the edge from currency X to Y the weight $-\ln(\text{rate})$. Suppose USD $\to$ EUR is 0.90, EUR $\to$ GBP is 0.90 and GBP $\to$ USD is 1.25. Round the cycle, \$1 becomes $0.90 \times 0.90 \times 1.25 = 1.0125$ dollars, and the weights sum to $-\ln 1.0125 < 0$: a **negative cycle**. Bellman–Ford (the table above) detects negative cycles, which is why it, not Dijkstra, is the textbook tool for this. (Real markets close such gaps within milliseconds.) *Try it:* Find the arbitrage cycle .

Networks. Internet routers inside one organisation run OSPF, which is Dijkstra on the link-state graph, recomputed whenever a link fails.

25.13. Dependency graphs and topological sort

25.13.1. The problem

Some tasks must happen before others. A course has prerequisites; a source file must be compiled before the program that links it; a package must be installed before the packages that depend on it. Given all the "must come before" constraints, find an order that respects every one of them.

This book is itself an example. The map at the front (The Knowledge Map) is a directed graph whose edges say "read this chapter first". Here is a fragment:

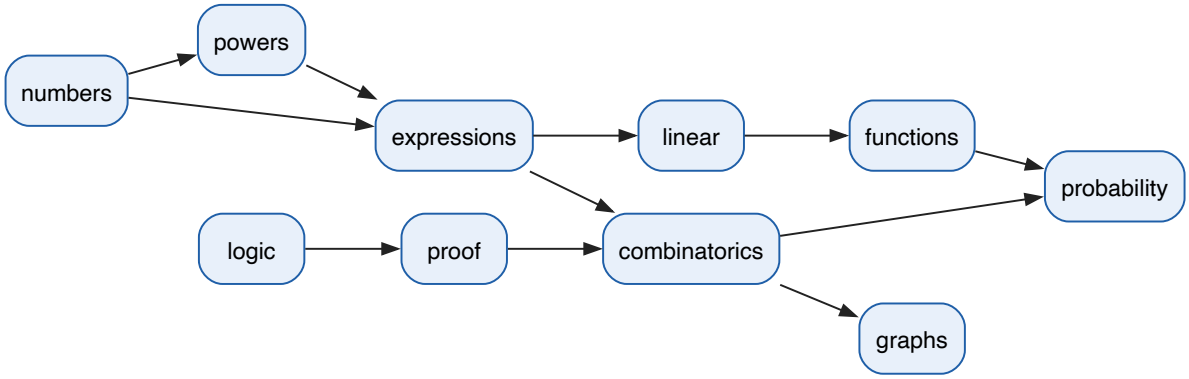


Figure 37. A fragment of this book's dependency map. An arrow X → Y means "X should be read before Y".



DAG and topological order

A **directed acyclic graph** (**DAG**) is a directed graph with no directed cycle. A **topological order** (or topological sort) of a directed graph is a listing of all its vertices such that for every edge $u \to v$, u comes before v. A vertex with in-degree 0 is a **source**; with out-degree 0, a **sink**.

Theorem. A directed graph has a topological order if and only if it is a DAG.

If there is a cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$, then v_1 must come before itself, which is impossible. Conversely, every DAG has a source (follow edges *backwards* from any vertex; with no cycles and finitely many vertices you must reach a vertex with no incoming edge). Put a source first, delete it, and the rest is still a DAG; repeat. That argument is Kahn's algorithm.

Example 33. Procedure — topological sort (Kahn's algorithm)

Input: a directed graph.

Steps:

1. Compute the in-degree of every vertex (how many unmet prerequisites it has).
2. Put every vertex of in-degree 0 into a set **ready**.
3. While **ready** is not empty: remove a vertex u from it and append u to the output. For each edge $u \to v$, decrease v's in-degree by 1; if it reaches 0, add v to **ready**.
4. If the output contains every vertex, it is a topological order. If not, the remaining vertices all lie on or behind a cycle: report it.

Output: a topological order, or a proof that none exists.

25.13.2. Worked trace

In-degrees at the start: numbers 0, logic 0, powers 1, expressions 2, linear 1, functions 1, proof 1, combinatorics 2, graphs 1, probability 2. When several vertices are ready, take them alphabetically (any choice gives a valid order). *Try it:* Step through Kahn's algorithm .

Take	In-degree changes	Ready afterwards
(start)		logic, numbers
logic	proof 1→0	numbers, proof
numbers	powers 1→0, expressions 2→1	powers, proof
powers	expressions 1→0	expressions, proof
expressions	linear 1→0, combinatorics 2→1	linear, proof
linear	functions 1→0	functions, proof
functions	probability 2→1	proof
proof	combinatorics 1→0	combinatorics
combinatorics	graphs 1→0, probability 1→0	graphs, probability
graphs		probability
probability		(empty)

Order: logic, numbers, powers, expressions, linear, functions, proof, combinatorics, graphs, probability. Every arrow points forward.

```python
import heapq

# edge u -> v means "u must come before v"
prereqs = {
    "numbers":       ["powers", "expressions"],
    "powers":        ["expressions"],
    "expressions":   ["linear", "combinatorics"],
    "linear":        ["functions"],
    "functions":     ["probability"],
    "logic":         ["proof"],
    "proof":         ["combinatorics"],
    "combinatorics": ["graphs", "probability"],
    "graphs":        [],
    "probability":   [],
}

def topological_sort(graph):
    """Kahn's algorithm. Ties broken alphabetically for a reproducible order."""
    indegree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    ready = [u for u in graph if indegree[u] == 0]   # no unmet prerequisites
    heapq.heapify(ready)
    order = []
    while ready:
        u = heapq.heappop(ready)
        order.append(u)
        for v in graph[u]:                  # u is done: v has one fewer blocker
            indegree[v] -= 1
            if indegree[v] == 0:
                heapq.heappush(ready, v)
    if len(order) < len(graph):
        raise ValueError("graph has a cycle; no topological order exists")
    return order

print(topological_sort(prereqs))
prereqs["graphs"].append("logic")          # introduce a cycle
try:
    topological_sort(prereqs)
except ValueError as e:
    print(e)
```

Try it: Add the cycle, watch it stall .

```
['logic', 'numbers', 'powers', 'expressions', 'linear', 'functions', 'proof', 'combinatorics',
'graphs', 'probability']
graph has a cycle; no topological order exists
```

Alternative method. Run DFS and list vertices in order of *finishing* time, then reverse the list. A vertex finishes only after everything reachable from it has finished, so in the reversed list it comes before all of its descendants. Both methods are $O(n + m)$. Kahn's algorithm is easier to explain, naturally reports cycles, and shows which tasks can run *in parallel* (everything in **ready** at the same moment); the DFS method is shorter to code when you already have DFS.



Topological orders are rarely unique

A DAG usually has many valid topological orders (here, "numbers" could equally have gone before "logic"). Code that silently depends on one particular order, for instance a build that happens to work only because two independent steps ran in a fixed sequence, will break when the tie-breaking changes. If two tasks really must be ordered, add the edge.

Computer-science connection: dependency graphs everywhere

- **Build systems.** `make`, Bazel and Gradle build a DAG of targets and run them in topological order, running independent targets in parallel and rebuilding only the descendants of changed files.
- **Package managers.** `pip`, `npm` and `apt` resolve a dependency graph and install in topological order; a "circular dependency" error is Kahn's algorithm finding a leftover cycle.
- **Compilers.** Expressions become DAGs with shared subexpressions; instruction scheduling and data-flow analysis work over the control-flow graph. Spreadsheets recompute cells in topological order of their formula dependencies.
- **Data pipelines and workflows.** Airflow, dbt and CI systems literally call their job graphs "DAGs".
- **Databases and distributed systems.** Deadlock detection looks for a cycle in the "waits-for" graph; version histories (git commits, vector clocks) are DAGs, and "is commit X an ancestor of Y?" is a reachability question.

25.13.3. Where you'd use this

Project planning: the critical path. A small building job has five tasks: foundation (3 weeks), then walls (4 weeks) and plumbing (2 weeks), roof after the walls (2 weeks), and finishing after both roof and plumbing (1 week). Any topological order is a valid schedule. The shortest possible project length is the **longest** path through the DAG: foundation $\rightarrow$ walls $\rightarrow$ roof $\rightarrow$ finishing, $\mathbf{3+4+2+1=10}$ weeks. Plumbing lies on a path of only $\mathbf{3+2+1=6}$ weeks, so it has 4 weeks of **slack**; a delay on the critical path delays the whole job. Engineers call this the critical path method, and it was developed in the late 1950s to schedule large construction and defence projects. *Try it:* Find the critical path .

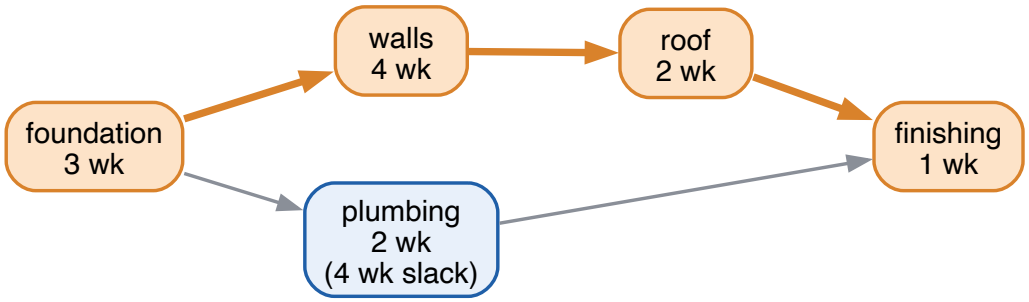


Figure 38. The building project as a DAG. The critical path (thick orange) takes $3 + 4 + 2 + 1 = 10$ weeks; plumbing can slip by 4 weeks without delaying the finish.

Everyday. A recipe is a DAG: you cannot sauté onions you have not chopped, but you can boil the water while you chop. Reading the steps as a dependency graph tells you what can happen in parallel and which step sets the time the meal is ready.

25.14. Minimum spanning trees

25.14.1. The problem

You must connect six sites with fibre. Each possible link has a cost. Every site must be able to reach every other (possibly indirectly), and you want the cheapest network. By Section 25.5, a cheapest such network has no cycles (removing a cycle edge saves money and keeps everything connected), so it is a spanning tree: a **minimum spanning tree** (MST).

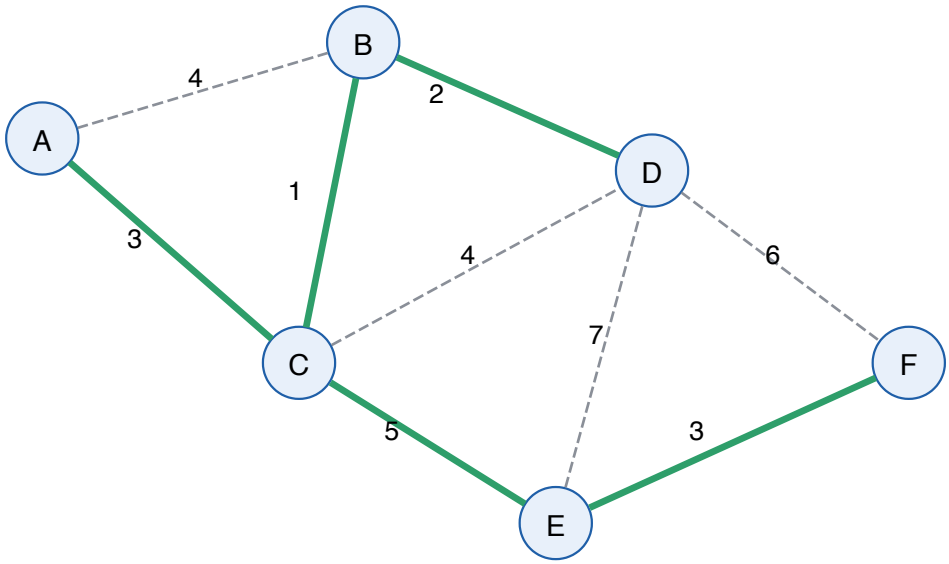


Figure 39. A weighted graph and its minimum spanning tree (solid green edges), total weight 14.

25.14.2. Kruskal's algorithm

Sort the edges from cheapest to dearest. Take each edge in turn, and keep it unless it would form a cycle with the edges already kept. Stop when you have $n - 1$ edges.

Example 34. Procedure — Kruskal's algorithm

Input: a connected weighted graph with n vertices.

Steps:

1. Sort all edges by weight, smallest first.
2. Start with each vertex in its own component and an empty tree.
3. For each edge uv in sorted order: if u and v are in different components, add uv to the tree and merge their components; otherwise skip it (it would close a cycle).
4. Stop when the tree has $n - 1$ edges.

Output: a minimum spanning tree.

Worked trace (edges: BC 1, BD 2, AC 3, EF 3, AB 4, CD 4, CE 5, DF 6, DE 7): *Try it:* Step through Kruskal .

Edge	Decision	Components afterwards
BC 1	keep	{B, C}, {A}, {D}, {E}, {F}
BD 2	keep	{B, C, D}, {A}, {E}, {F}
AC 3	keep	{A, B, C, D}, {E}, {F}
EF 3	keep	{A, B, C, D}, {E, F}
AB 4	skip: A, B already connected	unchanged
CD 4	skip: C, D already connected	unchanged
CE 5	keep (5th edge = $n - 1$; stop)	{A, B, C, D, E, F}

Total weight $1 + 2 + 3 + 3 + 5 = 14$.

```python
edges = [  # (weight, u, v)
    (4, "A", "B"), (3, "A", "C"), (1, "B", "C"), (2, "B", "D"),
    (4, "C", "D"), (5, "C", "E"), (7, "D", "E"), (6, "D", "F"), (3, "E", "F"),
]

def kruskal(vertices, edges):
    parent = {v: v for v in vertices}      # union-find forest

    def find(v):                           # root of v's component
        while parent[v] != v:
            parent[v] = parent[parent[v]]  # path halving
            v = parent[v]
        return v

    tree = []
    for w, u, v in sorted(edges):          # cheapest edges first
        ru, rv = find(u), find(v)
        if ru != rv:                       # joins two components: no cycle
            parent[ru] = rv
            tree.append((u, v, w))
    return tree

mst = kruskal("ABCDEF", edges)
print(mst, sum(w for _, _, w in mst))
```

```
[('B', 'C', 1), ('B', 'D', 2), ('A', 'C', 3), ('E', 'F', 3), ('C', 'E', 5)] 14
```

The "same component?" test uses a **union–find** (disjoint-set) structure, which answers it in nearly constant time; sorting dominates, so Kruskal runs in $O(m \log m)$.

Why greedy works here (the cut property). Split the vertices into any two groups. The cheapest edge crossing between the groups belongs to some MST: if a spanning tree avoided it, adding it would create a cycle that crosses the split at least twice, and swapping out the other, dearer crossing edge would give a tree that is no more expensive. Every edge Kruskal keeps is the cheapest edge crossing between its own component and the rest, so every choice is safe.

25.14.3. Prim's algorithm, briefly

Prim's algorithm grows a single tree from a start vertex: repeatedly add the cheapest edge joining the tree to a vertex not yet in it. It is Dijkstra's algorithm with one change (the priority of a vertex is the weight of the single edge connecting it, not the total path length) and runs in $O((n + m) \log n)$ with a heap. **Choosing:** Kruskal is simple and natural when edges arrive as a list; Prim suits dense graphs and adjacency structures. Both give a minimum total weight; if several edges have equal weights, different MSTs of the same total can result. *Try it:* Grow the tree with Prim .



An MST is not a shortest-path tree

A minimum spanning tree minimises the *total* cost of the network, not the distance between any particular pair. In the MST above, A to E costs $3 + 5 = 8$ along the tree, but if the graph had a direct A–E edge of weight 7 it would not be in the MST, and the shortest A–E route would use it. Pick MST when you're paying to build every link once (cabling, pipelines, clustering); pick shortest paths when you're paying per trip (routing).

25.14.4. Where you'd use this

Infrastructure. The first MST algorithm was published in 1926 by Otakar Borůvka for a practical problem: designing an economical electricity network for southern Moravia. Treat the example graph above as possible cable routes between six villages with lengths in kilometres. Laying every route would cost $4 + 3 + 1 + 2 + 4 + 5 + 7 + 6 + 3 = 35$ km of cable; the MST connects all six villages with 14 km. Pipelines, fibre networks and irrigation channels are planned the same way, usually as a starting point before adding a few redundant links for resilience.

Science and data. Biologists and data scientists build an MST on the pairwise distances between samples and then delete its longest edges; the pieces that remain are **clusters** (single-linkage clustering). The same trick groups similar images, documents or customer records.

Algorithms. For the travelling salesman problem (Section 25.8) with ordinary distances, walking around an MST and skipping repeated cities gives a tour at most twice the optimal length, a quick guaranteed-quality answer to an intractable problem.

25.15. Graph colouring

25.15.1. The problem

Six exams must be scheduled. Two exams that share a student cannot be in the same time slot. What is the smallest number of slots?

Build a **conflict graph**: one vertex per exam, an edge when two exams share a student. A schedule is an assignment of slots to vertices with adjacent vertices in different slots. Call the slots colours, and this is graph colouring.



Graph colouring and chromatic number

A (proper) **colouring** of a graph assigns a colour to each vertex so that adjacent vertices get different colours. The **chromatic number** $\chi(G)$ (Greek letter chi) is the fewest colours that suffice. $\chi(G) \leq 2$ exactly when G is bipartite; $\chi(K_n) = n$; a graph containing a triangle needs at least 3.

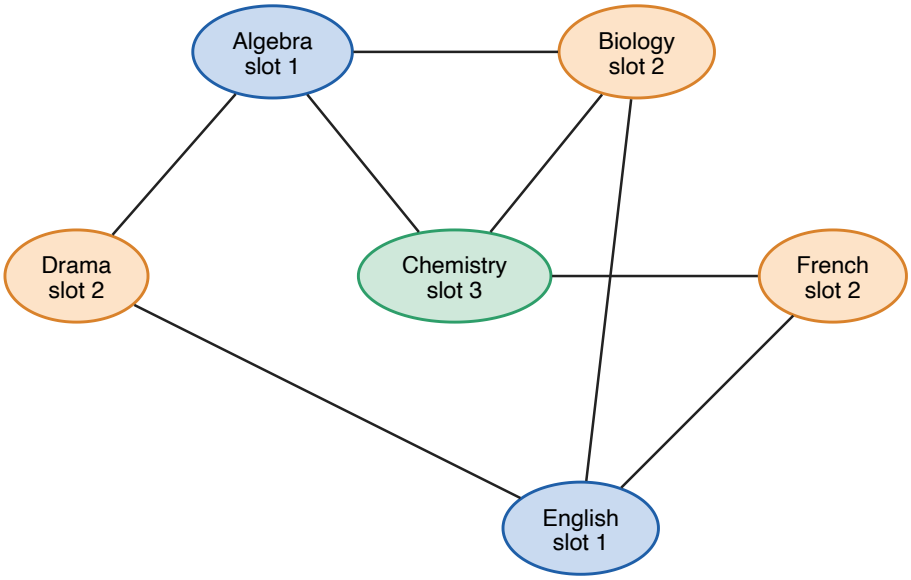


Figure 40. The exam conflict graph, coloured with 3 time slots. Algebra, Biology and Chemistry form a triangle, so 3 slots is the minimum.

25.15.2. The greedy method

Go through the vertices in some order and give each the smallest colour not already used by a coloured neighbour. *Try it:* Greedy-colour the exams .

Example 35. Procedure — greedy colouring

Input: a graph and an ordering of its vertices.

Steps:

1. For each vertex u in the given order: collect the colours already used by u's coloured neighbours, and give u the smallest colour number (0, 1, 2, …) not in that collection.

Output: a proper colouring using at most $\Delta + 1$ colours, where Δ is the largest degree (each vertex has at most Δ neighbours blocking colours).

PYTHON

```python
def greedy_colouring(graph, order):
    """Give each vertex, in the given order, the smallest colour not used by
    an already-coloured neighbour. Colours are 0, 1, 2, ..."""
    colour = {}
    for u in order:
        taken = {colour[v] for v in graph[u] if v in colour}
        c = 0
        while c in taken:
            c += 1
        colour[u] = c
    return colour

# exams that share a student cannot be in the same slot
conflicts = {
    "Algebra":   ["Biology", "Chemistry", "Drama"],
    "Biology":   ["Algebra", "Chemistry", "English"],
    "Chemistry": ["Algebra", "Biology", "French"],
    "Drama":     ["Algebra", "English"],
    "English":   ["Biology", "Drama", "French"],
    "French":    ["Chemistry", "English"],
}
c = greedy_colouring(conflicts, list(conflicts))
print(c, max(c.values()) + 1)

# a bipartite "crown" graph where a bad order wastes colours
crown = {f"a{i}": [f"b{j}" for j in range(1, 5) if j != i] for i in range(1, 5)}
crown.update({f"b{j}": [f"a{i}" for i in range(1, 5) if i != j] for j in range(1, 5)})
bad = ["a1", "b1", "a2", "b2", "a3", "b3", "a4", "b4"]
good = ["a1", "a2", "a3", "a4", "b1", "b2", "b3", "b4"]
print(max(greedy_colouring(crown, bad).values()) + 1,
      max(greedy_colouring(crown, good).values()) + 1)
```

```
{'Algebra': 0, 'Biology': 1, 'Chemistry': 2, 'Drama': 1, 'English': 0, 'French': 1} 3
4 2
```

Trace for the exams: Algebra gets 0. Biology is adjacent to Algebra (0), so gets 1. Chemistry is adjacent to Algebra (0) and Biology (1), so gets 2. Drama is adjacent to Algebra (0): gets 1. English is adjacent to Biology (1) and Drama (1): gets 0. French is adjacent to Chemistry (2) and English (0): gets 1. Three slots, and the Algebra–Biology–Chemistry triangle proves three is optimal.

25.15.3. Failure mode: greedy depends on the order

The second test colours a **crown graph** (each a_i joined to every b_j with $j \neq i$). It is bipartite, so 2 colours suffice, and greedy finds 2 when it colours all the a's first. But in the interleaved order $a_1, b_1, a_2, b_2, \ldots$, greedy uses 4; with k pairs it would use k. Finding $\chi(G)$ exactly is NP-hard (even deciding whether 3 colours suffice), so practical systems use good heuristic orders (largest degree first, or DSatur, which colours the most-constrained vertex next) and accept near-optimal answers. *Try it:* Greedy in a bad order .

25.15.4. Applications

- **Scheduling:** exams, meetings, sports fixtures, with conflicts as edges and time slots as colours.

- **Register allocation:** a compiler builds an *interference graph* whose vertices are program variables and whose edges join variables that are live at the same time. Colours are CPU registers. If the graph can't be coloured with the available registers, some variables are "spilled" to memory. Chaitin's 1981 register allocator made this the standard approach.
- **Frequency assignment:** radio transmitters that are close together (edges) need different frequencies (colours).
- **Sudoku** is a 9-colouring of an 81-vertex graph with some colours pre-assigned.
- **Map colouring:** countries are vertices, shared borders are edges.



The four-colour theorem

In 1852 Francis Guthrie noticed that he could colour a map of the counties of England with four colours so that neighbouring counties differed, and asked whether four always suffice. The question resisted proof for 124 years. In 1976 Kenneth Appel and Wolfgang Haken proved that every planar map can be four-coloured, by reducing the problem to 1,936 configurations checked by computer. It was the first major theorem proved with essential computer assistance, and it provoked a long debate about whether a proof that no human can check by hand counts as a proof. A formally verified version (Gonthier, 2005, in the Coq proof assistant) has since settled most doubts.

25.16. Planar graphs and Euler's formula

25.16.1. The problem

Three houses must each be connected to three utilities (water, gas, electricity) by pipes that do not cross. Can it be done on flat ground? Circuit designers ask the same question: can this circuit be laid out on a single layer without wires crossing?



Planar graph, face

A graph is **planar** if it can be drawn in the plane with no edges crossing. Such a drawing is a **plane embedding**. The edges divide the plane into regions called **faces**, including one unbounded **outer face**. A planar graph can also have non-planar drawings; planarity asks whether *some* crossing-free drawing exists.

25.16.2. Euler's formula

For any connected plane graph with V vertices, E edges and F faces,

$$V - E + F = 2.$$

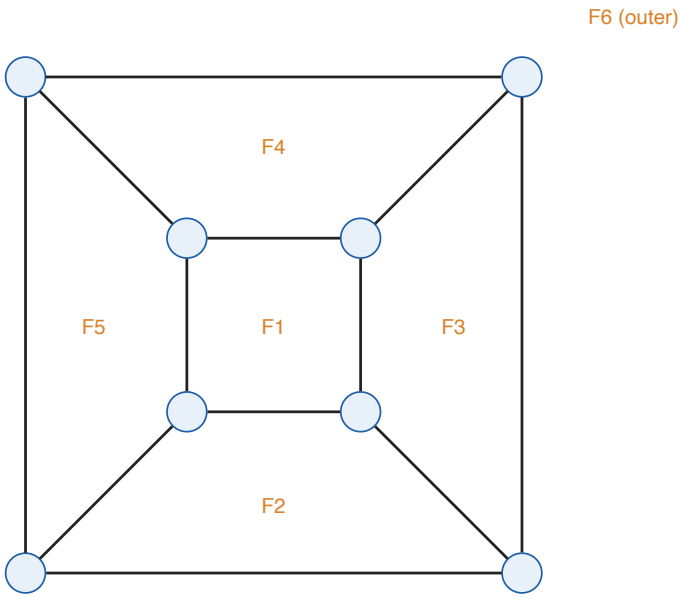


Figure 41. The cube's graph drawn in the plane without crossings: $V = 8$, $E = 12$, $F = 6$ (five inner faces plus the outer one), and $8 - 12 + 6 = 2$.

Proof by induction on the number of edges. If the graph is a tree, it has no cycles, so there is only the outer face: $F = 1$, and $E = V - 1$ (Section 25.5), giving $V - (V - 1) + 1 = 2$. ✓ If it is not a tree, it has a cycle. Remove one edge of that cycle. The graph stays connected (the rest of the cycle still joins the two endpoints), and the two faces on either side of that edge merge into one. So E and F each drop by 1, and $V - E + F$ is unchanged. Repeat until a tree remains, where the value is 2. ■

The cube itself (as a solid) has 8 corners, 12 edges and 6 faces, and $8 - 12 + 6 = 2$: this is Euler's polyhedron formula, of which the graph version is a flattened form. Every convex polyhedron obeys it.

25.16.3. Consequence: an edge limit, and the utilities puzzle

In a simple planar graph with $V \geq 3$, every face is bounded by at least 3 edges, and every edge borders at most 2 faces. Counting (edge, face) border pairs both ways gives $3F \leq 2E$. Substitute $F = 2 - V + E$ from Euler's formula:

$$\begin{aligned} 3(2 - V + E) &\leq 2E \\ 6 - 3V + 3E &\leq 2E \\ E &\leq 3V - 6. \end{aligned}$$

K_5 is not planar. It has $V = 5$ and $E = \binom{5}{2} = 10$, but $3V - 6 = 9 < 10$. *Try it:* Test K5 for planarity.

$K_{3,3}$ (three houses, three utilities) is not planar. It is bipartite, so it has no triangles, and every face has at least 4 edges. The same counting gives $4F \leq 2E$, hence $E \leq 2V - 4 = 2 \times 6 - 4 = 8$. But $K_{3,3}$ has $3 \times 3 = 9$ edges. The puzzle is impossible. *Try it:* Test K3,3 for planarity.

These two graphs are, in a precise sense, the *only* obstacles. **Kuratowski's theorem** (1930): a graph is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$. Linear-time planarity-testing algorithms exist (Hopcroft and Tarjan, 1974).



The edge test only proves non-planarity

$E \le 3V - 6$ is a necessary condition. A graph that satisfies it may still be non-planar ($K_{3,3}$ satisfies $9 \le 12$, which is why you need the sharper triangle-free bound). Use the inequality to rule planarity *out* quickly; never to rule it *in*. Also, Euler's formula as stated needs the graph to be connected; with c components it becomes $V - E + F = 1 + c$.

Planarity in practice

Single-layer printed circuit boards and chip layouts need planar wiring, and the number of extra layers needed relates to how far a circuit is from planar. Planar graphs are also algorithmically friendlier: by $E \le 3V - 6$ they are always sparse, they can always be 4-coloured, and many problems that are hard in general (for example, some shortest-path and separator problems) have faster algorithms on planar inputs, which is useful because road networks are nearly planar.

25.17. A teaser: PageRank

How should a search engine decide which of a million matching web pages is most important? Larry Page and Sergey Brin's 1998 answer was to use the link graph: a page is important if important pages link to it. That sounds circular, and it is, in the way that eigenvector equations are. *Try it:* Run this PageRank example .

Imagine a random surfer who, at each step, follows a random outgoing link from the current page with probability $d = 0.85$, or jumps to a page chosen uniformly at random with probability 0.15. Over a long time, the fraction of time spent on each page settles down to a fixed distribution: that is the PageRank. If M is the matrix whose column i spreads page i's score equally over the pages it links to, the scores $\mathbf{r}$ satisfy

$$\mathbf{r} = \frac{1 - d}{n}\mathbf{1} + d\,M\mathbf{r},$$

and you can find $\mathbf{r}$ by simply iterating that equation:

```python
import numpy as np

# links[i] = pages that page i links to
links = {0: [1, 2], 1: [2], 2: [0], 3: [2]}
n, d = 4, 0.85                          # d: probability of following a link

M = np.zeros((n, n))
for i, outs in links.items():
    for j in outs:
        M[j, i] = 1 / len(outs)         # column i spreads page i's score

r = np.full(n, 1 / n)                   # start with equal scores
for _ in range(100):
    r = (1 - d) / n + d * M @ r         # follow a link, or jump at random
print(np.round(r, 3))
```

```
[0.373 0.196 0.394 0.038]
```

Page 2, linked from three pages, ranks highest. Page 0 ranks second even though only one page links to it, because that one page (page 2) is itself important and gives page 0 *all* of its score. Page 3, which nothing links to, gets only the random-jump share. Why the iteration converges, and why the answer is an *eigenvector* of a matrix built from the graph, is the story of Volume 2, Chapter 28 ; the random surfer is a Markov chain, met again in Chapter 29.

Computer-science connection: graphs across computing

- **Compilers:** control-flow graphs (basic blocks and jumps), call graphs, data-dependency DAGs, interference graphs for register allocation. Dead code is code unreachable from the entry vertex, found by DFS.
- **Package managers and build tools:** dependency DAGs and topological sort; version resolution as a constraint problem over the graph.
- **Networks:** routing is shortest paths; resilience is connectivity; spanning trees prevent loops in Ethernet switching (the Spanning Tree Protocol).
- **Distributed systems:** deadlock detection (cycles in the waits-for graph), causal ordering (happened-before is a DAG), gossip and overlay networks (expander graphs that stay connected under failures).
- **Garbage collection:** "mark" is a graph traversal from the roots; anything not reached is garbage.
- **Machine learning:** neural networks are computation DAGs, and backpropagation visits them in reverse topological order; graph neural networks learn on molecular and social graphs.

US curriculum

Adjacency matrices sometimes appear as an application of matrices in **Algebra II** or **Precalculus**, and Euler's polyhedron formula appears in **Geometry**. Graph algorithms (BFS, DFS, Dijkstra, MST, topological sort) belong to college **Data Structures and Algorithms**, and to the USA Computing Olympiad (USACO) for motivated high-school students.

25.18. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Graph algorithms

25.19. Exercises

1. **(Recognition)** A simple graph has 10 vertices, each of degree 3. How many edges does it have? Could a simple graph have 9 vertices, each of degree 3?
2. **(Recognition)** Which of these are trees? (a) 6 vertices, 5 edges, connected. (b) 6 vertices, 5 edges, containing a cycle. (c) 6 vertices, 6 edges, connected.
3. **(Application)** For the graph with edges AB, AC, BC, BD, CD, DE, decide whether it has an Euler circuit, an Euler path, or neither. If a path exists, give one. *Try it:* Check your Euler answer .
4. **(Application)** Run BFS from vertex 1 on the graph with adjacency lists 1: [2, 3], 2: [1, 4, 5], 3: [1, 5], 4: [2, 6], 5: [2, 3, 6], 6: [4, 5]. Give the visit order and the distance of each vertex. *Try it:* Check your BFS answer .
5. **(Application)** Run Dijkstra's algorithm from A on the directed graph with edges A→B 7, A→C 2, C→B 3, B→D 1, C→D 8, D→E 2, C→E 10. Show the table of tentative distances and give the shortest path to E. *Try it:* Check your Dijkstra answer .
6. **(Application)** Find a topological order of the tasks: `fetch` before `compile`; `configure` before `compile`; `compile` before `test` and `package`; `test` before `deploy`; `package` before `deploy`. How many tasks can run in parallel at the first step? *Try it:* Check your build order .
7. **(Problem solving)** A connected planar simple graph has 10 vertices, each of degree 3. How many faces does any plane drawing of it have?
8. **(Problem solving)** Prove that in any graph with at least 2 vertices, two vertices have the same degree. (Hint: Chapter 24.)
9. **(Problem solving, CS)** You store a road network with 50 million intersections and 120 million road segments. Estimate the memory of an adjacency matrix (1 bit per entry) versus adjacency lists (8 bytes per stored neighbour entry, each segment stored twice). Which do you choose?

Answers

1. Handshake lemma: $2|E| = 10 \times 3 = 30$, so $|E| = 15$. With 9 vertices the degree sum would be 27, which is odd and cannot equal $2|E|$: impossible.
2. (a) Yes: connected with $n - 1$ edges. (b) No: a tree has no cycles (such a graph must also be disconnected). (c) No: a tree on 6 vertices has exactly 5 edges.
3. Degrees: A 2, B 3, C 3, D 3, E 1. Four odd vertices (B, C, D, E): neither.
4. Queue process: remove 1 → discover 2, 3; remove 2 → discover 4, 5; remove 3 → (5 already found); remove 4 → discover 6; remove 5, remove 6. Order 1, 2, 3, 4, 5, 6. Distances: 1:0, 2:1, 3:1, 4:2, 5:2, 6:3.
5. Init A0, others ∞. Finish A (0): B 7 (A), C 2 (A). Finish C (2): B min(7, 2+3=5) = 5 ©; D 2+8 = 10 ©; E 2+10 = 12 ©. Finish B (5): D min(10, 5+1=6) = 6 (B). Finish D (6): E min(12, 6+2=8) = 8 (D). Finish E (8). Path: $A \to C \to B \to D \to E$, length $2 + 3 + 1 + 2 = 8$.
6. In-degree 0 at the start: `configure`, `fetch`, so 2 tasks can run in parallel. One order: configure, fetch, compile, package, test, deploy (package and test can also run in parallel once compile finishes).
7. $E = 10 \times 3/2 = 15$. Euler: $F = 2 - V + E = 2 - 10 + 15 = 7$.
8. A simple graph on n vertices has degrees in $\{0, 1, \ldots, n - 1\}$: n values. But 0 and $n - 1$ cannot both occur (a vertex adjacent to all others rules out an isolated vertex). So at most $n - 1$ distinct values are available for n vertices; by the pigeonhole principle two share a degree. (This is the handshake puzzle from Chapter 24.)
9. Matrix: $(5 \times 10^7)^2 = 2.5 \times 10^{15}$ bits $\approx 3.1 \times 10^{14}$ bytes, about 312 TB. Lists: $2 \times 1.2 \times 10^8 \times 8 \approx 1.9 \times 10^9$ bytes, about 2 GB, plus a small per-vertex overhead. Adjacency lists, by a factor of over 100,000: the graph is extremely sparse (average degree under 5).

25.20. Mastery check

1. State the handshake lemma and use it to show that a 7-vertex graph cannot have every vertex of degree 5.
2. A graph has degrees 2, 2, 4, 4, 3, 3 and is connected. Does it have an Euler circuit, an Euler path or neither? Between which vertices?
3. Why can BFS find shortest paths in an unweighted graph but not in a weighted one? Which algorithm do you use instead, and what condition on the weights does it need?
4. Run Dijkstra from S on: S–A 1, S–B 5, A–B 2, A–C 6, B–C 1. Give final distances.
5. Your package manager reports "circular dependency". Which algorithm detected it, and what property of the dependency graph failed?
6. Find the weight of a minimum spanning tree of the complete graph on vertices P, Q, R, S with weights PQ 1, PR 3, PS 4, QR 2, QS 5, RS 6.
7. Show that K_5 is not planar.
8. A compiler has 6 variables whose live ranges pairwise overlap for every pair except (a, f). How many registers are needed to avoid spilling?

Decision rule: All correct without hesitation → move on to Volume 2, Chapter 26 , with matrices and Volume 2, Chapter 28 (PageRank) to come. Stuck on 1–2 → reread the relevant section (Section 25.12 and Section 25.7 cause the most trouble). Stuck on more → work the chapter exercises, and trace BFS and Dijkstra by hand on a small graph of your own before running the code.

Try it: Check your Dijkstra distances ; Check your MST weight .

▼ **Mastery check answers**

1. $\sum \deg(v) = 2|E|$. With 7 vertices of degree 5 the sum is 35, which is odd: impossible.
2. Exactly two odd vertices (the two of degree 3), and the graph is connected, so an Euler path exists, starting at one degree-3 vertex and ending at the other. No Euler circuit.
3. BFS processes vertices in order of number of edges from the start, which equals distance only when every edge costs the same. With weights, a path of more, cheaper edges can be shorter. Use Dijkstra's algorithm, which requires non-negative weights (otherwise Bellman–Ford).
4. Finish S (0): A 1, B 5. Finish A (1): B min(5, 3) = 3, C 7. Finish B (3): C min(7, 4) = 4. Finish C (4). Distances: S 0, A 1, B 3, C 4.
5. A topological sort (e.g. Kahn's algorithm) failed to place every package: the dependency graph is not acyclic. It contains a directed cycle, so no install order satisfies every dependency.
6. Kruskal: PQ 1 (keep), QR 2 (keep), PR 3 (skip, cycle), PS 4 (keep). Three edges for 4 vertices: weight $1 + 2 + 4 = 7$.

7. $V = 5$, $E = \binom{5}{2} = 10$. A simple planar graph with $V \geq 3$ has $E \leq 3V - 6 = 9$. Since $10 > 9$, K_5 is not planar.

8. The interference graph is K_6 minus the edge af. Vertices b, c, d, e and a form K_5, so at least 5 colours are needed; a and f are not adjacent, so they can share a colour, and 5 suffice. Five registers.

Part VIII: Probability and Statistics

Chapter 29. Probability

A medical test is 99% accurate, and you test positive. How worried should you be? A game-show host opens a door you didn't pick: should you switch? Twenty-three people are in a room: is it likely two share a birthday? Each question has a definite answer, and intuition gets all three wrong. This chapter builds the language of probability (sample spaces, events, the axioms and the rules that follow from them), then spends real time on the idea that matters most: ***conditioning****, which means updating a probability when you learn something. Bayes' theorem falls out of the definition in two lines. Along the way you will meet randomized algorithms, a working spam filter, and the arithmetic of Bloom filters, and you will check every answer by simulation.*

Where this chapter sits

You need: fractions, decimals and percentages (Volume 1, Chapter 2), basic algebra (Volume 1, Chapter 4), set language (union, intersection, complement) from Chapter 21, and counting (permutations and combinations) from Chapter 24. One optional aside uses ***e*** from Volume 1, Chapter 12 .

This chapter unlocks: random variables and distributions (Chapter 30), statistics and inference (Chapter 31), and the probabilistic side of machine learning (Chapter 33). Randomized algorithms, hashing and caching analyses all start here.

Why this chapter matters. Almost every decision you make is under uncertainty: whether to take an umbrella, whether a test result means you're ill, whether a bet or an insurance policy is worth it, whether a new deploy is safe. Probability is the arithmetic that keeps those judgements honest, and it is also how genetics predicts inheritance, how engineers rate a system's reliability, and how spam filters, search engines and machine-learning models decide anything at all.

29.1. What probability means

29.1.1. The problem

"There is a 70% chance of rain tomorrow." "This coin lands heads with probability 1/2." "There is a 1 in 10 chance this patch introduces a regression." All three sentences use the same word, but they don't obviously mean the same thing. Tomorrow happens only once, so what could "70% of the time" mean? A coin can be tossed again and again. A patch either has a bug or it doesn't; the uncertainty is in your head, not in the code.

Before computing anything, it pays to know what a probability *is*.

29.1.2. Intuition: two readings of one number

There are two classic interpretations, and you will use both.

Frequentist (long-run frequency). The probability of an event is the fraction of times it happens when you repeat the experiment many, many times under the same conditions. Toss a fair coin 10 times and you might get 7 heads. Toss it 10,000 times and the fraction of heads will be very close to 0.5. The probability is the number that fraction settles down to.

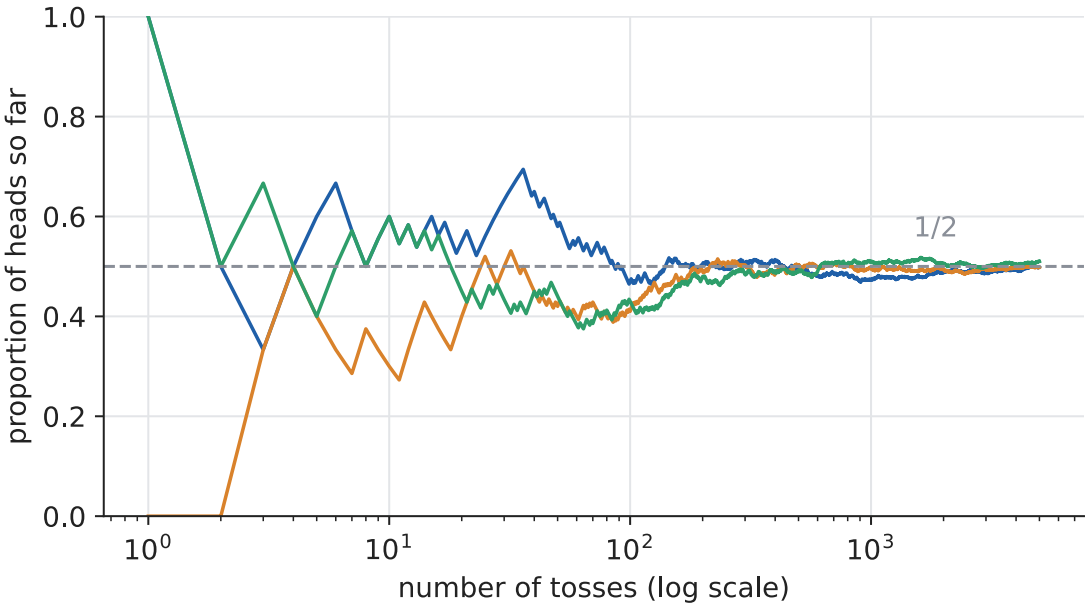


Figure 42. Three runs of 5,000 simulated coin tosses. Early on the proportion of heads swings wildly; as tosses accumulate, every run settles towards 1/2.

Bayesian (degree of belief). The probability of an event is a number between 0 and 1 measuring how strongly you believe it, given what you know. "70% chance of rain" means the forecaster is about as confident of rain as of drawing a red ball from an urn with 7 red and 3 blue. This reading makes sense for one-off events (tomorrow's weather, whether a patch is buggy) and, crucially, tells you how beliefs should change when evidence arrives. That change is Bayes' theorem, coming later in Section 29.9.

The good news: both interpretations obey exactly the same rules. Whatever probability "really" is, the arithmetic in this chapter applies to it. Statisticians argue about interpretation (we return to the argument in Chapter 31); they don't argue about the rules.



Probability, experiment, outcome

A **random experiment** is any process whose result is uncertain: rolling a die, sampling a user, running a randomized algorithm. Each possible result is an **outcome**. A **probability** is a number from 0 to 1 attached to a set of outcomes: 0 means impossible, 1 means certain. Probabilities are often written as fractions ($\frac{1}{4}$), decimals (0.25) or percentages (25%); these are the same number in different clothes (Volume 1, Chapter 2).

Odds are a different way to express the same information: odds of $a{:}b$ in favour means probability $\frac{a}{a+b}$. Probability 0.2 is odds $1{:}4$.



From gambling to mathematics

Probability theory began in 1654 with letters between Blaise Pascal and Pierre de Fermat about the "problem of points": how should the stakes of an unfinished dice game be split fairly? Gamblers had known rough odds for centuries, but nobody had a systematic method. The modern axioms, which turn probability into a branch of mathematics in its own right, were only written down in 1933, by Andrey Kolmogorov.

29.2. Sample spaces and events

29.2.1. The problem

To say anything precise you need to pin down *what can happen*. "Roll two dice; what is the probability of a total of 8?" requires a list of everything that could happen, and a way to pick out the cases that count as "total of 8". Set language from Chapter 21 does exactly this.

29.2.2. Vocabulary and notation



Sample space and event

The **sample space**, written Ω (capital Greek *omega*) or S, is the set of all possible outcomes of an experiment.

An **event** is a subset of the sample space: a collection of outcomes. We say "event A occurs" if the outcome that actually happens is a member of A.

$P(A)$, read "the probability of A", is the probability that event A occurs.

Examples. One die: $\Omega = \{1, 2, 3, 4, 5, 6\}$; "roll is even" is the event $\{2, 4, 6\}$. One coin tossed twice: $\Omega = \{HH, HT, TH, TT\}$; "at least one head" is $\{HH, HT, TH\}$.

Because events are sets, all the set operations you know become ways of combining events. The translation table is worth memorising, because every probability word problem is written in the left-hand column and every calculation happens in the right-hand one.

In words	Set notation	Meaning
"not A"	A^c (also $\bar{A}$, A')	the **complement**: outcomes not in A
"A or B" (or both)	$A \cup B$	**union**: outcomes in at least one
"A and B"	$A \cap B$	**intersection**: outcomes in both
"A and B can't both happen"	$A \cap B = \varnothing$	**disjoint**, or **mutually exclusive**
"if A happens, so does B"	$A \subseteq B$	A is a **subset** of B
"something happens" / "nothing"	Ω / $\varnothing$	the certain event / the impossible event

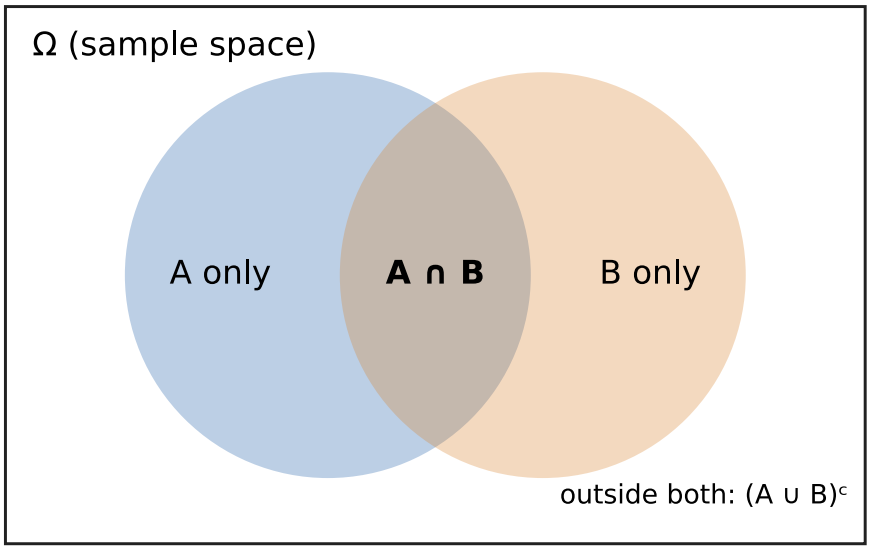


Figure 43. Events as regions of the sample space. The rectangle is everything that could happen; each circle is an event.

Complement, union, intersection, mutually exclusive

Complement A^c: everything in Ω not in A. **Union** $A \cup B$: "A or B or both" (mathematical "or" is always inclusive). **Intersection** $A \cap B$: "both A and B". Events are **mutually exclusive** (or **disjoint**) if they share no outcomes, $A \cap B = \varnothing$. The symbol $\varnothing$ is the empty set.

29.2.3. Worked example: two dice

Roll a red die and a blue die. An outcome is a pair (r, b) with $r, b \in \{1, \ldots, 6\}$, so by the product rule (Chapter 24) the sample space has $6 \times 6 = 36$ outcomes. Picture them as a grid: *Try it:* Two-dice events on a grid .

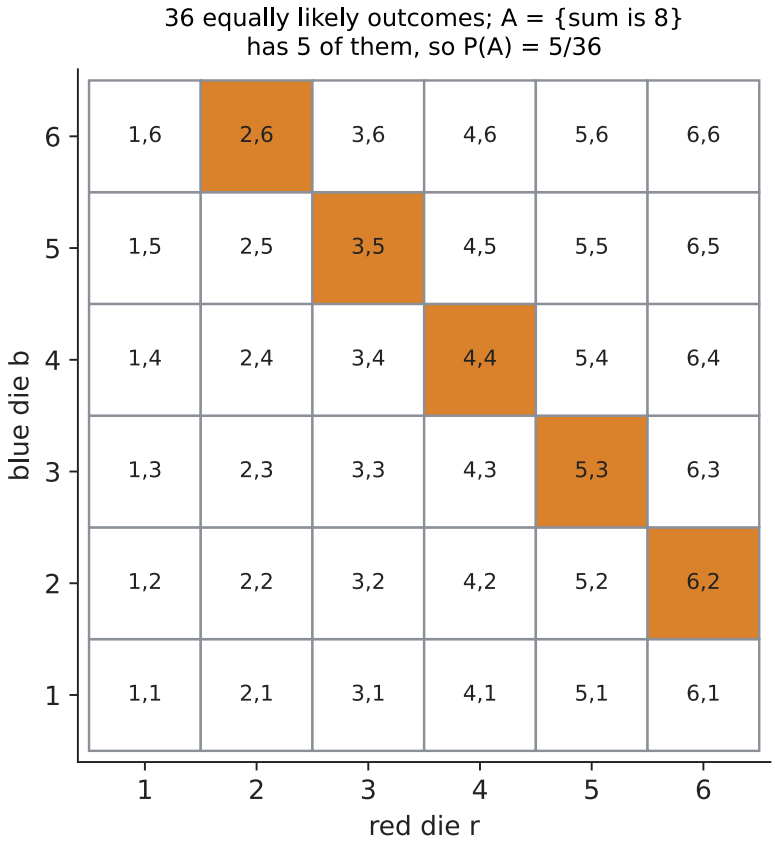


Figure 44. The sample space for two dice as a 6 × 6 grid. The shaded squares form the event "the total is 8".

Let A = "total is 8" $= \{(2,6),(3,5),(4,4),(5,3),(6,2)\}$, and D = "doubles" $= \{(1,1),(2,2),\ldots,(6,6)\}$. Then

- $A \cap D = \{(4,4)\}$: a total of 8 *and* doubles;
- $A \cup D$ has $5 + 6 - 1 = 10$ outcomes (subtract the one counted twice);
- D^c is "not doubles", with $36 - 6 = 30$ outcomes.

Notice how the question "how many outcomes?" is already doing most of the work. Probability, in the simplest setting, is counting.



Choosing the sample space

The same experiment can be described by different sample spaces, and a poor choice causes errors. For two dice you could take $\Omega = \{2,3,\ldots,12\}$ (the possible totals). That's a legitimate sample space, but its outcomes are *not* equally likely: a total of 7 can happen six ways, a total of 2 only one way. Choose a sample space whose outcomes are equally likely whenever you plan to count, and keep the dice distinguishable (red and blue) even if they look identical.

29.3. The axioms and the basic rules

29.3.1. The problem

We need rules that every assignment of probabilities must obey, whether the numbers come from counting, from long-run frequencies or from belief. Remarkably, three rules suffice, and everything else in this chapter is derived from them.

29.3.2. The axioms



Kolmogorov's axioms

A **probability** assigns to each event A a number $P(A)$ such that

1. $P(A) \geq 0$ for every event A (no negative probabilities);
2. $P(\Omega) = 1$ (something certainly happens);
3. if A and B are mutually exclusive, $P(A \cup B) = P(A) + P(B)$. (The full axiom says this for any countable list of pairwise disjoint events.)

An **axiom** is a starting rule we accept without proof; everything else is derived from the axioms.

These match intuition for frequencies. A fraction of trials can't be negative; the fraction of trials in which *something* happens is 1; and if A and B never happen together, the fraction of trials in which one of them happens is the fraction for A plus the fraction for B.

29.3.3. Why the other rules follow

The complement rule. A and A^c are disjoint and together make up Ω. So

$$\begin{aligned} P(A) + P(A^c) &= P(A \cup A^c) && \text{(axiom 3: disjoint)} \\ &= P(\Omega) && \text{(together they are everything)} \\ &= 1 && \text{(axiom 2)} \end{aligned}$$

and therefore $P(A^c) = 1 - P(A)$. As special cases, $P(\varnothing) = 1 - P(\Omega) = 0$, and since $P(A^c) \geq 0$, $P(A) \leq 1$.

The addition rule (general). If A and B overlap, adding $P(A) + P(B)$ counts the overlap $A \cap B$ twice. Split the union into three disjoint pieces: "A only", "B only", "both". Then

$$\begin{aligned} P(A) &= P(A \text{ only}) + P(A \cap B) \\ P(B) &= P(B \text{ only}) + P(A \cap B) \\ P(A \cup B) &= P(A \text{ only}) + P(B \text{ only}) + P(A \cap B) \\ &= \big[P(A) - P(A \cap B)\big] + \big[P(B) - P(A \cap B)\big] + P(A \cap B) \\ &= P(A) + P(B) - P(A \cap B). \end{aligned}$$

This is inclusion–exclusion from Chapter 24, with probabilities in place of counts.



Complement rule and addition rule

Complement rule: $P(A^c) = 1 - P(A)$.

Addition rule: $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. If A and B are mutually exclusive, the last term is 0.

Monotonicity: if $A \subseteq B$ then $P(A) \leq P(B)$. (A more specific event can't be more probable.)

29.3.4. Worked examples

Example 1 (complement). What is the probability of at least one six in four rolls of a die? "At least one" has many cases (exactly one six, exactly two, ...). The complement, "no sixes", is one case. Each roll avoids six in 5 of 6 ways, so by counting, $P(\text{no sixes}) = \frac{5^4}{6^4} = \frac{625}{1296}$. Then

$$P(\text{at least one six}) = 1 - \frac{625}{1296} = \frac{671}{1296} \approx 0.518.$$

Slightly better than even. This was one of the bets that sparked the Pascal–Fermat letters.

Example 2 (addition rule). In a codebase, 30% of files have no tests, 20% have a lint warning, and 8% have both. What fraction of files have at least one of these problems?

$$\begin{aligned} P(\text{no tests} \cup \text{lint}) &= P(\text{no tests}) + P(\text{lint}) - P(\text{both}) \\ &= 0.30 + 0.20 - 0.08 \\ &= 0.42. \end{aligned}$$

So 58% of files are clean.



Adding probabilities that overlap

"There's a 50% chance of rain on Saturday and 50% on Sunday, so it's certain to rain this weekend." Wrong: the events aren't mutually exclusive, so you must subtract the overlap. If the days were independent (Section 29.7) the chance of rain at some point would be $1 - 0.5 \times 0.5 = 0.75$. A quick sanity check: any calculation that produces a probability above 1 has added overlapping events.

29.3.5. Where you'd use this

- **Everyday: reading betting odds.** Fractional odds of "3/1" mean you win 3 for every 1 staked, which is a fair bet only if the probability is $1/(3+1) = 0.25$. For a football match priced home 6/4, draw 9/4, away 7/4, the implied probabilities are $\frac{4}{10} = 0.400$, $\frac{4}{13} \approx 0.308$ and $\frac{4}{11} \approx 0.364$. The three outcomes are mutually exclusive and exhaustive, so the axioms say true probabilities must add to 1; these add to about 1.071. That extra 7% (the **overround**) is the bookmaker's built-in margin, and the complement-and-addition rules are all you need to spot it.
- **Science:** a "70% chance of rain" from a modern forecast typically means that in roughly 70% of an **ensemble** of slightly perturbed simulations, rain fell at your location: the frequentist reading applied to computer runs. Geneticists, epidemiologists and insurers all start from the same three

axioms.

- **Computing:** a classifier's output layer (such as softmax) must produce non-negative numbers that sum to 1 across the possible labels; if your "probabilities" sum to 1.3, you have a bug, and the addition rule is the unit test. The complement rule is the standard trick for "at least one" questions in reliability and testing: "probability that at least one of 50 flaky tests fails" is $1 - P(\text{all pass})$.

> **US curriculum**
>
> Sample spaces, simple events and the complement rule appear in middle school (grade 7, Common Core 7.SP). The addition rule and "or/and" language belong to the high-school **Statistics and Probability** strand (S-CP), usually taught inside **Algebra II** or **Geometry**, and revisited in **AP Statistics**.

29.4. Equally likely outcomes: probability as counting

29.4.1. The problem

For fair dice, shuffled cards and random selections, every outcome in a well-chosen sample space has the same probability. Then probability reduces to counting. But the counts can be large, which is why Chapter 24 came first.

29.4.2. Why it works

Suppose Ω has N outcomes, all equally likely, each with probability p. The outcomes are disjoint and together make up Ω, so by axioms 2 and 3, $Np = 1$ and $p = 1/N$. An event A with $|A|$ outcomes is the disjoint union of those outcomes, so

$$P(A) = \underbrace{\frac{1}{N} + \frac{1}{N} + \cdots + \frac{1}{N}}_{|A| \text{ terms}} = \frac{|A|}{|\Omega|} = \frac{\text{number of favourable outcomes}}{\text{total number of outcomes}}.$$

Example 36. Procedure — probability by counting

Input: an experiment and an event A.

Steps:

1. Choose a sample space Ω whose outcomes are equally likely. (Make objects distinguishable; decide whether order matters and stay consistent.)
2. Count $|\Omega|$.
3. Count the outcomes in A, using the *same* convention (ordered or unordered) as step 2.
4. If A is awkward to count, count A^c instead and use $P(A) = 1 - P(A^c)$.
5. Compute $P(A) = |A|/|\Omega|$ and sanity-check that it lies in $[0, 1]$.

Output: $P(A)$.

29.4.3. Worked example: a poker hand

What is the probability that a 5-card hand dealt from a shuffled 52-card deck contains exactly two aces? *Try it:* Two aces, computed .

Step 1. Outcomes: unordered 5-card hands, all equally likely.

Step 2. $|\Omega| = \binom{52}{5} = 2\,598\,960$.

Step 3. Build a favourable hand in two stages: choose 2 of the 4 aces, then 3 of the 48 non-aces.

$$\begin{aligned} |A| &= \binom{4}{2}\binom{48}{3} && \text{(product rule)} \\ &= 6 \times 17\,296 \\ &= 103\,776. \end{aligned}$$

Step 5. $P(A) = \frac{103\,776}{2\,598\,960} \approx 0.0399$, about 4%.

29.4.4. Worked example: a defective batch

A batch of 20 chips has 3 defective ones. You test 4 chosen at random. What is the probability you catch at least one defective chip? *Try it:* Defective batch sampler .

Counting "at least one" directly means cases 1, 2 and 3. The complement, "no defectives", is one case: all 4 chosen from the 17 good chips.

$$\begin{aligned} P(\text{no defective}) &= \frac{\binom{17}{4}}{\binom{20}{4}} = \frac{2380}{4845} \approx 0.491 \\ P(\text{at least one}) &= 1 - 0.491 = 0.509. \end{aligned}$$

Testing 4 of 20 chips gives only about a coin-flip's chance of spotting a problem that affects 15% of the batch. Quality-control sampling plans are designed with exactly this calculation.

29.4.5. Where you'd use this

- **Everyday: lotteries.** The UK Lotto draws 6 numbers from 59, so there are $\binom{59}{6} = 45\,057\,474$ equally likely tickets and one jackpot: a probability of about 1 in 45 million per ticket. The US Powerball (5 from 69, plus 1 from 26) has $\binom{69}{5} \times 26 = 292\,201\,338$ combinations. Buying 100 tickets a week for 50 years (260,000 tickets) still gives less than a 0.1% chance of a Powerball jackpot ($260\,000/292\,201\,338 \approx 0.0009$).
- **Science:** in a lab, choosing 4 mice at random from 20 for a treatment group is the same counting problem as the chip batch; **random assignment** guarantees every group of 4 was equally likely, which is what lets the statistics in Chapter 31 work. Geneticists count equally likely allele combinations in a Punnett square.
- **Computing:** the chance that a brute-force attacker guesses a random 8-character password drawn from 62 letters and digits on one try is $1/62^8 \approx 4.6 \times 10^{-15}$; password-strength meters are counting arguments. Random sampling for audits and canary deploys ("send 1% of traffic to the new version") relies on every unit being equally likely to be picked.



Equally likely is an assumption

$|A|/|\Omega|$ only works if outcomes are equally likely. "Either I win the lottery or I don't, so it's 50–50" uses a two-outcome sample space whose outcomes are wildly unequal. Likewise, mixing ordered counts in the numerator with unordered counts in the denominator gives nonsense. A safe habit: write the words "ordered" or "unordered" next to both counts.

29.5. Conditional probability: what conditioning means

This is the most important section of the chapter. The notation $P(A \mid B)$ looks innocent, but the idea behind it (how a probability changes when you learn something) underlies diagnostics, spam filters, recommendation systems, A/B tests and most of machine learning.

29.5.1. The problem

Roll two dice. The probability that the total is 8 is $5/36 \approx 0.139$. Now a friend peeks at the red die and tells you "it's even". Should your probability for "total is 8" change? Intuitively yes: an even red die rules out some ways of getting 8 (such as $(3, 5)$) and keeps others (such as $(2, 6)$). But by how much?

29.5.2. Intuition: the information shrinks the world

Before the hint, any of 36 outcomes could be the true one. After "the red die is even", only the 18 outcomes with $r \in \{2, 4, 6\}$ are still possible. The other 18 are ruled out. Nothing has changed about the relative chances of the outcomes that remain: they were equally likely before, and nothing in the hint favours one over another, so they are still equally likely. The new sample space is just smaller. *Try it:* Watch conditioning shrink the world .

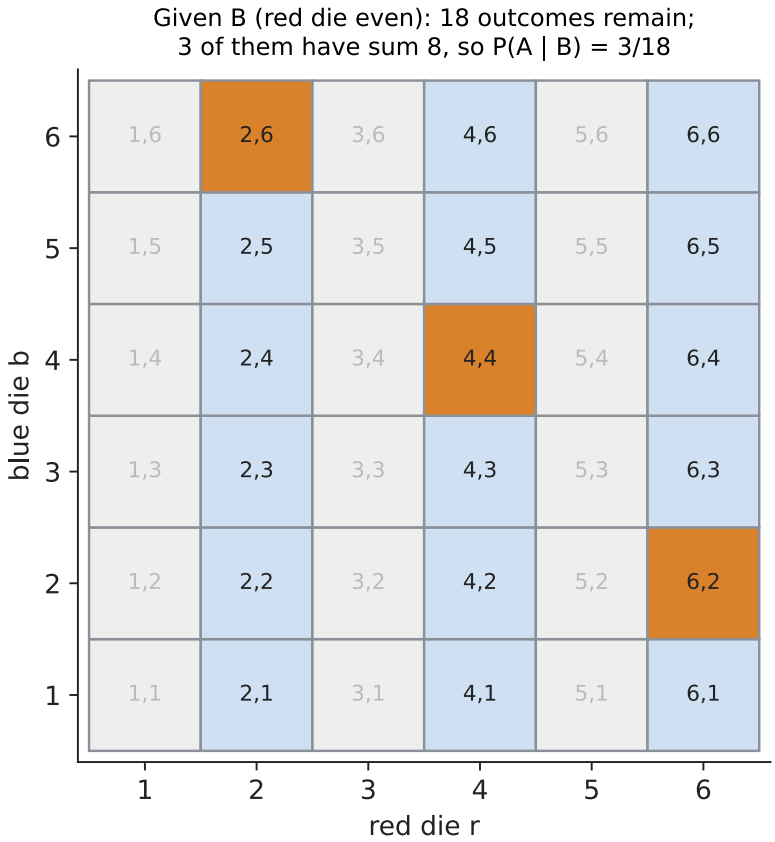


Figure 45. Conditioning on B = "red die is even". Greyed outcomes are ruled out; of the 18 that remain, 3 give a total of 8.

Among the 18 survivors, the totals of 8 are $(2,6)$, $(4,4)$, $(6,2)$: three of them. So the updated probability is $3/18 = 1/6 \approx 0.167$. The hint made "total is 8" a little more likely.

That's all conditioning is: **restrict attention to the outcomes consistent with what you know, and re-measure the event as a fraction of that restricted world.**

29.5.3. Why the definition looks the way it does

In the dice example the calculation was

$$P(A \mid B) = \frac{\text{outcomes in both } A \text{ and } B}{\text{outcomes in } B} = \frac{|A \cap B|}{|B|} = \frac{3}{18}.$$

Divide top and bottom by $|\Omega| = 36$:

$$\frac{|A \cap B|}{|B|} = \frac{|A \cap B|/36}{|B|/36} = \frac{P(A \cap B)}{P(B)} = \frac{3/36}{18/36}.$$

That last form doesn't mention counting at all, so it makes sense even when outcomes aren't equally likely. It becomes the definition.



Conditional probability

For events A and B with $P(B) > 0$, the **conditional probability of A given B** is

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

Read the vertical bar $\mid$ as "given". B is the **conditioning event** (what you know); A is the event you're asking about. $P(A)$, without a condition, is sometimes called the **unconditional** or **marginal** probability.

Meaning: of all the probability sitting on B, what fraction also lies in A?
Geometrically: zoom into B and rescale it to have total probability 1.

Read the formula as an instruction: *zoom in on B, then measure A's share of it*. The division by $P(B)$ is a rescaling so that the new, smaller world has total probability 1 again. You can check that $P(\cdot \mid B)$ satisfies all three axioms: it is a genuine probability, just on a smaller world.

29.5.4. Two-way tables: conditioning by rows and columns

Real data often arrives as a table of counts, and then conditioning is literally "restrict to a row or column". Here are 1,000 deploys of a service, classified by whether they were made on a Friday and whether they caused an incident:

	Incident	No incident	Total
Friday	24	176	200
Not Friday	40	760	800
Total	64	936	1000

- Unconditional: $P(\text{incident}) = 64/1000 = 0.064$.
- Given Friday, restrict to the Friday row: $P(\text{incident} \mid \text{Friday}) = 24/200 = 0.12$.
- Given not Friday: $P(\text{incident} \mid \text{not Friday}) = 40/800 = 0.05$.
- Reverse the question, restrict to the Incident column: $P(\text{Friday} \mid \text{incident}) = 24/64 = 0.375$.

Check the definition gives the same thing: $P(\text{incident} \cap \text{Friday})/P(\text{Friday}) = 0.024/0.2 = 0.12$. ✓

Friday deploys are more than twice as likely to cause incidents. (Whether Fridays *cause* incidents is a different question; see correlation versus causation in Chapter 31.)

Example 37. Procedure — computing P(A | B)

Input: events A and B, and a way to find probabilities (counts, a table, or given numbers).

Steps:

1. Identify which event is *known* (it goes after the bar) and which is *asked about*. Rewrite the question as "of the cases where [known], what fraction have [asked]?"
2. Find $P(B)$, or count $|B|$ in a table or equally-likely sample space.
3. Find $P(A \cap B)$, or count $|A \cap B|$.
4. Divide: $P(A \mid B) = P(A \cap B)/P(B)$.

Output: $P(A \mid B)$.

29.5.5. Worked example: two children

A family has two children. Assume each child is independently a boy or a girl with probability 1/2. Given that *at least one* is a girl, what is the probability that both are girls?

Sample space (ordered by birth, so equally likely): $\{BB, BG, GB, GG\}$.

1. Known: B = "at least one girl" $= \{BG, GB, GG\}$. Asked: A = "both girls" $= \{GG\}$.
2. $P(B) = 3/4$.
3. $A \cap B = \{GG\}$, so $P(A \cap B) = 1/4$.
4. $P(A \mid B) = \dfrac{1/4}{3/4} = \dfrac{1}{3}$.

Not 1/2, which is most people's first answer. The information "at least one girl" removes only BB, leaving three equally likely worlds, and only one of them is GG. If instead you're told "the *elder* child is a girl", the world shrinks to $\{GB, GG\}$ and the answer *is* 1/2. Different information, different restricted world, different answer. Conditioning is sensitive to exactly what you learned.

29.5.6. Where you'd use this

- **Everyday:** conditional probabilities are what you actually want from your own records. If over a year you were late on 9 of 30 rainy days but only 20 of 200 dry ones, then $P(\text{late} \mid \text{rain}) = 0.3$ while $P(\text{late} \mid \text{dry}) = 0.1$: leave earlier when it rains. Car-insurance premiums are set the same way, from the claim rate **given** the driver's age band and postcode.
- **Science and medicine:** survival rates, side-effect rates and test accuracy are all conditional ("5-year survival **given** diagnosis at stage 1"), and quoting one without its condition is the most common way statistics mislead.
- **Computing: "customers who bought this also bought".** Suppose 2,000 of 10,000 supermarket baskets contain pasta, 800 of those also contain parmesan, and 500 baskets in total contain parmesan. Then $P(\text{parmesan} \mid \text{pasta}) = 800/2000 = 0.4$, against

$P(\text{parmesan}) = 500/10\,000 = 0.05$ overall: buying pasta makes parmesan 8 times as likely. Association-rule mining calls 0.4 the rule's **confidence** and the factor 8 its **lift**; recommendation engines and shelf layouts are built from such numbers.



Confusing P(A | B) with P(B | A)

$P(\text{incident} \mid \text{Friday}) = 0.12$ but $P(\text{Friday} \mid \text{incident}) = 0.375$. These are different questions, divided by different totals (a row versus a column). Swapping them is called the **confusion of the inverse**, and in court it's the **prosecutor's fallacy**: "If the defendant were innocent, there's only a 1 in a million chance the DNA would match, so there's only a 1 in a million chance he's innocent." The first probability is $P(\text{match} \mid \text{innocent})$; the jury needs $P(\text{innocent} \mid \text{match})$, which depends on how many other people could have matched. In a city of 10 million, around 10 innocent people would match. Bayes' theorem (below) is the tool for turning one into the other.

29.6. The multiplication rule and tree diagrams

29.6.1. The problem

Often you know conditional probabilities and want a joint one. A bag holds 5 red and 3 blue marbles. Draw two without replacement. What's the chance both are red? The second draw depends on the first.

29.6.2. Why it works

Multiply both sides of the definition of conditional probability by $P(B)$:

$$P(A \cap B) = P(B)\,P(A \mid B) = P(A)\,P(B \mid A).$$

This is the **multiplication rule**: the chance that both happen is the chance that the first happens times the chance the second happens *given* the first. For three events it chains:
$P(A \cap B \cap C) = P(A)\,P(B \mid A)\,P(C \mid A \cap B)$.

For the marbles, with R_1 = "first is red", R_2 = "second is red":

$$\begin{aligned} P(R_1 \cap R_2) &= P(R_1)\,P(R_2 \mid R_1) \\ &= \frac{5}{8} \times \frac{4}{7} && \text{(after one red is gone: 4 red of 7 left)} \\ &= \frac{20}{56} = \frac{5}{14} \approx 0.357. \end{aligned}$$

Check by counting: $\binom{5}{2}/\binom{8}{2} = 10/28 = 5/14$. ✓ Two different methods, same answer.

29.6.3. Tree diagrams

A **tree diagram** draws the multiplication rule. Each level is a stage of the experiment; each branch carries the conditional probability of that branch given the path so far. The probability of a complete path is the product of the probabilities along it, and the probability of an event is the sum of the paths that make it up.

Example 38. Procedure — tree diagram

Input: a multi-stage experiment.

Steps:

1. Draw one level per stage. At each node, draw a branch for every possible result of the next stage.
2. Label each branch with its probability *conditional on the path leading to it*. Branches leaving any node must sum to 1.
3. Multiply along each root-to-leaf path to get the probability of that path (multiplication rule).
4. Add the probabilities of all paths in your event (addition rule: paths are disjoint).

Output: the probability of the event.

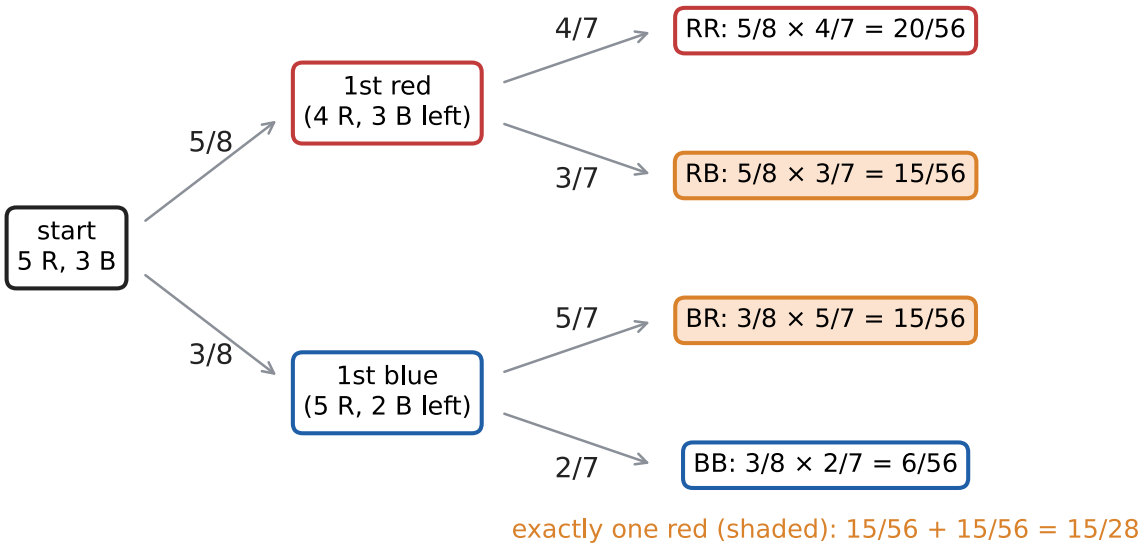


Figure 46. The tree for drawing two marbles without replacement from a bag of 5 red and 3 blue. Each branch carries a probability conditional on the path so far; multiply along a path to get its probability, and add paths to get an event. The shaded leaves make up "exactly one red".

For the marbles, "exactly one red" consists of two paths, RB and BR:

$$P(\text{RB}) = \tfrac{5}{8} \times \tfrac{3}{7} = \tfrac{15}{56}$$
$$P(\text{BR}) = \tfrac{3}{8} \times \tfrac{5}{7} = \tfrac{15}{56}$$
$$P(\textbf{exactly one red}) = \tfrac{15}{56} + \tfrac{15}{56} = \tfrac{30}{56} = \tfrac{15}{28}.$$

And $P(\text{BB}) = \tfrac{3}{8} \times \tfrac{2}{7} = \tfrac{6}{56}$. The four paths sum to $\tfrac{20+15+15+6}{56} = 1$. ✓

29.6.4. Where you'd use this

A tree diagram is the right tool whenever an outcome happens in stages. **Everyday:** the chance of being dealt two aces from a shuffled deck is $\frac{4}{52} \times \frac{3}{51} = \frac{12}{2652} = \frac{1}{221}$, about 0.45%, which is why poker players treat it as a once-in-an-evening event. **Science:** a diagnostic pathway (screen, then confirm, then treat) is a tree, as is a multi-stage chemical synthesis whose overall yield is the product of the stage yields. **Computing:** a funnel in product analytics (visit, then sign up, then pay) multiplies conditional conversion rates; 20% sign up and 10% of those pay gives $0.2 \times 0.1 = 2\%$ of visitors paying.

29.7. Independence

29.7.1. The problem

Sometimes learning B tells you nothing about A. Knowing the red die shows 3 says nothing about the blue die. Knowing it rained in Tokyo says nothing about a coin toss in London. We need a word for this, because it simplifies calculations enormously, and because wrongly assuming it causes some of the worst mistakes in applied probability.

29.7.2. Definition and intuition

A and B are **independent** if conditioning on B doesn't change the probability of A: $P(A \mid B) = P(A)$. Substituting the definition of conditional probability and multiplying by $P(B)$ gives the symmetric form usually taken as the definition.



Independent events

Events A and B are **independent** if

$$P(A \cap B) = P(A)\,P(B).$$

Equivalently (when $P(B) > 0$), $P(A \mid B) = P(A)$: knowing B happened doesn't change the probability of A. Events that aren't independent are **dependent**.

Several events $A_1, \dots, A_n$ are **mutually independent** if the product rule holds for every sub-collection, e.g. $P(A_1 \cap A_2 \cap A_3) = P(A_1)P(A_2)P(A_3)$.

In the dice example, "total is 8" and "red is even" are *dependent*: $P(A \mid B) = 1/6 \neq 5/36 = P(A)$. In the deploy table, "incident" and "Friday" are dependent: $0.12 \neq 0.064$.

By contrast, let C = "total is 7" and B = "red is even". The totals of 7 with even red are $(2,5), (4,3), (6,1)$, so $P(C \mid B) = 3/18 = 1/6$, and $P(C) = 6/36 = 1/6$. Equal, so C and B *are* independent. Independence is a numerical fact, not always something you can see in advance.

29.7.3. Applications: reliability

Independence is how engineers compute reliability. Suppose each of three servers is up 99% of the time, independently. *Try it:* Build your own reliability diagram ; Three replicas in parallel ; Three servers in series .

- **In series** (the request needs all three, e.g. a chain of microservices): $P(\text{all up}) = 0.99^3 \approx 0.970$. Chaining services multiplies their unreliability.
- **In parallel** (any one suffices, e.g. redundant replicas): use the complement. $P(\text{all down}) = 0.01^3 = 10^{-6}$, so $P(\text{at least one up}) = 1 - 10^{-6} = 0.999999$: "six nines".

29.7.4. More applications: genetics and everyday risk

- **Science: inheritance.** Cystic fibrosis is recessive: a child is affected only if it inherits the faulty allele from both parents. If both parents are carriers, each child independently has probability $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$ of being affected. For a family of three children, $P(\text{none affected}) = (\frac{3}{4})^3 = \frac{27}{64}$, so $P(\text{at least one affected}) = 1 - \frac{27}{64} = \frac{37}{64} \approx 0.58$. Genetic counsellors give exactly these numbers. Mendel's peas worked because separate genes are (mostly) inherited independently.
- **Everyday:** a thief with your phone gets 3 guesses at a random 4-digit PIN: $3/10\,000 = 0.03\%$. Those guesses aren't independent trials (a sensible thief never repeats a guess), which is why the answer is exactly $3/10\,000$ rather than $1 - 0.9999^3$; the two differ only in the eighth decimal place here, but the habit of asking "independent or not?" is the point.
- **Engineering:** twin-engine aircraft, dual brake circuits in cars and backup generators in hospitals are the "parallel" calculation above, and the reason regulators insist the redundant parts share as little as possible (separate fuel lines, separate wiring) is precisely to keep failures independent.



Independent is not the same as disjoint

Mutually exclusive events with positive probability are always *dependent*: if A and B are disjoint and B happens, A certainly didn't, so $P(A \mid B) = 0 \neq P(A)$. "Can't happen together" and "don't influence each other" are opposite ideas, not the same one.

The other classic trap is **assuming** independence. The replica calculation above fails badly if the three servers share a power supply, a network switch or a buggy config push: then failures are correlated and $P(\text{all down})$ might be 10^{-3}, not 10^{-6}. The 2008 financial crisis was, in part, a correlated-failure miscalculation: mortgage defaults were modelled as nearly independent, and they weren't.



The gambler's fallacy

On 18 August 1913, at the casino in Monte Carlo, a roulette wheel came up black 26 times in a row. Gamblers lost millions betting on red, reasoning that red was "due". But successive spins are independent: the wheel has no memory, and after 25 blacks the next spin is still red with probability 18/37. Believing that independent events "balance out" in the short run is the **gambler's fallacy**. The law of large numbers (Chapter 30) says the *proportion* evens out over the long run, but it does so by swamping early streaks with many later spins, not by correcting them.

29.8. The law of total probability

29.8.1. The problem

A factory has two production lines. Line 1 makes 60% of the chips and 2% of its chips are defective; line 2 makes 40% and 5% of its chips are defective. What fraction of all chips is defective? You know the defect rate *conditional* on each line; you want the overall rate.

29.8.2. Why it works

The lines split the sample space into disjoint pieces that cover everything. Every defective chip came from exactly one line, so

$$\begin{aligned} P(D) &= P(D \cap L_1) + P(D \cap L_2) && \text{(disjoint pieces of } D\text{)} \\ &= P(L_1)P(D \mid L_1) + P(L_2)P(D \mid L_2) && \text{(multiplication rule)} \\ &= 0.6 \times 0.02 + 0.4 \times 0.05 \\ &= 0.012 + 0.020 = 0.032. \end{aligned}$$

The overall defect rate is 3.2%, a **weighted average** of the line rates with weights equal to each line's share.



Partition and the law of total probability

Events $B_1, B_2, \ldots, B_n$ form a **partition** of Ω if they are mutually exclusive and together cover Ω: every outcome lies in exactly one B_i. The **law of total probability** says, for any event A,

$$P(A) = \sum_{i=1}^{n} P(B_i)\, P(A \mid B_i).$$

Here $\sum_{i=1}^{n}$ ("sigma") means "add the following expression for $i = 1, 2, \ldots, n$". In a tree diagram, this is "add up every path that ends in A". Simplest case:
$P(A) = P(B)P(A \mid B) + P(B^c)P(A \mid B^c)$.

The law is a "divide and conquer" strategy: when $P(A)$ is hard, split the world into cases where it's easy, and recombine with weights.

29.8.3. Where you'd use this

- **Everyday:** you take the train 70% of the time and are late 10% of those days; you take the bus otherwise and are late 25% of those days. Then $P(\text{late}) = 0.7 \times 0.10 + 0.3 \times 0.25 = 0.07 + 0.075 = 0.145$: about one day in seven.
- **Medicine:** what fraction of everyone tested will see a positive result? Split by whether they have the disease: $0.01 \times 0.99 + 0.99 \times 0.05 = 0.0099 + 0.0495 = 0.0594$, so 5.94% of tests come back positive. That denominator is exactly the 594 positives out of 10,000 in the next section, where it becomes the bottom line of Bayes' theorem.

- **Computing:** the overall error rate of a service running in several regions is the traffic-weighted average of the regional error rates, and the overall false-positive rate of a classifier on a mixed dataset is the weighted average over its sub-populations. When an aggregate number looks surprising (a "better" model that is worse in every subgroup, **Simpson's paradox**, Chapter 31), the cause is almost always a change in the weights.

29.9. Bayes' theorem

29.9.1. The problem

A disease affects 1% of people. A test for it has a 99% **true-positive rate** (if you have the disease, it's positive 99% of the time) and a 5% **false-positive rate** (if you don't, it's still positive 5% of the time). You test positive. What is the probability you have the disease? *Try it:* The test as 1,000 people .

Most people, including, in published studies, most doctors, answer something like 95% or 99%. The true answer is about 17%. Let's see why, two ways.

29.9.2. Intuition: natural frequencies

Imagine 10,000 people taking the test. Replace each probability with a count: *Try it:* Medical test as frequencies .

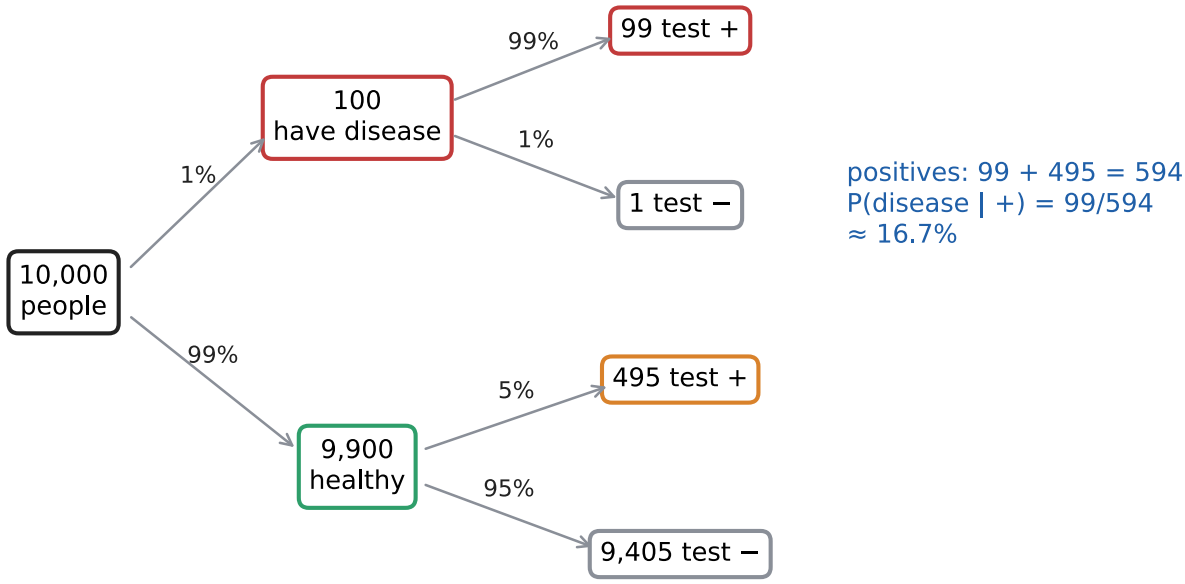


Figure 47. The medical test as natural frequencies. Of 594 positive results, only 99 come from people who actually have the disease.

1. 1% have the disease: 100 people. 9,900 don't.
2. Of the 100 with the disease, 99% test positive: 99 people.
3. Of the 9,900 healthy people, 5% test positive anyway: 495 people.
4. Total positives: $\mathbf{99 + 495 = 594}$.

You are one of the 594 positives. Of those, 99 are sick. So

$$P(\text{disease} \mid +) = \frac{99}{594} = \frac{1}{6} \approx 16.7\%.$$

The test is good, but the disease is rare. Healthy people are 99 times as numerous as sick ones, so even a small false-positive rate among the healthy produces five times as many false alarms as true alarms. Conditioning on "positive" restricts you to the 594 positives, and in that restricted world, sick people are a minority. This is the **base-rate fallacy**: ignoring how common the condition was in the first place (its **base rate** or **prior**).

29.9.3. Why it works: deriving Bayes' theorem

Nothing new is needed; Bayes' theorem is the definition of conditional probability, used twice. The multiplication rule gives $P(A \cap B)$ two ways:

$$P(A \cap B) = P(A \mid B)\,P(B) \qquad \text{and} \qquad P(A \cap B) = P(B \mid A)\,P(A).$$

Set them equal and divide by $P(B)$:

$$\begin{aligned} P(A \mid B)\,P(B) &= P(B \mid A)\,P(A) \\ P(A \mid B) &= \frac{P(B \mid A)\,P(A)}{P(B)}. \end{aligned}$$

The denominator $P(B)$ is usually not given directly, so expand it with the law of total probability over A and A^c:

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B \mid A)\,P(A) + P(B \mid A^c)\,P(A^c)}.$$

That’s Bayes' theorem. It "inverts" a conditional probability: from $P(B \mid A)$ (which a lab can measure: how does the test behave on sick people?) to $P(A \mid B)$ (which the patient wants: given my result, am I sick?).

Bayes' theorem: prior, likelihood, posterior

$$\underbrace{P(H \mid E)}_{\text{posterior}} = \frac{\overbrace{P(E \mid H)}^{\text{likelihood}}\overbrace{P(H)}^{\text{prior}}}{\underbrace{P(E)}_{\text{evidence}}}$$

H is a **hypothesis** (has the disease), E the **evidence** (tested positive).

- **Prior** $P(H)$: how plausible H was before the evidence (the base rate).
- **Likelihood** $P(E \mid H)$: how probable the evidence is if H is true.
- **Posterior** $P(H \mid E)$: how plausible H is after seeing the evidence.
- **Evidence** or **marginal likelihood** $P(E)$: overall probability of the evidence, from the law of total probability.

For tests: **sensitivity** = $P(+ \mid \text{disease})$ (true-positive rate); **specificity** = $P(- \mid \text{healthy})$; false-positive rate = $1-$ specificity.

29.9.4. Worked example: the formula, every step

With D = disease and $+$ = positive test: $P(D) = 0.01$, $P(D^c) = 0.99$, $P(+ \mid D) = 0.99$, $P(+ \mid D^c) = 0.05$.

$$\begin{aligned} P(+) &= P(+ \mid D)P(D) + P(+ \mid D^c)P(D^c) && \text{(total probability)} \\ &= 0.99 \times 0.01 + 0.05 \times 0.99 \\ &= 0.0099 + 0.0495 = 0.0594 \end{aligned}$$

$$\begin{aligned} P(D \mid +) &= \frac{P(+ \mid D)P(D)}{P(+)} && \text{(Bayes)} \\ &= \frac{0.0099}{0.0594} \\ &= \frac{1}{6} \approx 0.167. \end{aligned}$$

Same answer as the tree, because the tree *is* the formula: 99 is $10\,000 \times 0.0099$ and 594 is $10\,000 \times 0.0594$.

What if you take a second, independent test and it's also positive? Use the posterior as the new prior: $P(D) = 1/6$. *Try it:* The second positive test ; Add a second test .

$$\begin{aligned} P(+) &= 0.99 \times \tfrac{1}{6} + 0.05 \times \tfrac{5}{6} = 0.165 + 0.041\overline{6} = 0.20\overline{6} \\ P(D \mid ++) &= \frac{0.165}{0.20\overline{6}} \approx 0.798. \end{aligned}$$

Two positives take you from 1% to about 80%. Evidence accumulates; this is why doctors confirm a positive screening test with a second, different test (which also helps ensure the "independent" assumption is reasonable).

Example 39. Procedure — Bayes' theorem for a yes/no hypothesis

Input: prior $P(H)$, likelihoods $P(E \mid H)$ and $P(E \mid H^c)$.

Steps:

1. Compute the two joint probabilities: $a = P(E \mid H)P(H)$ (evidence *and* hypothesis true) and $b = P(E \mid H^c)\,(1 - P(H))$ (evidence *and* hypothesis false).
2. The evidence probability is $P(E) = a + b$.
3. The posterior is $P(H \mid E) = a/(a + b)$.
4. For further independent evidence, repeat with the posterior as the new prior.

Output: the posterior $P(H \mid E)$.

Check: draw the natural-frequency tree for 10,000 (or 100,000) cases; the leaf counts should be $10\,000a$ and $10\,000b$.

PYTHON

```python
import random

def posterior(prior, sensitivity, false_pos_rate):
    """P(disease | positive test) by Bayes' theorem."""
    p_pos = sensitivity * prior + false_pos_rate * (1 - prior)  # total probability
    return sensitivity * prior / p_pos

print(round(posterior(0.01, 0.99, 0.05), 4))

# Check by simulating a million patients
random.seed(2)
sick_and_pos = pos = 0
for _ in range(1_000_000):
    sick = random.random() < 0.01
    if sick:
        positive = random.random() < 0.99
    else:
        positive = random.random() < 0.05
    if positive:
        pos += 1
        sick_and_pos += sick
print(round(sick_and_pos / pos, 4))
```

```
0.1667
0.166
```

The simulation does exactly what conditioning means: it throws away every simulated patient who tested negative and asks what fraction of the rest are sick.

29.9.5. Alternative approach: the odds form

Bayes' theorem is even simpler in odds. Divide the Bayes formula for H by the one for H^c; the awkward denominator $P(E)$ cancels:

$$\underbrace{\frac{P(H \mid E)}{P(H^c \mid E)}}_{\text{posterior odds}} = \underbrace{\frac{P(H)}{P(H^c)}}_{\text{prior odds}} \times \underbrace{\frac{P(E \mid H)}{P(E \mid H^c)}}_{\text{likelihood ratio}}.$$

For the test: prior odds $1{:}99$; likelihood ratio $0.99/0.05 = 19.8$; posterior odds $19.8{:}99 = 1{:}5$, i.e. probability $1/6$. ✓ A second positive multiplies by 19.8 again: $19.8{:}5 \approx 4{:}1$, about 80%. ✓

29.9.6. Choosing a method

- **Natural frequency tree:** best for explaining to anyone (including yourself at 2 a.m.), and best for catching errors. Use it first whenever the numbers are percentages.
- **Formula:** best when you need a general expression or code.
- **Odds form:** best for sequential evidence (just keep multiplying by likelihood ratios) and for quick mental estimates. It is the form used by naive Bayes classifiers.
- **Simulation:** best as an independent check, and for problems too messy for algebra.

29.9.7. Where you'd use this

- **Everyday:** any time a "pretty accurate" signal fires for a rare event: a smoke alarm, a fraud alert on your card, a home pregnancy or COVID test, a "your account has been compromised" email. The base rate of the event matters as much as the accuracy of the signal, and the natural-frequency tree tells you how much to trust the alarm.
- **Science and engineering:** Bayesian search theory updates a map of where a lost object might be after each unsuccessful search, multiplying the probability of each search area by the chance the search would have missed it there. It located the wreck of the submarine USS *Scorpion* in 1968, and in 2011 a Bayesian reanalysis of the earlier, unsuccessful searches pointed searchers to the wreckage of Air France 447 on the floor of the Atlantic, found within about a week of resuming. Self-driving cars and phone GPS fuse noisy sensor readings with the **Kalman filter**, which is Bayes' theorem applied repeatedly to continuous quantities.
- **Computing:** spam filtering (below), A/B testing that reports "probability variant B is better", and probabilistic programming all apply this one formula.

Where Bayes goes wrong in practice

- **Wrong prior.** The 17% answer assumes you're a random member of the population. If you were tested *because* you had symptoms, your prior is much higher than 1%, and so is your posterior. The prior must describe the population you were drawn from.
- **Non-independent evidence.** Repeating the *same* test on the same sample may repeat the same error; multiplying likelihood ratios then overstates the evidence.
- **A prior of exactly 0 or 1.** If $P(H) = 0$, no evidence can ever move it ($0\times$ anything = 0). Never assign probability 0 to something merely unlikely.

Reverend Bayes and Mr Price

Thomas Bayes, an English Presbyterian minister, never published his theorem. After his death in 1761 his friend Richard Price found the essay among his papers, edited it and presented it to the Royal Society in 1763. Pierre-Simon Laplace independently discovered the general form and used it for decades to analyse astronomical and census data. For much of the 20th century "Bayesian" methods were unfashionable; cheap computing brought them back, and today Bayesian updating runs inside spam filters, robot localisation, medical decision aids and search-and-rescue planning.

Bayes slider

Three sliders (prevalence, sensitivity, false-positive rate) drive a live natural-frequency tree and an "icon array" of 1,000 people, colouring true positives, false positives, true negatives and false negatives. Watch the posterior collapse as you slide prevalence from 50% down to 0.1%, and see how much specificity you'd need to make a rare-disease screen useful. A second mode adds a second, independent test.

Try it: Slide the prevalence · Bayes slider lab

US curriculum

Conditional probability, two-way tables and independence are in the high-school Common Core strand S-CP ("Conditional Probability and the Rules of Probability"), typically in **Algebra II** or **Geometry**, with Bayes-style tree-diagram problems in **AP Statistics** (Unit 4, "Probability, Random Variables, and Probability Distributions"). The formal name "Bayes' theorem" is often saved for college, but the tree-diagram reasoning is expected in high school.

29.10. Two famous puzzles

29.10.1. The Monty Hall problem

In a game show there are three doors: a car behind one, goats behind the other two. You pick a door, say door 1. The host, who *knows* where the car is, opens one of the other doors to reveal a goat (if both hide goats he picks one at random), and offers you the chance to switch to the remaining closed door. Should you? *Try it:* Play Monty Hall .

The almost universal first reaction is "two doors left, so it's 50–50; switching doesn't matter". That's wrong: switching wins with probability 2/3.

Argument 1: what your first pick tells you. Your first pick is right with probability 1/3 and wrong with probability 2/3. The host's action never changes that: he can always open a goat door, whatever you picked. If you were right (prob. 1/3), switching loses. If you were wrong (prob. 2/3), the host has been forced to reveal the *other* goat, so the remaining door must hide the car, and switching wins. Switching wins exactly when your first pick was wrong: probability 2/3.

Argument 2: Bayes. Let C_i = "car behind door i" and O_3 = "host opens door 3". You picked door 1. Priors: $P(C_i) = 1/3$. Likelihoods of the host's action:

- if the car is behind 1, the host chooses 2 or 3 at random: $P(O_3 \mid C_1) = 1/2$;
- if the car is behind 2, the host must open 3: $P(O_3 \mid C_2) = 1$;
- if the car is behind 3, the host can't open 3: $P(O_3 \mid C_3) = 0$.

$$P(O_3) = \tfrac{1}{3}\cdot\tfrac{1}{2} + \tfrac{1}{3}\cdot 1 + \tfrac{1}{3}\cdot 0 = \tfrac{1}{2}$$

$$P(C_2 \mid O_3) = \frac{P(O_3 \mid C_2)P(C_2)}{P(O_3)} = \frac{1\cdot\frac{1}{3}}{\frac{1}{2}} = \frac{2}{3}$$

$$P(C_1 \mid O_3) = \frac{\frac{1}{2}\cdot\frac{1}{3}}{\frac{1}{2}} = \frac{1}{3}.$$

The host's choice carries information because it is *constrained*: he will never open the car's door. The intuition "50–50" would be right if the host opened a door at random and happened to reveal a goat; that's a different experiment (try it in the simulation below).

PYTHON

```python
import random

def play(switch, rng):
    car = rng.randrange(3)
    pick = rng.randrange(3)
    # host opens a door that is neither the pick nor the car
    opened = rng.choice([d for d in range(3) if d != pick and d != car])
    if switch:
        pick = next(d for d in range(3) if d != pick and d != opened)
    return pick == car

rng = random.Random(3)
n = 100_000
print("stay  :", sum(play(False, rng) for _ in range(n)) / n)
print("switch:", sum(play(True, rng) for _ in range(n)) / n)
```

```
stay  : 0.33329
switch: 0.66607
```

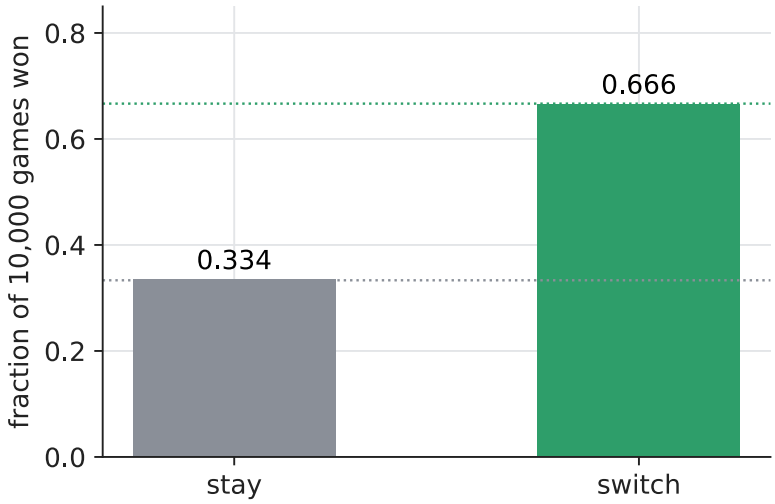


Figure 48. Simulated win rates over 10,000 games. Dotted lines mark 1/3 and 2/3.



> *Ten thousand letters*
>
> When Marilyn vos Savant gave the "switch" answer in *Parade* magazine in 1990, she received around 10,000 letters, nearly a thousand of them from people with PhDs, many telling her she was wrong. The mathematician Paul Erdős reportedly remained unconvinced until he watched a computer simulation. Simulation is a legitimate way to settle a probability argument, and a humbling one.

29.10.2. The birthday problem

In Chapter 24 you counted the ways for n people to have distinct birthdays. Now read the same calculation as a chain of conditional probabilities. Ignore leap years and assume all 365 days equally likely. Add people one at a time. Given that the first k people all have different birthdays, person $k+1$ avoids all of them with probability $(365-k)/365$. By the multiplication rule,

$$P(\text{all } n \text{ distinct}) = \frac{365}{365}\cdot\frac{364}{365}\cdot\frac{363}{365}\cdots\frac{365-n+1}{365}$$
$$P(\text{some shared birthday}) = 1 - P(\text{all distinct}).$$

For $n = 23$ this gives 0.507: better than even. The surprise is resolved by counting *pairs*, not people: 23 people make $\binom{23}{2} = 253$ pairs, and each pair matches with probability $1/365$. *Try it:* Birthday problem calculator .

An approximation (optional, uses e from Volume 1, Chapter 12). For small x, $1 - x \approx e^{-x}$. Each factor is $1 - k/365 \approx e^{-k/365}$, so the product is about

$$e^{-(0+1+\cdots+(n-1))/365} = e^{-n(n-1)/(2\cdot 365)}.$$

The exponent $n(n-1)/2$ is the number of pairs. For $n = 23$, $1 - e^{-253/365} \approx 0.500$. Setting the probability to 1/2 and solving gives $n \approx 1.18\sqrt{365} \approx 22.5$. In general, with N equally likely values, a repeat becomes likely after about $\sqrt{N}$ draws.

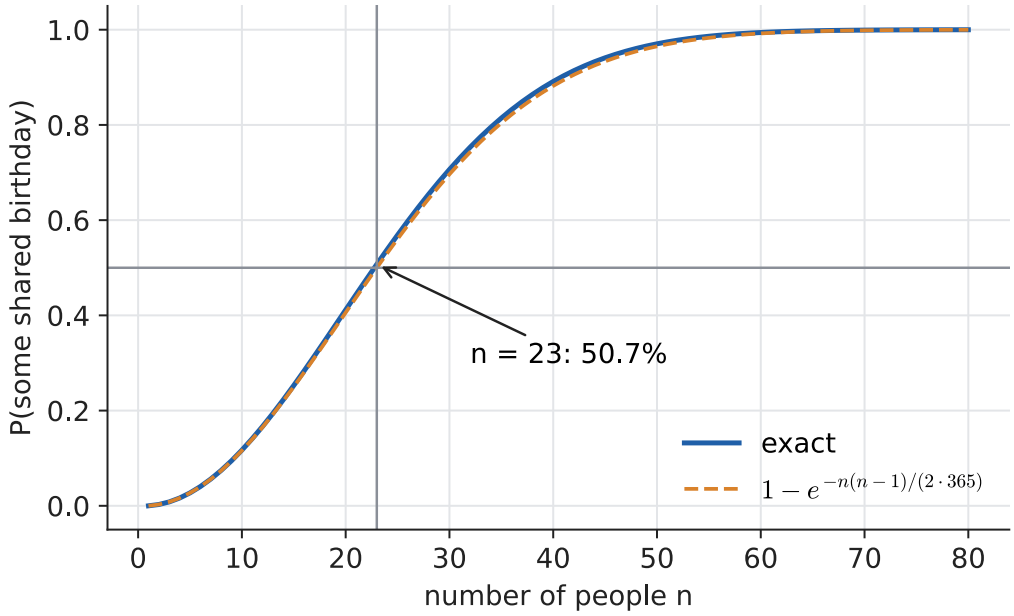


Figure 49. Probability of at least one shared birthday. The approximation (dashed) tracks the exact curve closely.

PYTHON

```python
import math, random

def p_shared(n, days=365):
    p_distinct = 1.0
    for k in range(n):
        p_distinct *= (days - k) / days
    return 1 - p_distinct

def simulate(n, days=365, trials=100_000, seed=4):
    rng = random.Random(seed)
    hits = 0
    for _ in range(trials):
        seen = set()
        for _ in range(n):
            b = rng.randrange(days)
            if b in seen:
                hits += 1
                break
            seen.add(b)
    return hits / trials

for n in (10, 23, 50):
    approx = 1 - math.exp(-n * (n - 1) / (2 * 365))
    print(n, round(p_shared(n), 4), round(approx, 4), simulate(n))
```

```
10 0.1169 0.116 0.1175
23 0.5073 0.5 0.50933
50 0.9704 0.9651 0.97013
```

Computer-science connection: hash collisions

Replace "days" with hash buckets. With a hash function producing N equally likely values, expect the first collision after roughly $\sqrt{N}$ items; the chance of a collision reaches 50% at about $1.18\sqrt{N}$: about 77,000 items for a 32-bit hash, about 5 billion for 64 bits. This is the **birthday bound**, and it's why a hash with b output bits offers only about $b/2$ bits of collision resistance, why cryptographic hashes are 256 bits, and why hash tables must handle collisions long before they are "full". (Chapter 24 works through these numbers.)

Try it: Birthday bound for hashes .

29.11. Probability at work in computing

29.11.1. Randomized algorithms

The problem. Quicksort picks a *pivot*, partitions the array into elements smaller and larger than the pivot, and recurses. If the pivot is always the first element, a sorted input makes every partition maximally lopsided, and quicksort takes about $n^2/2$ comparisons instead of about $n \log n$. An adversary (or just unlucky real-world data, which is often nearly sorted) can always find the bad input for a fixed rule.

The idea. Choose the pivot *uniformly at random*. Now there are no bad inputs, only bad luck, and bad luck is rare: for *every* input, the expected number of comparisons is about $2n \ln n \approx 1.39\, n \log_2 n$. (The derivation uses linearity of expectation; see Chapter 30.) The probability shifts from the input, which you don't control, to the coin flips, which you do.



Randomized algorithm, Las Vegas, Monte Carlo

A **randomized algorithm** makes random choices during its execution. Two families:

- **Las Vegas** algorithms are always correct; their *running time* is random. Randomized quicksort is Las Vegas: it always sorts correctly, but some runs take longer.
- **Monte Carlo** algorithms run in bounded time; their *answer* may be wrong with small probability. Repeating them independently drives the error probability down exponentially.

Mnemonic: in Las Vegas you always leave with a (correct) answer but you don't know how long you'll be there; in Monte Carlo you leave on time but might have lost.

A Monte Carlo example: Freivalds' algorithm. Given $n \times n$ matrices A, B, C (Volume 2, Chapter 27), is $AB = C$? Multiplying out costs about n^3 operations. Instead, pick a random vector r of 0s and 1s and check whether $A(Br) = Cr$: three matrix–vector products, about $3n^2$ operations. If $AB = C$, the

check always passes. If $AB \neq C$, one can show it fails with probability at least 1/2. After 20 independent rounds that all pass, the chance of being fooled is at most $2^{-20} \approx 10^{-6}$, by independence of the rounds.

PYTHON

```
import random

def matmul_vec(M, v):
    return [sum(a * b for a, b in zip(row, v)) for row in M]

def freivalds(A, B, C, rounds=20, rng=random):
    """Monte Carlo check that A @ B == C. Never wrong when it says False."""
    n = len(A)
    for _ in range(rounds):
        r = [rng.randint(0, 1) for _ in range(n)]
        if matmul_vec(A, matmul_vec(B, r)) != matmul_vec(C, r):
            return False          # definitely not equal
    return True                   # equal, with error prob <= 2**-rounds

A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
good = [[19, 22], [43, 50]]
bad = [[19, 22], [43, 51]]
random.seed(5)
print(freivalds(A, B, good), freivalds(A, B, bad))
```

```
True False
```

Choosing between them. Prefer Las Vegas when a wrong answer is unacceptable and you can tolerate variable latency (sorting, hashing with retries). Prefer Monte Carlo when a fixed time budget matters and you can make the error probability smaller than, say, the chance of a cosmic-ray bit flip (primality testing in cryptography, as in Chapter 23, is the classic case).

29.11.2. A naive Bayes spam filter

The problem. Decide whether an incoming message is spam from the words it contains. You have labelled history, from which you can measure how often each word appears in spam and in legitimate mail ("ham"). This is the medical test again, with words as test results. *Try it:* Train the spam filter .

The model. Let S = "spam". For a message containing words $w_1, \dots, w_k$, Bayes gives

$$P(S \mid w_1, \dots, w_k) = \frac{P(w_1, \dots, w_k \mid S)\, P(S)}{P(w_1, \dots, w_k)}.$$

Estimating $P(w_1, \dots, w_k \mid S)$ for every combination of words is hopeless: there are too many combinations. The **naive** assumption is that words are conditionally independent given the class:

$$P(w_1, \dots, w_k \mid S) \approx P(w_1 \mid S)\, P(w_2 \mid S) \cdots P(w_k \mid S).$$

It's obviously false ("Nigerian" and "prince" are not independent in spam), but the classifier only needs to rank spam above ham, and it does that well. Compute an unnormalized score for each class, then divide by their sum (the law of total probability, in disguise).

Worked example. Training data: 4 spam and 6 ham messages.

Class	Messages
spam	"free money now", "free offer meeting", "win free prize", "cheap money offer"
ham	"meeting at noon", "project meeting notes", "free for lunch", "meeting moved", "code review today", "lunch plans"

Estimates (fraction of messages in the class that contain the word):

$$\begin{aligned} P(S) &= \tfrac{4}{10} = 0.4, & P(H) &= \tfrac{6}{10} = 0.6, \\ P(\text{free} \mid S) &= \tfrac{3}{4}, & P(\text{free} \mid H) &= \tfrac{1}{6}, \\ P(\text{meeting} \mid S) &= \tfrac{1}{4}, & P(\text{meeting} \mid H) &= \tfrac{3}{6} = \tfrac{1}{2}. \end{aligned}$$

A new message contains "free" and "meeting". Scores:

$$\begin{aligned} \text{spam score} &= P(S)\,P(\text{free} \mid S)\,P(\text{meeting} \mid S) = 0.4 \times \tfrac{3}{4} \times \tfrac{1}{4} = 0.075 \\ \text{ham score} &= P(H)\,P(\text{free} \mid H)\,P(\text{meeting} \mid H) = 0.6 \times \tfrac{1}{6} \times \tfrac{1}{2} = 0.05 \\ P(S \mid \text{free, meeting}) &= \frac{0.075}{0.075 + 0.05} = \frac{0.075}{0.125} = 0.6. \end{aligned}$$

Mildly spammy. "Free" alone would give $0.3/(0.3 + 0.1) = 0.75$; the word "meeting" pulled it back towards ham.

```python
spam = ["free money now", "free offer meeting", "win free prize",
        "cheap money offer"]
ham = ["meeting at noon", "project meeting notes", "free for lunch",
       "meeting moved", "code review today", "lunch plans"]

def doc_freq(docs, word, alpha=0):
    """Fraction of docs containing word (alpha = add-alpha smoothing)."""
    hits = sum(word in d.split() for d in docs)
    return (hits + alpha) / (len(docs) + 2 * alpha)

def p_spam(words, alpha=0):
    n = len(spam) + len(ham)
    score_s = len(spam) / n
    score_h = len(ham) / n
    for w in words:
        score_s *= doc_freq(spam, w, alpha)
        score_h *= doc_freq(ham, w, alpha)
    return score_s / (score_s + score_h)

print(round(p_spam(["free", "meeting"]), 3))
print(round(p_spam(["free"]), 3))
print(round(p_spam(["prize"]), 3), round(p_spam(["prize", "meeting"]), 3))
print(round(p_spam(["prize", "meeting"], alpha=1), 3))
```

```
0.6
0.75
1.0 1.0
0.542
```



The zero-count problem

"prize" never appears in the ham training set, so $P(\text{prize} \mid H) = 0$, the ham score is 0 whatever else the message says, and the filter is *certain* it's spam, even for "prize" plus "meeting". This is the "prior of 0" failure from the Bayes Watch out, in another form. The standard fix is **Laplace (add-one) smoothing**: pretend every word was seen once more in each class, and adjust the denominators so probabilities still sum to 1 (in our binary "contains the word or not" model, add 2 to the denominator). With smoothing the same message scores 0.542. Real filters also add logarithms of probabilities instead of multiplying them, to avoid floating-point underflow when multiplying hundreds of small numbers (Volume 1, Chapter 12 , Volume 2, Chapter 32).

Try it: Fix it with smoothing ; See the zero-count problem .

29.11.3. Bloom filters: designing for a false-positive rate

The problem. You want a compact set that answers "have I seen this URL / key / username before?" using far less memory than storing the items. You can accept occasional false positives ("probably yes" when the answer is no) but never false negatives. Databases use this to skip disk reads; browsers once used it to check URLs against malware lists. *Try it:* Bloom filter explorer .

The data structure. A **Bloom filter** is an array of m bits, all 0, plus k hash functions. To add an item, hash it k ways and set those k bits to 1. To query, check the k bits: if any is 0, the item was definitely never added; if all are 1, it *probably* was.

The probability. Insert n items. Treat each hash as a uniformly random, independent position. Focus on one particular bit. (The last step uses $(1 - \frac{1}{m})^m \approx e^{-1}$, an optional fact from Volume 1, Chapter 12 .)

$$\begin{aligned} P(\text{one hash misses this bit}) &= 1 - \tfrac{1}{m} \\ P(\text{all } kn \text{ hashes miss it}) &= \left(1 - \tfrac{1}{m}\right)^{kn} && \text{(independence)} \\ &\approx e^{-kn/m} && \text{(since } (1 - \tfrac{1}{m})^m \approx e^{-1}\text{)} \end{aligned}$$

So a bit is 1 with probability about $1 - e^{-kn/m}$. A query for a new item gives a false positive when all k of its bits happen to be 1. Treating those as independent:

$$P(\text{false positive}) \approx \left(1 - e^{-kn/m}\right)^k.$$

There's a tradeoff in k: more hashes give each query more chances to find a 0, but also fill the array faster. The best choice turns out to be $k \approx (m/n) \ln 2$. With $m/n = 10$ bits per item, $k \approx 6.9$, and the false-positive rate is under 1%.

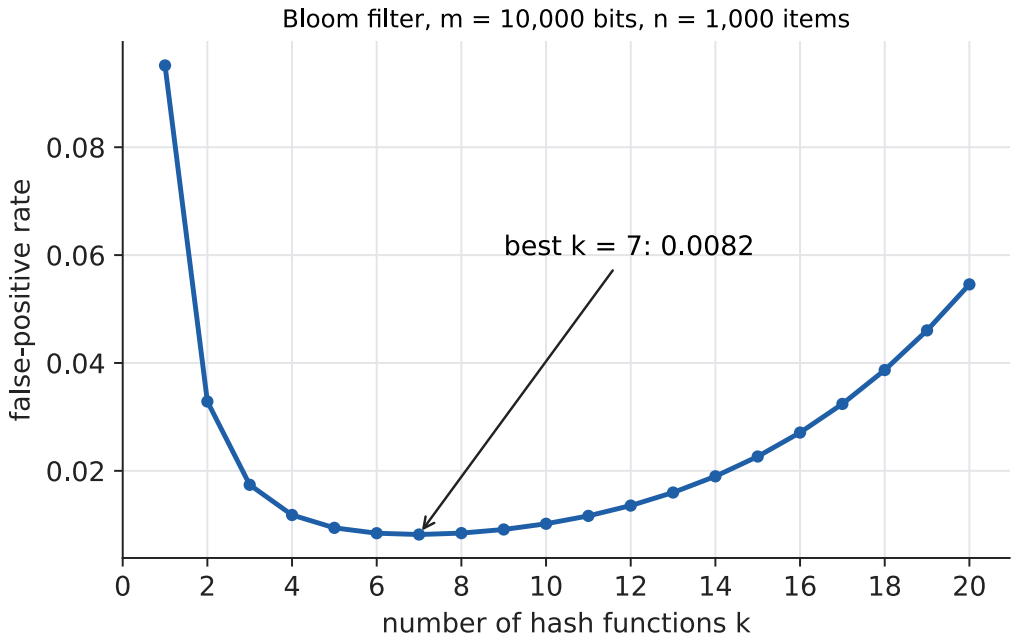


Figure 50. False-positive rate against the number of hash functions, for 10 bits per item. Too few hashes and too many are both bad.

PYTHON

```
import hashlib, math, random

class BloomFilter:
    def __init__(self, m, k):
        self.m, self.k = m, k
        self.bits = bytearray(m)          # one byte per bit, for clarity

    def _positions(self, item):
        for i in range(self.k):
            h = hashlib.sha256(f"{i}:{item}".encode()).digest()
            yield int.from_bytes(h[:8], "big") % self.m

    def add(self, item):
        for p in self._positions(item):
            self.bits[p] = 1

    def might_contain(self, item):
        return all(self.bits[p] for p in self._positions(item))

m, n, k = 10_000, 1_000, 7
bf = BloomFilter(m, k)
for i in range(n):
    bf.add(f"user{i}")

tests = 20_000
false_pos = sum(bf.might_contain(f"other{j}") for j in range(tests))
print("measured :", false_pos / tests)
print("predicted:", round((1 - math.exp(-k * n / m)) ** k, 5))
```

```
measured : 0.00815
predicted: 0.00819
```

The independence assumptions in the derivation are not exactly true, but measurement agrees with prediction to within the noise. That's a pattern worth noticing: a model built on slightly false assumptions can still be extremely useful, provided you *check* it.

Computer-science connection: simulation as a test oracle

Every result in this chapter was checked by simulation. That's not a crutch; it's a professional habit. Probability arguments fail silently: a wrong conditioning event or a hidden dependence gives a confident, wrong number. A 10-line Monte Carlo simulation is an independent implementation of the same question, like a second engineer writing the same function. When the analytic answer and the simulation agree to within sampling noise (about $1/\sqrt{\text{trials}}$; why is explained in Chapter 30), you can trust both. When they disagree, one of them encodes a misunderstanding of the problem, and finding which one is where the learning happens.



Simulate before you believe

A sandbox with a two-dice grid where you click cells to define events A and B. The lab shades $A \cap B$, displays $P(A)$, $P(B)$, $P(A \mid B)$ and $P(B \mid A)$, flags whether the events are independent, and animates "conditioning" as the non-B cells fading away. A second panel runs Monty Hall with a "host knows" / "host is random" toggle, so you can see the 2/3 answer turn into 1/2 when the host stops carrying information.

Try it: Two-dice event sandbox · Monty Hall: host knows or random

29.12. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Sampling and simulation
- Probability: dice, cards, Bayes
- Test accuracy and risk
- Counting, visually

29.13. Exercises

1. **(Recognition)** Translate into set notation and say whether each pair is mutually exclusive: (a) "a card is a heart" and "a card is a king"; (b) "a die shows an even number" and "a die shows 5"; (c) "a request times out" and "a request returns 200 OK".
2. **(Recognition)** Which of these is $P(A \mid B)$ and which $P(B \mid A)$, with A = "email is spam", B = "email contains a link"? (i) "Of spam emails, 90% contain a link." (ii) "Of emails with links, 30% are spam."

3. **(Application)** Two dice are rolled. Find (a) $P(\text{total} \geq 10)$; (b) $P(\text{at least one six})$; (c) $P(\text{at least one six} \mid \text{total} \geq 10)$. *Try it:* Two dice distribution ; Check exercise 3 on the grid .
4. **(Application)** A drawer holds 4 black and 6 white socks. You pull two at random. Find the probability they match, using a tree diagram; check by counting.
5. **(Application)** $P(A) = 0.5$, $P(B) = 0.4$, $P(A \cup B) = 0.7$. Find $P(A \cap B)$ and $P(A \mid B)$. Are A and B independent?
6. **(Application)** 2% of transactions are fraudulent. A fraud model flags 95% of fraudulent transactions and 3% of legitimate ones. What fraction of flagged transactions are actually fraudulent? Draw the natural-frequency tree for 100,000 transactions. *Try it:* Fraud flags tree .
7. **(Problem solving)** Show that if A and B are independent, so are A and B^c.
8. **(Problem solving)** A Monty Hall variant has 4 doors; the host knowingly opens 2 goat doors from the 3 you didn't pick, and offers a switch to the one remaining. What is the probability of winning by switching? *Try it:* Monty Hall with 100 doors ; Monty Hall with 4 doors .
9. **(Problem solving, CS)** A Monte Carlo primality test declares a composite number "probably prime" with probability at most 1/4 per round, independently. How many rounds guarantee an error probability below 10^{-12}?
10. **(Problem solving, CS)** A Bloom filter has $m = 8 \times 10^6$ bits and will hold $n = 10^6$ items. Find the optimal k (nearest integer) and the resulting false-positive rate. *Try it:* Check your Bloom design .

▼ **Answers**

1. (a) $H \cap K$ is {king of hearts}, not empty: not mutually exclusive. (b) $\{2, 4, 6\} \cap \{5\} = \varnothing$: mutually exclusive. (c) Mutually exclusive (assuming a timed-out request returns no status).
2. (i) is $P(B \mid A) = 0.9$ (restricted to spam). (ii) is $P(A \mid B) = 0.3$ (restricted to emails with links).
3. (a) Totals ≥ 10: (4,6),(5,5),(6,4),(5,6),(6,5),(6,6): 6 outcomes, $6/36 = 1/6$. (b) Complement: $1 - 25/36 = 11/36$. (c) Restrict to the 6 outcomes in (a); those with a six: (4,6),(6,4),(5,6),(6,5), (6,6) = 5. So $5/6$.
4. Tree: $P(BB) = \frac{4}{10} \cdot \frac{3}{9} = \frac{12}{90}$, $P(WW) = \frac{6}{10} \cdot \frac{5}{9} = \frac{30}{90}$; match $= \frac{42}{90} = \frac{7}{15} \approx 0.467$. Counting: $\frac{\binom{4}{2}+\binom{6}{2}}{\binom{10}{2}} = \frac{6+15}{45} = \frac{21}{45} = \frac{7}{15}$. ✓
5. $P(A \cap B) = P(A) + P(B) - P(A \cup B) = 0.5 + 0.4 - 0.7 = 0.2$. $P(A \mid B) = 0.2/0.4 = 0.5 = P(A)$, so independent (also $0.5 \times 0.4 = 0.2$ ✓).
6. Of 100,000: 2,000 fraudulent → 1,900 flagged; 98,000 legitimate → 2,940 flagged. Flagged total 4,840. $P(\text{fraud} \mid \text{flag}) = 1900/4840 \approx 0.393$. Fewer than 40% of alerts are real fraud, despite a "95% accurate" model.
7. $A = (A \cap B) \cup (A \cap B^c)$, disjoint, so $P(A \cap B^c) = P(A) - P(A \cap B) = P(A) - P(A)P(B) = P(A)(1 - P(B)) = P(A)P(B^c)$. ✓

8. Your first pick is right with probability 1/4; switching wins whenever it's wrong (probability 3/4), because the host has eliminated the other two goats among the three doors you didn't pick. Switching wins with probability 3/4.
9. Need $(1/4)^r < 10^{-12}$, i.e. $4^r > 10^{12}$, $r > 12/\log_{10} 4 = 12/0.602 \approx 19.9$. So 20 rounds.
10. $k = (m/n)\ln 2 = 8 \times 0.693 \approx 5.5$; try 5 and 6. $k = 5$:
    $(1 - e^{-5/8})^5 = (1 - 0.5353)^5 = 0.4647^5 \approx 0.0217$. $k = 6$:
    $(1 - e^{-0.75})^6 = 0.5276^6 \approx 0.0216$. Either gives about 2.2%; take $k = 6$ (marginally better) or 5 (cheaper).

29.14. Mastery check

1. Explain in one sentence what $P(A \mid B)$ means, without using the word "given".
2. A fair coin is tossed 3 times. Find the probability of at least two heads, and the probability of at least two heads given that the first toss is a head.
3. $P(A) = 0.3$ and $P(B) = 0.5$. Find $P(A \cup B)$ if (a) A and B are mutually exclusive; (b) they are independent.
4. 30% of a site's users are on mobile. Mobile users convert at 2%, desktop users at 5%. Find the overall conversion rate, and the probability that a converting user was on mobile.
5. Derive Bayes' theorem from the definition of conditional probability.
6. In a room of 30 people, is a shared birthday more or less likely than not? Estimate it with the $\sqrt{N}$ rule of thumb or the exponential approximation.
7. Explain the difference between a Las Vegas and a Monte Carlo algorithm, with one example of each.

Decision rule: All correct without hesitation → move on to Chapter 30. Stuck on 1–2 → reread Section 29.5 and Section 29.9, then redo exercises 3 and 6. Stuck on more → work the chapter exercises first, checking every answer with a ten-line simulation.

▼ **Mastery check answers**

1. Of the probability (or the cases) in which B happens, $P(A \mid B)$ is the fraction in which A also happens: restrict the world to B, then measure A.
2. 8 equally likely outcomes. At least two heads: HHH, HHT, HTH, THH = 4/8 = 1/2. Given first is H: restrict to HHH, HHT, HTH, HTT; at least two heads in 3 of them: 3/4.
3. (a) $0.3 + 0.5 = 0.8$. (b) $0.3 + 0.5 - 0.3 \times 0.5 = 0.65$.
4. $P(C) = 0.3 \times 0.02 + 0.7 \times 0.05 = 0.006 + 0.035 = 0.041$.
   $P(M \mid C) = 0.006/0.041 \approx 0.146$.

5. $P(A \cap B) = P(A \mid B)P(B)$ and $P(A \cap B) = P(B \mid A)P(A)$ (both from the definition). Equate and divide by $P(B)$: $P(A \mid B) = P(B \mid A)P(A)/P(B)$; expand $P(B) = P(B \mid A)P(A) + P(B \mid A^c)P(A^c)$ by total probability.

6. More likely: 30 exceeds $1.18\sqrt{365} \approx 23$. Approximation: $1 - e^{-30 \cdot 29/730} = 1 - e^{-1.19} \approx 0.70$ (exact 0.706).

7. Las Vegas: always correct, random running time (randomized quicksort). Monte Carlo: bounded running time, small probability of a wrong answer, reducible by repetition (Freivalds' matrix check, Miller–Rabin primality testing).

Chapter 30. Random Variables and Distributions

*Chapter 29 asked "how likely is this event?". Most practical questions are about quantities instead: how many requests arrive in the next second, how long until a disk fails, how many comparisons quicksort makes, what the average of a thousand measurements will be. A **random variable** is a number whose value is decided by chance, and its **distribution** says how likely each value is. This chapter introduces expected value and variance (the centre and spread of a distribution), the handful of named distributions that model most real situations (binomial, geometric, Poisson, uniform, exponential, normal), and the two great theorems that explain why averages are predictable: the law of large numbers and the central limit theorem. The computing payoff is large: expected running times, the coupon collector, load balancing, and how a program turns a uniform random number into any distribution you like.*

Why this chapter matters. Insurance premiums, drug-trial sample sizes, the tolerances on a machined part, the number of servers a website needs and the "±3%" on an opinion poll are all computed from a distribution's mean and spread. Once you can name the distribution behind a situation, a page of reasoning collapses to a formula.

Where this chapter sits

You need: the rules of probability, conditional probability and independence (Chapter 29), counting and binomial coefficients (Chapter 24), functions and their graphs (Volume 1, Chapter 11), and e and natural logarithms (Volume 1, Chapter 12). Sigma notation is used throughout. A few optional passages use integrals (Volume 2, Chapter 19); they are flagged and can be skipped.

This chapter unlocks: statistics and inference (Chapter 31), which turns these ideas around to learn about the world from data; the probabilistic side of machine learning (Chapter 33); and the analysis of randomized algorithms and systems.

30.1. Random variables

30.1.1. The problem

Roll two dice. Chapter 29 would ask "what's the probability of a total of 8?". But often you care about the total itself as a quantity: what total should you expect on average? How variable is it? You want to treat "the total" as a single object you can compute with, the way a program treats a variable.

30.1.2. Intuition

A random variable is a rule that turns each outcome of an experiment into a number. For two dice, the rule "add the faces" turns the outcome $(3, 5)$ into 8 and $(6, 6)$ into 12. Before the dice are rolled you don't know the value; afterwards you do. Different rules on the same experiment give different random variables: "the larger face", "the number of sixes", "1 if doubles else 0".



Random variable

A **random variable** is a function from the sample space Ω to the real numbers: it assigns a number to every outcome. Random variables are written with capital letters (X, Y, S); their particular values with lower-case letters (x, y, s).

$X = x$ denotes the *event* "X takes the value x", i.e. the set of outcomes $\{\omega \in \Omega : X(\omega) = x\}$. So $P(X = x)$ is an ordinary probability of an event.

A random variable is **discrete** if its possible values can be listed (finitely many, or a list like 0, 1, 2, …). It is **continuous** if it can take any value in an interval (a time, a length, a temperature).

The name is a historical accident: a random variable is neither random (it's a fixed rule) nor a variable (it's a function). The randomness is in which outcome occurs.

30.1.3. Probability mass functions

For a discrete random variable, the **distribution** is the list of possible values with their probabilities. Let S be the total of two dice. Counting cells in the 6×6 grid from Chapter 29 gives:

s	2	3	4	5	6	7	8	9	10	11	12
$P(S = s)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

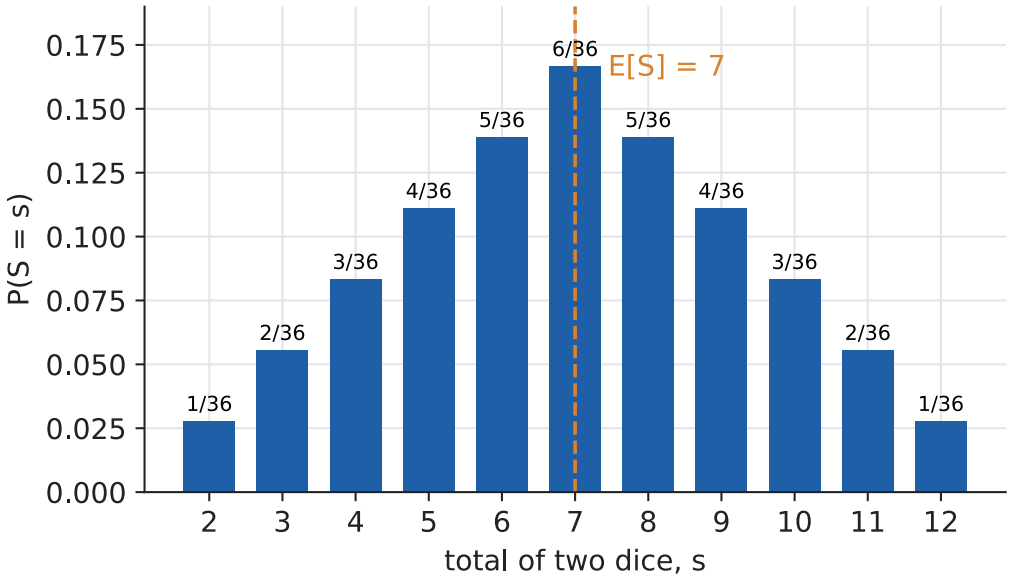


Figure 51. The probability mass function of the total of two dice. The bar heights sum to 1.

Probability mass function (PMF)

The **probability mass function** of a discrete random variable X is $p(x) = P(X = x)$. It satisfies $p(x) \geq 0$ for every x, and $\sum_x p(x) = 1$ (the sum over all possible values). "Mass" because each value carries a lump of probability, like beads of different weights on a number line.

The **support** of X is the set of values with $p(x) > 0$.

Example 40. Procedure — finding a PMF

Input: an experiment and a random variable X defined on it.

Steps:

1. List the possible values of X.
2. For each value x, identify the event $X = x$ (which outcomes give that value).
3. Find its probability by counting, a tree diagram, or a known formula.
4. Check that the probabilities sum to 1.

Output: a table or formula for $p(x) = P(X = x)$.

30.2. Expected value

30.2.1. The problem

A game costs £1 to play. You roll a die; on a 6 you win £5, otherwise nothing. Should you play? Each game is a gamble, but over a thousand games, what happens to your money?

30.2.2. Intuition: the long-run average

Over 6,000 games you'd expect about 1,000 sixes, winning $1000 \times 5 = 5000$ pounds, while paying 6,000. You lose about £1,000, or about 17p per game. That per-game average is the **expected value**. It's what the average of many repetitions settles down to (the frequency interpretation from Chapter 29, applied to quantities).

Computed directly: you win £5 with probability $\frac{1}{6}$ and £0 with probability $\frac{5}{6}$, so the average winnings per game are $5 \times \frac{1}{6} + 0 \times \frac{5}{6} = \frac{5}{6} \approx 0.83$; subtract the £1 stake and the expected profit is $-\frac{1}{6}$ pound.



Expected value

The **expected value** (or **expectation**, or **mean**) of a discrete random variable X is

$$E[X] = \sum_x x\, P(X = x),$$

the sum of each possible value times its probability: a **weighted average** of the values, weighted by probability. It is often written μ (Greek *mu*) or μ_X.

Interpretation: the long-run average of X over many independent repetitions.

30.2.3. Worked example: one die

Let X be the face of a fair die.

$$\begin{aligned} E[X] &= 1 \cdot \tfrac{1}{6} + 2 \cdot \tfrac{1}{6} + 3 \cdot \tfrac{1}{6} + 4 \cdot \tfrac{1}{6} + 5 \cdot \tfrac{1}{6} + 6 \cdot \tfrac{1}{6} \\ &= \frac{1+2+3+4+5+6}{6} \\ &= \frac{21}{6} = 3.5. \end{aligned}$$

You'll never roll 3.5. The expected value is a balance point, not a "most likely value". Picture the PMF as weights on a seesaw: $E[X]$ is where it balances. For the two-dice total, the symmetric PMF balances at 7, so $E[S] = 7$.

The same idea works for any function of X: $E[g(X)] = \sum_x g(x)\, P(X = x)$. For example $E[X^2] = \frac{1+4+9+16+25+36}{6} = \frac{91}{6}$. Note that $E[X^2] = 15.17 \neq (E[X])^2 = 12.25$. In general you can't push E through a nonlinear function.

30.2.4. Linearity of expectation

The most useful fact about expectation is almost embarrassingly simple:

$$E[X + Y] = E[X] + E[Y] \qquad \text{and} \qquad E[aX + b] = a\,E[X] + b,$$

for *any* random variables X and Y, **whether or not they are independent**.

Why it works. Think of the sample space as a finite list of outcomes ω, each with probability $P(\omega)$. Then $E[X] = \sum_\omega X(\omega) P(\omega)$ (group the outcomes by value and you get the earlier formula back). Now

$$\begin{aligned} E[X+Y] &= \sum_{\omega} \big(X(\omega)+Y(\omega)\big)P(\omega) \\ &= \sum_{\omega} X(\omega)P(\omega) + \sum_{\omega} Y(\omega)P(\omega) \qquad \text{(split the sum)} \\ &= E[X] + E[Y]. \end{aligned}$$

Nothing about independence was used. A sum is a sum.

So the total of two dice has $E[S] = E[X_1] + E[X_2] = 3.5 + 3.5 = 7$, without the PMF table. Ten dice: 35. A hundred: 350.

30.2.5. The indicator trick

Linearity becomes a superpower when combined with indicators.



Indicator random variable

The **indicator** of an event A, written I_A or $\mathbf{1}_A$, is the random variable that equals 1 if A happens and 0 otherwise. Its expectation is the probability of the event:

$$E[I_A] = 1 \cdot P(A) + 0 \cdot P(A^c) = P(A).$$

To find the expected *number* of things that happen, write the count as a sum of indicators, one per thing, and add their probabilities. You never need the distribution of the count.

Example 41. Procedure — expected count by indicators

Input: a random count X = "how many of the items $1, \dots, n$ have property Q".

Steps:

1. For each item i, define $I_i = 1$ if item i has property Q, else 0. Then $X = I_1 + I_2 + \cdots + I_n$.
2. For each i, compute $P(\text{item } i \text{ has } Q)$. Often, by symmetry, they're all equal.
3. Add: $E[X] = \sum_i P(\text{item } i \text{ has } Q)$.

Output: $E[X]$. Dependence between items does not matter.

30.2.6. Worked example: the hat-check problem

n people check their hats at a restaurant. The attendant loses the tickets and hands the hats back in a uniformly random order. On average, how many people get their own hat back?

The events "person i gets their own hat" are heavily dependent (if the first $n-1$ people all get their own hats, so does the last). Computing the distribution of the number of matches is a genuinely hard counting problem. Linearity ignores all that.

1. Let $I_i = 1$ if person i gets their own hat. The number of matches is $M = I_1 + \cdots + I_n$.

2. In a random permutation, person i's hat is equally likely to go to any of the n people, so $P(I_i = 1) = \frac{1}{n}$.
3. $E[M] = n \times \frac{1}{n} = 1$.

Exactly one match on average, whether there are 3 people or 3 million. (A harder inclusion–exclusion argument shows that the probability of *no* matches tends to $1/e \approx 0.368$.)

PYTHON

```python
import random

def fixed_points(n, rng):
    hats = list(range(n))
    rng.shuffle(hats)                       # random permutation
    return sum(hats[i] == i for i in range(n))

rng = random.Random(6)
for n in (3, 10, 100):
    trials = 100_000
    results = [fixed_points(n, rng) for _ in range(trials)]
    mean = sum(results) / trials
    p_none = results.count(0) / trials
    print(f"n={n:5d}  average matches={mean:.3f}  P(no match)={p_none:.3f}")
```

```
n=    3  average matches=1.005  P(no match)=0.332
n=   10  average matches=1.005  P(no match)=0.366
n=  100  average matches=0.996  P(no match)=0.369
```



When expectation misleads

- **The mean is not the typical value.** Expected household wealth is pulled far above what most households have by a few very rich ones. Expected latency hides the slow tail that users notice. Report medians and percentiles too (Chapter 31).
- $E[XY] = E[X]E[Y]$ **needs independence**, unlike $E[X + Y]$. And $E[g(X)] \neq g(E[X])$ for nonlinear g: the expected value of $1/X$ is not $1/E[X]$.
- **Some expectations don't exist.** In the *St Petersburg game* a coin is tossed until the first head; if that takes k tosses you win 2^k pounds. Then $E = \sum_k 2^k \cdot 2^{-k} = 1 + 1 + 1 + \cdots$, which is infinite, yet nobody would pay £1,000 to play. Heavy-tailed quantities (file sizes, wealth, some latencies) can behave like this.

30.2.7. Where you'd use this

- **Everyday: is the extended warranty worth it?** A £400 washing machine comes with an offer of a two-year warranty for £40. Suppose the chance of a breakdown in those two years is 5% and a typical repair costs £200. Your expected repair bill without the warranty is $0.05 \times 200 = 10$ pounds. The warranty costs four times the repair bill it saves you on average; that gap is the seller's profit margin. (Buying it can still be rational if a £200 surprise would genuinely hurt you, but now you know the price of that peace of mind.)

- **Chemistry: atomic masses are expectations.** Chlorine is 75.8% chlorine-35 and 24.2% chlorine-37. Pick an atom at random; its expected mass is $35 \times 0.758 + 37 \times 0.242 = 35.48$, the value printed on the periodic table. A weighted average *is* an expected value.
- **Computing: average memory access time.** A cache hit costs 1 ns, a miss (a trip to main memory) costs 100 ns, and the hit rate is 90%. Expected access time: $0.9 \times 1 + 0.1 \times 100 = 10.9$ ns. Raising the hit rate to 99% gives $0.99 + 1 = 1.99$ ns, five times faster, which is why so much engineering goes into the last few percent of cache hits.

30.3. Variance and standard deviation

30.3.1. The problem

Two servers both average 100 ms per request. One always takes 95–105 ms. The other takes 10 ms most of the time and occasionally 2 seconds. The means agree; the experiences don't. You need a number for spread.

30.3.2. Intuition and definition

Measure how far X typically lands from its mean μ. The deviation $X - \mu$ averages to zero (positives and negatives cancel: $E[X - \mu] = \mu - \mu = 0$), so square it first, making every deviation count positively, then average.



Variance and standard deviation

The **variance** of X, with $\mu = E[X]$, is the expected squared deviation from the mean:

$$\mathrm{Var}(X) = E\big[(X - \mu)^2\big].$$

It is often written σ^2 (*sigma squared*). The **standard deviation** is its square root, $\sigma = \sqrt{\mathrm{Var}(X)}$, which has the same units as X (milliseconds rather than milliseconds squared), so it's the one to quote.

Rough meaning: σ is a typical distance between X and its mean.

30.3.3. Why it works: the computing formula

Expanding the square and using linearity gives a formula that's usually easier:

$$\begin{aligned} \mathrm{Var}(X) &= E[X^2 - 2\mu X + \mu^2] \\ &= E[X^2] - 2\mu E[X] + \mu^2 && \text{(linearity; } \mu \text{ is a constant)} \\ &= E[X^2] - 2\mu^2 + \mu^2 \\ &= E[X^2] - \mu^2 = E[X^2] - (E[X])^2. \end{aligned}$$

"Mean of the square minus square of the mean."

30.3.4. Worked example: one die

From before, $E[X] = 3.5$ and $E[X^2] = 91/6$.

$$\mathrm{Var}(X) = \tfrac{91}{6} - 3.5^2 = \tfrac{91}{6} - \tfrac{49}{4} = \tfrac{182-147}{12} = \tfrac{35}{12} \approx 2.917$$
$$\sigma = \sqrt{35/12} \approx 1.708.$$

A die typically lands about 1.7 away from 3.5, which matches the eye: the values 1 to 6 are between 0.5 and 2.5 from the centre.

30.3.5. Rules for variance

$$\mathrm{Var}(aX + b) = a^2\,\mathrm{Var}(X), \qquad \mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) \ \text{ if } X, Y \text{ independent.}$$

Adding a constant b shifts everything and doesn't change spread; scaling by a scales deviations by a, so squared deviations by a^2. The sum rule needs independence (in general there's an extra covariance term, Section 30.8). Note the consequence: for the **average** of n independent copies of X,

$$\mathrm{Var}\left(\frac{X_1 + \cdots + X_n}{n}\right) = \frac{1}{n^2} \cdot n\sigma^2 = \frac{\sigma^2}{n}, \qquad \text{so its standard deviation is } \frac{\sigma}{\sqrt{n}}.$$

Averaging n measurements shrinks the spread by a factor of $\sqrt{n}$. To halve the noise, take four times as many measurements. Hold on to this: it's the engine of the law of large numbers, the central limit theorem and nearly all of Chapter 31.



Variance is not additive for dependent quantities

Standard deviations *never* add ($\sigma_{X+Y} \neq \sigma_X + \sigma_Y$ even for independent variables; it's the variances that add). And if two latencies are positively correlated (same overloaded database), the variance of their sum is *larger* than the sum of variances. Capacity plans that assume independence underestimate bad days.

30.3.6. Where you'd use this

- **Engineering: tolerance stacking.** Three machined spacers are stacked; each has length error with standard deviation 0.1 mm, independently. The worst case adds to 0.3 mm, but the standard deviation of the total is $\sqrt{3 \times 0.1^2} \approx 0.17$ mm, because variances add, not standard deviations. Engineers call this **root-sum-square tolerancing**; it lets them specify cheaper parts without over-designing for a worst case that almost never happens.
- **Lab science and sport timing:** averaging four independent readings halves the standard deviation of the result ($\sigma/\sqrt{4}$). That's why a chemistry practical asks for repeat titrations and why a hand-timed 100 m is less trustworthy than the average of several timekeepers.
- **Computing: latency budgets.** A page makes three independent backend calls in sequence, each with standard deviation 20 ms. The total's standard deviation is $\sqrt{3} \times 20 \approx 35$ ms, not 60 ms. Variance (as "jitter") also matters in its own right: video calls and games care as much about consistent packet timing as about the average.

30.4. The named discrete distributions

A small family of distributions models a huge range of situations. Learn each as a *story*: if your situation matches the story, the formulas come for free.

30.4.1. Bernoulli: one yes/no trial

Story: one trial that succeeds with probability p (a coin flip, a click or not, a packet lost or not). $X = 1$ on success, 0 on failure.

$$P(X = 1) = p, \quad P(X = 0) = 1 - p, \qquad E[X] = p, \qquad \mathrm{Var}(X) = p(1 - p).$$

The variance: $E[X^2] = p$ (since $X^2 = X$ when X is 0 or 1), so $\mathrm{Var} = p - p^2 = p(1 - p)$. It's largest at $p = 1/2$, the most uncertain coin.

30.4.2. Binomial: number of successes in n trials

Story: n independent Bernoulli trials, each with success probability p. X = number of successes. Write $X \sim \mathrm{Bin}(n, p)$, read "$X$ has the binomial distribution with parameters n and p".

Deriving the PMF by counting. Take $n = 5$, and ask for exactly $k = 2$ successes. One particular sequence, say SSFFF, has probability $p \cdot p \cdot (1 - p)(1 - p)(1 - p) = p^2(1 - p)^3$ by independence. *Every* sequence with exactly 2 S's and 3 F's has the same probability, just in a different order. How many such sequences? Choose which 2 of the 5 positions are S: $\binom{5}{2} = 10$ (Chapter 24). The sequences are disjoint, so add: $P(X = 2) = 10\,p^2(1 - p)^3$. In general:

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}, \qquad k = 0, 1, \ldots, n.$$

The binomial theorem confirms these sum to $(p + (1 - p))^n = 1^n = 1$; that's where the distribution gets its name.

Mean and variance, without algebra. $X = X_1 + \cdots + X_n$, a sum of independent Bernoullis. By linearity $E[X] = np$; by independence $\mathrm{Var}(X) = np(1 - p)$.

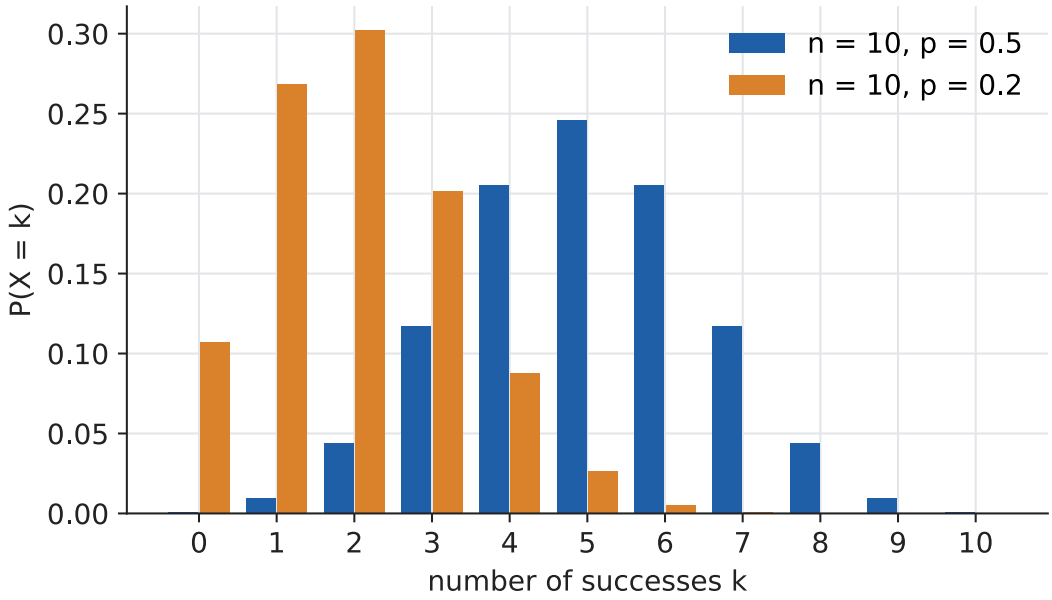


Figure 52. Binomial PMFs for 10 trials. With p = 0.5 the shape is symmetric about 5; with p = 0.2 it piles up near 2 and is skewed right.

Worked example. Ten fair coin flips. Probability of exactly 7 heads? *Try it:* Binomial(10, 0.5) at 7 .

$$P(X = 7) = \binom{10}{7}(0.5)^7(0.5)^3 = 120 \times \frac{1}{1024} = \frac{120}{1024} \approx 0.117.$$

Probability of 8 or more heads: $\frac{\binom{10}{8}+\binom{10}{9}+\binom{10}{10}}{1024} = \frac{45+10+1}{1024} = \frac{56}{1024} \approx 0.055$.

PYTHON

```
from math import comb

def binom_pmf(k, n, p):
    return comb(n, k) * p**k * (1 - p)**(n - k)

n, p = 10, 0.5
print(round(binom_pmf(7, n, p), 4))
print(round(sum(binom_pmf(k, n, p) for k in range(8, 11)), 4))
mean = sum(k * binom_pmf(k, n, p) for k in range(n + 1))
var = sum((k - mean)**2 * binom_pmf(k, n, p) for k in range(n + 1))
print(round(mean, 6), round(var, 6))
```

```
0.1172
0.0547
5.0 2.5
```

The brute-force mean and variance match $np = 5$ and $np(1-p) = 2.5$.

30.4.3. Geometric: waiting for the first success

Story: repeat independent trials with success probability p until the first success. X = number of trials needed (including the success).

The first success is on trial k exactly when the first $k-1$ trials fail and trial k succeeds:

$$P(X = k) = (1-p)^{k-1}p, \qquad k = 1, 2, 3, \ldots$$

The mean, by a conditioning trick. Let $\mu = E[X]$. Look at the first trial. With probability p it succeeds, and you're done after 1 trial. With probability $1 - p$ it fails, you've used 1 trial, and, because trials are independent, you're back where you started, facing another μ trials on average. So

$$\begin{aligned} \mu &= p \cdot 1 + (1-p)(1+\mu) \\ \mu &= 1 + (1-p)\mu \\ p\mu &= 1 \\ \mu &= \frac{1}{p}. \end{aligned}$$

If a request fails with probability 0.25, so succeeds with probability $p = 0.75$, a retry loop needs $1/0.75 \approx 1.33$ attempts on average. Rolling a die until you get a six takes 6 rolls on average. (The variance is $(1-p)/p^2$.)

The geometric distribution is **memoryless**: having already failed 10 times doesn't make success any more likely on trial 11. That's the gambler's fallacy from Chapter 29 in distribution form.

30.4.4. Poisson: counting rare events

The problem. A web server receives on average 3 requests per second, from a huge population of users each of whom is very unlikely to send a request in any particular second. What's the probability of 0 requests in a given second? Of 6 or more?

Story: count events that happen independently, at a constant average rate, where each tiny time slice has only a tiny chance of an event. $X \sim \text{Poisson}(\lambda)$, where λ (*lambda*) is the average count per interval:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}, \qquad k = 0, 1, 2, \ldots, \qquad E[X] = \text{Var}(X) = \lambda.$$

Where it comes from. Chop the second into n tiny slices, each containing a request with probability $p = \lambda/n$. The count is binomial, $\text{Bin}(n, \lambda/n)$. As $n \to \infty$ with $np = \lambda$ fixed,

$$P(X = 0) = \left(1 - \frac{\lambda}{n}\right)^n \to e^{-\lambda}$$

by the compound-interest limit from Volume 1, Chapter 12 , and similar limits give the other terms. So the Poisson is the binomial for "many trials, each rare". Numerically: $(1 - 3/1000)^{1000} \approx 0.0496$, already close to $e^{-3} \approx 0.0498$.

Worked example. With $\lambda = 3$: *Try it:* Poisson with λ = 3 .

$$
\begin{aligned}
P(X = 0) &= e^{-3} \approx 0.0498 \\
P(X \le 5) &= e^{-3}\left(1 + 3 + \frac{3^2}{2!} + \frac{3^3}{3!} + \frac{3^4}{4!} + \frac{3^5}{5!}\right) \\
&= e^{-3}(1 + 3 + 4.5 + 4.5 + 3.375 + 2.025) \\
&= 0.04979 \times 18.4 \approx 0.916 \\
P(X \ge 6) &= 1 - 0.916 = 0.084.
\end{aligned}
$$

About 1 second in 20 sees no traffic, and about 1 in 12 sees double the average or more. If the server can handle 5 requests per second, it will be overloaded 8% of the time, even though the average load is only 60% of capacity. Randomness creates queues.

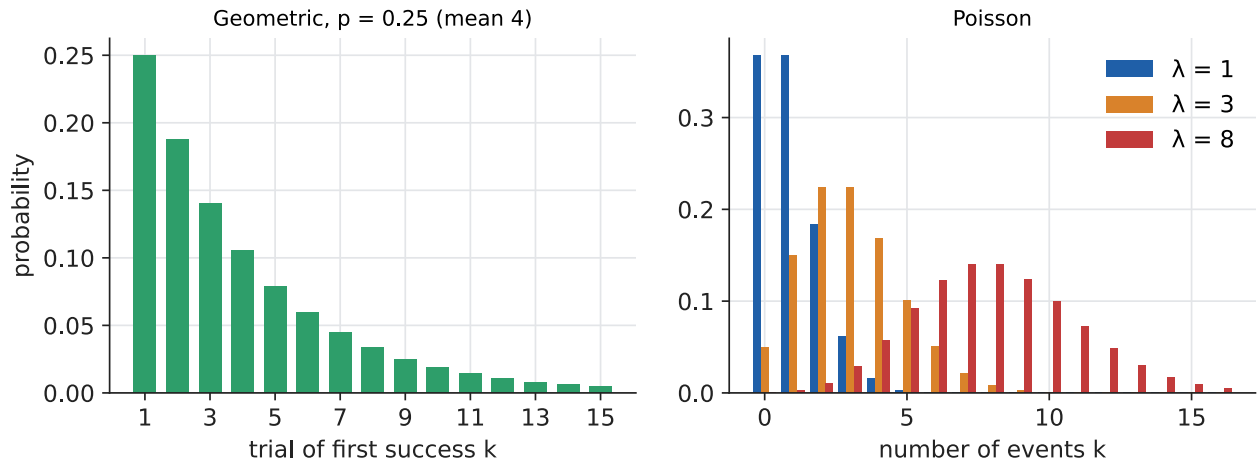


Figure 53. Left: the geometric distribution, always decreasing. Right: Poisson distributions for three rates; as λ grows the shape becomes more symmetric.

30.4.5. Choosing a discrete model

Question	Distribution	Typical use
Did one trial succeed?	Bernoulli(p)	a click, a bit flip, a test pass
How many successes in n fixed, independent trials?	Binomial(n, p)	conversions out of n visitors; defective chips in a batch
How many trials until the first success?	Geometric(p)	retries until success; attempts until a hash finds an empty slot
How many events in a fixed interval, at a constant rate?	Poisson(λ)	requests per second, typos per page, mutations per genome
Every one of n values equally likely?	Discrete uniform	dice, random array index, shuffles



Model assumptions

Every named distribution packages assumptions. Binomial needs a *fixed* number of trials with the *same* p and *independence*; users who visit twice, or a p that varies by user, break it. Poisson needs a constant rate and independent events; real traffic has daily cycles and bursts (one tweet sends thousands of users at once), so the variance of real request counts is often far larger than the mean. A quick diagnostic: for count data, if the sample variance is much bigger than the sample mean, a Poisson model is wrong.



Why "Poisson"? And the Prussian cavalry

Siméon Denis Poisson derived the distribution in 1837 while studying wrongful convictions. Its most famous early application was by Ladislaus Bortkiewicz in 1898, who showed that the number of Prussian cavalrymen killed by horse kicks per corps per year (20 years, 14 corps) fitted a Poisson distribution with $\lambda \approx 0.7$ almost perfectly. The same mathematics now counts photon arrivals in a camera sensor, radioactive decays, and packets arriving at a router.

30.4.6. Where you'd use this

- **Sport (Poisson).** Football goals are rare, roughly independent events: a typical top-flight match averages about 2.7 goals. Treating the total as Poisson(2.7), $P(\text{0--0}) = e^{-2.7} \approx 0.067$ and $P(\text{at most 1 goal}) = e^{-2.7}(1 + 2.7) \approx 0.25$. Betting companies price their odds from models built on exactly this, with a separate rate for each team.
- **Manufacturing and medicine (binomial).** A production line makes chips with a 2% defect rate. In a batch of 20, $P(\text{no defects}) = 0.98^{20} \approx 0.67$, so a third of batches contain at least one dud: that's how acceptance-sampling plans are set. Clinical trials use the same model: "how many of 200 patients will respond if the true rate is 30%?" is a $\text{Bin}(200, 0.3)$ question, with mean 60 and standard deviation $\sqrt{42} \approx 6.5$.
- **Computing (binomial): replication.** A database keeps 3 replicas, each independently down 1% of the time. All three are down with probability $0.01^3 = 10^{-6}$. A majority-quorum write needs at least 2 of 3 up: it fails if 2 or more are down, with probability $3(0.01)^2(0.99) + 0.01^3 \approx 0.0003$, so availability is 99.97%. (The independence assumption is the weak point: replicas in the same data centre fail together.)
- **Computing (geometric): retries.** A flaky network call succeeds 75% of the time. The number of attempts is geometric, so expect $1/0.75 \approx 1.33$ attempts, and the chance of needing more than 3 is $0.25^3 \approx 1.6\%$. That's how you choose a retry limit.

US curriculum

Discrete random variables, expected value and the binomial distribution appear in Common Core high-school standards (S-MD, "Using Probability to Make Decisions"), often in **Algebra II**, and are central to **AP Statistics** Unit 4 (random variables, binomial and geometric distributions). The Poisson distribution is usually first met in a college probability or statistics course.

30.5. Continuous distributions

30.5.1. The problem

How long until the next request arrives? It could be 0.3 s, 0.31 s, 0.3141 s: any real number. Now ask "what's the probability it's exactly 0.3 seconds, to infinitely many decimal places?". There are infinitely many possible values; if each had positive probability, the total would be infinite. So each *individual* value must have probability 0. Yet "between 0.2 and 0.4 seconds" clearly has positive probability. Lists of probabilities can't work; we need a different description.

30.5.2. Intuition: probability as area

Think of a histogram of a million measured waiting times, with very narrow bins, scaled so the total area of the bars is 1. The probability of landing in any range is the area of the bars over that range. As the bins get narrower, the bar tops trace out a smooth curve: the **probability density function**. Probability is area under it.



Probability density function (PDF)

A continuous random variable X has a **probability density function** $f(x)$ if

$$P(a \le X \le b) = \text{area under } f \text{ between } a \text{ and } b \quad \left(= \int_a^b f(x)\,dx\right).$$

(The integral notation is from Volume 2, Chapter 19 ; "area under the curve" is all you need here.)

A density satisfies $f(x) \ge 0$ and has total area 1. The height $f(x)$ is **not** a probability (it can exceed 1); it's probability *per unit length*, like population density is people per square mile. For a short interval of width Δx, $P(x \le X \le x + \Delta x) \approx f(x)\,\Delta x$.

For a continuous variable, $P(X = a) = 0$ for every single value a, so $P(a \le X \le b) = P(a < X < b)$: endpoints don't matter.

30.5.3. Cumulative distribution functions

One description works for discrete and continuous variables alike.



Cumulative distribution function (CDF)

The **cumulative distribution function** of X is

$$F(x) = P(X \le x).$$

It rises from 0 (far left) to 1 (far right) and never decreases. Then $P(a < X \le b) = F(b) - F(a)$. For a discrete variable F is a staircase, jumping by $p(x)$ at each value; for a continuous one it's a smooth curve whose slope is the density ($f = F'$, Volume 2, Chapter 17).

The **median** is the value where $F(x) = 0.5$; more generally the q-th **quantile** (or $100q$-th **percentile**) is where $F(x) = q$. "p99 latency" is the 0.99 quantile.

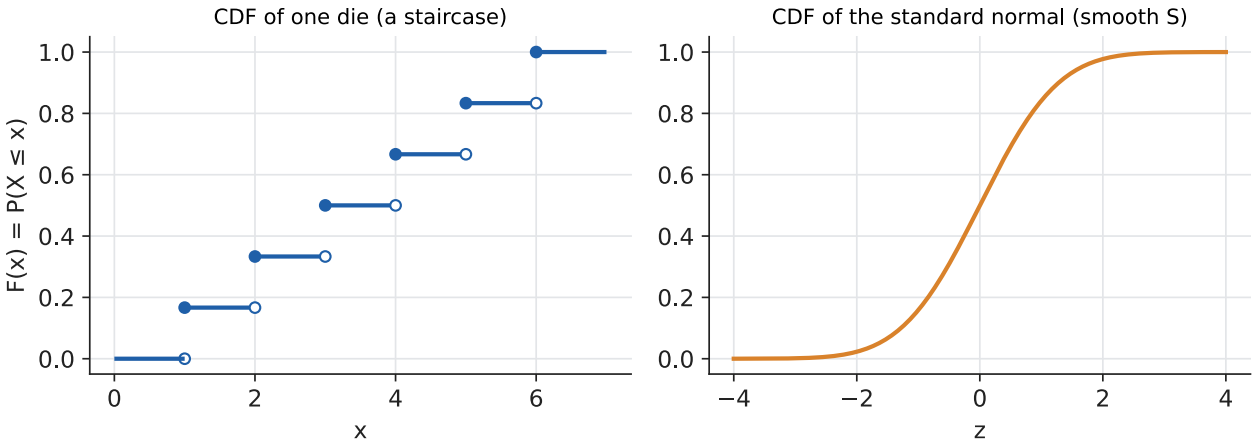


Figure 54. CDFs: a staircase for a discrete variable (one die), a smooth S-curve for a continuous one (standard normal).

30.5.4. Uniform: equally likely anywhere in an interval

Story: X is equally likely to land anywhere in $[a, b]$. The density is flat: $f(x) = \frac{1}{b-a}$ on the interval (so the rectangle has area 1), and 0 outside. Probabilities are proportional to length: $P(c \le X \le d) = \frac{d-c}{b-a}$. The mean is the midpoint $\frac{a+b}{2}$, and the variance is $\frac{(b-a)^2}{12}$.

The **standard uniform**, on $[0, 1]$, is what `random.random()` approximates, and as you'll see in [sec-dist-sampling], every other distribution can be built from it.

30.5.5. Exponential: waiting times

Story: the waiting time until the next event in a Poisson process with rate λ events per unit time. The time until the next request, the next radioactive decay, the next failure of a component that doesn't age.

Deriving it from Poisson. The wait T exceeds t exactly when there are zero events in $[0, t]$. In time t the count is Poisson with mean λt, so

$$P(T > t) = P(0 \text{ events in time } t) = e^{-\lambda t}, \qquad F(t) = P(T \le t) = 1 - e^{-\lambda t}.$$

The density is the slope of F: $f(t) = \lambda e^{-\lambda t}$ for $t \ge 0$. The mean is $1/\lambda$ (at 3 requests per second, the average gap is a third of a second), and the standard deviation is also $1/\lambda$.

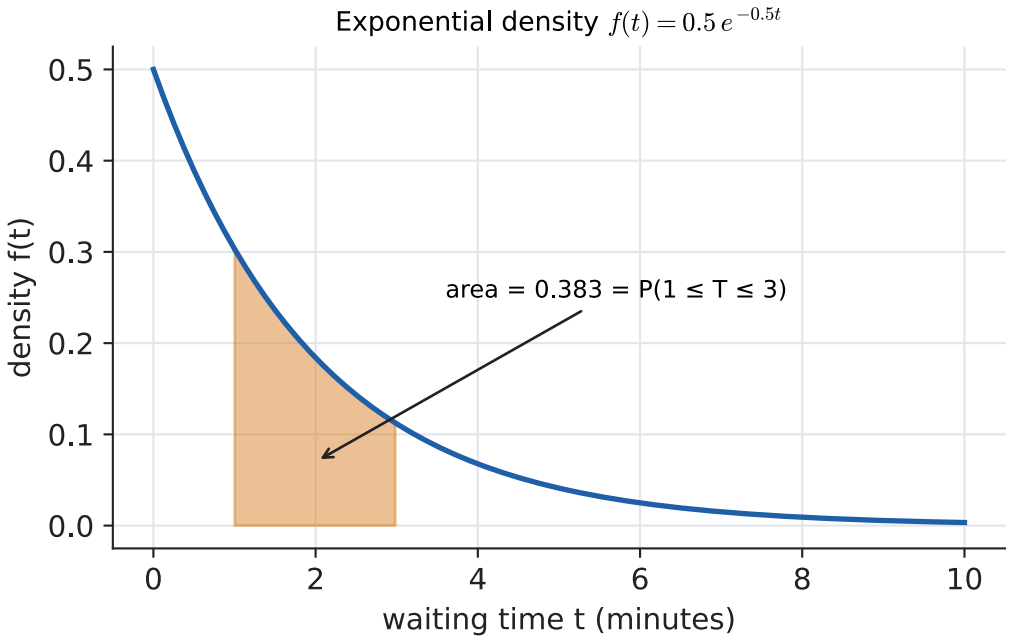


Figure 55. Exponential waiting time with rate 0.5 per minute (mean 2 minutes). The shaded area is the probability that the wait is between 1 and 3 minutes.

Worked example. Buses arrive as a Poisson process at rate $\lambda = 0.5$ per minute. What's the probability your wait is between 1 and 3 minutes? *Try it:* Exponential waiting time .

$$\begin{aligned} P(1 \le T \le 3) &= F(3) - F(1) \\ &= (1 - e^{-1.5}) - (1 - e^{-0.5}) \\ &= e^{-0.5} - e^{-1.5} \\ &= 0.6065 - 0.2231 = 0.383. \end{aligned}$$

Memorylessness. $P(T > s + t \mid T > s) = \frac{e^{-\lambda(s+t)}}{e^{-\lambda s}} = e^{-\lambda t} = P(T > t)$. If you've already waited 10 minutes, the remaining wait has the same distribution as if you'd just arrived. That's right for radioactive atoms and for well-behaved request streams, and wrong for things that wear out (hard disks, people), which need other models.



Optional: expectation as an integral

(Uses Volume 2, Chapter 19 ; skip freely.) For a continuous variable the sum $\sum x\,p(x)$ becomes an integral:

$$E[X] = \int_{-\infty}^{\infty} x f(x)\,dx, \qquad \mathrm{Var}(X) = \int_{-\infty}^{\infty} (x-\mu)^2 f(x)\,dx.$$

For the exponential, integrating by parts: $\int_0^\infty t\,\lambda e^{-\lambda t}\,dt = \left[-te^{-\lambda t}\right]_0^\infty + \int_0^\infty e^{-\lambda t}\,dt = 0 + \frac{1}{\lambda}$. Every rule from the discrete case (linearity, the variance formulas) carries over unchanged.

30.5.6. Where you'd use this

- **Physics: radioactive decay and half-life.** Each atom's lifetime is exponential with rate λ. The **half-life** is the median lifetime, where $e^{-\lambda t} = \frac{1}{2}$, i.e. $t_{1/2} = \ln 2/\lambda$. For carbon-14, $t_{1/2} = 5730$ years, so $\lambda = \ln 2/5730 \approx 1.21 \times 10^{-4}$ per year, the number behind radiocarbon dating.
- **Reliability engineering.** A component with a constant failure rate and a mean time between failures (MTBF) of 50,000 hours survives a year of continuous use (8,760 hours) with probability $e^{-8760/50000} \approx 0.84$. A rack of 100 such components can expect about 16 of them to fail within the year, which is how spare-parts stock and on-call rotas are planned.
- **Computing: simulation and load testing.** To mimic real users arriving at 3 per second, a load generator sleeps for exponentially distributed gaps with mean 1/3 s (generated as in [sec-dist-sampling]), not for exactly 1/3 s. Evenly spaced test traffic never produces the bursts that make real queues form.
- **Uniform:** rounding a measurement to the nearest millimetre leaves an error that is uniform on $[-0.5, 0.5]$ mm, with standard deviation $1/\sqrt{12} \approx 0.29$ mm; the same formula gives the "quantization noise" of an analogue-to-digital converter.

30.6. The normal distribution

30.6.1. The problem

Measure the heights of 10,000 adults, the errors of 10,000 repeated measurements of a length, or the average response time across 10,000 batches of requests. Plot histograms, and the same symmetric bell shape keeps appearing. Why one shape, and how do you compute with it?

The "why" is the central limit theorem (Section 30.7): any quantity that is the sum of many small, independent contributions comes out approximately bell-shaped. First, the "how".

30.6.2. The bell curve



Normal distribution

The **normal** (or **Gaussian**) distribution with mean μ and standard deviation σ, written $N(\mu, \sigma^2)$, has density

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

It is symmetric about μ (mean = median = mode), bell-shaped, and its tails thin out very fast. The **standard normal** is $Z \sim N(0, 1)$; its CDF is written $\Phi(z)$ (capital *phi*). The $\sqrt{2\pi}$ makes the total area 1.

Note: the second parameter in $N(\mu, \sigma^2)$ is the *variance*. $N(100, 225)$ has standard deviation 15.

You don't need to work with the formula directly. Two facts do almost all the work.

Standardizing. If $X \sim N(\mu, \sigma^2)$, then

$$Z = \frac{X - \mu}{\sigma} \sim N(0, 1).$$

Z (the **z-score**) counts how many standard deviations X is from its mean. So every normal question becomes a question about one curve.

The 68–95–99.7 rule. For any normal distribution, about 68% of the probability lies within 1σ of the mean, 95% within 2σ, and 99.7% within 3σ.

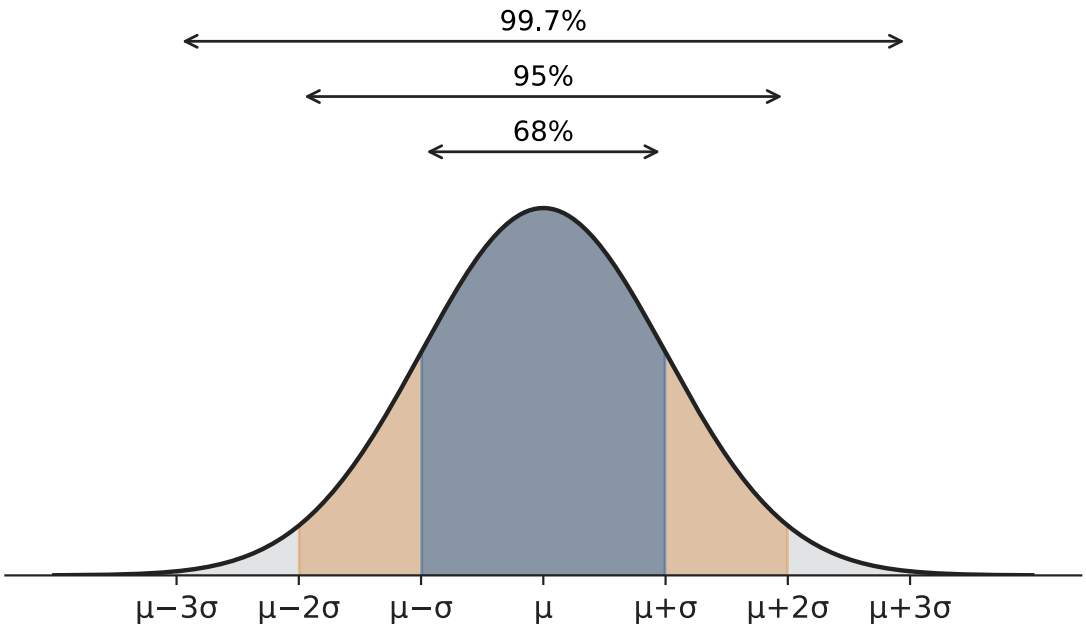


Figure 56. The 68–95–99.7 rule. The same percentages hold for every normal distribution, once distances are measured in standard deviations.

Example 42. Procedure — normal probabilities

Input: $X \sim N(\mu, \sigma^2)$ and an interval.

Steps:

1. Convert each endpoint x to a z-score: $z = (x - \mu)/\sigma$.
2. Sketch the bell curve and shade the region you want. Decide whether it's a left tail, right tail, or middle.
3. Use Φ (a table, a calculator, or code): left tail $\Phi(z)$; right tail $1 - \Phi(z)$; middle $\Phi(z_2) - \Phi(z_1)$. For a quick estimate, use 68–95–99.7.
4. Check the answer against the sketch (a right tail beyond $z = 2$ must be small).

Output: the probability.

Worked example. IQ scores are scaled to be approximately $N(100, 15^2)$. What fraction score above 130? *Try it:* Normal tail above 130 .

$$z = \frac{130 - 100}{15} = 2$$
$$P(X > 130) = P(Z > 2) = 1 - \Phi(2) \approx 1 - 0.9772 = 0.0228.$$

Quick check with the rule: 95% lie within $\pm 2\sigma$, so 5% lie outside, half of it above: 2.5%. ✓

PYTHON

```
from math import erf, sqrt

def normal_cdf(x, mu=0.0, sigma=1.0):
    """P(X <= x) for X ~ N(mu, sigma^2)."""
    z = (x - mu) / sigma
    return 0.5 * (1 + erf(z / sqrt(2)))

for k in (1, 2, 3):
    print(k, round(normal_cdf(k) - normal_cdf(-k), 4))
print(round(1 - normal_cdf(130, 100, 15), 4))
print(round(normal_cdf(250, 200, 40) - normal_cdf(150, 200, 40), 4))
```

```
1 0.6827
2 0.9545
3 0.9973
0.0228
0.7887
```

(The function `erf`, the *error function*, is in Python's `math` module; $\Phi(z) = \frac{1}{2}\left(1 + \operatorname{erf}(z/\sqrt{2})\right)$. The last line computes $P(150 \le X \le 250)$ for $X \sim N(200, 40^2)$, i.e. $P(-1.25 \le Z \le 1.25)$.)



Not everything is normal

Request latencies, file sizes, incomes and city populations are strongly right-skewed with long tails. Modelling them as normal predicts negative latencies and badly underestimates extreme values: under a normal model, a value 6 standard deviations above the mean has probability about one in a billion; in real latency data such outliers happen daily. Always plot the data before assuming a bell curve. Monitoring uses percentiles (p50, p99, p99.9) precisely because they don't assume a shape.



Gauss and the ten-mark note

The normal curve was first found by Abraham de Moivre in 1733 as an approximation to the binomial distribution for large n. Carl Friedrich Gauss used it around 1809 to model errors in astronomical observations, which is why it's often called Gaussian. Germany's old 10-Deutsche-Mark banknote carried Gauss's portrait with the bell curve and its formula printed beside him.

30.6.3. Where you'd use this

- **Shopping and the law: filling packets.** A machine fills crisp packets labelled 25 g with weights that are approximately $N(26, 0.5^2)$. The fraction under the label weight is $P(X < 25) = P(Z < -2) \approx 2.3\%$. Setting the machine's target is a z-score calculation: aim too low and you break trading-standards rules, too high and you give product away.
- **Medicine: reference ranges.** A blood-test "normal range" is usually the middle 95% of healthy people, mean ± 2 standard deviations. The flip side: a perfectly healthy person who has 20 independent tests can expect $20 \times 0.05 = 1$ result flagged as "abnormal".
- **Computing:** sensor noise is modelled as Gaussian in GPS filters and robotics; image "Gaussian blur" weights neighbouring pixels by a bell curve; neural-network weights are initialised with small normal random numbers; and anomaly detectors raise an alert when a metric's z-score passes 3, while remembering the watch-out above about heavy tails.

30.7. The law of large numbers and the central limit theorem

30.7.1. The problem

A single die roll is unpredictable. The average of 10,000 rolls is extremely predictable: it will be very close to 3.5. A single user's session length is unpredictable; the mean session length across a million users barely moves day to day. Why does averaging tame randomness, and how fast?

30.7.2. The law of large numbers

Let $X_1, X_2, \ldots, X_n$ be independent random variables, all with the same distribution (**independent and identically distributed**, or **i.i.d.**), with mean μ and standard deviation σ. Their **sample mean** is

$$\bar{X}_n = \frac{X_1 + X_2 + \cdots + X_n}{n}.$$

($\bar{X}$ is read "X-bar".) We computed in Section 30.3 that $E[\bar{X}_n] = \mu$ and $\mathrm{SD}(\bar{X}_n) = \sigma/\sqrt{n}$. As n grows, the spread of the average around μ shrinks to zero. That is the **law of large numbers**: the sample mean converges to the true mean.

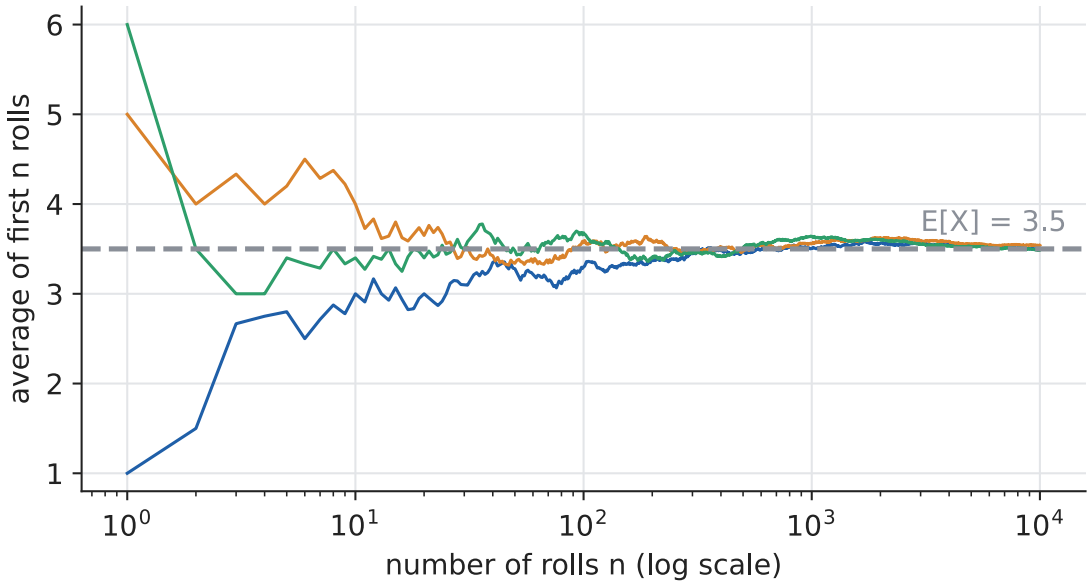


Figure 57. The law of large numbers: three runs of 10,000 die rolls. The running average wanders early, then is pinned ever closer to 3.5.

To make "converges" precise, **Chebyshev's inequality** says that any random variable is unlikely to be many standard deviations from its mean: $P(|Y - E[Y]| \geq k\sigma_Y) \leq 1/k^2$. Applied to $\bar{X}_n$, with a fixed tolerance ε (*epsilon*, a small positive number), *Try it:* Run the law of large numbers .

$$P(|\bar{X}_n - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{n\varepsilon^2} \xrightarrow{n\to\infty} 0.$$

This is why the frequentist interpretation of probability (Chapter 29) is coherent: the proportion of heads is the sample mean of Bernoulli indicators, so it converges to p. *Try it:* Proportion of heads converging .

30.7.3. The central limit theorem

The law of large numbers says the sample mean concentrates around μ. The **central limit theorem** (CLT) says what *shape* its fluctuations have: approximately normal, no matter what distribution the individual X_i have (provided their variance is finite).



Central limit theorem

If $X_1, \ldots, X_n$ are i.i.d. with mean μ and standard deviation σ, then for large n,

$$\bar{X}_n \approx N\left(\mu, \frac{\sigma^2}{n}\right), \qquad \text{equivalently} \qquad \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} \approx N(0,1).$$

The sum $X_1 + \cdots + X_n$ is likewise approximately $N(n\mu, n\sigma^2)$. The quantity $\sigma/\sqrt{n}$ is the **standard error** of the mean.

Watch it happen with an extremely un-normal starting distribution, the exponential (all mass on the positive side, a long right tail):

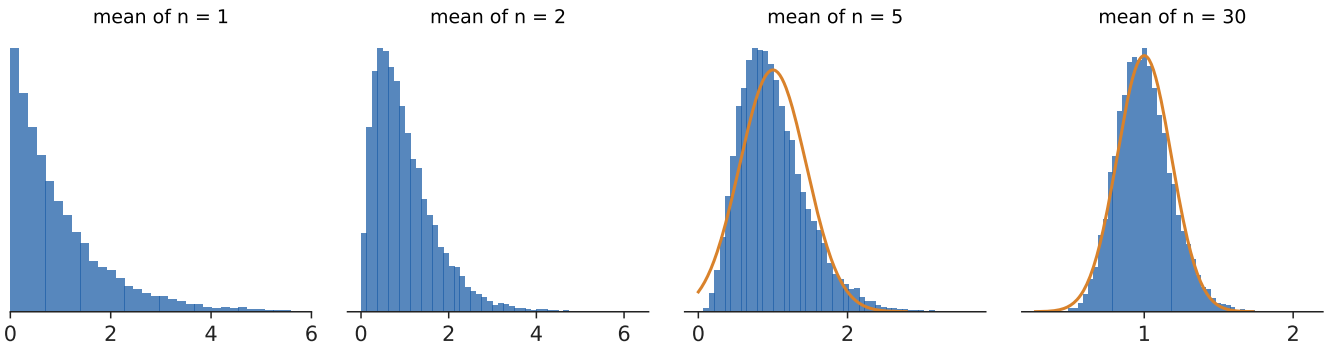


Figure 58. Histograms of 20,000 sample means of exponential(1) data. A single value (n = 1) is highly skewed; averages of 5 are already bell-like; averages of 30 are close to the normal curve (orange) with mean 1 and standard deviation 1/√n.

Why should sums become normal? A rough intuition: each extra term adds a small independent nudge; the fine details of each nudge's distribution wash out, and only the mean and variance survive. A sum of many small independent nudges is the situation the normal distribution describes. (The full proof uses more machinery than this book develops.) *Try it:* Build the bell curve yourself ; Exponential means, n = 1 .

PYTHON

```
import numpy as np

rng = np.random.default_rng(12)
print("  n   sd of means   1/sqrt(n)   below -1.96 SE   above +1.96 SE")
for n in (1, 5, 30, 100):
    # 100,000 experiments, each the mean of n exponential(1) values
    means = rng.exponential(1.0, size=(100_000, n)).mean(axis=1)
    se = 1 / np.sqrt(n)              # population sd is 1
    below = np.mean(means < 1 - 1.96 * se)
    above = np.mean(means > 1 + 1.96 * se)
    print(f"{n:3d}   {means.std():.3f}         {se:.3f}       "
          f"{below:.3f}            {above:.3f}")
```

```
  n   sd of means   1/sqrt(n)   below -1.96 SE   above +1.96 SE
  1   1.001         1.000       0.000            0.052
  5   0.448         0.447       0.000            0.044
 30   0.182         0.183       0.014            0.033
100   0.100         0.100       0.019            0.029
```

Read the output carefully. The standard deviation of the mean matches $\sigma/\sqrt{n}$ exactly at every n: that part is not an approximation. The *shape* approaches normal more slowly. For a normal distribution, 2.5% would fall in each tail beyond ± 1.96 standard errors; with skewed data, even at $n = 30$ the tails are lopsided (1.4% versus 3.3%).



The limits of the limit theorems

- **"$n \geq 30$ is enough"** is a classroom rule of thumb, not a theorem. For skewed data (latencies, revenue per user) you may need hundreds or thousands of samples before normal approximations are accurate in the tails.
- **Infinite variance breaks the CLT.** For very heavy-tailed distributions (the Cauchy distribution, some power laws), averages don't settle down at the $1/\sqrt{n}$ rate, or at all.
- **Independence matters.** Averaging 1,000 correlated measurements (the same user counted 1,000 times, time-series data with trends) gives far less than a $\sqrt{1000}$ reduction in noise.
- **The law of large numbers is not a law of averages.** It doesn't say a run of bad luck will be "corrected"; it says it will be *diluted*.

Try it: Watch the CLT fail (Cauchy) ; Cauchy running means never settle .



Galton's bean machine

Francis Galton built a device in the 1870s, the *quincunx*: beads fall through rows of pins, bouncing left or right at each one, and pile up in bins at the bottom. Each bead's final position is a sum of many independent ±1 bounces, a binomial variable, and the pile forms a bell curve every time. Galton wrote that the law "would have been personified by the Greeks and deified, if they had known of it".

30.7.4. Where you'd use this

- **Casinos and insurers.** A bet of £1 on red in European roulette wins £1 with probability $\frac{18}{37}$ and loses £1 with probability $\frac{19}{37}$: mean $-\frac{1}{37}$ pound, standard deviation almost exactly £1. Over a million such bets the casino's expected profit is $10^6/37 \approx 27{,}000$ pounds and, by the CLT, its standard deviation is only about $\sqrt{10^6} \times 1 = 1{,}000$ pounds. A loss would be a 27-standard-deviation event. Gamblers face randomness; the house faces arithmetic. Insurance works the same way.
- **Opinion polls.** Ask 1,000 random voters. If support is near 50%, each answer is a Bernoulli with standard deviation 0.5, so the poll's standard error is $0.5/\sqrt{1000} \approx 0.016$, and ± 1.96 standard errors is the familiar "margin of error ±3%". To get ±1% you need about 9,600 people: the $\sqrt{n}$ law again (Chapter 31).
- **Computing: Monte Carlo and A/B tests.** A Monte Carlo estimate (rendering a pixel by averaging random light paths, pricing an option, estimating π) has error proportional to $1/\sqrt{n}$: 100 times more samples for one more correct digit. The same law tells an A/B test how many users it needs before a 1% difference in click rate stands out from the noise.

Build your own bell curve

Choose any starting distribution, including a die, an exponential, a two-spike "0 or 100" distribution, or one you draw by hand. Choose n. The lab draws thousands of samples of size n and builds the histogram of their means live, overlaying the predicted normal curve $N(\mu, \sigma^2/n)$. A tail meter shows the fraction outside ± 1.96 standard errors versus the normal's 5%. Then switch to a Cauchy distribution and watch the CLT fail.

Try it: Bell curve from a die · Build your own bell curve · Bell curve from 0 or 100

US curriculum

Normal distributions, z-scores and the 68–95–99.7 ("empirical") rule appear in high-school standard S-ID.4, usually in **Algebra II**. Sampling distributions and the central limit theorem are a core part of **AP Statistics** (Unit 5, "Sampling Distributions"). Continuous distributions defined by integrals belong to college probability.

30.8. Joint distributions, covariance and correlation

30.8.1. The problem

Do slower pages have higher bounce rates? Does CPU load rise with request count? These questions are about *two* random variables at once and whether they move together.

30.8.2. Joint distributions

The **joint distribution** of discrete X and Y gives $P(X = x, Y = y)$ (the comma means "and") for every pair of values. It's a two-way table, exactly like the deploy table in Chapter 29.

Example. Roll a die. Let $X = 1$ if the roll is even (else 0), and $Y = 1$ if the roll is at least 4 (else 0). Rolls: 1 → (0,0), 2 → (1,0), 3 → (0,0), 4 → (1,1), 5 → (0,1), 6 → (1,1).

	$Y = 0$	$Y = 1$	$P(X = x)$
$X = 0$	$2/6$	$1/6$	$1/2$
$X = 1$	$1/6$	$2/6$	$1/2$
$P(Y = y)$	$1/2$	$1/2$	1

The row and column totals, written in the margins, are the **marginal distributions** of X and Y on their own. X and Y are **independent** if every cell equals the product of its marginals: $P(X = x, Y = y) = P(X = x)P(Y = y)$. Here $P(X = 1, Y = 1) = 2/6 \neq \frac{1}{2} \cdot \frac{1}{2}$, so they're dependent: knowing the roll is even makes "at least 4" more likely.

30.8.3. Covariance



Covariance

The **covariance** of X and Y, with means μ_X and μ_Y, is

$$\operatorname{Cov}(X,Y) = E\big[(X-\mu_X)(Y-\mu_Y)\big] = E[XY] - \mu_X\mu_Y.$$

Positive when X and Y tend to be above (or below) their means together; negative when one tends to be high as the other is low; zero (**uncorrelated**) when there's no linear tendency. $\operatorname{Cov}(X,X) = \operatorname{Var}(X)$.

Variance of a sum, in general: $\operatorname{Var}(X+Y) = \operatorname{Var}(X) + \operatorname{Var}(Y) + 2\operatorname{Cov}(X,Y)$.

Why the product? If both deviations have the same sign, their product is positive; if opposite signs, negative. Averaging the products measures which tendency wins.

For the die example: $E[XY] = P(X=1, Y=1) = \frac{2}{6} = \frac{1}{3}$ (since XY is 1 only when both are 1), and $\mu_X = \mu_Y = \frac{1}{2}$. So $\operatorname{Cov}(X,Y) = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$: positive, as expected.

30.8.4. Correlation

Covariance has awkward units (milliseconds times bounces) and depends on scale. Dividing by both standard deviations gives a pure number between -1 and 1:

$$\rho(X,Y) = \frac{\operatorname{Cov}(X,Y)}{\sigma_X\,\sigma_Y}.$$

ρ (*rho*) is the **correlation coefficient**. $\rho = 1$ means Y is an exact increasing linear function of X; $\rho = -1$, an exact decreasing one; $\rho = 0$, no linear relationship. For the die example, $\sigma_X = \sigma_Y = \sqrt{\frac{1}{2}\cdot\frac{1}{2}} = \frac{1}{2}$, so $\rho = \frac{1/12}{1/4} = \frac{1}{3}$.

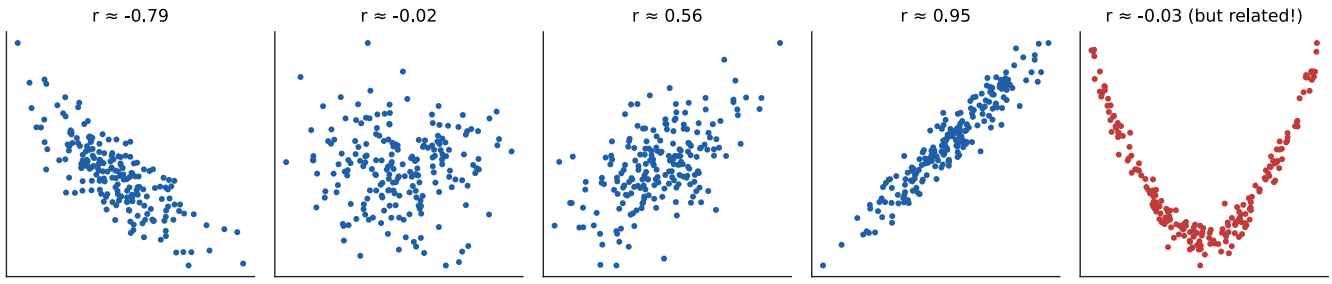


Figure 59. Scatter plots of 200 points with various correlations. The last panel has a strong relationship ($y \approx x^2$) but correlation near zero, because the relationship isn't linear.



Uncorrelated does not mean independent

Independent variables always have zero covariance ($E[XY] = E[X]E[Y]$ under independence). The converse fails: in the last scatter plot Y is almost determined by X, yet $\rho \approx 0$, because the upward and downward halves cancel. Correlation measures *linear* association only. And correlation is not causation; that's a theme of Chapter 31.

30.8.5. Where you'd use this

- **Finance: why diversification works.** Put half your money in each of two investments, each with yearly return standard deviation 10%. Using $\mathrm{Var}(\frac{1}{2}X + \frac{1}{2}Y) = \frac{1}{4}\sigma_X^2 + \frac{1}{4}\sigma_Y^2 + \frac{1}{2}\rho\sigma_X\sigma_Y$, the portfolio's standard deviation is 10% if $\rho = 1$, about 7.1% if $\rho = 0$, and 5% if $\rho = -0.5$. Diversification buys you risk reduction only to the extent that the investments are *not* correlated, which is why crashes, when everything falls together, hurt so much.
- **Science: finding relationships.** Epidemiologists start with correlations (smoking and lung cancer, height of parents and children) and then work hard to rule out confounders before claiming cause (Chapter 31).
- **Computing:** recommender systems measure how strongly two users' ratings are correlated ("people like you also liked…"); monitoring dashboards flag metrics that suddenly become correlated (latency and one particular host); and the covariance matrix in the sidebar below drives PCA.

Computer-science connection: covariance in machine learning

Collect d features and you get a $d \times d$ **covariance matrix** with $\mathrm{Cov}(X_i, X_j)$ in entry (i, j) and variances on the diagonal. Its eigenvectors (Volume 2, Chapter 28) are the directions of greatest variance: that's principal component analysis, the workhorse of dimensionality reduction. Correlation is also cosine similarity in disguise: for centred data vectors, ρ is the cosine of the angle between them (Volume 2, Chapter 26), which is why the same dot product appears in statistics, search ranking and embeddings.

30.9. Distributions at work in computing

30.9.1. Expected running time

Linear search. Search an array of n distinct items for one that's equally likely to be in any position. The number of comparisons is uniform on $\{1, \dots, n\}$, so the expected number is $\frac{1+2+\cdots+n}{n} = \frac{n+1}{2}$.

Hash table chains. Insert n keys into m buckets with a good hash. For a given bucket, the number of keys in it is a sum of n indicators each with probability $1/m$, so by linearity its expected length is n/m, the **load factor** α. An unsuccessful lookup scans a chain of expected length α: constant time if you keep α bounded by resizing.

Randomized quicksort (promised in Chapter 29). Let the sorted values be $z_1 < z_2 < \cdots < z_n$. Quicksort only ever compares an element with the current pivot, and any pair is compared at most once. Let $I_{ij} = 1$ if z_i and z_j ($i < j$) are ever compared. Total comparisons $C = \sum_{i<j} I_{ij}$.

Focus on the block $z_i, z_{i+1}, \dots, z_j$ ($j - i + 1$ values). Until a pivot is chosen from this block, all of them stay together in the same subarray. When the first pivot from the block is chosen, if it's z_i or z_j, those two get compared; if it's anything in between, they're split into different subarrays and never meet. Every element of the block is equally likely to be that first pivot, so

$$
\begin{aligned}
P(I_{ij} = 1) &= \frac{2}{j - i + 1} \\
E[C] &= \sum_{i<j} \frac{2}{j-i+1} && \text{(linearity)} \\
&\le \sum_{i=1}^{n} \sum_{d=1}^{n} \frac{2}{d+1} && (d = j - i\text{; extend the inner sum}) \\
&\le 2nH_n \approx 2n \ln n,
\end{aligned}
$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n + 0.577$ is the n-th **harmonic number**. (The exact value is $2(n+1)H_n - 4n$.) No input is bad; only the coin flips are random.

```python
import random

def quicksort_comparisons(a, rng):
    """Return number of comparisons randomized quicksort makes on list a."""
    if len(a) <= 1:
        return 0
    pivot = a[rng.randrange(len(a))]
    less = [x for x in a if x < pivot]
    more = [x for x in a if x > pivot]
    # every non-pivot element is compared with the pivot once
    return (len(a) - 1) + quicksort_comparisons(less, rng) \
                        + quicksort_comparisons(more, rng)

n = 1000
H = sum(1 / k for k in range(1, n + 1))
exact = 2 * (n + 1) * H - 4 * n
rng = random.Random(7)
runs = [quicksort_comparisons(list(range(n)), rng) for _ in range(200)]
print(round(exact, 1), sum(runs) / len(runs))
```

```
10985.9 10957.005
```

The input `list(range(n))` is already sorted, the worst case for a first-element pivot (about 500,000 comparisons); with random pivots it costs about 11,000.

Try it: Collect 50 coupons . ==== The coupon collector

The problem. A cereal brand puts one of n equally likely toys in each box. How many boxes until you have all n? The same question: how many random requests until every one of n servers has been hit; how many random tests until every one of n code paths is covered.

The method. Split the process into phases. Phase k starts when you have $k-1$ distinct toys and ends when you get a new one. During phase k, each box is new with probability $p_k = \frac{n-k+1}{n}$, so the phase length is geometric with mean $\frac{n}{n-k+1}$. By linearity,

$$E[T] = \frac{n}{n} + \frac{n}{n-1} + \frac{n}{n-2} + \cdots + \frac{n}{1}$$
$$= n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) = nH_n \approx n \ln n.$$

For $n = 50$: $50 \times H_{50} \approx 50 \times 4.499 = 225$ boxes. The last toy alone takes 50 boxes on average; the first takes 1.

PYTHON

```python
import random

def draws_to_collect(n, rng):
    seen, draws = set(), 0
    while len(seen) < n:
        seen.add(rng.randrange(n))
        draws += 1
    return draws

n = 50
H = sum(1 / k for k in range(1, n + 1))
rng = random.Random(8)
sim = sum(draws_to_collect(n, rng) for _ in range(20_000)) / 20_000
print(round(n * H, 2), round(sim, 2))
```

```
224.96 224.48
```

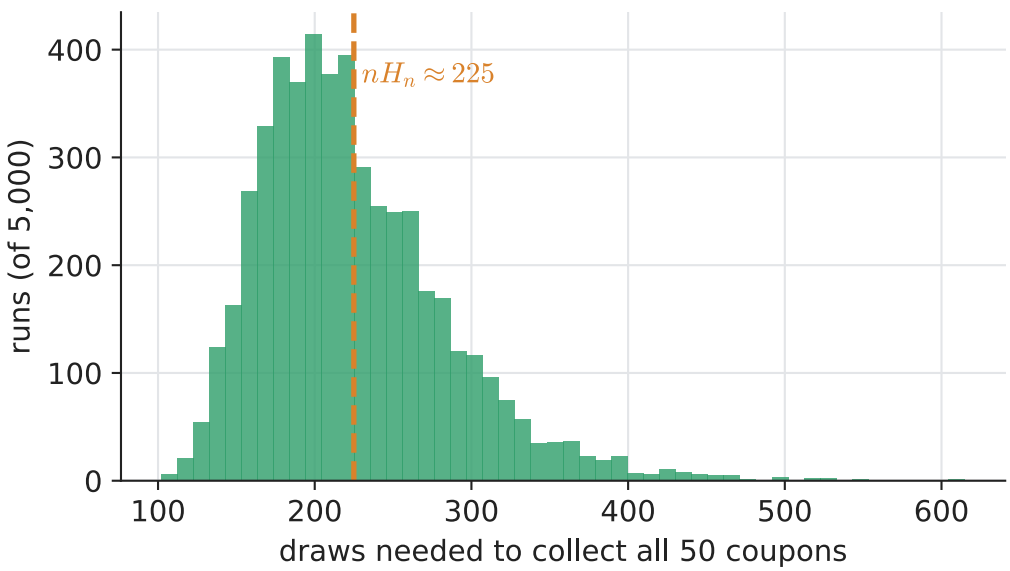


Figure 60. Distribution of the number of draws needed to collect 50 coupons, from 5,000 simulated runs. The mean is near $nH_n \approx 225$, but the right tail is long.

Try it: Balls into bins, two choices . ==== Balls into bins: load balancing

Throw n requests (balls) at n servers (bins) uniformly at random.

- **Expected load per server:** exactly 1, by linearity.

- **Expected fraction of idle servers:** a given server misses every ball with probability $(1 - \frac{1}{n})^n \approx e^{-1} \approx 0.368$. So about 37% of servers sit idle while others are double- or triple-loaded.
- **Maximum load:** grows slowly, like $\frac{\ln n}{\ln \ln n}$, but it does grow.

The **power of two choices**: for each ball, pick two random bins and put the ball in the less loaded one. This tiny change cuts the maximum load to about $\frac{\ln \ln n}{\ln 2}$, an exponential improvement. It's used in real load balancers (for example, NGINX's `random two` method) and in hash tables (cuckoo hashing).

PYTHON

```python
import random

def max_load(n, choices, rng):
    bins = [0] * n
    for _ in range(n):
        if choices == 1:
            b = rng.randrange(n)
        else:
            c1, c2 = rng.randrange(n), rng.randrange(n)
            b = c1 if bins[c1] <= bins[c2] else c2
        bins[b] += 1
    return max(bins), bins.count(0) / n

rng = random.Random(9)
n = 100_000
print("one choice :", max_load(n, 1, rng))
print("two choices:", max_load(n, 2, rng))
```

```
one choice : (8, 0.36719)
two choices: (4, 0.23785)
```

Try it: Inverse-transform sampling, animated . ==== Sampling from a distribution: inverse transform sampling

The problem. Your language gives you `random()`, uniform on $[0, 1)$. A simulation needs exponential inter-arrival times, or a weighted choice among options. How do you turn uniform randomness into any distribution?

Intuition. Look at the CDF F. It maps values to probabilities in $[0, 1]$. Run it backwards: pick a height u uniformly on the vertical axis, move across to the curve, and drop down to the x axis. Where the CDF is steep (high density), a wide band of heights maps to a narrow band of x, so those x values get picked often. Exactly right.

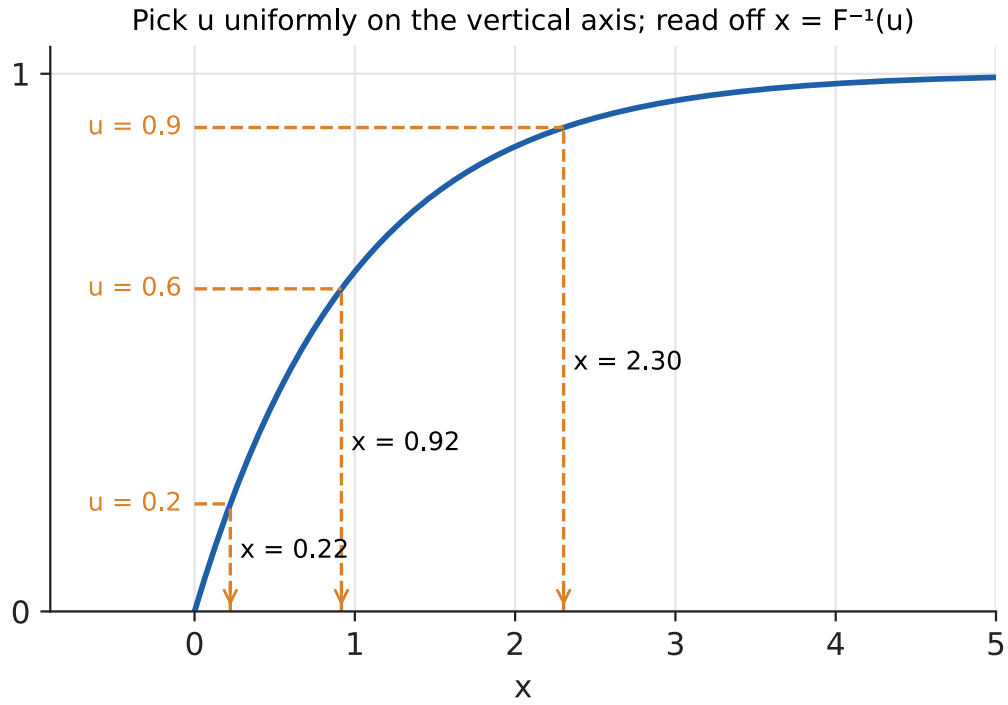


Figure 61. Inverse transform sampling for the exponential distribution. Uniform heights u map to x = -ln(1 - u); the steep part of the CDF near 0 collects many of them.

Why it works. Let U be uniform on $[0, 1]$ and $X = F^{-1}(U)$ (the inverse function, Volume 1, Chapter 11). Since F is increasing, $X \le x$ exactly when $U \le F(x)$, so $P(X \le x) = P(U \le F(x)) = F(x)$: the uniform's CDF is $P(U \le u) = u$. So X has CDF F, as required.

For the exponential, solve $u = 1 - e^{-\lambda x}$ for x:

$$\begin{aligned} e^{-\lambda x} &= 1 - u \\ -\lambda x &= \ln(1 - u) \qquad \text{(take natural logs)} \\ x &= -\frac{\ln(1 - u)}{\lambda}. \end{aligned}$$

For a discrete distribution, the CDF is a staircase, and "inverting" it means: walk up the staircase until the cumulative probability first exceeds u. *Try it:* Walk up the CDF staircase .

Example 43. Procedure — inverse transform sampling

Input: a target distribution with CDF F; a source of uniform random numbers on $[0, 1)$.

Steps:

1. Draw u uniformly from $[0, 1)$.
2. **Continuous case:** solve $F(x) = u$ for x (by algebra if possible, else numerically, as in Volume 2, Chapter 32) and return $x = F^{-1}(u)$.
3. **Discrete case:** with values $v_1, v_2, \ldots$ and probabilities $p_1, p_2, \ldots$, accumulate $c = p_1 + p_2 + \cdots$ and return the first v_i for which $u < c$.
4. Repeat for each sample needed (fresh u each time).

Output: samples with distribution F.

```python
import math, random

def sample_exponential(rate, rng=random):
    u = rng.random()                # uniform on [0, 1)
    return -math.log(1 - u) / rate  # inverse of F(x) = 1 - exp(-rate x)

def sample_discrete(values, probs, rng=random):
    u = rng.random()
    cumulative = 0.0
    for v, p in zip(values, probs):
        cumulative += p             # walk up the CDF staircase
        if u < cumulative:
            return v
    return values[-1]               # guard against rounding

random.seed(10)
xs = [sample_exponential(0.5) for _ in range(100_000)]
print(round(sum(xs) / len(xs), 3))                    # should be near 1/0.5 = 2
print(round(sum(x > 3 for x in xs) / len(xs), 3), round(math.exp(-1.5), 3))
draws = [sample_discrete(["a", "b", "c"], [0.5, 0.3, 0.2]) for _ in range(100_000)]
print({v: round(draws.count(v) / len(draws), 3) for v in "abc"})
```

```
1.994
0.223 0.223
{'a': 0.499, 'b': 0.302, 'c': 0.199}
```

Alternatives and choosing. Inverse transform is the default whenever F^{-1} has a formula. The normal distribution doesn't, so libraries use special methods (the Box–Muller transform, or the faster ziggurat algorithm). For discrete distributions with many values sampled many times, the linear walk is slow; use binary search on the cumulative array ($O(\log n)$, this is what Python's `random.choices` does) or the **alias method** ($O(1)$ per sample after setup). **Rejection sampling** handles densities you can evaluate but not invert.



> *Using 1 - u, not u*
> `random()` can return exactly 0 but never 1. `-log(u)` would then crash on $\ln 0$; `-log(1 - u)` never does. (Statistically, U and $1 - U$ have the same distribution, so both formulas are correct in principle; the difference is purely a floating-point edge case, the kind of thing Volume 2, Chapter 32 is about.)

Try it: Step through reservoir sampling . ==== Reservoir sampling

The problem. Choose k items uniformly at random from a stream (log lines, tweets, packets) whose length you don't know in advance and which is too big to store. You get one pass.

The method (Algorithm R). Keep the first k items in a "reservoir". For the i-th item ($i > k$), keep it with probability k/i, replacing a uniformly random reservoir slot.

Why it works (the case $k = 1$). Keep item i with probability $1/i$. Item i survives to the end of a stream of length n if it's chosen at step i and not replaced at any later step $i+1, \dots, n$:

$$P(\text{item } i \text{ is the final choice}) = \frac{1}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \cdots \times \frac{n-1}{n}$$
$$= \frac{1}{n}. \qquad \text{(the product telescopes)}$$

Every item ends up equally likely, whatever n turns out to be. The general-k argument is the same with k/i and a factor $\left(1 - \frac{1}{j}\right)$ for "replaced at step j *and* that slot was mine", and gives k/n for every item (an induction proof, Chapter 22).

```python
import random
from collections import Counter

def reservoir_sample(stream, k, rng=random):
    """Uniform random sample of k items from an iterable of unknown length."""
    reservoir = []
    for i, item in enumerate(stream, start=1):   # i = items seen so far
        if i <= k:
            reservoir.append(item)
        else:
            j = rng.randrange(i)                 # 0 .. i-1
            if j < k:                            # happens with prob k/i
                reservoir[j] = item
    return reservoir

random.seed(11)
counts = Counter()
for _ in range(100_000):
    counts.update(reservoir_sample(range(10), 3))
print([round(counts[x] / 100_000, 3) for x in range(10)])   # each ~ 0.3
```

```
[0.301, 0.299, 0.3, 0.301, 0.301, 0.3, 0.299, 0.298, 0.3, 0.299]
```

Each of the 10 items appears in the sample with probability $3/10$, as promised.

Computer-science connection: where distributions live in systems

- **Queueing.** Poisson arrivals and exponential service times give the M/M/1 queue, whose average wait grows like $\frac{1}{1-\rho}$ as utilisation ρ approaches 1, the mathematical reason a server at 95% CPU feels far worse than one at 80%.
- **Retry and backoff.** The number of attempts is geometric; exponential backoff with random "jitter" spreads retries out so that clients don't stampede together.
- **Sketches.** HyperLogLog estimates the number of distinct items in a stream from the maximum run of leading zeros in their hashes, a geometric random variable.
- **Testing.** Property-based testing and fuzzing draw inputs from distributions; choosing them well (more small values and edge cases) is applied probability.



Balls, bins and two choices

Animate requests falling into a row of servers. Toggle between "one random choice" and "best of two random choices" and watch the tallest stack, the number of idle servers, and a histogram of loads evolve. A second panel collects coupons: watch the long final wait for the last missing coupon, and compare the average against nH_n.

Try it: Balls, bins and two choices · Coupon collector simulation

30.10. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Statistics
- Sampling and simulation

30.11. Exercises

1. **(Recognition)** Which distribution (Bernoulli, binomial, geometric, Poisson, uniform, exponential, normal) best models each: (a) the number of 500 errors in a minute on a busy service; (b) whether a single user clicks an ad; (c) the number of heads in 20 flips; (d) the time until the next earthquake in a region; (e) the number of rolls until the first six; (f) the measurement error of a well-calibrated scale.
2. **(Recognition)** X takes values 0, 1, 2 with probabilities 0.5, 0.3, 0.2. Find $E[X]$, $E[X^2]$, $\mathrm{Var}(X)$ and σ.
3. **(Application)** A test has 20 multiple-choice questions with 4 options each. A student guesses every answer. Find the expected number correct, the standard deviation, and the probability of exactly 5 correct.

4. **(Application)** Emails arrive at an average rate of 2 per hour. Find the probability of (a) no emails in an hour; (b) at least 3 emails in an hour; (c) waiting more than 30 minutes for the next email.
5. **(Application)** Adult heights in a population are approximately $N(170, 10^2)$ cm. Use the 68–95–99.7 rule to estimate the fraction taller than 190 cm and the fraction between 160 and 180 cm.
6. **(Application)** A sample of $n = 400$ response times has population standard deviation $\sigma = 200$ ms. What is the standard error of the sample mean? How large must n be to make the standard error 5 ms?
7. **(Problem solving)** Use indicators to find the expected number of pairs of people sharing a birthday in a room of 30 (365 equally likely days).
8. **(Problem solving)** You roll a die until you've seen every face. What's the expected number of rolls?
9. **(Problem solving)** Derive an inverse-transform formula for sampling from the density $f(x) = 2x$ on $[0, 1]$ (CDF $F(x) = x^2$).
10. **(Problem solving, CS)** A hash table has 1,000 buckets and holds 2,000 keys. Find the expected chain length and the expected number of empty buckets.

▼ **Answers**

1. (a) Poisson; (b) Bernoulli; (c) binomial(20, 0.5); (d) exponential (if quakes occur at a constant rate, which is only roughly true); (e) geometric(1/6); (f) normal.
2. $E[X] = 0 + 0.3 + 0.4 = 0.7$; $E[X^2] = 0 + 0.3 + 0.8 = 1.1$; $\text{Var} = 1.1 - 0.49 = 0.61$; $\sigma \approx 0.781$.
3. $X \sim \text{Bin}(20, 0.25)$: $E = 5$; $\sigma = \sqrt{20 \times 0.25 \times 0.75} = \sqrt{3.75} \approx 1.94$.
   $P(X = 5) = \binom{20}{5}(0.25)^5(0.75)^{15} = 15504 \times 0.000977 \times 0.01336 \approx 0.202$.
4. (a) $e^{-2} \approx 0.135$. (b) $1 - e^{-2}(1 + 2 + 2) = 1 - 5e^{-2} \approx 1 - 0.677 = 0.323$. (c) Rate 2 per hour = 1 per 30 min; $P(T > 0.5\text{ h}) = e^{-2 \times 0.5} = e^{-1} \approx 0.368$.
5. 190 is $z = 2$: about 2.5% taller. 160 to 180 is $\pm 1\sigma$: about 68%.
6. $200/\sqrt{400} = 10$ ms. Need $200/\sqrt{n} = 5$, so $\sqrt{n} = 40$, $n = 1600$. (Halving the error took four times the data.)
7. There are $\binom{30}{2} = 435$ pairs, each matching with probability $1/365$. Expected matching pairs $= 435/365 \approx 1.19$. (Indicators are dependent; linearity doesn't care.)
8. Coupon collector with $n = 6$: $6H_6 = 6(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}) = 6 \times 2.45 = 14.7$ rolls.
9. Solve $u = x^2$: $x = \sqrt{u}$. So sample u uniform and return $\sqrt{u}$. (Check: $P(\sqrt{U} \le x) = P(U \le x^2) = x^2$ ✓.)
10. Load factor $2000/1000 = 2$. A bucket is empty with probability $(1 - \frac{1}{1000})^{2000} \approx e^{-2} \approx 0.135$, so about 135 empty buckets.

30.12. Mastery check

1. In your own words: what is a random variable, and why is "variable" a misleading name?
2. A game pays £10 with probability 0.1, £2 with probability 0.3, and nothing otherwise. Find the expected payout and its standard deviation.
3. Derive the binomial formula $P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$, explaining where each factor comes from.
4. Explain why $P(X = 3) = 0$ for a continuous variable, and what a density value $f(3) = 0.4$ does mean.
5. $X \sim N(50, 4^2)$. Find $P(X > 58)$ approximately, without a calculator.
6. State the central limit theorem and explain why the standard deviation of a sample mean is $\sigma/\sqrt{n}$.
7. 10 people are randomly seated at a round table of 10 seats where they had assigned places. Expected number in their assigned seat?
8. Write (in words or code) how to sample from an exponential distribution with mean 4 using `random()`.

Decision rule: All correct without hesitation → move on to Chapter 31. Stuck on 1–2 → reread Section 30.2.4 and Section 30.7. Stuck on more → work the exercises first, simulating each answer in Python.

▼ **Mastery check answers**

1. A function that assigns a number to every outcome of a random experiment. It's not a variable in the algebra sense (an unknown) or the programming sense (a mutable store): it's a fixed rule, and the randomness lives in which outcome occurs.
2. $E = 10(0.1) + 2(0.3) = 1.6$. $E[X^2] = 100(0.1) + 4(0.3) = 11.2$; $\text{Var} = 11.2 - 2.56 = 8.64$; $\sigma = 2.94$.
3. Any particular sequence with k successes and $n-k$ failures has probability $p^k(1-p)^{n-k}$ by independence; there are $\binom{n}{k}$ such sequences (choose the success positions); they're disjoint, so add.
4. There are infinitely many possible values, so each individual value has probability zero; probability lives in intervals (areas). $f(3) = 0.4$ means probability is 0.4 per unit near 3: $P(3 \le X \le 3.01) \approx 0.4 \times 0.01 = 0.004$.
5. $z = (58 - 50)/4 = 2$; about 2.5% (exactly 2.28%).
6. For i.i.d. X_i with mean μ, sd σ, the sample mean is approximately $N(\mu, \sigma^2/n)$ for large n. Variance of the sum of n independent terms is $n\sigma^2$; dividing by n divides variance by n^2, giving σ^2/n, so sd $\sigma/\sqrt{n}$.

7. Indicators: each person in their seat with probability $1/10$; expected number = 1.

8. $\lambda = 1/4$. Draw $u =$ `random()`, return $-4\ln(1-u)$.

Chapter 31. Statistics and Inference

*Probability runs forwards: given the coin is fair, how many heads should you expect? Statistics runs backwards: given 61 heads in 100 tosses, what can you say about the coin? Every A/B test, latency dashboard, opinion poll, clinical trial and model evaluation is this backwards question in disguise. This chapter starts with describing data honestly (centre, spread, shape, outliers, and the pictures that show them), then builds the machinery of inference: sampling, standard errors, confidence intervals, hypothesis tests and p-values, regression. It spends as much time on what these tools do **not** mean as on how to compute them, because statistical mistakes are rarely arithmetic mistakes. It finishes with an A/B test run end to end, the bootstrap, and the precision/recall metrics that evaluate classifiers and search engines.*

Where this chapter sits

You need: probability and conditional probability (Chapter 29), random variables, expectation, variance, the normal distribution and the central limit theorem (Chapter 30), and straight lines (Volume 1, Chapter 5). One optional derivation uses partial derivatives (Volume 2, Chapter 20).

This chapter unlocks: experimentation and data science in practice; the evaluation side of machine learning and information retrieval (Chapter 33); and least squares in matrix form (Section 33.2), which generalizes the regression here to many variables.

31.1. Describing data: centre

31.1.1. The problem

A service logged these 11 response times, in milliseconds:

$$120,\ 95,\ 110,\ 130,\ 105,\ 98,\ 102,\ 115,\ 940,\ 108,\ 125.$$

Someone asks "how fast is the service?". You need one number, or a few, that summarise the list honestly. Which?

31.1.2. Intuition: three kinds of "typical"

- The **mean** is the balance point: total divided by count.
- The **median** is the middle value when sorted: half the data below, half above.
- The **mode** is the most common value (most useful for categories, such as "most common browser").

Sorted, the data are 95, 98, 102, 105, 108, **110**, 115, 120, 125, 130, 940. The median is the 6th of 11 values: 110 ms. The mean is

$$\bar{x} = \frac{95 + 98 + 102 + 105 + 108 + 110 + 115 + 120 + 125 + 130 + 940}{11} = \frac{2048}{11} \approx 186.2 \text{ ms}.$$

Ten of the eleven requests took 95–130 ms, yet the mean is 186 ms, larger than every one of them. A single slow request (perhaps a cache miss or a garbage-collection pause) dragged it up. Remove the 940 and the mean drops to 110.8 while the median barely moves (110 to 109).



Mean, median, mode

For data $x_1, x_2, \ldots, x_n$:

- **Mean** (arithmetic mean, average): $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. The bar over x means "sample mean". It uses every value, so it is sensitive to extreme values.
- **Median**: the middle value of the sorted data (the average of the two middle values if n is even). It is **robust**, meaning a few extreme values can't move it far.
- **Mode**: the most frequent value.

A **statistic** is any number computed from data. Statistics (the subject) studies how to compute and interpret them.

31.1.3. Choosing a measure of centre

- **Mean**, when the data are roughly symmetric without wild outliers, or when totals matter. (Total server cost depends on the mean cost per request, outliers included.)
- **Median**, for skewed data or data with outliers: incomes, house prices, latencies, time-on-page. "Typical user experience" is a median question.
- **Mode**, for categories, or to notice that data have two peaks (**bimodal**: perhaps cached and uncached requests), in which case no single "centre" is honest.



Averages of averages

The mean of per-day means is not the overall mean unless every day has the same number of requests. A server handling 1,000 requests at 100 ms on Monday and 10 requests at 1,000 ms on Tuesday has an overall mean of $(100\,000 + 10\,000)/1010 \approx 109$ ms, not $(100 + 1000)/2 = 550$ ms. Dashboards that average pre-aggregated means or percentiles get this wrong constantly. (You cannot average p99s at all; you must recompute them from the raw data or from mergeable sketches.)

31.2. Describing data: spread, shape and outliers

31.2.1. The problem

Two services both have a median latency of 110 ms. One is between 100 and 120 ms almost always; the other ranges from 20 ms to 3 s. You need measures of spread, and pictures that show shape.

31.2.2. Standard deviation

From Chapter 30, the variance of a random variable is the expected squared deviation from its mean. For data, average the squared deviations from $\bar{x}$, with one twist:

Sample variance and standard deviation

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2, \qquad s = \sqrt{s^2}.$$

s is the **sample standard deviation**, roughly the typical distance of a data point from the mean, in the data's units. Greek letters (μ, σ) denote **population** quantities, the true values for the whole population or process; Latin letters ($\bar{x}$, s) denote **sample** quantities computed from data.

Why $n-1$? The deviations are measured from $\bar{x}$, which was fitted to the same data and so sits closer to the data than the true mean μ does. Dividing by n would systematically underestimate σ^2; dividing by $n-1$ (**Bessel's correction**) exactly fixes the average bias. For large n it hardly matters.

For the 11 response times, $s \approx 250$ ms; without the 940, $s \approx 11.6$ ms. Like the mean, the standard deviation is wrecked by one outlier, since it squares the deviations.

31.2.3. Quartiles and the five-number summary

Robust measures of spread use positions in the sorted data. The **quartiles** split it into four equal parts.

Quartiles, IQR, percentiles, five-number summary

- **First quartile Q_1**: a quarter of the data lie below it (the 25th **percentile**).
- **Median Q_2** (50th percentile), **third quartile Q_3** (75th percentile).
- **Interquartile range $\mathrm{IQR} = Q_3 - Q_1$**: the spread of the middle half of the data. Robust to outliers.
- **Five-number summary:** minimum, Q_1, median, Q_3, maximum.
- **The 1.5 × IQR rule:** a value below $Q_1 - 1.5\,\mathrm{IQR}$ or above $Q_3 + 1.5\,\mathrm{IQR}$ is flagged as a possible **outlier**.

Example 44. Procedure — five-number summary and outliers

Input: a list of n numbers.

Steps:

1. Sort the data.
2. The median is the middle value (or mean of the two middle values).
3. Q_1 is the median of the lower half, Q_3 the median of the upper half. (Convention used here: if n is odd, leave the overall median out of both halves.)
4. Compute $\text{IQR} = Q_3 - Q_1$ and the fences $Q_1 - 1.5\,\text{IQR}$ and $Q_3 + 1.5\,\text{IQR}$.
5. Flag values outside the fences as possible outliers. Report min, Q_1, median, Q_3, max.

Output: the five-number summary and a list of flagged values.

Worked example. Sorted: 95, 98, 102, 105, 108, 110, 115, 120, 125, 130, 940. *Try it:* Quartiles and box plot .

$$\begin{aligned} \text{median} &= 110 && \text{(6th of 11)} \\ Q_1 &= \text{median of } 95, 98, 102, 105, 108 = 102 \\ Q_3 &= \text{median of } 115, 120, 125, 130, 940 = 125 \\ \text{IQR} &= 125 - 102 = 23 \\ \text{fences} &= 102 - 1.5 \times 23 = 67.5 \quad \text{and} \quad 125 + 1.5 \times 23 = 159.5. \end{aligned}$$

The 940 is far beyond 159.5: flagged. Nothing else is.

```python
import statistics as st

times = [120, 95, 110, 130, 105, 98, 102, 115, 940, 108, 125]
print("mean  :", round(st.mean(times), 2))
print("median:", st.median(times))
print("stdev :", round(st.stdev(times), 2))      # sample sd, divides by n-1
trimmed = [t for t in times if t != 940]
print("without 940: mean", round(st.mean(trimmed), 2),
      "median", st.median(trimmed), "stdev", round(st.stdev(trimmed), 2))
print("quartiles (inclusive):", st.quantiles(times, n=4, method="inclusive"))
print("quartiles (exclusive):", st.quantiles(times, n=4))
```

```
mean  : 186.18
median: 110
stdev : 250.25
without 940: mean 110.8 median 109.0 stdev 11.57
quartiles (inclusive): [103.5, 110.0, 122.5]
quartiles (exclusive): [102.0, 110.0, 125.0]
```

Notice the last two lines: there are several conventions for quartiles, and software disagrees. For large data sets the differences vanish; for small ones, say which convention you used.

31.2.4. Box plots and histograms

A **box plot** draws the five-number summary: a box from Q_1 to Q_3 with a line at the median, "whiskers" out to the most extreme non-outlier values, and outliers as individual points.

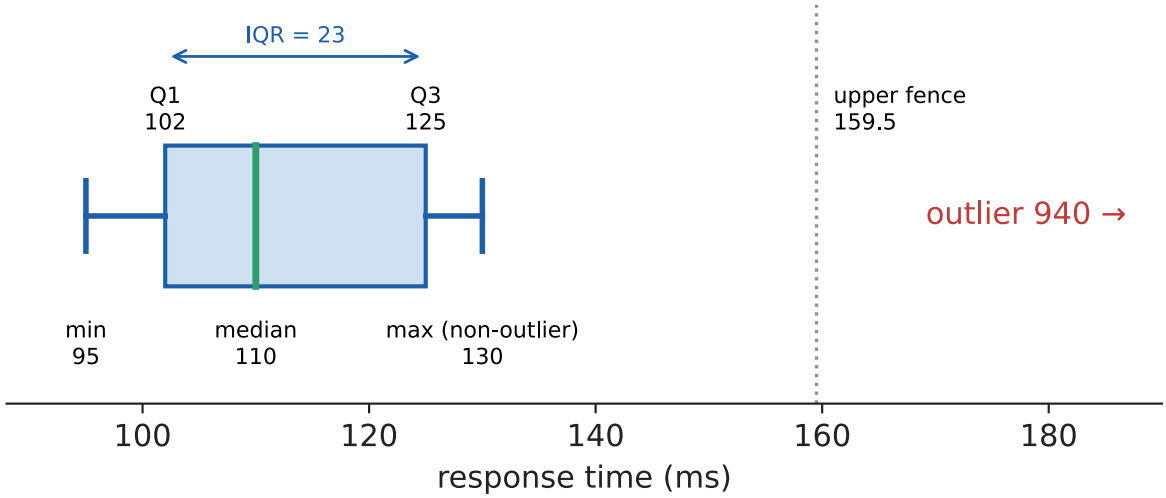


Figure 62. Anatomy of a box plot, for the 11 response times. The box is the middle half of the data; the whiskers reach the most extreme values inside the fences.

A **histogram** divides the range into bins and draws a bar for the count in each bin. It shows shape: one peak or two, symmetric or skewed, gaps, clusters. Box plots are compact and good for comparing many groups side by side; histograms show more detail about one group. Use both.

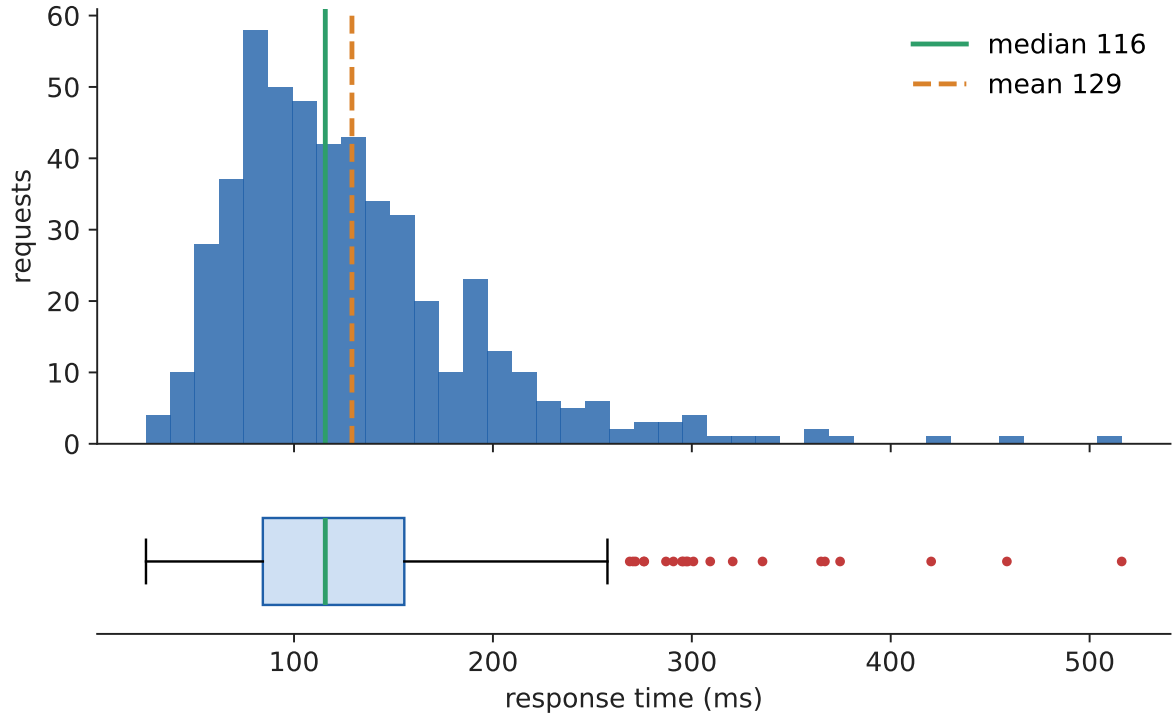


Figure 63. 500 response times: histogram (top) and box plot (bottom) of the same data. The long right tail pulls the mean above the median, and the box plot flags the tail values as outliers.

31.2.5. Skew

A distribution is **right-skewed** (positively skewed) if it has a long tail to the right: most values are moderate and a few are very large. Latencies, incomes, file sizes and city populations are right-skewed. **Left-skewed** is the mirror image (exam scores on an easy test). In a right-skewed distribution the mean is pulled into the tail, above the median; in a left-skewed one it's below.

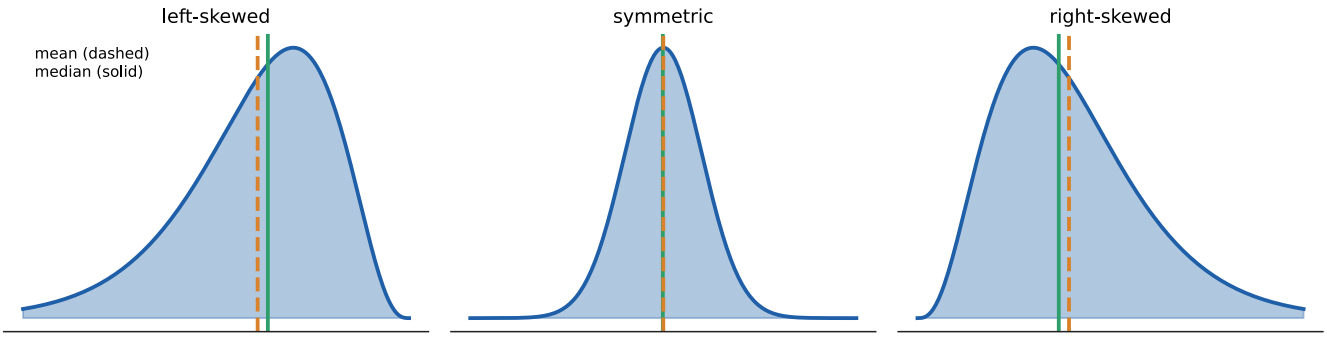


Figure 64. Skew pulls the mean (dashed) towards the long tail; the median (solid) stays with the bulk of the data.



Outliers are information

Never delete an outlier just because the IQR rule flags it. Ask what produced it. A 940 ms request might be a measurement glitch (remove it, and say so) or the one user in a hundred who hits a cold cache (that's the most important number on the dashboard). Report robust summaries (median, IQR, percentiles) *and* say how many outliers there were and what they were. For right-skewed data, the 1.5 × IQR rule flags many perfectly legitimate tail values, as in the box plot above.

31.2.6. Where you'd use this

- **Everyday: house prices and pay.** A street has nine houses worth £250,000 and one mansion worth £2.5 million. The mean value is $(9 \times 250\,000 + 2\,500\,000)/10 = 475\,000$ pounds, which describes no house on the street; the median, £250,000, describes nine of them. That's why official statistics quote *median* house prices and *median* earnings.
- **Health: percentiles.** A child "on the 75th percentile for height" is taller than 75% of children of the same age and sex. Growth charts are nothing but quantile curves, and a doctor watches for a child crossing several percentile lines, not for any single value.
- **Science and manufacturing: reporting spread.** A lab result is written mean $\pm$ standard deviation (or standard error, Section 31.4), and a factory tracks the standard deviation of a part's diameter as closely as its mean: a machine can be perfectly centred and still produce scrap if its spread grows.
- **Sport:** two basketball players both average 20 points a game; the one with the smaller standard deviation is the more predictable choice for a must-win game. The same comparison picks between two servers with equal median latency but different IQRs.

Computer-science connection: metrics and dashboards

Production monitoring is descriptive statistics under time pressure.

- **Percentiles, not means.** p50 (median), p95, p99 and p99.9 latencies describe the experience of typical and unlucky users; a mean hides both. A page that makes 100 backend calls will hit the p99 of at least one of them on most loads ($1 - 0.99^{100} \approx 63\%$), so tail percentiles of components become the median experience of users.
- **Mergeable summaries.** Percentiles can't be averaged across servers. Systems store histograms or sketches (t-digest, HDR histograms) that can be merged, and compute percentiles from the merged result.
- **Noise bands.** A metric that moved 3% today may just be noise. Plot a band of ± 3 standard deviations of historical day-to-day variation (a **control chart**) and investigate only moves outside it.
- **Goodhart's law.** "When a measure becomes a target, it ceases to be a good measure." Optimize for p50 latency and engineers may move slow work into the tail.

US curriculum

Mean, median, mode, range, box plots and histograms are taught in middle school (Common Core 6.SP and 7.SP). Standard deviation, interquartile range, outliers and comparing distributions by shape, centre and spread are high-school standards S-ID.1–3, usually in **Algebra I**, and form Unit 1 of **AP Statistics** ("Exploring One-Variable Data").

31.3. Samples, populations and bias

31.3.1. The problem

You want the average session length of all 40 million users, or the fraction of voters supporting a candidate, or the true conversion rate of a checkout page. You can't measure everyone (or, for a conversion rate, "everyone" includes future users who don't exist yet). You measure a **sample** and generalize. When is that legitimate, and how wrong might you be?



Population, sample, parameter, statistic

- **Population:** the whole group or process you want to know about.
- **Sample:** the part you actually observe.
- **Parameter:** a number describing the population (true mean μ, true proportion p). Usually unknown.
- **Statistic:** a number computed from the sample ($\bar{x}$, $\hat{p}$). Used to **estimate** a parameter. The "hat" in $\hat{p}$ (read "p-hat") marks an estimate.
- **Simple random sample:** every subset of size n is equally likely to be the sample. This is what makes the probability of Chapter 30 apply.

31.3.2. Bias: when samples lie

Random variation (noise) shrinks as samples grow. **Bias** doesn't: it's a systematic error that makes the sample unrepresentative no matter how large.

- **Selection bias.** Surveying your users about your product says nothing about the people who left. Measuring latency only on successful requests ignores those that timed out, the slowest ones.
- **Survivorship bias.** In the Second World War, analysts studied bullet holes in returning bombers and proposed armouring the most-hit areas. The statistician Abraham Wald pointed out that the planes hit in the *other* areas hadn't returned: armour where the survivors had no holes.
- **Non-response bias.** People who answer surveys differ from those who don't.
- **Convenience samples.** "The first 1,000 users who saw the new feature" are the most active, most engaged users, not typical ones.



The 1936 Literary Digest poll

In 1936 the *Literary Digest* mailed 10 million ballots, received 2.4 million back, and predicted a landslide for Alf Landon over Franklin Roosevelt. Roosevelt won 46 of 48 states. The mailing list came from telephone directories and car registrations, which in the Depression over-represented the wealthy, and those who replied were a self-selected group. George Gallup, with a sample of about 50,000 chosen to be representative, called the result correctly. A huge biased sample loses to a modest random one.



Big data does not cure bias

Logs from millions of users feel like the whole population, but they are a sample of *who used the product, in the way the product let them, while logging worked*. The formulas in this chapter quantify random error only. They say nothing about bias, and a confidence interval around a biased estimate is a precise answer to the wrong question. The best cure is design: randomize who gets what (as in an A/B test) and sample at random rather than by convenience.

31.4. Estimation and standard error

31.4.1. The problem

You sample 100 page loads and find a mean of 2.30 seconds. The true mean over all page loads is not exactly 2.30; a different sample of 100 would give a slightly different answer. How much would estimates bounce around from sample to sample?

31.4.2. Intuition: the sampling distribution

Imagine repeating the whole study many times, each time drawing a fresh sample of 100 and computing $\bar{x}$. The collection of all those $\bar{x}$ values has a distribution, the **sampling distribution** of the mean. Chapter 30 already told you its shape and size: by the central limit theorem it's approximately normal, centred on the true μ, with standard deviation $\sigma/\sqrt{n}$.



Standard error

The **standard error** (SE) of a statistic is the standard deviation of its sampling distribution: how much the statistic would vary from sample to sample.

- For a sample mean: $\mathrm{SE}(\bar{x}) = \sigma/\sqrt{n}$, estimated by $s/\sqrt{n}$.
- For a sample proportion $\hat{p}$ (a mean of 0/1 values): $\mathrm{SE}(\hat{p}) = \sqrt{p(1-p)/n}$, estimated by $\sqrt{\hat{p}(1-\hat{p})/n}$.

Standard **deviation** describes the spread of individual data points; standard **error** describes the uncertainty of an estimate. Collecting more data doesn't change the first and shrinks the second like $1/\sqrt{n}$.

If the 100 page loads have sample standard deviation $s = 0.80$ s, then $\mathrm{SE} = 0.80/\sqrt{100} = 0.08$ s. Individual loads vary by about 0.8 s, but the *average* of 100 is uncertain by only about 0.08 s.

An estimator is **unbiased** if its sampling distribution is centred on the true parameter: $E[\bar{x}] = \mu$, for instance. Good estimators are unbiased (or nearly) and have small standard errors.

31.5. Confidence intervals

31.5.1. The problem

"Average load time is 2.30 s" sounds more certain than it is. You want to report a range that honestly conveys the uncertainty, with a stated reliability.

31.5.2. Why it works

By the CLT, $\bar{x}$ is approximately normal around μ with standard deviation SE. The 68–95–99.7 rule (more precisely, 95% lies within 1.96 standard deviations) says:

$$P\left(\mu - 1.96\,\mathrm{SE} \le \bar{x} \le \mu + 1.96\,\mathrm{SE}\right) \approx 0.95.$$

"$\bar{x}$ is within 1.96 SE of μ" is the same statement as "μ is within 1.96 SE of $\bar{x}$". So

$$P\left(\bar{x} - 1.96\,\mathrm{SE} \le \mu \le \bar{x} + 1.96\,\mathrm{SE}\right) \approx 0.95.$$

Read this carefully. μ is a fixed (unknown) number; the randomness is in $\bar{x}$, so in the *interval*. Before you collect data, the procedure "take $\bar{x} \pm 1.96\,\mathrm{SE}$" has a 95% chance of producing an interval that captures μ.



Confidence interval

A **95% confidence interval** for a parameter is an interval computed from the data by a procedure that captures the true parameter in 95% of repeated samples. For a mean:

$$\bar{x} \pm z^* \cdot \frac{s}{\sqrt{n}},$$

where $z^* = 1.96$ for 95% (1.645 for 90%, 2.576 for 99%). The quantity $z^* \cdot \mathrm{SE}$ is the **margin of error**. The **confidence level** (95%) is the long-run success rate of the procedure.

For small samples, replace z^* by a slightly larger t^* from the **t-distribution** (Section 31.6.6) to account for estimating σ by s.

Try it: When z* is too narrow .

Example 45. Procedure — confidence interval for a mean or proportion

Input: a random sample; a confidence level (say 95%).

Steps:

1. Compute the estimate: $\bar{x}$ (mean) or $\hat{p}$ (proportion).
2. Compute the standard error: $s/\sqrt{n}$ or $\sqrt{\hat{p}(1-\hat{p})/n}$.
3. Find the critical value: $z^* = 1.96$ for 95% (or t^* with $n-1$ degrees of freedom for a small-sample mean).
4. Report estimate $\pm$ critical value $\times$ SE.
5. Check assumptions: random sample, independent observations, n large enough for the CLT (for proportions, at least about 10 successes and 10 failures).

Output: an interval and its confidence level.

Worked example. 100 page loads: $\bar{x} = 2.30$ s, $s = 0.80$ s.

$$\begin{aligned} \text{SE} &= 0.80/\sqrt{100} = 0.08 \\ \text{margin} &= 1.96 \times 0.08 = 0.157 \quad (t^* = 1.98 \text{ for 99 degrees of freedom gives } 0.158) \\ 95\%\ \text{CI} &= 2.30 \pm 0.16 = (2.14,\ 2.46) \text{ s.} \end{aligned}$$

Worked example (proportion). A poll of 1,000 randomly chosen users finds 540 prefer the new design.

$$\begin{aligned} \hat{p} = 0.54, \qquad \text{SE} &= \sqrt{0.54 \times 0.46/1000} = \sqrt{0.0002484} \approx 0.0158 \\ 95\%\ \text{CI} &= 0.54 \pm 1.96 \times 0.0158 = 0.54 \pm 0.031 = (0.509,\ 0.571). \end{aligned}$$

That "±3 percentage points" is the margin of error quoted under news polls of about 1,000 people. To halve it, poll 4,000.

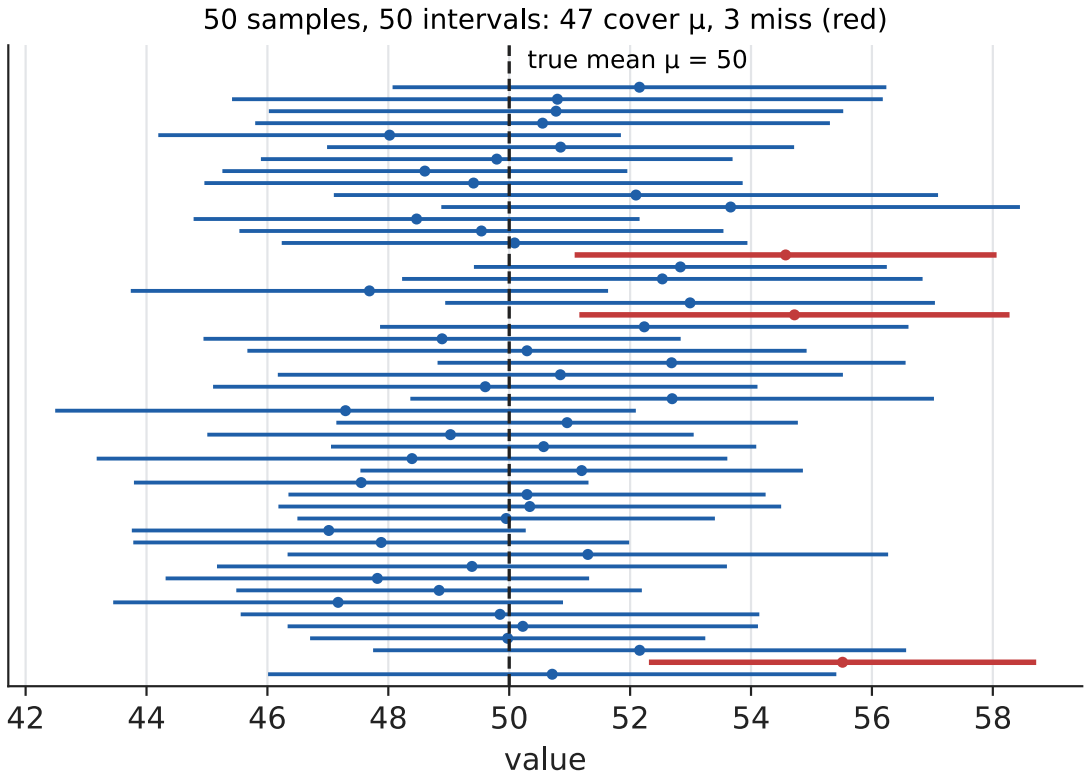


Figure 65. Fifty 95% confidence intervals, each from a different random sample of 25 values from a population with mean 50. Most capture the true mean; a few (red) miss. The procedure is right about 95% of the time; any one interval is either right or wrong.



What a confidence interval does not mean

The common misreading is: "there is a 95% probability that μ is between 2.14 and 2.46". In the frequentist framework that produced the interval, μ is a fixed number, not a random one: it either is in $(2.14, 2.46)$ or it isn't. The 95% describes the *method*: if you repeated the study many times, 95% of the intervals so produced would contain μ. The figure above is the picture to remember.

Two other misreadings: a 95% CI does *not* contain 95% of the data (that would be roughly $\bar{x} \pm 2s$, a much wider range), and it does *not* account for bias. If you want a statement like "95% probability that μ lies here", that's a Bayesian **credible interval** (Section 31.10), which requires a prior.

Try it: Fifty intervals, live .

31.5.3. Where you'd use this

- **Physics: measuring g.** Ten timings of a pendulum give values of g with mean $9.79\ \text{m/s}^2$ and $s = 0.05$. Then $\text{SE} = 0.05/\sqrt{10} \approx 0.016$, and with $t^* = 2.262$ (9 degrees of freedom) the 95% interval is $9.79 \pm 0.036 = (9.754, 9.826)$. The accepted value 9.81 is inside it, so the experiment agrees with theory. Every "result ± uncertainty" in a physics or chemistry write-up is a confidence interval or a standard error.

- **Medicine:** a trial reports that a drug lowers blood pressure by "8 mmHg (95% CI 5 to 11)". A doctor reads the lower end: even in the least favourable plausible case the benefit is 5 mmHg, which is clinically worthwhile.
- **Computing: benchmarks.** Twenty runs of a benchmark average 412 ms with $s = 30$ ms. The standard error is $30/\sqrt{20} \approx 6.7$ ms and the 95% interval is $412 \pm 2.093 \times 6.7 \approx 412 \pm 14$ ms. A claimed "3% speed-up" (about 12 ms) is inside that noise, so one run of each version proves nothing; report intervals, not single timings.



Catch the mean

Set a population (normal, skewed, or two-humped), a sample size and a confidence level, then press "sample" repeatedly. Each press draws a new sample and adds its interval to a growing stack, coloured by whether it covers the true mean. A counter shows the running coverage rate approaching the stated level, and a slider for n shows the intervals narrowing like $1/\sqrt{n}$. A "biased sampling" switch shifts the population that samples come from, and coverage collapses even as intervals narrow.

Try it: Catch the mean · Biased sampling breaks coverage

31.6. Hypothesis testing

31.6.1. The problem

A coin is tossed 100 times and lands heads 61 times. Is it biased? A fair coin *could* produce 61 heads; it would just be unusual. How unusual is unusual enough? The same question, with higher stakes: the new checkout page converted 11.0% of users versus 10.0% for the old one. Is the new page better, or did it get lucky?

31.6.2. Intuition: proof by contradiction, with probabilities

Hypothesis testing borrows the shape of proof by contradiction (Chapter 22). Assume the boring explanation, nothing is going on, and work out what data you'd expect. If the data you actually got would be very surprising under that assumption, you have evidence against it.

Hypotheses, test statistic, p-value, significance level

- **Null hypothesis H_0**: the "nothing is going on" claim, precise enough to compute probabilities (the coin is fair: $p = 0.5$; the two pages convert at the same rate).
- **Alternative hypothesis H_1** (or H_a): what you suspect instead ($p \neq 0.5$: **two-sided**; or $p > 0.5$: **one-sided**).
- **Test statistic:** a number computed from the data that measures how far it departs from H_0, often a z-score: $\frac{\text{estimate} - \text{null value}}{\text{SE}}$.
- **p-value:** the probability, *computed assuming H_0 is true*, of getting a test statistic at least as extreme as the one observed.
- **Significance level α**: a threshold chosen in advance (commonly 0.05). If $p \leq \alpha$, **reject H_0** ("statistically significant"); otherwise **fail to reject H_0**.

31.6.3. Worked example: the coin

$H_0: p = 0.5$. $H_1: p \neq 0.5$. Significance level $\alpha = 0.05$. *Try it:* Run the z-test .

Under H_0, the number of heads $X \sim \text{Bin}(100, 0.5)$, with mean 50 and standard deviation $\sqrt{100 \times 0.5 \times 0.5} = 5$.

Normal approximation. The test statistic is $z = \frac{61 - 50}{5} = 2.2$. "At least as extreme, in either direction" means $|Z| \geq 2.2$:

$$p = P(|Z| \geq 2.2) = 2\big(1 - \Phi(2.2)\big) = 2(1 - 0.9861) \approx 0.028.$$

Exact binomial. Add the binomial probabilities of 61 or more heads and 39 or fewer: $p \approx 0.035$. (The normal approximation is a little optimistic here; a "continuity correction" using 60.5 instead of 61 would bring it closer.)

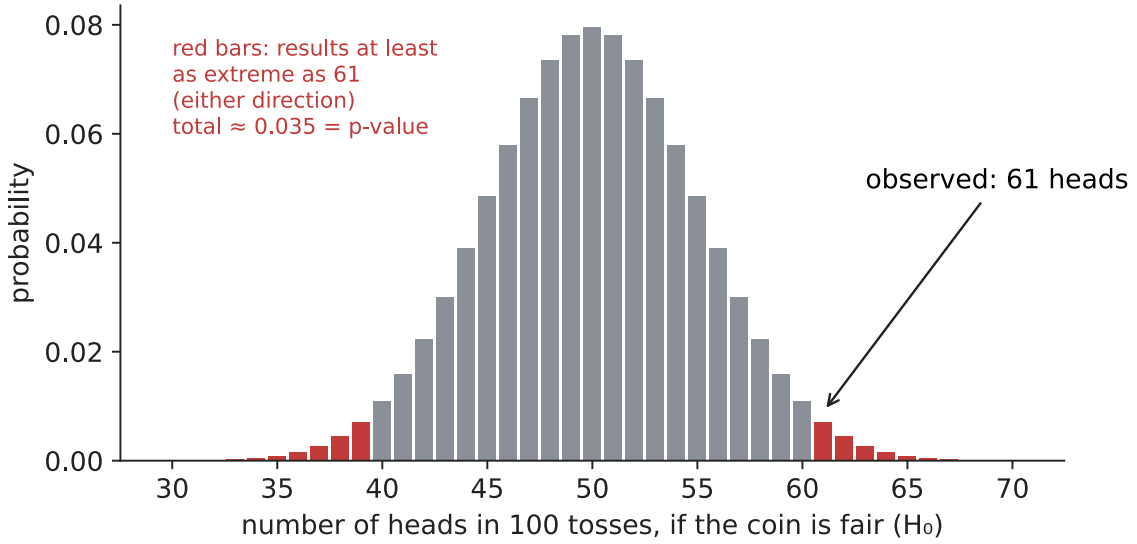


Figure 66. The null distribution: the number of heads in 100 tosses of a fair coin. The p-value is the total height of the red bars, the outcomes at least as far from 50 as the observed 61.

Either way $p < 0.05$, so we reject H_0 at the 5% level: 61 heads is evidence (moderate, not overwhelming) that the coin isn't fair.

PYTHON

```python
from math import comb, erf, sqrt

n, observed = 100, 61
pmf = [comb(n, k) / 2**n for k in range(n + 1)]
# two-sided: results at least as far from 50 as 61 is
p_exact = sum(p for k, p in enumerate(pmf) if abs(k - 50) >= abs(observed - 50))
z = (observed - 50) / 5
p_normal = 2 * (1 - 0.5 * (1 + erf(z / sqrt(2))))
print(round(p_exact, 4), round(z, 2), round(p_normal, 4))
```

```
0.0352 2.2 0.0278
```

Example 46. Procedure — a hypothesis test

Input: data; a question.

Steps:

1. State H_0 and H_1 in terms of a parameter. Choose α **before** looking at the data.
2. Choose a test statistic and work out its distribution assuming H_0 (the **null distribution**), by formula, normal approximation or simulation.
3. Compute the observed test statistic.
4. Compute the p-value: the probability under H_0 of a statistic at least as extreme as observed (both tails for a two-sided H_1).
5. If $p \le \alpha$, reject H_0; otherwise fail to reject. Report the p-value *and* the estimated effect with a confidence interval.

Output: a decision, a p-value and an effect estimate.

31.6.4. What a p-value is, and what it isn't

The p-value is $P(\text{data this extreme} \mid H_0)$. Recall the warning from Chapter 29 about confusing $P(A \mid B)$ with $P(B \mid A)$.

- It is **not** $P(H_0 \mid \text{data})$, the probability that the null hypothesis is true. That's the prosecutor's fallacy again. To get $P(H_0 \mid \text{data})$ you need Bayes' theorem and a prior: how plausible was H_0 to begin with?
- It is **not** the probability that the result is "due to chance".
- It does **not** measure the size or importance of an effect. With a million users, a 0.01% improvement can have $p < 0.001$ and be worthless; with 20 users, a huge improvement can have $p = 0.2$.
- $p > 0.05$ does **not** show that H_0 is true. "Fail to reject" means "not enough evidence", not "evidence of no effect". Absence of evidence is not evidence of absence, especially in a small study.
- The 0.05 threshold is a convention (from R. A. Fisher in the 1920s), not a law of nature. $p = 0.049$ and $p = 0.051$ are essentially the same evidence.

31.6.5. Type I and type II errors, and power

A test makes a binary decision about an unknown truth, so it can be wrong in two ways:

	H_0 true (no effect)	H_0 false (real effect)
Reject H_0	**Type I error** (false positive). Probability α.	Correct: a true discovery. Probability = **power**.
Fail to reject	Correct.	**Type II error** (false negative). Probability β.



Type I and type II errors, power

- **Type I error:** rejecting a true H_0 (crying wolf). Its probability is α, which you choose.
- **Type II error:** failing to reject a false H_0 (missing a real effect). Its probability is β.
- **Power** $= 1 - \beta$: the probability of detecting an effect of a given size if it really exists. Conventionally, studies are designed for 80% power.

Power increases with larger samples, larger true effects, less noisy data, and a larger α. For a fixed sample size, lowering α (fewer false alarms) raises β (more misses).

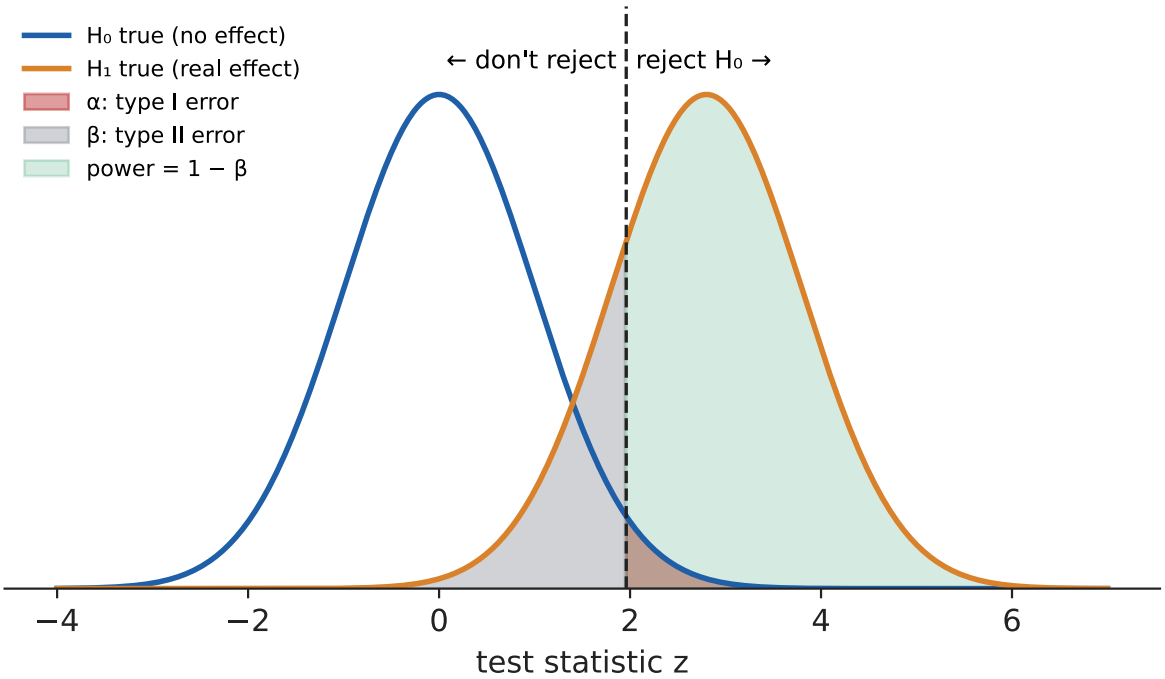


Figure 67. The two error types. If there's no effect, the test statistic follows the left curve, and the red tail beyond the cut-off is the type I error rate α. If there's a real effect, it follows the right curve: the grey area left of the cut-off is β (missed), the green area right of it is the power.

The analogy is a smoke alarm. Make it more sensitive and it catches more fires (more power) but also goes off for toast (more type I errors). The only way to improve both is a better detector, which in statistics means more data or less noise.



Underpowered studies

An experiment with too little data is worse than useless. It usually fails to find a real effect, and when it does reach significance, it's typically because noise exaggerated the effect: significant results from low-powered studies systematically overestimate effect sizes (the "winner's curse"). Work out the sample size you need *before* running the experiment (Section 31.11).

31.6.6. Comparing two means: the t-test, intuitively

The problem. A new build's latencies (8 runs, ms): 190, 204, 185, 199, 210, 188, 195, 202. The old build: 212, 198, 225, 240, 205, 219, 230, 201. Means 196.6 and 216.3, a difference of 19.6 ms. Real, or noise? *Try it:* Run Welch's t-test .

The idea. Every two-sample test asks the same thing: is the difference between groups large compared with how much it would vary by chance? The **t statistic** is

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\dfrac{s_1^2}{n_1} + \dfrac{s_2^2}{n_2}}} = \frac{\text{difference in means}}{\text{standard error of the difference}}.$$

The denominator combines the standard errors of the two means (variances of independent quantities add, Chapter 30). Here $t \approx 3.23$: the observed difference is more than three standard errors from zero.

Why "t" and not "z"? With small samples, s is itself a noisy estimate of σ, which adds uncertainty. William Gosset worked out the exact null distribution of t for normal data in 1908: the **t-distribution**, a bell curve with heavier tails than the normal, controlled by the **degrees of freedom** (roughly $n - 1$ per group). Some 95% two-sided critical values:

degrees of freedom	5	10	11	30	∞
t^*	2.571	2.228	2.201	2.042	1.960

Welch's formula for unequal variances gives about 11 degrees of freedom here, and $|t| = 3.23 > 2.201$, so the difference is significant at the 5% level. As n grows, t^* approaches the normal's 1.96.

An alternative that needs no tables: the permutation test. If the build made no difference (H_0), the labels "old" and "new" are arbitrary: any 8 of the 16 numbers could equally have been the "old" ones. So shuffle the labels thousands of times, recompute t each time, and see how often a shuffle gives a result as extreme as the real one. That fraction *is* the p-value, computed directly from its definition. *Try it:* Permutation test vs t-test .

```python
import random, statistics as st

old = [212, 198, 225, 240, 205, 219, 230, 201]   # ms, old build
new = [190, 204, 185, 199, 210, 188, 195, 202]   # ms, new build

def welch_t(a, b):
    se = (st.variance(a) / len(a) + st.variance(b) / len(b)) ** 0.5
    return (st.mean(a) - st.mean(b)) / se

t_obs = welch_t(old, new)
print("difference in means:", st.mean(old) - st.mean(new))
print("t =", round(t_obs, 3))

# Permutation test: if the build made no difference, labels are arbitrary.
rng = random.Random(13)
pooled = old + new
trials, extreme = 20_000, 0
for _ in range(trials):
    rng.shuffle(pooled)
    if abs(welch_t(pooled[:8], pooled[8:])) >= abs(t_obs):
        extreme += 1
print("permutation p-value ~", round(extreme / trials, 4))
```

Try it: Run this in your browser .

```
difference in means: 19.625
t = 3.227
permutation p-value ~ 0.0067
```

Choosing a method. Use a t-test for comparing means when data are roughly normal or samples are moderately large; it's standard and fast. Use a permutation test when samples are small and skewed, or for any statistic (medians, ratios) without a textbook formula; it costs only computer time. For paired data (the same 8 benchmarks run on both builds) use a **paired** test on the differences, which removes benchmark-to-benchmark variation and is far more powerful.



Student's t and Guinness

William Sealy Gosset was a chemist at the Guinness brewery in Dublin, testing barley and hops with small samples. Guinness forbade employees from publishing (a rival had leaked trade secrets), so he published his 1908 paper under the pseudonym "Student". The t-test is still called "Student's t-test", a brewery quality-control method that became one of the most used tools in science.

31.6.7. Counts in categories: the chi-squared test, briefly

For categorical data, compare observed counts with the counts expected under H_0. A die is rolled 120 times; if fair, each face is expected 20 times. Observed: 15, 22, 18, 25, 17, 23. The **chi-squared statistic** sums the squared discrepancies, each scaled by its expected count: *Try it:* Run the χ^2 test .

$$\chi^2 = \sum \frac{(O-E)^2}{E} = \frac{25+4+4+25+9+9}{20} = \frac{76}{20} = 3.8.$$

(χ is the Greek letter *chi*, pronounced "kai".) Under H_0 this statistic follows a **chi-squared distribution** with $6 - 1 = 5$ degrees of freedom, whose 95% critical value is 11.07. Since $3.8 < 11.07$, there's no evidence the die is unfair; deviations this size are routine. The same test checks whether two categorical variables are independent in a two-way table (does the browser affect whether checkout succeeds?), and whether a random number generator or hash function spreads values evenly across buckets.

31.6.8. Where you'd use this

- **Biology: Mendel's peas.** Mendel's genetics predicts round and wrinkled seeds in a 3 : 1 ratio. Of 7,324 seeds he counted 5,474 round and 1,850 wrinkled; the expected counts are 5,493 and 1,831. Then $\chi^2 = \frac{19^2}{5493} + \frac{19^2}{1831} \approx 0.26$, far below the 1-degree-of-freedom critical value 3.84: the data fit the theory. (Fisher later argued that Mendel's results overall fit *too* well, a statistical accusation of tidying the data.)
- **Medicine:** every new drug must beat a placebo in a randomized trial at a pre-agreed significance level (typically two-sided $\alpha = 0.05$) with a pre-computed sample size giving 80–90% power. The type I error is approving a useless drug; the type II error is abandoning a useful one.
- **Everyday: "hot streaks".** A basketball player who usually makes 50% of shots makes 8 in a row. Is that evidence of a hot hand? For one specific run of 8 shots, $P = 0.5^8 \approx 0.004$; but a player takes hundreds of shots a season, so a run of 8 somewhere is not surprising at all. Whether a "streak" is significant depends on how many chances it had to happen (Section 31.9).
- **Computing: catching performance regressions.** Continuous-integration systems run each benchmark several times on the old and new commit and apply a t-test or permutation test; only differences with a small p-value *and* a meaningful size raise an alert, which keeps noisy benchmarks from crying wolf on every commit.

US curriculum

Sampling, bias, margin of error and simulation-based inference appear in high-school standards S-IC.1–6 (often **Algebra II**). Formal confidence intervals, significance tests, type I and II errors, power, t-procedures and chi-squared tests are the heart of **AP Statistics** (Units 6–8), the most popular AP mathematics exam after Calculus AB, and a common first college statistics course.

31.7. Correlation, causation and Simpson's paradox

31.7.1. Correlation is not causation

Chapter 30 defined the correlation coefficient. For data pairs (x_i, y_i), the **sample correlation** is

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2}\,\sqrt{\sum (y_i - \bar{y})^2}},$$

a number from -1 to 1 measuring the strength of a *linear* association. Ice-cream sales correlate with drownings. Countries' chocolate consumption correlates with Nobel prizes per capita. Users who enable dark mode have higher retention. None of these shows that one thing causes the other. The usual explanations are:

- **Confounding:** a third variable drives both. Hot weather causes both ice-cream sales and swimming; engaged users both find settings like dark mode *and* stay longer.
- **Reverse causation:** Y causes X. Do happy users write reviews, or does writing reviews make users happy?
- **Selection:** the sample was filtered in a way that creates a relationship.
- **Chance:** with enough variables, some will correlate by luck (Section 31.9).

The gold standard for causation is a **randomized controlled experiment**: assign the treatment at random, so that confounders are balanced between groups on average. That's exactly what an A/B test does, and why it can answer causal questions that no amount of observational log analysis can.

31.7.2. Simpson's paradox

A 1986 study compared two treatments for kidney stones: *Try it:* Slide through Simpson's paradox .

	Treatment A	**Treatment B**
Small stones	**81/87 = 93%**	234/270 = 87%
Large stones	**192/263 = 73%**	55/80 = 69%
All patients	273/350 = 78%	**289/350 = 83%**

Treatment A is better for small stones *and* better for large stones, yet B looks better overall. How?

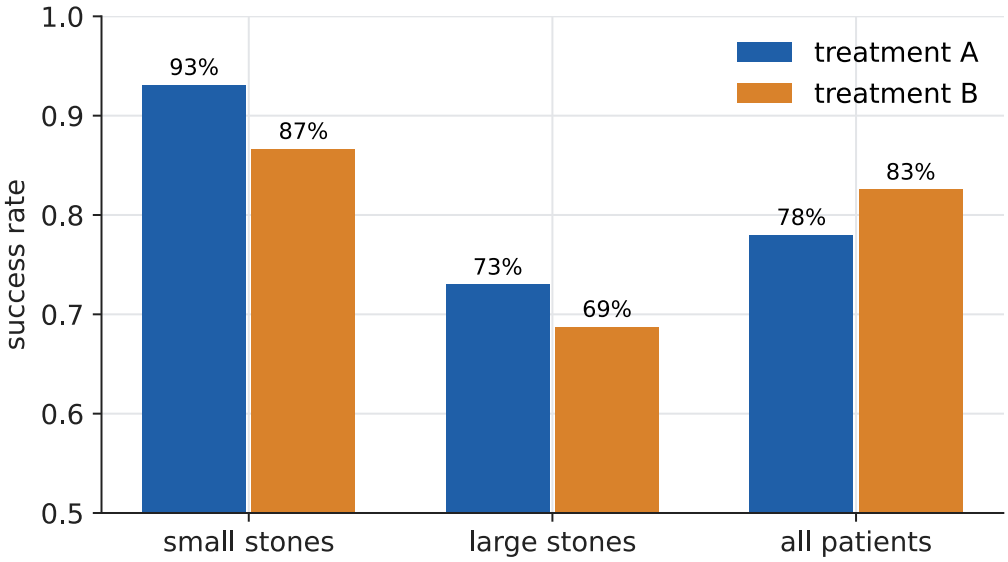


Figure 68. Simpson's paradox in the kidney-stone data: A wins within each group but loses in the combined data.

Look at who got which treatment. Treatment A was given mostly to large stones (263 of 350), the hard cases; B mostly to small stones (270 of 350), the easy ones. The overall rate is a weighted average of the group rates (the law of total probability from Chapter 29), and the weights differ: A's average is dominated by its 73% group, B's by its 87% group. Stone size is a confounder: it affects both which treatment doctors chose and the chance of success. *Try it:* Same case mix, no paradox .

The right comparison here is within groups: A is better. But Simpson's paradox doesn't come with a rule that "always disaggregate"; which comparison is right depends on the causal story. If the grouping variable were itself *caused* by the treatment, aggregating would be correct.



Spotting Simpson's paradox in product data

"Conversion fell after the redesign" might be entirely because the mix of traffic changed (more mobile users, who always convert less), while conversion rose within every platform. Whenever group sizes differ between the things you compare, break the comparison down by the obvious grouping variables (platform, country, new versus returning) before concluding anything. Randomization protects A/B tests from this, because it makes the mix the same in both arms on average.

31.8. Simple linear regression

31.8.1. The problem

A service's median response time at six load levels (load in hundreds of concurrent users):

load x	**1**	**2**	**3**	**4**	**5**	**6**
response y (ms)	51	52	59	63	64	71

Response time clearly grows with load. You want a line through the data to summarize the trend and predict, say, the response at load 8. The points don't lie exactly on any line, so which line is "best"?

31.8.2. Intuition: least squares

For any candidate line $\hat{y} = a + bx$, each point has a **residual**: the vertical miss $e_i = y_i - \hat{y}_i$ (observed minus predicted). A good line makes residuals small. The **least-squares** line makes the **sum of squared residuals** as small as possible. Squaring treats overshoots and undershoots alike, punishes big misses heavily, and (the decisive practical reason) gives a clean formula.



Least squares regression line and residuals

The **least-squares regression line** (or **line of best fit**) of y on x is $\hat{y} = a + bx$ with

$$b = \frac{S_{xy}}{S_{xx}} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}, \qquad a = \bar{y} - b\,\bar{x}.$$

b is the **slope** (change in predicted y per unit of x); a is the **intercept**. The line always passes through the point of means $(\bar{x}, \bar{y})$. $\hat{y}$ ("y-hat") denotes a predicted value. A **residual** is $y_i - \hat{y}_i$.

31.8.3. Why it works (light derivation)

Write the total squared error as $\mathrm{SSE}(a, b) = \sum(y_i - a - bx_i)^2$. Two observations give the formulas.

The intercept. For a fixed slope b, SSE is the sum of squared distances from the numbers $y_i - bx_i$ to a single number a. The number that minimizes a sum of squared distances is the mean (you can verify this by completing the square, Volume 1, Chapter 8): $a = \overline{y - bx} = \bar{y} - b\bar{x}$. So the best line passes through $(\bar{x}, \bar{y})$.

The slope. Substitute $a = \bar{y} - b\bar{x}$ and write $X_i = x_i - \bar{x}$, $Y_i = y_i - \bar{y}$ (centred data). Then $\mathrm{SSE}(b) = \sum(Y_i - bX_i)^2 = S_{yy} - 2bS_{xy} + b^2 S_{xx}$, a quadratic in b that opens upward. Its vertex (Volume 1, Chapter 8) is at $b = S_{xy}/S_{xx}$.

*(Optional, with calculus: setting $\partial\,\mathrm{SSE}/\partial a = 0$ and $\partial\,\mathrm{SSE}/\partial b = 0$ gives the same two equations, the **normal equations**; see Volume 2, Chapter 20 . With many input variables they become the matrix equation of Section 33.2.)*

31.8.4. Worked example

$\bar{x} = 21/6 = 3.5$ and $\bar{y} = 360/6 = 60$. *Try it:* Fit the load data ; Fit the regression line .

x	y	$x-\bar{x}$	$y-\bar{y}$	$(x-\bar{x})(y-\bar{y})$	$(x-\bar{x})^2$
1	51	-2.5	-9	22.5	6.25
2	52	-1.5	-8	12	2.25
3	59	-0.5	-1	0.5	0.25
4	63	0.5	3	1.5	0.25
5	64	1.5	4	6	2.25
6	71	2.5	11	27.5	6.25
			Sum	$S_{xy} = 70$	$S_{xx} = 17.5$

$$
\begin{aligned}
b &= \frac{70}{17.5} = 4 \\
a &= 60 - 4 \times 3.5 = 46 \\
\hat{y} &= 46 + 4x.
\end{aligned}
$$

Each extra 100 concurrent users adds about 4 ms. The prediction at load 8 is $46 + 32 = 78$ ms, an **extrapolation** beyond the data, to be trusted only if the trend stays linear (queueing theory says it won't near saturation; see the Watch out below).

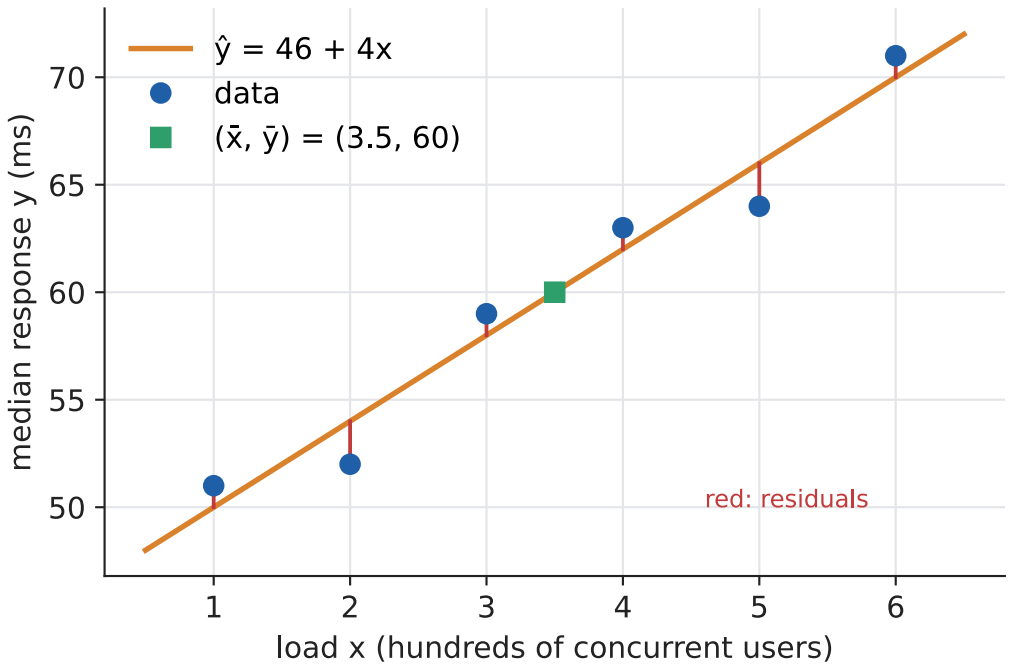


Figure 69. The least-squares line for the load data. Residuals are the vertical red segments; the line passes through the point of means.

31.8.5. How good is the fit? R^2

Compare the line's errors with the errors of the dumbest prediction, "always predict $\bar{y}$":

$$R^2 = 1 - \frac{\mathrm{SS}_{\mathrm{res}}}{\mathrm{SS}_{\mathrm{tot}}} = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}.$$

Here the residuals are $51-50=1$, $52-54=-2$, $59-58=1$, $63-62=1$, $64-66=-2$, $71-70=1$, so $\mathrm{SS}_{\mathrm{res}} = 1+4+1+1+4+1 = 12$. And $\mathrm{SS}_{\mathrm{tot}} = 81+64+1+9+16+121 = 292$. So

$$R^2 = 1 - \frac{12}{292} \approx 0.959.$$

The line explains about 96% of the variation in response time. For simple linear regression, $R^2 = r^2$, so $r = \sqrt{0.959} \approx 0.979$.

```python
def least_squares(xs, ys):
    n = len(xs)
    xbar, ybar = sum(xs) / n, sum(ys) / n
    sxy = sum((x - xbar) * (y - ybar) for x, y in zip(xs, ys))
    sxx = sum((x - xbar) ** 2 for x in xs)
    b = sxy / sxx
    a = ybar - b * xbar
    ss_res = sum((y - (a + b * x)) ** 2 for x, y in zip(xs, ys))
    ss_tot = sum((y - ybar) ** 2 for y in ys)
    return a, b, 1 - ss_res / ss_tot

a, b, r2 = least_squares([1, 2, 3, 4, 5, 6], [51, 52, 59, 63, 64, 71])
print(a, b, round(r2, 4))
```

```
46.0 4.0 0.9589
```



Regression pitfalls

- **Plot first.** Very different data sets (a curve, a line with one outlier, a vertical cluster) can share the same slope, intercept and R^2; Anscombe's quartet (1973) is the famous demonstration. Always look at a scatter plot and at the residuals; a pattern in the residuals means a line is the wrong model.
- **Extrapolation.** Latency is roughly linear at low load and explodes near capacity. A line fitted at 10–60% utilisation will badly underpredict at 95%.
- **Regression is not causation.** A slope describes association in the data you have; it doesn't tell you what would happen if you intervened, unless the data came from an experiment.
- **Regression to the mean.** Extreme measurements tend to be followed by less extreme ones, purely because part of what made them extreme was noise. The slowest day last month will probably look better next month, with or without your fix.

31.8.6. Where you'd use this

- **Physics: Hooke's law.** Hang loads of 1, 2, 3, 4, 5 N on a spring and measure extensions 2.1, 3.9, 6.2, 7.8, 10.1 mm. The least-squares line is $\hat{y} = 0.05 + 1.99x$, so the spring stretches about 1.99 mm per newton and its spring constant is $k \approx 1/1.99 \approx 0.50$ N/mm (about 500 N/m). The intercept near 0 is a check that the law holds; the slope is the physical constant you wanted. Calibration curves in chemistry (absorbance against concentration) work the same way.
- **Everyday and finance:** fit your monthly fuel cost against miles driven, and the slope is your cost per mile while the intercept is what you'd spend standing still. In finance, a stock's **beta** is the regression slope of its returns on the market's returns: beta 1.5 means it tends to move 1.5% when the market moves 1%.
- **Computing: measuring complexity.** A function takes 0.012, 0.049, 0.19 and 0.78 s on inputs of size 1,000, 2,000, 4,000 and 8,000. Regress $\ln(\text{time})$ on $\ln n$ (if time $\approx c\,n^k$, then $\ln \text{time} = \ln c + k \ln n$, a straight line); the slope comes out at 2.0, so the algorithm is quadratic and 100,000 items will take about $0.78 \times 12.5^2 \approx 120$ s. Fitting straight lines to transformed data is how you read growth rates from measurements (Volume 1, Chapter 12).

31.9. Multiple comparisons

31.9.1. The problem

You run an A/B test and check 20 metrics: conversion, revenue, clicks, time on page, each on mobile and desktop, in five countries. One metric shows $p = 0.03$. Discovery?

31.9.2. Why it's a problem

At $\alpha = 0.05$, each test of a true null has a 5% chance of a false positive. With 20 independent tests of true nulls, *Try it:* Simulate twenty tests .

$$P(\text{at least one false positive}) = 1 - 0.95^{20} \approx 1 - 0.358 = 0.642.$$

You're more likely than not to find "something", even if the change did nothing. Testing many hypotheses and reporting the significant ones is called **p-hacking** or **data dredging**, and is a central cause of the **replication crisis** in psychology, medicine and other fields: many published "findings" were simply the lucky tail of many tries. The same thing happens if you test one hypothesis but try many analyses (different outlier rules, different subgroups, different stopping times) and report the one that worked.

31.9.3. Remedies

- **Pre-register.** Decide on one **primary metric** and the analysis before collecting data; treat the rest as exploratory, to be confirmed in a new experiment.
- **Bonferroni correction.** With m tests, use threshold α/m for each (0.0025 for 20 tests). This bounds the chance of *any* false positive by α. Simple and conservative (it loses power). *Try it:* Bonferroni in action .

- **False discovery rate (FDR) control,** such as the Benjamini–Hochberg procedure, bounds the expected *fraction* of false positives among the discoveries instead. Better when you're screening many candidates (genes, features, metrics). *Try it:* Benjamini–Hochberg on 100 tests .



> *The dead salmon*
> In 2009 neuroscientists put a dead Atlantic salmon in an fMRI scanner, showed it photographs of people in social situations, and "asked" it to judge their emotions. Analysing tens of thousands of brain voxels without a multiple-comparisons correction, they found a cluster of "significant" activity in the fish's brain. The poster won an Ig Nobel Prize, and the point landed: with enough tests, noise always produces significance somewhere.

31.10. Bayesian and frequentist perspectives

Everything above is **frequentist**: probabilities describe long-run frequencies over repeated samples, parameters are fixed unknowns, and procedures (intervals, tests) are judged by how often they'd be right. That's why a confidence interval's 95% describes the procedure, and why a p-value conditions on H_0 rather than computing its probability.

The **Bayesian** approach (from Chapter 29's degree-of-belief interpretation) treats an unknown parameter as uncertain and describes that uncertainty with a probability distribution. Start with a **prior** distribution, observe data, and use Bayes' theorem to get the **posterior**:

$$\text{posterior}(\theta \mid \text{data}) \propto \text{likelihood}(\text{data} \mid \theta) \times \text{prior}(\theta),$$

where θ (*theta*) is the parameter and $\propto$ means "proportional to" (the denominator just rescales so the posterior has total probability 1). Then you can say directly: "there is a 95% probability that θ is in this range" (a **credible interval**), or "there's a 99.8% probability that version B converts better than A".

	Frequentist	Bayesian
Parameter	fixed, unknown	uncertain, has a distribution
Probability means	long-run frequency	degree of belief
Answers	"how surprising is this data if H_0 holds?" (p-value)	"how probable is the hypothesis, given the data?" (posterior)
Needs	a sampling model	a sampling model **and** a prior
Strengths	no prior to argue about; well-understood error rates	direct, intuitive answers; combines evidence naturally; good with small data
Weaknesses	answers are easy to misread	results depend on the prior; computation can be heavy

In practice, with plenty of data and a mild prior, the two approaches give nearly the same numbers; they differ in interpretation and in small-data situations. Many practitioners use both: frequentist error control for decisions that must be defensible, Bayesian models where prior knowledge is real and direct probability statements are wanted. A Bayesian treatment of the A/B test follows in the next section.

31.11. A/B testing, end to end

31.11.1. The problem

Your checkout page converts 10% of visitors. A redesign might help. You'll randomly show half of new visitors the old page (A, **control**) and half the new one (B, **treatment**). How many visitors do you need, how do you analyse the result, and what mistakes must you avoid?

31.11.2. Step 1: decide what you're testing, before you start

- **Primary metric:** conversion rate (fraction of visitors who purchase).
- **Hypotheses:** $H_0 : p_B = p_A$; $H_1 : p_B \neq p_A$ (two-sided, so you'd also detect harm).
- $\alpha = 0.05$, power 80%.
- **Minimum detectable effect (MDE):** the smallest improvement worth detecting. Say 1 percentage point (10% to 11%). Smaller effects exist, but you've decided they're not worth the engineering cost, or not worth the time to detect.

31.11.3. Step 2: sample size

The test statistic is the difference in sample proportions divided by its standard error. For the test to have power 80%, the true difference must be large enough that the test statistic's distribution under H_1 sits mostly beyond the cut-off 1.96 (the figure in the type I/II section). The standard approximation is

Try it: Plan the sample size .

$$n \text{ per group} \approx \frac{(z_{\alpha/2} + z_{\beta})^2 \left[p_A(1-p_A) + p_B(1-p_B)\right]}{(p_B - p_A)^2},$$

where $z_{\alpha/2} = 1.96$ (two-sided 5%) and $z_{\beta} = 0.84$ ($\Phi(0.84) \approx 0.80$, for 80% power). With $p_A = 0.10$, $p_B = 0.11$:

$$\begin{aligned} n &\approx \frac{(1.96 + 0.84)^2 \,(0.10 \times 0.90 + 0.11 \times 0.89)}{0.01^2} \\ &= \frac{7.84 \times (0.0900 + 0.0979)}{0.0001} \\ &= \frac{7.84 \times 0.1879}{0.0001} \approx 14\,700 \text{ visitors per group.} \end{aligned}$$

About 30,000 visitors in total. Note the denominator: the required sample grows with the **inverse square** of the effect. Detecting a 0.5-point lift needs about four times as many users (roughly 58,000 per group); a 0.1-point lift, a hundred times as many. *Try it:* Half-point lift sample size .

31.11.4. Step 3: run it properly

- **Randomize** at the right unit (usually the user, not the page view, so the same person sees a consistent version), using a hash of the user ID.
- **Run for the planned sample size,** and for whole weeks to average over weekly cycles.
- **Don't peek and stop early** (see the simulation below).
- **Check the split:** if you intended 50/50 and got 50.8/49.2 on 30,000 users, a chi-squared test will flag a **sample ratio mismatch**, which usually means a bug in assignment or logging, and invalidates the test.

31.11.5. Step 4: analyse

Results: A converted 1,500 of 15,000 (10.0%); B converted 1,650 of 15,000 (11.0%). *Try it:* Run the two-proportion test .

$$\begin{aligned} \hat{p}_A &= 0.100, \quad \hat{p}_B = 0.110, \quad \text{difference} = 0.010 \\ \hat{p}_{\text{pooled}} &= \frac{1500 + 1650}{30\,000} = 0.105 \qquad \text{(best estimate of the common rate if } H_0 \text{ holds)} \\ \text{SE}_0 &= \sqrt{0.105 \times 0.895 \times \left(\tfrac{1}{15\,000} + \tfrac{1}{15\,000}\right)} = \sqrt{1.253 \times 10^{-5}} \approx 0.00354 \\ z &= \frac{0.010}{0.00354} \approx 2.83 \\ p &= 2\big(1 - \Phi(2.83)\big) \approx 0.0047. \end{aligned}$$

Significant at the 5% level. For the confidence interval, use the unpooled standard error (we no longer assume the rates are equal):

$$\text{SE} = \sqrt{\frac{0.10 \times 0.90}{15\,000} + \frac{0.11 \times 0.89}{15\,000}} \approx 0.00354$$

$$95\%\ \text{CI} = 0.010 \pm 1.96 \times 0.00354 = 0.010 \pm 0.0069 = (0.0031,\ 0.0169).$$

Report: "B increased conversion by 1.0 percentage point (95% CI 0.3 to 1.7 points; $p = 0.005$)." The interval matters as much as the p-value: the true lift could plausibly be as small as 0.3 points. Is that still worth shipping? That's a business decision the statistics informs but doesn't make.

PYTHON

```python
from math import erf, sqrt

def phi(z):
    return 0.5 * (1 + erf(z / sqrt(2)))

def ab_test(conv_a, n_a, conv_b, n_b):
    pa, pb = conv_a / n_a, conv_b / n_b
    pooled = (conv_a + conv_b) / (n_a + n_b)
    se0 = sqrt(pooled * (1 - pooled) * (1 / n_a + 1 / n_b))   # under H0
    z = (pb - pa) / se0
    p_value = 2 * (1 - phi(abs(z)))
    se = sqrt(pa * (1 - pa) / n_a + pb * (1 - pb) / n_b)      # for the CI
    ci = (pb - pa - 1.96 * se, pb - pa + 1.96 * se)
    return z, p_value, ci

z, p, ci = ab_test(1500, 15_000, 1650, 15_000)
print(f"z = {z:.2f}, p = {p:.4f}, 95% CI for lift = "
      f"({ci[0]*100:.2f}, {ci[1]*100:.2f}) percentage points")

def sample_size(p1, p2, z_alpha=1.96, z_beta=0.8416):
    return (z_alpha + z_beta) ** 2 * (p1 * (1 - p1) + p2 * (1 - p2)) / (p2 - p1) ** 2

print(round(sample_size(0.10, 0.11)))
```

```
z = 2.83, p = 0.0047, 95% CI for lift = (0.31, 1.69) percentage points
14748
```

31.11.6. Step 5: check the whole procedure by simulation

Does the procedure really have a 5% false-positive rate and 80% power? And what does peeking do? Simulate thousands of experiments where you know the truth.

```python
import numpy as np

rng = np.random.default_rng(14)
n, experiments = 15_000, 20_000

def z_stats(conv_a, conv_b, n):
    pooled = (conv_a + conv_b) / (2 * n)
    se = np.sqrt(pooled * (1 - pooled) * 2 / n)
    return (conv_b - conv_a) / n / se

# A/A tests: both arms truly convert at 10%
a = rng.binomial(n, 0.10, experiments)
b = rng.binomial(n, 0.10, experiments)
print("false positive rate:", np.mean(np.abs(z_stats(a, b, n)) > 1.96))

# A/B tests: B truly converts at 11%
b = rng.binomial(n, 0.11, experiments)
print("power              :", np.mean(np.abs(z_stats(a, b, n)) > 1.96))

# Peeking: check after every 1,000 users per arm, stop at first "significant" result
checks = np.arange(1000, n + 1, 1000)
stopped = 0
for _ in range(2000):
    xa = rng.random(n) < 0.10
    xb = rng.random(n) < 0.10            # no real difference
    ca, cb = np.cumsum(xa)[checks - 1], np.cumsum(xb)[checks - 1]
    pooled = (ca + cb) / (2 * checks)
    z = (cb - ca) / checks / np.sqrt(pooled * (1 - pooled) * 2 / checks)
    stopped += np.any(np.abs(z) > 1.96)
print("false positive rate with peeking:", stopped / 2000)
```

Try it: Watch peeking inflate false positives .

```
false positive rate: 0.04755
power              : 0.80715
false positive rate with peeking: 0.2265
```

The planned test behaves as designed: about 5% false positives when nothing changed, about 80% detection of a real 1-point lift. But checking 15 times and stopping at the first significant result raises the false-positive rate from 5% to about 23%. Each look is another chance for noise to cross the line: a multiple-comparisons problem in time. If you need to monitor continuously, use methods designed for it (**sequential tests**, or "always-valid" p-values), which set stricter thresholds for early looks.

31.11.7. A Bayesian reading of the same data

With a uniform prior on each conversion rate, the posterior for a rate after s successes and f failures is a **Beta** distribution, Beta($s+1, f+1$). Sampling from both posteriors answers the question stakeholders actually ask:

```python
import numpy as np

rng = np.random.default_rng(15)
# Uniform prior on each conversion rate -> Beta(conversions+1, failures+1) posterior
post_a = rng.beta(1500 + 1, 13_500 + 1, 200_000)
post_b = rng.beta(1650 + 1, 13_350 + 1, 200_000)
print("P(B better than A) =", round(np.mean(post_b > post_a), 4))
print("P(lift > 0.5 points) =", round(np.mean(post_b - post_a > 0.005), 3))
```

```
P(B better than A) = 0.9978
P(lift > 0.5 points) = 0.921
```

"There's a 99.8% probability B is better, and a 92% probability the lift exceeds half a point", given the model and the (flat) prior. With this much data the prior barely matters, and the conclusions agree with the frequentist analysis.



A/B testing failure modes

- **Peeking and early stopping** inflate false positives (simulated above).
- **Too many metrics or segments** without correction (Section 31.9).
- **Underpowered tests** that "find nothing" or find exaggerated effects.
- **Novelty effects:** users click the new thing because it's new; the effect fades.
- **Interference:** in social or marketplace products, treated users affect control users (a seller discount in B changes what A buyers see), violating independence.
- **Sample ratio mismatch** signals a broken experiment; don't analyse it, fix it.

Computer-science connection: experimentation platforms

Large companies run thousands of concurrent experiments on shared infrastructure. The statistics above is built into those platforms: deterministic hash-based assignment, automatic sample-ratio checks, pre-registered primary metrics, variance-reduction techniques that use each user's pre-experiment behaviour as a covariate (CUPED), sequential testing for safe early stopping, and guardrail metrics (latency, crash rate, unsubscribes) that block a launch even if the primary metric improves. "Engineering" an experiment platform is mostly applied statistics plus careful distributed logging.



Plan an experiment

Sliders for baseline rate, minimum detectable effect, α and power compute the required sample size live, with a plot of the two sampling distributions (null and alternative) moving apart as n grows. A "run" button simulates the experiment day by day, showing the z-score's random walk and the 1.96 boundary; a "peek" toggle lets you stop at the first crossing, while a counter over many replays shows the false-positive rate climbing.

Try it: Plan an experiment

Try it: Watch the bootstrap resample . === The bootstrap

31.11.8. The problem

You have 200 response times and want a confidence interval for the *median*, or the p99, or the ratio of two medians. There's no simple standard-error formula like $s/\sqrt{n}$ for these. The sampling distribution you need comes from imagining many repeated samples from the population, but you only have one sample.

31.11.9. Intuition: resample your sample

Your sample is your best picture of the population. So treat it *as* the population: draw a new sample of the same size from it, **with replacement** (some values appear twice, some not at all), and compute the statistic. Repeat thousands of times. The spread of these **bootstrap** statistics approximates the spread of the statistic across real repeated samples.

The name comes from the impossible feat of "pulling yourself up by your bootstraps". It does feel like getting something for nothing, but it works remarkably well when the sample is reasonably large and representative.

Example 47. Procedure — percentile bootstrap confidence interval

Input: a sample $x_1, \ldots, x_n$; a statistic (median, p99, ...); a number of resamples B (e.g. 5,000); a level (e.g. 95%).

Steps:

1. Compute the statistic on the original sample: the estimate.
2. Repeat B times: draw n values from the sample uniformly **with replacement**; compute the statistic on this resample; store it.
3. Sort the B stored values. The 2.5th and 97.5th percentiles are the ends of the 95% interval.
4. Look at the histogram of the stored values: a lumpy or very skewed shape is a warning.

Output: the estimate and a bootstrap confidence interval.

```python
import numpy as np

def bootstrap_ci(data, stat=np.median, resamples=5000, level=0.95, seed=0):
    rng = np.random.default_rng(seed)
    data = np.asarray(data)
    stats = np.array([stat(rng.choice(data, size=data.size, replace=True))
                      for _ in range(resamples)])
    tail = (1 - level) / 2 * 100
    return stat(data), np.percentile(stats, [tail, 100 - tail])

rng = np.random.default_rng(2024)
latencies = rng.lognormal(mean=np.log(120), sigma=0.6, size=200)  # stand-in data
est, (lo, hi) = bootstrap_ci(latencies, seed=1)
print(f"median = {est:.1f} ms, 95% bootstrap CI = ({lo:.1f}, {hi:.1f})")
est, (lo, hi) = bootstrap_ci(latencies, stat=lambda x: np.percentile(x, 99), seed=1)
print(f"p99    = {est:.1f} ms, 95% bootstrap CI = ({lo:.1f}, {hi:.1f})")
```

```
median = 121.8 ms, 95% bootstrap CI = (109.8, 132.4)
p99    = 462.8 ms, 95% bootstrap CI = (371.7, 511.1)
```

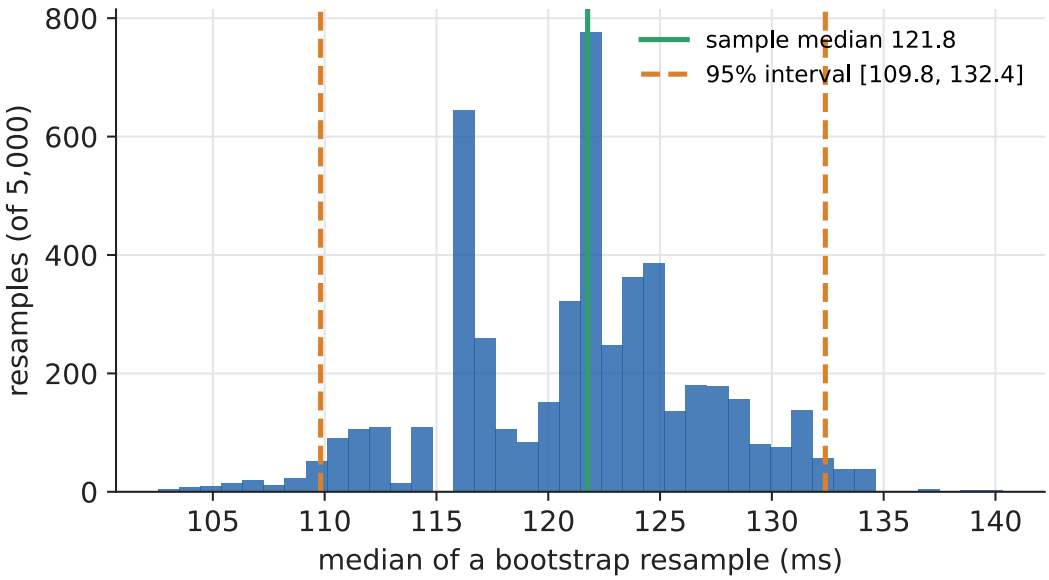


Figure 70. Bootstrap distribution of the median from 5,000 resamples of 200 response times. Its lumpiness reflects that a median can only take values from (or between) the data points.

The p99 interval is wide and lopsided: with 200 data points, only the top two values inform the 99th percentile, so it is very uncertain. That's useful knowledge before anyone alerts on a p99 from a small window of requests. *Try it:* Bootstrap the p99 .

Choosing a method. Use formula-based intervals for means and proportions with decent sample sizes: they're exact enough and instant. Use the bootstrap for medians, percentiles, ratios, correlations, model-evaluation metrics, or anything complicated. The bootstrap struggles for extreme statistics (maximum, very high percentiles with small samples), for very small samples, and for dependent data (time series need block resampling).



Bradley Efron's computer-age statistics

Bradley Efron introduced the bootstrap in 1979. It replaced pages of approximate algebra with brute-force computation, and was initially regarded with suspicion for exactly that reason. It is now one of the most used methods in applied statistics, and Efron received the inaugural International Prize in Statistics in 2016 for it. It's a clear case where cheap computing changed which mathematics was practical.

31.12. Evaluating classifiers and search: precision, recall, F1

31.12.1. The problem

A fraud model flags transactions; a search engine returns documents; a spam filter sends email to junk. How good are they? "Accuracy" (fraction of correct decisions) is often useless: if 1% of transactions are fraudulent, a model that never flags anything is 99% accurate. You need measures that focus on the class you care about.

31.12.2. The confusion matrix

Every decision lands in one of four cells:

	Actually positive	**Actually negative**
Predicted positive	True positive (TP)	False positive (FP)
Predicted negative	False negative (FN)	True negative (TN)



Precision, recall, F1

- **Precision** $= \dfrac{TP}{TP + FP}$: of the items flagged (or retrieved), what fraction are truly positive (relevant)? "When it says yes, how often is it right?"
- **Recall** $= \dfrac{TP}{TP + FN}$: of the truly positive (relevant) items, what fraction were flagged (retrieved)? "Of everything it should find, how much does it find?" Also called **sensitivity** or **true-positive rate**.
- **F1 score** $= \dfrac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$, the **harmonic mean** of the two. It's high only if both are high.

In probability language (Chapter 29): recall is $P(\text{flagged} \mid \text{positive})$ and precision is $P(\text{positive} \mid \text{flagged})$, the two directions of conditioning. Precision is the medical test's "probability of disease given a positive test", and so it depends on the base rate.

Worked example: search. A query has 30 relevant documents in the collection. The engine returns 20 results, of which 12 are relevant.

$$TP = 12, \quad FP = 20 - 12 = 8, \quad FN = 30 - 12 = 18$$
$$\text{precision} = \frac{12}{20} = 0.60, \qquad \text{recall} = \frac{12}{30} = 0.40$$
$$F_1 = \frac{2 \times 0.60 \times 0.40}{0.60 + 0.40} = \frac{0.48}{1.00} = 0.48.$$

Worked example: a rare-class classifier. A fraud model on 10,000 transactions, 100 of them fraudulent, catches 90 frauds and wrongly flags 150 legitimate transactions. Recall $= 90/100 = 0.90$; precision $= 90/240 = 0.375$; F1 ≈ 0.53. Accuracy $= (90 + 9\,750)/10\,000 = 98.4\%$, which sounds great and tells you almost nothing; it's lower than the 99% of the do-nothing model. *Try it:* Precision, recall and F1 .

PYTHON

```python
def precision_recall_f1(tp, fp, fn):
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f1 = 2 * precision * recall / (precision + recall)
    return precision, recall, f1

print(precision_recall_f1(tp=12, fp=8, fn=18))     # search example
print(precision_recall_f1(tp=90, fp=150, fn=10))   # rare-class classifier
```

Try it: Run this in your browser .

```
(0.6, 0.4, 0.48)
(0.375, 0.9, 0.5294117647058825)
```

31.12.3. The tradeoff

Most classifiers output a score, and you choose a threshold. Lower the threshold and you flag more: recall rises, precision usually falls. Raise it and the reverse happens. Which balance is right depends on the costs. Missing fraud is expensive, so favour recall; showing users junk results in the top 10 is expensive, so favour precision. *Try it:* Move the threshold .

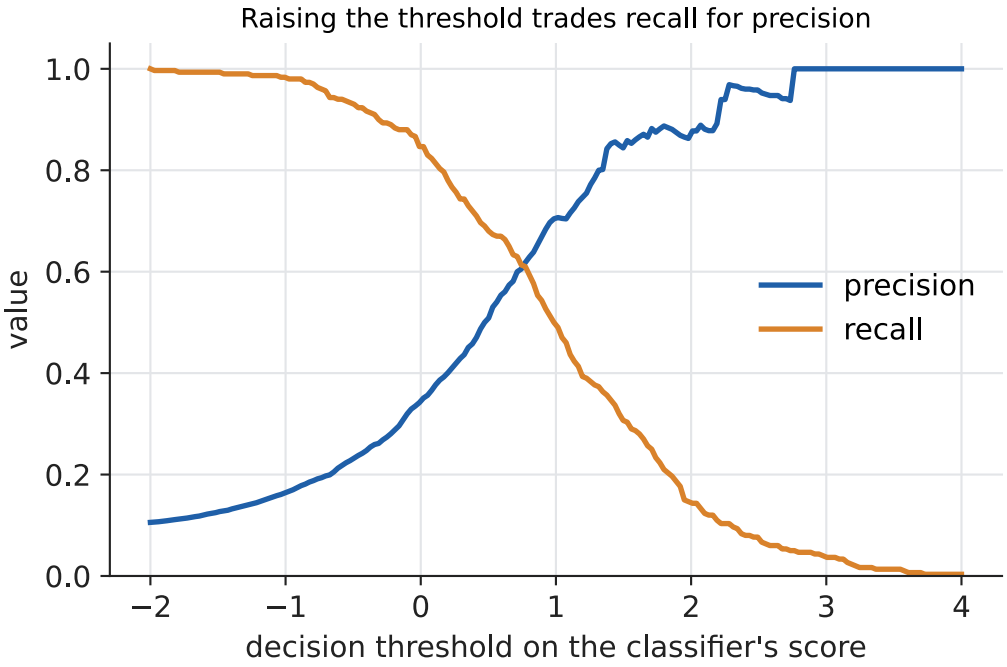


Figure 71. As the threshold rises, fewer items are flagged: precision rises and recall falls.

Computer-science connection: information retrieval metrics

Search engines care most about the top of the ranking, so IR uses rank-aware variants. **Precision@k** is the precision of the first k results. **Mean average precision** (MAP) averages precision at each rank where a relevant result appears. **NDCG** (normalized discounted cumulative gain) gives graded relevance scores and discounts results lower down by $1/\log_2(\text{rank}+1)$. Comparing two rankers on a set of queries is itself a statistical question: use paired tests or the bootstrap over queries, because performance varies far more between queries than between systems. Offline metrics are then validated online, by an A/B test on real users.



Precision, recall and the base rate

Recall and the false-positive rate are properties of the classifier; precision also depends on how common positives are. A model evaluated on a balanced test set (50% fraud) will show far higher precision than it achieves in production (1% fraud). Always evaluate on data with the real-world class balance, or recompute precision with Bayes' theorem for the production base rate.

31.12.4. Where you'd use this

- **Health: home tests.** A rapid antigen test with recall (sensitivity) 80% and a false-positive rate of 0.5% is used when 1% of people are infected. Per 10,000 people: 100 infected, 80 test positive; 9,900 not infected, about 50 test positive. Precision is $80/130 \approx 0.62$: more than a third of positive results are false, which is why positives were often confirmed by a lab test. Meanwhile 20 of the 100 infected people get a reassuring negative: low recall is why a negative test was never a free pass.

- **Everyday: airport security and smoke alarms** are tuned for recall (missing a weapon or a fire is catastrophic) and accept poor precision (most bag searches and alarms are false). A music app's "you might like" list is tuned the other way: a few bad suggestions annoy, a missed song costs nothing.

> ### US curriculum
>
> Scatter plots, correlation, lines of best fit and residuals appear in grade 8 (8.SP) and high-school standards S-ID.6–9 (**Algebra I**), including the explicit standard "distinguish between correlation and causation". Least-squares regression, R^2 and inference for slopes form Units 2 and 9 of **AP Statistics**. Bootstrap methods, Bayesian inference and classifier metrics are usually first met in college statistics, data science or machine-learning courses.

31.13. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Statistics
- Sampling and simulation
- Language sandbox
- The maths of machine learning
- Test accuracy and risk

31.14. Exercises

1. **(Recognition)** For each, say whether the mean or the median is the better summary of "typical", and why: (a) household incomes in a city; (b) heights of adult women; (c) time users spend on a page; (d) the number of bugs per 1,000 lines in a well-tested codebase.
2. **(Recognition)** Classify each as type I error, type II error or correct decision: (a) an A/B test declares a useless change "significant"; (b) a smoke alarm stays silent during a fire; (c) a test fails to find an effect that doesn't exist.
3. **(Application)** Find the five-number summary, IQR and any outliers (1.5 × IQR rule) for: 3, 7, 8, 5, 12, 14, 21, 13, 18.
4. **(Application)** A random sample of 400 users has mean session length 6.2 minutes and standard deviation 4.0 minutes. Give a 95% confidence interval for the mean session length of all users, and say in one sentence what "95%" refers to.
5. **(Application)** In 200 rolls of a die, a six appears 45 times. Test at the 5% level whether the die is biased towards six (two-sided, normal approximation).
6. **(Application)** Fit a least-squares line to $(1,2), (2,4), (3,5), (4,4), (5,5)$ and compute R^2.

7. **(Problem solving)** A study finds that people who take vitamin supplements live longer. Give two non-causal explanations, and describe a study design that could establish causation.
8. **(Problem solving)** A team runs 10 independent A/B tests on changes that, unknown to them, do nothing. At $\alpha = 0.05$, what's the probability at least one is "significant"? What per-test threshold does Bonferroni give for a 5% family-wide error rate?
9. **(Problem solving, CS)** You want to detect a lift in click-through rate from 5.0% to 5.5% with 80% power at $\alpha = 0.05$ (two-sided). Roughly how many users per group do you need?
10. **(Problem solving, CS)** A spam filter is tested on 1,000 emails, 200 of them spam. It flags 190 emails, 170 of which are spam. Compute precision, recall, F1 and accuracy. In production 2% of email is spam; if recall and false-positive rate stay the same, what precision should you expect?

▼ **Answers**

1. (a) Median: right-skewed, a few very high incomes inflate the mean. (b) Mean (or either): roughly symmetric, no extreme outliers. (c) Median: right-skewed, many short visits plus a few tabs left open for hours. (d) Probably median: counts with occasional bad files are right-skewed; if planning total effort, the mean (totals) matters.
2. (a) Type I. (b) Type II (H_0 "no fire" is false but not rejected). (c) Correct decision.
3. Sorted: 3, 5, 7, 8, 12, 13, 14, 18, 21. Median 12. Lower half 3, 5, 7, 8: $Q_1 = 6$. Upper half 13, 14, 18, 21: $Q_3 = 16$. IQR = 10. Fences $6 - 15 = -9$ and $16 + 15 = 31$: no outliers. Five-number summary: 3, 6, 12, 16, 21.
4. $\text{SE} = 4.0/\sqrt{400} = 0.2$; CI $6.2 \pm 1.96 \times 0.2 = 6.2 \pm 0.39 = (5.81, 6.59)$ minutes. "95%" is the long-run fraction of such intervals, over repeated samples, that would contain the true mean.
5. $H_0 : p = 1/6$. Expected $200/6 = 33.3$; SD $\sqrt{200 \times \frac{1}{6} \times \frac{5}{6}} = \sqrt{27.8} = 5.27$.
   $z = (45 - 33.3)/5.27 = 2.21$. Two-sided $p \approx 2(1 - 0.9864) = 0.027 < 0.05$: reject H_0; evidence of bias.
6. $\bar{x} = 3, \bar{y} = 4$. Deviations x: $-2, -1, 0, 1, 2$; y: $-2, 0, 1, 0, 1$. $S_{xy} = 4 + 0 + 0 + 0 + 2 = 6$, $S_{xx} = 10$. $b = 0.6$, $a = 4 - 1.8 = 2.2$. $\hat{y} = 2.2 + 0.6x$. Predictions 2.8, 3.4, 4.0, 4.6, 5.2; residuals $-0.8, 0.6, 1.0, -0.6, -0.2$; $\text{SS}_{\text{res}} = 0.64 + 0.36 + 1 + 0.36 + 0.04 = 2.4$; $\text{SS}_{\text{tot}} = 4 + 0 + 1 + 0 + 1 = 6$; $R^2 = 1 - 2.4/6 = 0.6$.
7. Confounding: people who take supplements tend to be wealthier, exercise more, smoke less. Reverse causation or selection: very ill people may stop taking supplements. Design: a randomized controlled trial, randomly assigning supplements or placebo, ideally double-blind.
8. $1 - 0.95^{10} \approx 1 - 0.599 = 0.401$. Bonferroni: $0.05/10 = 0.005$ per test.
9. $n \approx \frac{7.84\,(0.05 \times 0.95 + 0.055 \times 0.945)}{0.005^2} = \frac{7.84 \times (0.0475 + 0.0520)}{0.000025} = \frac{7.84 \times 0.0995}{0.000025} \approx 31\,200$ per group.
10. TP = 170, FP = 20, FN = 30, TN = 780. Precision $170/190 = 0.895$; recall $170/200 = 0.85$; F1 $= 2(0.895)(0.85)/(1.745) \approx 0.872$; accuracy $(170 + 780)/1000 = 0.95$. False-positive rate $20/800 = 0.025$. Production, per 10,000 emails: 200 spam → 170 flagged; 9,800 ham → 245 flagged. Precision $170/415 \approx 0.41$. Most flagged mail would be legitimate.

31.15. Mastery check

1. Why is the median latency usually more informative than the mean, and what does the mean tell you that the median doesn't?
2. Explain the difference between a standard deviation and a standard error, and how each changes as the sample grows.
3. A 95% confidence interval for a conversion rate is (4.1%, 5.3%). A colleague says "so there's a 95% chance the true rate is in there". What would you say?
4. State precisely what a p-value of 0.03 means. Give two things it does not mean.
5. Describe how you'd decide the sample size for an A/B test, and why stopping when the result first becomes significant is a mistake.
6. Explain Simpson's paradox using a two-group example and the law of total probability.
7. Write the steps of a bootstrap confidence interval for a median.
8. A classifier has 95% recall and a 5% false-positive rate. In a population with 1% positives, estimate its precision.

Decision rule: All correct without hesitation → move on to Volume 2, Chapter 32 . Stuck on 1–2 → reread Section 31.5 and Section 31.6, the sections with the most commonly misread ideas. Stuck on more → work the exercises first, and simulate each procedure in Python until its behaviour feels obvious.

▼ **Mastery check answers**

1. Latency is right-skewed, so a few slow requests pull the mean above what most users see; the median is the typical experience. The mean, multiplied by the request count, gives total time consumed, which matters for capacity and cost, and it reflects the tail that the median ignores.
2. The SD describes the spread of individual observations; it settles to the population value σ as n grows. The SE describes the uncertainty of an estimate such as $\bar{x}$; it shrinks like $1/\sqrt{n}$.
3. In the frequentist sense, the true rate is fixed: it either is or isn't in that interval. The 95% describes the method: 95% of intervals made this way capture the true rate. (A Bayesian credible interval would support the colleague's statement, given a prior.)
4. If the null hypothesis were true, a result at least as extreme as the one observed would occur with probability 0.03. It is not the probability that H_0 is true, not the probability the result is a fluke, and says nothing about the effect's size or importance.
5. Choose α, power, the baseline rate and the minimum effect worth detecting; compute $n \approx (z_{\alpha/2} + z_{\beta})^2[p_A(1-p_A) + p_B(1-p_B)]/(p_B - p_A)^2$; run to that size. Stopping at the first significant look gives noise many chances to cross the threshold, inflating false positives far above α (to about 23% with 15 looks in our simulation).

6. The kidney-stone table: A beats B within small stones and within large stones, but A was given mostly to large (hard) cases. The overall rate is a weighted average of group rates, $P(S) = \sum_g P(g)P(S \mid g)$, with different weights for A and B, so B's overall rate is inflated by its many easy cases.
7. Compute the sample median. Repeat 5,000 times: resample n values with replacement, compute the median. Take the 2.5th and 97.5th percentiles of those medians as the interval; check the histogram looks reasonable.
8. Per 10,000: 100 positives → 95 flagged; 9,900 negatives → 495 flagged. Precision $95/590 \approx 0.16$. (This is the medical-test calculation of Chapter 29.)

Part X: Horizons

Chapter 33. Horizons: Optimisation, Machine Learning, and Information

This final chapter is a capstone. Nothing in it needs a new branch of mathematics; instead it takes the branches you have rebuilt — calculus, linear algebra, probability, numerical methods — and shows them working together on the problems that dominate modern computing: fitting models to data, training neural networks, ranking documents, and measuring information. It closes with a guided tour of reading "real" mathematics, the skill this whole book has been aiming at, and a map of where to go next.

Why this chapter matters. The spam filter on your email, the photo search on your phone, the ranking of a web search, the "people also bought" row in a shop and the compression of every video you stream are all built from the handful of ideas in this chapter. So, increasingly, are tools in medicine, physics and finance. After this chapter you can open the box and read what is inside.

Where this chapter sits

Prerequisites: derivatives and the chain rule (Volume 2, Chapter 17 , Volume 2, Chapter 18), the gradient and gradient descent (Volume 2, Chapter 20), vectors and the dot product (Volume 2, Chapter 26), matrices (Volume 2, Chapter 27), exponentials and logarithms (Volume 1, Chapter 12), probability and expectation (Chapter 29, Chapter 30), and floating point and iterative methods (Volume 2, Chapter 32).

Unlocks: machine learning, information retrieval, information theory, convex optimisation, differential equations, and — most importantly — the ability to pick up a paper in any of those fields and read the mathematics in it.

How to read it: each section is a compact case study. You will recognise every tool; the new skill is seeing several of them combine in one place.

33.1. Optimisation: the art of going downhill

33.1.1. The problem

Almost every question of the form "what is the best ...?" is an optimisation problem. What is the best straight line through these data points? What are the best weights for this neural network? What is the cheapest route, the strongest beam for a given mass of steel, the portfolio with the least risk for a target return?

In each case there is a number you want to make as small (or large) as possible, and a set of knobs you are allowed to turn. Machine learning in particular is, at its core, the business of turning millions of knobs to make one number — the *loss* — as small as possible.

33.1.2. Intuition

Picture the loss as a landscape. Each point on the ground is a setting of the knobs; the height is the loss. You are standing somewhere on this landscape in thick fog. You cannot see the lowest valley, but you can feel the slope under your feet. The sensible strategy is to step downhill, feel the slope again, and repeat. That is gradient descent, which you met in Volume 2, Chapter 20 : the gradient points uphill, so you walk in the opposite direction.

The fog is the crucial part of the picture. Any optimisation method only ever has *local* information — the value and slope at the current point. Whether local information is enough to find the global answer depends entirely on the shape of the landscape.



Objective function, parameters, minimiser

The **objective function** (in ML, the **loss** or **cost** function) is the quantity being minimised, often written $L(\theta)$ or $f(\mathbf{x})$. The **parameters** θ (Greek *theta*) are the knobs; when there are many, θ is a vector. A **minimiser** θ^* (read "theta star") is a parameter value where the objective is smallest. The **minimum** is the smallest *value* $L(\theta^*)$; the minimiser is *where* it occurs.



argmin and argmax

$\min_\theta L(\theta)$ is the smallest value of L. $\operatorname{argmin}_\theta L(\theta)$ is the *argument* — the value of θ — that achieves it. For $L(\theta) = (\theta - 3)^2 + 1$, $\min_\theta L = 1$ but $\operatorname{argmin}_\theta L = 3$. Programmers know this distinction: `min(xs)` versus `xs.index(min(xs))`.

33.1.3. Why it works: convexity

Some landscapes are friendly. A bowl has exactly one low point, and from anywhere on its surface "downhill" eventually takes you there. Other landscapes are egg boxes, with many hollows; walking downhill takes you to the *nearest* hollow, which may not be the deepest.

The mathematical word for "bowl-shaped" is *convex*.



Convex function

A function f is **convex** if the straight line (chord) between any two points on its graph lies on or above the graph:

$$f\big(ta + (1-t)b\big) \le t f(a) + (1-t) f(b) \qquad \text{for all } a, b \text{ and } 0 \le t \le 1.$$

For a twice-differentiable function of one variable this is the same as $f''(x) \ge 0$ everywhere: the slope never decreases. For several variables the condition is that the Hessian matrix (Volume 2, Chapter 20) is positive semidefinite. Examples: x^2, e^x, $|x|$. Non-examples: $\sin x$, x^3, $-x^2$.



Local and global minimum

A **local minimum** is a point lower than everything *nearby*. A **global minimum** is lower than (or equal to) *everything*. A **saddle point** is a critical point (gradient zero) that is a minimum along some directions and a maximum along others — like the middle of a horse saddle.

The key theorem, stated without proof: **for a convex function, every local minimum is a global minimum.** The proof is short and worth seeing in outline. Suppose a is a local minimum but some point b is lower. Walk along the chord from a towards b. Convexity says the function along that chord is at most the chord's height, and the chord slopes down from $f(a)$ to $f(b) < f(a)$. So arbitrarily close to a there are points lower than a — contradicting "local minimum". Hence no such b exists.

That is why optimisers love convex problems: the fog does not matter. Any method that reliably goes downhill will find *the* answer.

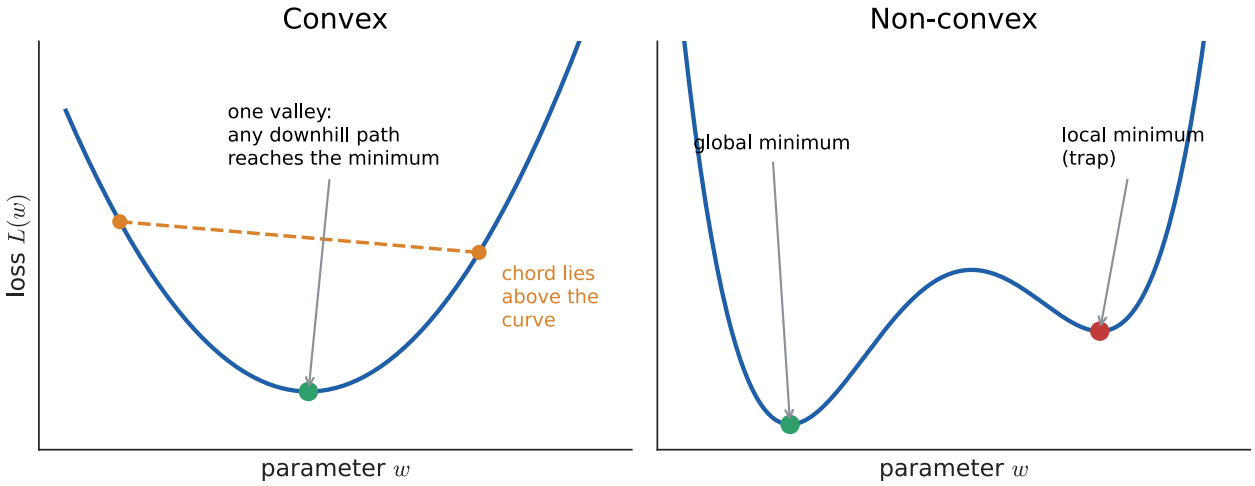


Figure 72. Convex and non-convex losses. On the left, every downhill path leads to the single minimum. On the right, gradient descent started on the right-hand slope settles into the local minimum and never sees the deeper one.

Linear regression (Section 33.2) and logistic regression (Section 33.3) have convex losses. Neural networks (Section 33.4) do not. One of the surprises of the last decade is that gradient descent still works astonishingly well on neural networks; why it works so well is an active research question.

33.1.4. The method: gradient descent and its variants

Recall the basic update from Volume 2, Chapter 20 :

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t).$$

Here t counts steps, η (Greek *eta*) is the **learning rate**, and $\nabla L(\theta_t)$ is the gradient of the loss at the current parameters. Two practical problems motivate two variants.

Problem 1: the gradient is expensive. In machine learning the loss is an average over a data set,

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell_i(\theta),$$

where ℓ_i is the loss on example i. If n is a billion, computing the exact gradient costs a billion evaluations *per step*. **Stochastic gradient descent (SGD)** instead estimates the gradient from a small random **mini-batch** of, say, 32 examples. The estimate is noisy but unbiased: on average it points the right way. You get a thousand cheap, slightly wobbly steps for the price of one exact step, and that is almost always a better deal.

Problem 2: narrow valleys. If the landscape is a long, narrow ravine, the gradient points mostly across the ravine (where it is steep) rather than along it (where the minimum is). Plain gradient descent zig-zags from wall to wall and crawls along the floor. **Momentum** fixes this by letting the step accumulate a running "velocity": components of the gradient that keep pointing the same way (along the valley) build up, while components that flip sign every step (across the valley) cancel out.



Learning rate, mini-batch, epoch

The **learning rate** η is the step-size multiplier in gradient descent. A **mini-batch** is a small random subset of the training data used to estimate the gradient for one step. An **epoch** is one full pass through the training data — with 1,000 examples and batches of 32, one epoch is 32 steps (the last batch is short). **Momentum** β (Greek *beta*, typically 0.9) controls how much of the previous velocity carries into the next step.

Example 48. Procedure — Mini-batch stochastic gradient descent with momentum

Input: a loss $L(\theta) = \frac{1}{n}\sum_i \ell_i(\theta)$, a way to compute $\nabla \ell_i$, initial parameters θ_0, learning rate η, momentum $\beta \in [0, 1)$, batch size B, number of epochs.

Steps:

1. Set the velocity $\mathbf{v} = \mathbf{0}$.
2. For each epoch: shuffle the example indices.
3. For each consecutive batch S of B indices:
   a. Estimate the gradient: $\mathbf{g} = \frac{1}{|S|}\sum_{i \in S} \nabla \ell_i(\theta)$.
   b. Update the velocity: $\mathbf{v} \leftarrow \beta \mathbf{v} + \mathbf{g}$.
   c. Step: $\theta \leftarrow \theta - \eta \mathbf{v}$.
4. Stop after the chosen number of epochs, or when the loss on held-out data stops improving.

Output: parameters θ with (approximately) minimal loss.

With $\beta = 0$ this is plain SGD; with $\beta = 0$ and $B = n$ it is plain (full-batch) gradient descent.

33.1.5. Worked example: momentum in a narrow valley

Take the ravine $f(x, y) = \frac{1}{2}(x^2 + 25y^2)$. Its minimum is at the origin, but it is 25 times steeper in y than in x. The gradient is *Try it:* Race momentum against descent .

$$\nabla f(x, y) = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} = \begin{bmatrix} x \\ 25y \end{bmatrix}.$$

Start at $(10, 1)$ with $\eta = 0.07$.

Plain gradient descent, two steps.

$$\begin{aligned} \nabla f(10, 1) &= (10,\ 25) \\ (x_1, y_1) &= (10, 1) - 0.07\,(10, 25) = (10 - 0.7,\ 1 - 1.75) = (9.3,\ -0.75) \\ \nabla f(9.3, -0.75) &= (9.3,\ -18.75) \\ (x_2, y_2) &= (9.3, -0.75) - 0.07\,(9.3, -18.75) = (9.3 - 0.651,\ -0.75 + 1.3125) = (8.649,\ 0.5625) \end{aligned}$$

The y-coordinate bounces $1 \to -0.75 \to 0.5625$: it overshoots each time. The x-coordinate creeps down by only 7% per step.

Momentum with $\beta = 0.7$, two steps. The first step is identical because the velocity starts at zero: $\mathbf{v}_1 = (10, 25)$, position $(9.3, -0.75)$. Then

$$\begin{aligned} \mathbf{v}_2 &= 0.7\,(10, 25) + (9.3, -18.75) && \text{(old velocity plus new gradient)} \\ &= (7 + 9.3,\ 17.5 - 18.75) = (16.3,\ -1.25) \\ (x_2, y_2) &= (9.3, -0.75) - 0.07\,(16.3, -1.25) = (9.3 - 1.141,\ -0.75 + 0.0875) = (8.159,\ -0.6625) \end{aligned}$$

Look at what happened to the velocity. In x, the old velocity and new gradient agreed, so the velocity grew to 16.3 and the step along the valley was 75% bigger. In y, they disagreed (25 versus -18.75), so they almost cancelled and the bounce was damped. That is the whole idea of momentum, visible in two steps of arithmetic.

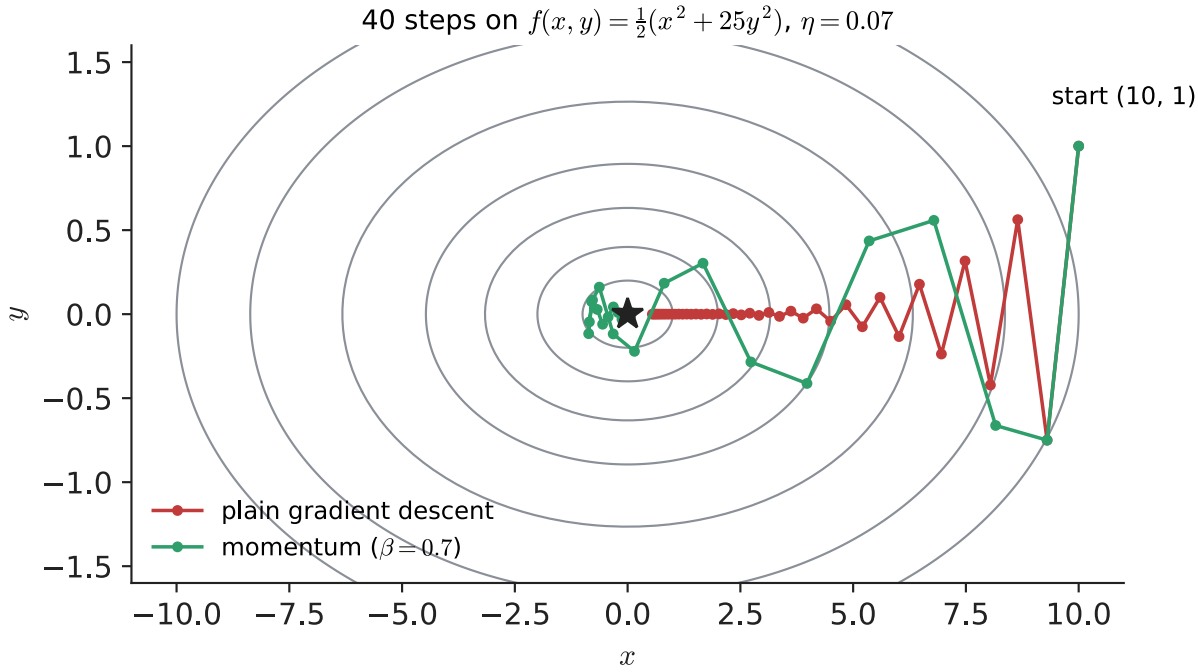


Figure 73. Forty steps of each method from the same start. Plain gradient descent (red) bounces across the narrow direction and then inches along the floor; momentum (green) builds speed along the valley and arrives. (The y-axis is stretched about 3.5 times so the zig-zag is visible; drawn to scale, each contour ellipse is five times longer than it is wide.)

Now implement and run it for 100 steps:

```python
import numpy as np

# A long, narrow valley: f(x, y) = 0.5*(x^2 + 25*y^2), minimum at (0, 0)
def grad(p):
    return np.array([p[0], 25.0 * p[1]])

def run(eta, beta, steps=100):
    p = np.array([10.0, 1.0])
    v = np.zeros(2)
    for _ in range(steps):
        v = beta * v + grad(p)          # velocity: running sum of gradients
        p = p - eta * v
    return np.linalg.norm(p)

print(f"plain GD  (beta=0.0): distance from minimum = {run(0.07, 0.0):.2e}")
print(f"momentum  (beta=0.7): distance from minimum = {run(0.07, 0.7):.2e}")
```

```
plain GD  (beta=0.0): distance from minimum = 7.05e-03
momentum  (beta=0.7): distance from minimum = 2.04e-07
```

You can predict the first number by hand: in x each plain step multiplies the coordinate by $1 - 0.07 = 0.93$, and $10 \times 0.93^{100} \approx 7.05 \times 10^{-3}$. Momentum is more than four orders of magnitude closer after the same number of gradient evaluations.

33.1.6. Alternative approaches and choosing a method

- **Closed form.** Some problems (least squares, Section 33.2) can be solved exactly by setting the gradient to zero and solving the resulting equations. Use this when n and the number of parameters are moderate.
- **Newton's method** in several variables uses the Hessian as well as the gradient: $\theta \leftarrow \theta - H^{-1}\nabla L$. It converges in very few steps near the minimum (quadratically, as in Volume 2, Chapter 32), but forming and inverting a $p \times p$ Hessian costs $O(p^3)$ — impossible for a network with $p = 10^9$ parameters.
- **Adaptive methods** such as **Adam** combine momentum with a per-parameter learning rate, scaled down for parameters whose gradients are consistently large. Adam is the default for training neural networks.
- **Derivative-free methods** (grid search, random search, simulated annealing, evolutionary algorithms) need only function values. Use them when there is no gradient — for example tuning discrete hyperparameters.

A reasonable rule of thumb: **closed form if it exists and fits in memory; Newton or quasi-Newton (L-BFGS) for smooth problems with up to a few thousand parameters; SGD with momentum or Adam for huge data and huge models; derivative-free when nothing else applies.**

33.1.7. Constraints

Real problems have rules: probabilities must be non-negative and sum to 1, a robot arm's joint angles are limited, a budget cannot be exceeded. A **constrained** problem is written

$$\min_{\theta} L(\theta) \quad \text{subject to} \quad g(\theta) \le 0.$$

Three standard ways to handle the "subject to":

1. **Lagrange multipliers** (Volume 2, Chapter 20). At a constrained optimum on the boundary, the gradient of the objective is parallel to the gradient of the constraint: $\nabla L = -\lambda \nabla g$. This turns a constrained problem into a system of equations, and generalises (as the Karush–Kuhn–Tucker conditions) to inequalities.
2. **Projected gradient descent.** Take an ordinary gradient step, then *project* back onto the allowed set — move to the nearest allowed point. For the constraint $\theta \ge 0$ the projection is simply `max(theta, 0)`, applied component by component.
3. **Penalties.** Replace the hard rule with a cost for breaking it: minimise $L(\theta) + \lambda \|\theta\|^2$ instead of requiring $\|\theta\| \le c$. This is exactly **regularisation** in machine learning (ridge regression, weight decay): it discourages large weights and reduces overfitting. The two forms are equivalent in the sense that every c corresponds to some λ.



Learning rate is still the number-one failure

Every failure mode of gradient descent from Volume 2, Chapter 20 survives here. Too large a learning rate and the loss explodes, often to `nan` (in the ravine above, any $\eta > 2/25 = 0.08$ makes the y-coordinate diverge for plain gradient descent). Too small and training crawls. With SGD, a constant learning rate means the parameters never settle; they jitter around the minimum at a distance set by the gradient noise, which is why practitioners *decay* the learning rate over time. And on non-convex losses, the answer depends on where you start: run it twice with different random initialisations and you may get two different models.

33.1.8. Where you'd use this

- **Packaging (everyday calculus).** What shape of 330 ml drink can uses the least metal? With radius r and height $h = 330/(\pi r^2)$, the surface area $A(r) = 2\pi r^2 + 660/r$ is convex for $r > 0$; setting $A'(r) = 4\pi r - 660/r^2 = 0$ gives $r = (330/2\pi)^{1/3} \approx 3.74$ cm and $h = 2r \approx 7.5$ cm, area about 264 cm². Real cans are narrower ($r \approx 3.3$ cm, so $h \approx 9.6$ cm) and use only about 1.5% more metal: near a smooth minimum the loss is flat, so designers can trade a little optimality for a can that fits a hand and a vending machine. *Try it:* Design the drink can .
- **Physics: nature optimises.** Light travels between two points along the path of least time (Fermat's principle); minimising travel time across a boundary between air and water gives Snell's law of refraction. A soap film minimises its area, and a hanging chain minimises its potential energy.
- **Engineering and logistics:** aircraft wings are shaped by optimisers that minimise drag subject to lift and strength constraints; airlines schedule crews and delivery companies plan routes with linear and integer programming, constrained optimisation on a huge scale.
- **Finance:** the minimum-variance portfolio of Chapter 30 is a constrained optimisation: minimise variance subject to the weights summing to 1.

Computer-science connection: optimisation is the engine of machine learning

"Training a model" means running an optimiser on a loss. Every ML framework (PyTorch, JAX, TensorFlow) is, underneath, three things: a way to describe a loss as a composition of functions, an automatic differentiator that computes its gradient (Section 33.4), and a library of update rules — `SGD`, `SGD(momentum=0.9)`, `Adam`. Outside ML, optimisation appears as linear programming in logistics and scheduling, as convex solvers in control systems, and as the "solver" in constraint-based layout engines.

Cauchy's gradient descent

Gradient descent was proposed by Augustin-Louis Cauchy in 1847, in a three-page note on solving systems of equations arising from astronomical orbit calculations. Momentum is usually credited to Boris Polyak (1964), who called it the "heavy ball method" — the image is a ball rolling down the landscape, carrying its speed through small bumps. Stochastic approximation, the theory behind SGD, dates from Robbins and Monro in 1951.

Optimiser race

Draw a 2-D loss landscape (a ravine, a bowl, an egg box). Drop a ball and choose plain gradient descent, momentum, or Adam, with sliders for η and β. Watch the paths, count steps to reach the minimum, and find the learning rate where each method first diverges. On the egg box, try ten random starting points and see how many reach the global minimum.

Try it: Ten starts on the egg box · Open the optimiser race

US curriculum

Single-variable optimisation ("find the maximum area") appears in **AP Calculus AB**. Convexity, gradient descent, and constrained optimisation belong to undergraduate courses: multivariable calculus, then **Optimisation** or **Operations Research**, typically in the second or third year of a university STEM degree.

33.1.9. Exercises

1. Which of these are convex on all of $\mathbb{R}$? w^2, $|w|$, $\sin w$, e^w, w^3.
2. Apply two steps of gradient descent to $f(w) = (w-3)^2$ from $w_0 = 0$ with $\eta = 0.1$. Then find a formula for w_t, and the range of η for which the iteration converges.
3. Minimise $f(w) = (w-3)^2$ subject to $w \le 2$ using projected gradient descent. What is the answer, and what does the projection do?

▼ **Answers**

1. w^2 ($f'' = 2 \ge 0$), $|w|$ (chord check: the V-shape always lies below its chords), and e^w ($f'' = e^w > 0$) are convex. $\sin w$ has $f'' = -\sin w$, negative on $(0, \pi)$. w^3 has $f'' = 6w$, negative for $w < 0$.
2. $f'(w) = 2(w-3)$. Step 1: $w_1 = 0 - 0.1 \times 2(0-3) = 0 + 0.6 = 0.6$. Step 2: $w_2 = 0.6 - 0.1 \times 2(0.6-3) = 0.6 + 0.48 = 1.08$. In general $w_{t+1} - 3 = (w_t - 3) - 2\eta(w_t - 3) = (1-2\eta)(w_t - 3)$, so $w_t = 3 + (1-2\eta)^t(w_0 - 3)$; with $\eta = 0.1$, $w_t = 3 - 3(0.8)^t$. It converges exactly when $|1 - 2\eta| < 1$, that is $0 < \eta < 1$.

3. The unconstrained minimum $w = 3$ is not allowed. Each gradient step pushes w towards 3; whenever it exceeds 2 the projection $w \leftarrow \min(w, 2)$ clips it back. The iteration settles at $w^* = 2$, on the boundary, with $f(2) = 1$. At that point $f'(2) = -2 \neq 0$: at a constrained optimum the gradient need not be zero, which is exactly what Lagrange multipliers account for.

33.2. Linear regression, end to end

33.2.1. The problem

You have recorded how many hours five students studied and their test scores (out of 6):

hours x	**1**	**2**	**3**	**4**	**5**
score y	2	4	5	4	5

The points do not lie on a line, but there is clearly a trend. What is the *best* straight line through them? And what does "best" even mean?

This tiny question contains, in miniature, the whole workflow of machine learning: choose a **model**, choose a **loss** that says how wrong the model is, and **optimise**.

33.2.2. Intuition

A line $\hat{y} = b + wx$ makes a prediction $\hat{y}_i$ (read "y-hat") for each x_i. The vertical gap $y_i - \hat{y}_i$ is the error, or residual. We want the residuals small, all at once. Adding them up fails — positive and negative errors cancel — so we square them first. Squaring makes every error count positively, punishes large errors more than small ones, and (crucially) produces a smooth, convex bowl that calculus can minimise exactly.



Residual, model, design matrix

A **model** is a function with adjustable parameters that maps inputs to predictions; here $\hat{y} = b + wx$ with **intercept** b and **slope** (or **weight**) w. A **residual** is $r_i = y_i - \hat{y}_i$, observed minus predicted. The **sum of squared errors** is $\mathrm{SSE} = \sum_i r_i^2$; dividing by n gives the **mean squared error** (MSE). The **design matrix** X has one row per example and one column per parameter; for a line, row i is $[1,\ x_i]$, the 1 multiplying the intercept.

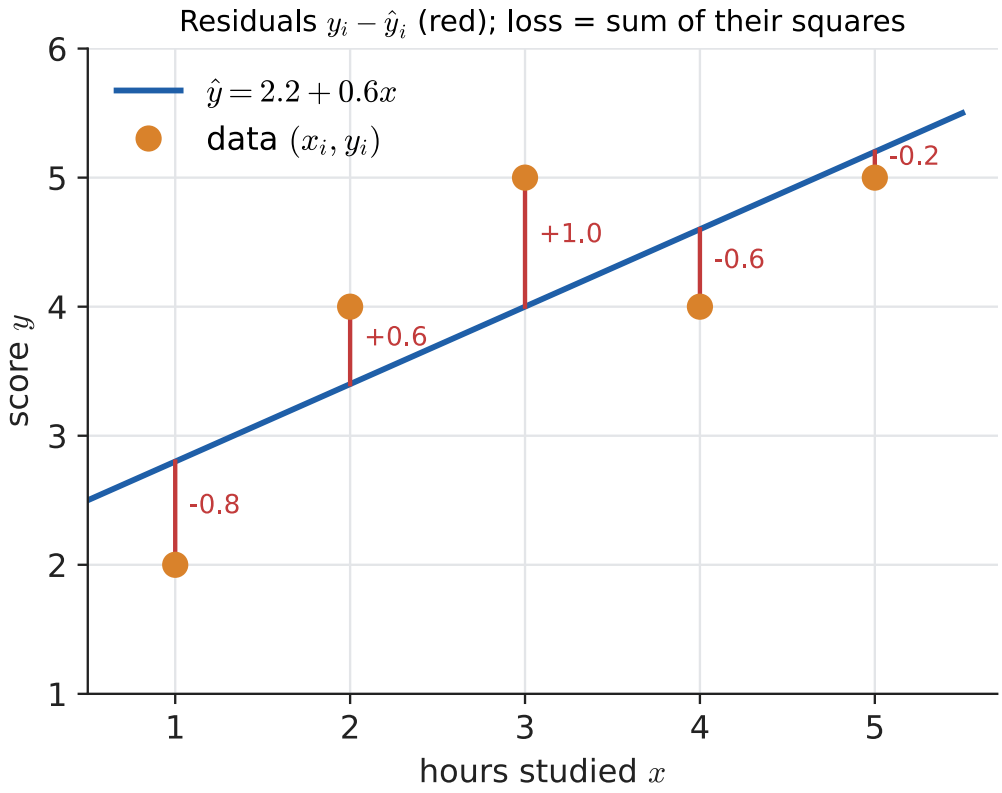


Figure 74. The fitted line and its residuals. Least squares chooses the line that makes the total area of the squares on these red segments as small as possible.

33.2.3. Why it works: deriving the best line by calculus

The loss is a function of the two parameters:

$$L(b, w) = \sum_{i=1}^{n} \big(y_i - b - w x_i\big)^2.$$

At the minimum of a smooth bowl, both partial derivatives are zero (Volume 2, Chapter 20). Differentiate using the chain rule; the outer function is "square", the inner is $y_i - b - w x_i$:

$$\frac{\partial L}{\partial b} = \sum_i 2\big(y_i - b - w x_i\big) \cdot (-1) = -2 \sum_i \big(y_i - b - w x_i\big)$$

$$\frac{\partial L}{\partial w} = \sum_i 2\big(y_i - b - w x_i\big) \cdot (-x_i) = -2 \sum_i x_i \big(y_i - b - w x_i\big)$$

Set each to zero, divide by -2, and split the sums:

$$\sum_i y_i = n b + w \sum_i x_i$$

$$\sum_i x_i y_i = b \sum_i x_i + w \sum_i x_i^2$$

These two linear equations in b and w are the **normal equations**. You know how to solve them from Volume 1, Chapter 6 .

33.2.4. The method

Example 49. Procedure — Least-squares line by the normal equations

Input: data points $(x_1, y_1), \dots, (x_n, y_n)$.

Steps:

1. Compute the four sums $\sum x_i$, $\sum y_i$, $\sum x_i^2$, $\sum x_i y_i$.
2. Write the normal equations $\sum y_i = nb + w\sum x_i$ and $\sum x_i y_i = b\sum x_i + w\sum x_i^2$.
3. Solve the 2-by-2 system for w and b (elimination, or the closed form $w = \dfrac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}$, $b = \bar{y} - w\bar{x}$).
4. Optionally compute the residuals and the SSE to judge the fit.

Output: the line $\hat{y} = b + wx$ minimising the sum of squared residuals.

33.2.5. Worked example

For the study data, $n = 5$: *Try it:* Normal equations, step by step ; Fit the least-squares line .

$$\begin{aligned} \sum x_i &= 1+2+3+4+5 = 15 \\ \sum y_i &= 2+4+5+4+5 = 20 \\ \sum x_i^2 &= 1+4+9+16+25 = 55 \\ \sum x_i y_i &= 1\cdot 2 + 2\cdot 4 + 3\cdot 5 + 4\cdot 4 + 5\cdot 5 = 2+8+15+16+25 = 66 \end{aligned}$$

The normal equations are

$$\begin{aligned} 20 &= 5b + 15w \\ 66 &= 15b + 55w \end{aligned}$$

Multiply the first by 3 and subtract it from the second:

$$\begin{aligned} 60 &= 15b + 45w && \text{(first equation} \times 3\text{)} \\ 66 - 60 &= (55-45)w && \text{(subtract)} \\ 6 &= 10w \quad \Rightarrow \quad w = 0.6 \\ 5b &= 20 - 15(0.6) = 11 \quad \Rightarrow \quad b = 2.2 \end{aligned}$$

So the best line is $\hat{y} = 2.2 + 0.6x$: each extra hour of study is associated with 0.6 more points. The predictions at $x = 1, \dots, 5$ are 2.8, 3.4, 4.0, 4.6, 5.2; the residuals are $-0.8, 0.6, 1.0, -0.6, -0.2$ (they sum to zero, as the first normal equation guarantees); and

$$\text{SSE} = 0.64 + 0.36 + 1.00 + 0.36 + 0.04 = 2.4, \qquad \text{MSE} = 2.4/5 = 0.48.$$

33.2.6. The same thing in matrix form

With many input features, writing out sums becomes hopeless. Matrices (Volume 2, Chapter 27) package it. Stack the parameters into $\theta = [b,\ w]^{\mathsf{T}}$, the targets into $\mathbf{y}$, and the inputs into the design matrix X. Then all predictions at once are $X\theta$, the residual vector is $\mathbf{y} - X\theta$, and

$$\begin{aligned} L(\theta) &= \|X\theta - \mathbf{y}\|^2 = (X\theta - \mathbf{y})^{\mathsf{T}}(X\theta - \mathbf{y}) \\ &= \theta^{\mathsf{T}}X^{\mathsf{T}}X\theta - 2\,\theta^{\mathsf{T}}X^{\mathsf{T}}\mathbf{y} + \mathbf{y}^{\mathsf{T}}\mathbf{y} && \text{(expand; the two cross terms are equal scalars)} \\ \nabla L(\theta) &= 2X^{\mathsf{T}}X\theta - 2X^{\mathsf{T}}\mathbf{y} && \text{(matrix analogues of } \tfrac{d}{d\theta}(a\theta^2 - 2c\theta) = 2a\theta - 2c\text{)} \\ \nabla L(\theta) = \mathbf{0} &\iff X^{\mathsf{T}}X\,\theta = X^{\mathsf{T}}\mathbf{y}. \end{aligned}$$

That last line is the **normal equations** in matrix form, valid for any number of features. For the study data,

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad X^{\mathsf{T}}X = \begin{bmatrix} 5 & 15 \\ 15 & 55 \end{bmatrix}, \quad X^{\mathsf{T}}\mathbf{y} = \begin{bmatrix} 20 \\ 66 \end{bmatrix},$$

which are precisely the coefficients we found by hand. Two derivations, one by sums and partial derivatives, one by matrices and gradients, land on the same equations. That is a good sign you understand both.



Why "normal"?
The equations $X^{\mathsf{T}}(\mathbf{y} - X\theta) = \mathbf{0}$ say that the residual vector is *perpendicular* ("normal", in the geometric sense) to every column of X. Least squares is orthogonal projection: the best prediction $X\theta$ is the shadow of $\mathbf{y}$ on the space spanned by the columns of X. Legendre published the method in 1805; Gauss claimed (credibly) to have used it since 1795, and famously used it to recover the lost dwarf planet Ceres from a handful of observations.

33.2.7. Alternative approach: gradient descent

Instead of solving the equations, we can walk downhill on the MSE. Its gradient is $\frac{2}{n}X^{\mathsf{T}}(X\theta - \mathbf{y})$. Starting from $\theta = \mathbf{0}$ with $\eta = 0.05$, the first step is *Try it:* Gradient descent versus the solve .

$$\nabla = \tfrac{2}{5}\left(X^{\mathsf{T}}X\,\mathbf{0} - X^{\mathsf{T}}\mathbf{y}\right) = -0.4\begin{bmatrix} 20 \\ 66 \end{bmatrix} = \begin{bmatrix} -8 \\ -26.4 \end{bmatrix}, \qquad \theta_1 = \mathbf{0} - 0.05\begin{bmatrix} -8 \\ -26.4 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 1.32 \end{bmatrix}.$$

Both methods in code:

PYTHON

```python
import numpy as np

# Data: hours studied (x) vs exam score (y)
x = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
y = np.array([2.0, 4.0, 5.0, 4.0, 5.0])

# --- Method 1: closed form via the normal equations ---
X = np.column_stack([np.ones_like(x), x])    # design matrix: [1, x]
theta = np.linalg.solve(X.T @ X, X.T @ y)    # solve (X^T X) theta = X^T y
print("normal equations:", theta)

# --- Method 2: gradient descent on the mean squared error ---
def mse(theta):
    r = X @ theta - y
    return (r @ r) / len(y)

def grad(theta):
    return 2 * X.T @ (X @ theta - y) / len(y)

theta_gd = np.zeros(2)
eta = 0.05                                   # learning rate
for step in range(2001):
    theta_gd -= eta * grad(theta_gd)
    if step in (0, 10, 100, 1000, 2000):
        print(f"step {step:4d}  theta={theta_gd.round(4)}  loss={mse(theta_gd):.4f}")
```

```
normal equations: [2.2 0.6]
step    0  theta=[0.4  1.32]  loss=1.6464
step   10  theta=[0.6341 1.0337]  loss=0.9263
step  100  theta=[1.8624 0.6935]  loss=0.5007
step 1000  theta=[2.2 0.6]  loss=0.4800
step 2000  theta=[2.2 0.6]  loss=0.4800
```

Gradient descent takes about a thousand steps to match what the normal equations gave in one solve. On five points the closed form obviously wins. So why does anybody use gradient descent for regression?

And the stochastic version, on a thousand noisy points from the line $y = 3x + 2$: *Try it:* Run SGD yourself .

```python
import numpy as np

rng = np.random.default_rng(0)
n = 1000
x = rng.uniform(0, 10, n)
y = 3.0 * x + 2.0 + rng.normal(0, 1.0, n)     # true line: slope 3, intercept 2

w, b = 0.0, 0.0
eta, batch = 0.01, 32
for epoch in range(20):
    order = rng.permutation(n)                 # shuffle each epoch
    for start in range(0, n, batch):
        idx = order[start:start + batch]
        err = (w * x[idx] + b) - y[idx]
        w -= eta * 2 * np.mean(err * x[idx])   # gradient estimated from 32 points
        b -= eta * 2 * np.mean(err)
print(f"SGD estimate: slope {w:.2f}, intercept {b:.2f}")
X = np.column_stack([np.ones(n), x])
b_exact, w_exact = np.linalg.solve(X.T @ X, X.T @ y)
print(f"Exact least squares: slope {w_exact:.2f}, intercept {b_exact:.2f}")
```

```
SGD estimate: slope 3.11, intercept 1.93
Exact least squares: slope 3.00, intercept 2.00
```

SGD lands close but not exactly on the answer: with a constant learning rate it keeps jittering around the minimum, as the Watch out box in Section 33.1 warned. Decaying η over the epochs would tighten it.

33.2.8. Choosing a method

- **Normal equations** (`np.linalg.solve(X.T @ X, X.T @ y)`): fine when the number of features p is up to a few thousand. Cost $O(np^2 + p^3)$.
- **QR decomposition or** `np.linalg.lstsq`: the numerically preferred closed form. Forming $X^\top X$ *squares* the condition number (Volume 2, Chapter 32), so nearly collinear features lose twice as many digits; QR works on X directly. This is what statistical software actually does.
- **Gradient descent / SGD:** when n is too large to fit in memory, when data arrive as a stream, or when the model is not linear (then there is no closed form at all, and gradient descent is the only game in town). Learning it on a problem where you can check the answer is the point of this section.



Least squares has blind spots

- **Outliers dominate.** Squaring means one point 10 units off contributes as much as a hundred points 1 unit off. If your data contain wild values, consider absolute error (more robust, but not smooth) or the Huber loss.
- **Extrapolation.** The study line predicts a score of $2.2 + 0.6 \times 20 = 14.2$ out of 6 for twenty hours of work. A model is only trustworthy near the data it was fitted to.
- **Correlation is not causation.** The slope says scores and hours move together in this data, not that studying *causes* the improvement (Chapter 31).
- **Collinear features.** If two columns of X are (nearly) multiples of each other, $X^\mathsf{T}X$ is (nearly) singular and the solution is unstable. Ridge regression, $(X^\mathsf{T}X + \lambda I)\theta = X^\mathsf{T}\mathbf{y}$, fixes this — the penalty from Section 33.1 in action.

Try it: Outliers: squared versus absolute .

33.2.9. Where you'd use this

- **Astronomy: the age of the universe.** Plot the recession speeds of galaxies against their distances and fit a line through the origin: the slope is the Hubble constant, about 70 km/s per megaparsec. Its reciprocal, converted to years, is about 14 billion years, a first estimate of the age of the universe from a least-squares slope.
- **Everyday: pricing.** Regress house prices on floor area in your town and the slope is the price per extra square metre; estate agents' "comparable sales" valuations are an informal version, and automated valuation websites use multiple regression with dozens of features.
- **Science and medicine:** dose against response, temperature against reaction rate (after taking logarithms, as in the Arrhenius equation), and height against age on a growth chart are all fitted by least squares; the fitted coefficients are the scientific result.

Computer-science connection: regression everywhere

Linear regression is the workhorse behind capacity planning (requests per second versus CPU), performance modelling (runtime versus input size on a log–log plot gives the exponent in $O(n^k)$), A/B-test analysis with covariates, and the "linear layer" that is the basic building block of every neural network. A linear layer is literally $X\theta$ with a learned θ; the network simply refuses to stop there.

Drag the points

Plot five to twenty points and the least-squares line. Drag any point and watch the line, the residual squares, and the SSE update live. Drag one point far away to feel how an outlier pulls the line. Toggle between squared and absolute error and compare.

Try it: Drag the points · Least-squares fit and residuals

US curriculum

Lines of best fit appear informally in **Algebra I** (grades 8–9), usually fitted by calculator. The least-squares regression line, residuals and r^2 are core content in **AP Statistics**. The derivation by calculus and the matrix normal equations belong to university **linear algebra** and **statistics** courses.

33.2.10. Exercises

1. Fit the least-squares line to $(0,1),(1,3),(2,5)$. What is the SSE?
2. Fit the least-squares line to $(1,1),(2,2),(3,2),(4,4)$, showing the four sums.
3. For a line through the origin, $\hat{y} = wx$, derive the least-squares slope.

▼ **Answers**

1. The points lie exactly on $y = 2x + 1$, so that line has every residual zero and SSE $= 0$; no line can do better. (Check with sums: $n = 3$, $\sum x = 3$, $\sum y = 9$, $\sum x^2 = 5$, $\sum xy = 13$; equations $9 = 3b + 3w$, $13 = 3b + 5w$; subtracting gives $4 = 2w$, so $w = 2$, $b = 1$.)
2. $n = 4$, $\sum x = 10$, $\sum y = 9$, $\sum x^2 = 30$, $\sum xy = 1 + 4 + 6 + 16 = 27$. Equations: $9 = 4b + 10w$ and $27 = 10b + 30w$. Multiply the first by 2.5: $22.5 = 10b + 25w$. Subtract: $4.5 = 5w$, so $w = 0.9$, and $4b = 9 - 9 = 0$, so $b = 0$. Line: $\hat{y} = 0.9x$.
3. $L(w) = \sum (y_i - wx_i)^2$,
   $L'(w) = -2\sum x_i(y_i - wx_i) = 0 \Rightarrow \sum x_i y_i = w\sum x_i^2 \Rightarrow w = \sum x_i y_i / \sum x_i^2$. Since $L''(w) = 2\sum x_i^2 > 0$, this is a minimum.

33.3. Logistic regression: from lines to probabilities

33.3.1. The problem

Now the question is yes/no. Ten students studied for between half an hour and five hours; some passed (1) and some failed (0):

hours	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
passed	0	0	0	1	0	1	0	1	1	1

We want a model that outputs the *probability* of passing. A straight line is wrong for the job: it happily predicts probabilities of -0.3 or 1.4, and a single extreme student drags it around. We need a function that takes any real number and squeezes it into $(0, 1)$.

33.3.2. Intuition: the sigmoid

The **sigmoid** (or **logistic**) function is

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Read it piece by piece. For large positive z, e^{-z} is tiny, so $\sigma(z) \approx 1/1 = 1$. For large negative z, e^{-z} is huge, so $\sigma(z) \approx 0$. At $z = 0$, $\sigma(0) = 1/(1+1) = 1/2$. It is an S-shaped curve that turns a "score" of any size into a probability.

Logistic regression computes a linear score and passes it through the sigmoid:

$$p = P(\text{pass} \mid x) = \sigma(wx + b).$$



Odds, logit, sigmoid

The **odds** of an event with probability p are $p/(1-p)$: a probability of 0.8 is odds of 4 to 1. The **logit** is the log-odds, $\text{logit}(p) = \ln \frac{p}{1-p}$, which runs over all of $\mathbb{R}$. The **sigmoid** $\sigma(z) = 1/(1+e^{-z})$ is the inverse of the logit. So logistic regression says: *the log-odds are a linear function of the input*. Each extra unit of x multiplies the odds by e^{w}.

33.3.3. Why it works: the loss comes from probability

What should the loss be? Squared error is tempting but leads to a non-convex problem here. A principled answer comes from probability (Chapter 29): choose w, b to make the observed data *as likely as possible* — **maximum likelihood**.

If the model says student i passes with probability p_i, then the probability of the observed outcome $y_i \in \{0, 1\}$ is p_i if they passed and $1 - p_i$ if they failed. A neat way to write both cases at once is $p_i^{y_i}(1-p_i)^{1-y_i}$ (one of the exponents is always zero). Treating students as independent, the likelihood of the whole data set is the product, and

$$\text{likelihood} = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i}$$

$$\ln(\text{likelihood}) = \sum_{i=1}^{n} \Big[y_i \ln p_i + (1-y_i)\ln(1-p_i) \Big] \qquad \text{(log turns products into sums)}$$

$$L(w,b) = -\frac{1}{n} \sum_{i=1}^{n} \Big[y_i \ln p_i + (1-y_i)\ln(1-p_i) \Big] \qquad \text{(negate and average: a loss to minimise)}$$

The symbol $\prod$ (capital Greek *pi*) is the product analogue of $\sum$: multiply the terms rather than add them. The final expression is the **binary cross-entropy** loss. You will see in Section 33.6 that "cross-entropy" is a precise term from information theory, not a coincidence of naming.



Cross-entropy loss

For a true label $y \in \{0, 1\}$ and predicted probability p, the **binary cross-entropy** is $-[y \ln p + (1 - y) \ln(1 - p)]$. If $y = 1$ it is $-\ln p$: zero when $p = 1$, growing without bound as $p \to 0$. Confident wrong answers are punished severely. It is also called **log loss** or **negative log-likelihood**.

33.3.4. Deriving the gradient

The gradient turns out to be beautifully simple. Two ingredients.

Derivative of the sigmoid. Write $\sigma(z) = (1 + e^{-z})^{-1}$ and use the chain rule:

$$\begin{aligned}\sigma'(z) &= -1 \cdot (1 + e^{-z})^{-2} \cdot (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} = \sigma(z)\,\big(1 - \sigma(z)\big) \qquad \text{since } \tfrac{e^{-z}}{1+e^{-z}} = 1 - \tfrac{1}{1+e^{-z}}.\end{aligned}$$

Derivative of the loss for one example. Let $\ell = -[y \ln p + (1 - y) \ln(1 - p)]$ with $p = \sigma(z)$, $z = wx + b$. Chain rule, link by link:

$$\begin{aligned}\frac{\partial \ell}{\partial p} &= -\frac{y}{p} + \frac{1 - y}{1 - p} \\ \frac{\partial \ell}{\partial z} &= \frac{\partial \ell}{\partial p} \cdot \frac{dp}{dz} = \left(-\frac{y}{p} + \frac{1 - y}{1 - p}\right) p(1 - p) = -y(1 - p) + (1 - y)p = p - y \\ \frac{\partial \ell}{\partial w} &= \frac{\partial \ell}{\partial z} \cdot \frac{\partial z}{\partial w} = (p - y)\,x, \qquad \frac{\partial \ell}{\partial b} = p - y.\end{aligned}$$

The gradient is "prediction minus truth, times input"—exactly the same shape as for linear regression. The messy sigmoid derivative cancelled perfectly against the logarithms. That cancellation is no accident; it is why cross-entropy and sigmoid are always used together.

33.3.5. The method and a worked run

Example 50. Procedure — Logistic regression by gradient descent

Input: examples (x_i, y_i) with $y_i \in \{0, 1\}$; learning rate η; number of steps.

Steps:

1. Initialise $w = 0$, $b = 0$.
2. Repeat:
   a. Compute predictions $p_i = \sigma(w x_i + b)$ for all i.
   b. Compute gradients $g_w = \frac{1}{n}\sum (p_i - y_i) x_i$ and $g_b = \frac{1}{n}\sum (p_i - y_i)$.
   c. Update $w \leftarrow w - \eta g_w$, $b \leftarrow b - \eta g_b$.
3. Stop when the gradients are near zero or the loss stops improving.

Output: w, b; predict $P(y = 1 \mid x) = \sigma(wx + b)$, and classify as 1 when this exceeds a threshold (often 0.5).

Try it: The fitted logistic curve .

By hand, the first step: with $w = b = 0$ every $p_i = \sigma(0) = 0.5$. Five students passed and five failed, so

$$g_b = \frac{1}{10}\sum_i (0.5 - y_i) = \frac{1}{10}\left(10 \times 0.5 - 5\right) = 0,$$

and b does not move yet. For w we need $\sum_i x_i = 27.5$ and the sum of the passers' hours, $2.0 + 3.0 + 4.0 + 4.5 + 5.0 = 18.5$:

$$g_w = \frac{1}{10}\sum_i (0.5 - y_i)\, x_i = \frac{1}{10}\left(0.5 \sum_i x_i - \sum_{\text{passers}} x_i\right) = \frac{1}{10}(13.75 - 18.5) = -0.475$$

$$w_1 = 0 - 0.1 \times (-0.475) = 0.0475$$

The slope starts positive: more hours, higher pass probability. The code runs the same update 5,000 times.

```python
import numpy as np

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

# Hours studied -> passed (1) or failed (0)
x = np.array([0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0])
y = np.array([0,   0,   0,   1,   0,   1,   0,   1,   1,   1  ])

w, b = 0.0, 0.0
eta = 0.1
for step in range(5000):
    p = sigmoid(w * x + b)                # predicted probabilities
    w -= eta * np.mean((p - y) * x)       # dL/dw
    b -= eta * np.mean(p - y)             # dL/db

p = sigmoid(w * x + b)
loss = -np.mean(y * np.log(p) + (1 - y) * np.log(1 - p))
print(f"w={w:.3f}  b={b:.3f}  cross-entropy={loss:.3f}")
print(f"P(pass | 1 hour)  = {sigmoid(w*1 + b):.3f}")
print(f"P(pass | 4 hours) = {sigmoid(w*4 + b):.3f}")
print(f"50% boundary at x = {-b/w:.2f} hours")
```

```
w=1.353  b=-3.721  cross-entropy=0.434
P(pass | 1 hour)  = 0.086
P(pass | 4 hours) = 0.844
50% boundary at x = 2.75 hours
```

Check one prediction by hand: at one hour, $z = 1.353 - 3.721 = -2.368$, $e^{2.368} \approx 10.68$, and $\sigma(-2.368) = 1/(1 + 10.68) \approx 0.086$. The **decision boundary** is where $p = 0.5$, that is where $z = 0$: $x = -b/w = 3.721/1.353 \approx 2.75$ hours. The slope means each extra hour multiplies the odds of passing by $e^{1.353} \approx 3.9$.

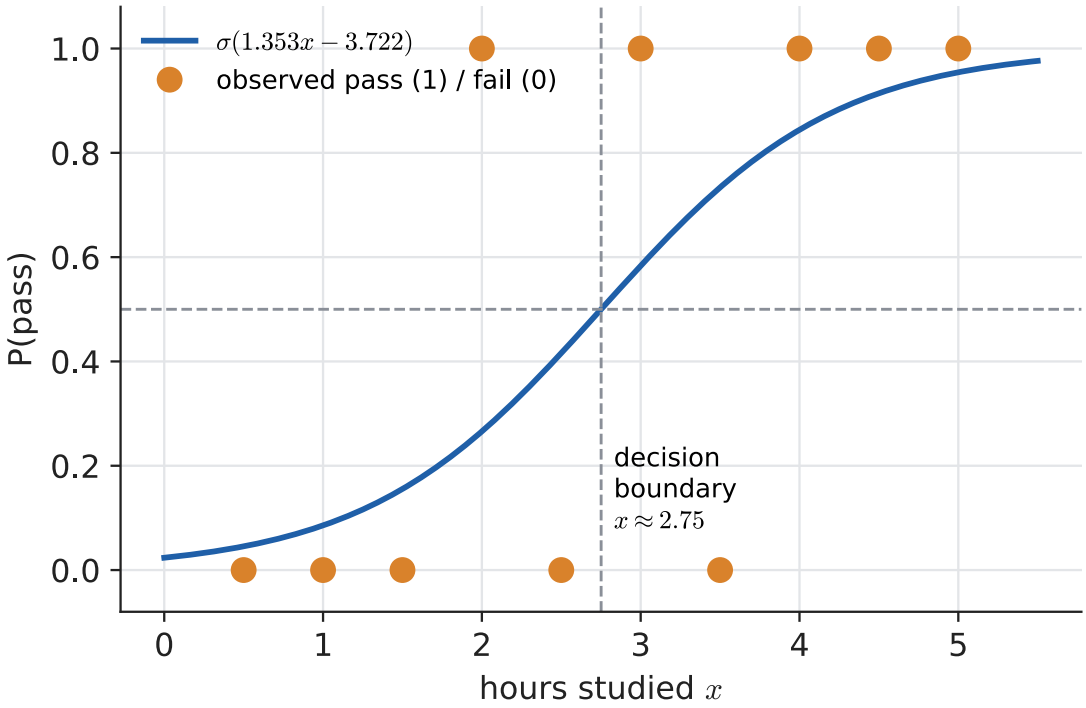


Figure 75. The fitted logistic curve. It never leaves the interval (0, 1), and crosses one half at the decision boundary.

33.3.6. Alternatives and choosing

Logistic regression is the natural first model for binary classification: convex (so training always finds the best fit), fast, and *interpretable*—each weight is a log-odds ratio you can explain to a colleague. Alternatives include decision trees and gradient-boosted trees (often more accurate on tabular data, less interpretable), support-vector machines, and neural networks, which are, as the next section shows, stacked logistic regressions. With more than two classes, the sigmoid generalises to the **softmax** (Section 33.9).



Perfect separation and log(0)

- If the classes are *perfectly* separable (every failer studied less than every passer), the loss keeps decreasing as $w \to \infty$; the weights grow without bound and the curve becomes a step. Regularisation (a penalty λw^2) fixes it.
- In floating point, $\sigma(z)$ rounds to exactly 1.0 for $z \gtrsim 37$, and then $\ln(1-p) = \ln 0 = -\infty$. Libraries compute the loss directly from z using stable formulas (for $y = 1$, $-\ln \sigma(z) = \ln(1 + e^{-z})$, evaluated with `log1p`)—a direct application of Volume 2, Chapter 32.
- A 0.5 threshold is a choice, not a law. For a cancer screen, a false negative costs far more than a false positive; lower the threshold.

33.3.7. Where you'd use this

- **Medicine: risk calculators.** Clinical risk scores that estimate, say, the 10-year probability of a heart attack from age, blood pressure, cholesterol and smoking are logistic (or closely related) regressions fitted to large patient studies. Each weight has a direct reading: a weight of 0.7 on "smoker"

multiplies the odds by $e^{0.7} \approx 2$.

- **Pharmacology: dose–response.** The fraction of patients who respond to a dose follows an S-curve in log-dose; the dose where the curve crosses 50% (the decision boundary $x = -b/w$) is the drug's ED50, the number on which dosing is based.
- **Finance:** credit scoring turns an applicant's history into a probability of default with a logistic model, and regulators like it precisely because each weight can be explained to a rejected customer.

Computer-science connection: classifiers in production

Logistic regression powered the first generation of spam filters and still drives large-scale click-through-rate prediction for advertising, where billions of sparse features make simple, fast, well-calibrated models valuable. Its probabilistic output is the point: a ranking system can sort by $P(\text{click})$, and an alerting system can pick its own threshold.



Logistic curve explorer

Plot pass/fail data and a sigmoid with sliders for w and b. Show the cross-entropy loss live, and a second panel with the loss surface over (w, b) as a contour plot with a dot for the current setting. Then press "train" and watch gradient descent slide the dot to the bottom of the (convex) bowl.

Try it: Sigmoid with w and b sliders · Logistic curve explorer

US curriculum

The logistic function appears in **Algebra II** and **Precalculus** as a model of limited population growth, and in **AP Calculus BC** as the solution of the logistic differential equation. Logistic regression as a classifier is university material (statistics, machine learning).

33.3.8. Exercises

1. Compute $\sigma(0), \sigma(2), \sigma(-2)$. What do you notice about the last two?
2. A model predicts $p = 0.9$ for an example whose true label is 1. What is the cross-entropy loss? What if it predicts $p = 0.1$?
3. Show algebraically that $\sigma(-z) = 1 - \sigma(z)$.

▼ **Answers**

1. $\sigma(0) = 0.5$; $\sigma(2) = 1/(1 + e^{-2}) = 1/1.1353 \approx 0.881$;
   $\sigma(-2) = 1/(1 + e^{2}) = 1/8.389 \approx 0.119$. They sum to 1 — see exercise 3.
2. $-\ln 0.9 \approx 0.105$; $-\ln 0.1 \approx 2.303$. Being confidently wrong costs about 22 times as much as being fairly confidently right.

3. $\sigma(-z) = \dfrac{1}{1+e^{z}} = \dfrac{e^{-z}}{e^{-z}+1}$ (multiply top and bottom by e^{-z})
$= \dfrac{(1+e^{-z})-1}{1+e^{-z}} = 1 - \dfrac{1}{1+e^{-z}} = 1-\sigma(z).$

33.4. Neurons, networks, and the chain rule

33.4.1. The problem

Logistic regression draws a single straight boundary. Many real problems need curved, tangled boundaries: recognising a handwritten 7, deciding whether a sentence is sarcastic. We want a model flexible enough to learn *any* reasonable function from data, yet still trainable by gradient descent.

33.4.2. Intuition: functions composed with functions

An artificial **neuron** is exactly logistic regression with several inputs:

$$a = \sigma(\mathbf{w}\cdot\mathbf{x} + b) = \sigma\Big(\sum_{j} w_j x_j + b\Big).$$

It takes a weighted sum of its inputs (a dot product, Volume 2, Chapter 26), adds a bias, and applies a nonlinear **activation function**. A **layer** is many neurons side by side, which in matrix form is $\mathbf{a} = \sigma(W\mathbf{x} + \mathbf{b})$ with σ applied to each component. A **network** is layers stacked, each feeding the next:

$$\hat{\mathbf{y}} = f_3\big(f_2\big(f_1(\mathbf{x})\big)\big), \qquad f_k(\mathbf{u}) = \sigma(W_k\mathbf{u} + \mathbf{b}_k).$$

That is all a neural network is: a composition of simple functions (Volume 1, Chapter 11), with a very large number of adjustable parameters.



Neuron, activation, layer

A **neuron** computes $\phi(\mathbf{w}\cdot\mathbf{x}+b)$ for weights $\mathbf{w}$, bias b, and **activation function** ϕ (Greek *phi*). Common activations: the sigmoid σ; $\tanh$; and the **ReLU** (rectified linear unit) $\mathrm{ReLU}(z) = \max(0, z)$. A **layer** applies many neurons to the same input. A **hidden layer** is any layer between input and output. The **forward pass** evaluates the network; the **backward pass** computes gradients.

33.4.3. Why the nonlinearity matters

Why not drop the activation and just stack linear layers? Because the composition of linear maps is linear: *Try it:* Train without an activation .

$$W_2(W_1\mathbf{x}) = (W_2W_1)\mathbf{x}.$$

A hundred linear layers collapse into one matrix (Volume 2, Chapter 27), so the "deep" network would be no more expressive than linear regression. The nonlinearity between layers is what lets each layer bend the space the next one sees. The **universal approximation theorem** makes this precise: one hidden layer with enough sigmoid (or ReLU) neurons can approximate any continuous function on a bounded region as closely as you like.

33.4.4. Why training works: backpropagation is the chain rule

To train with gradient descent, we need $\partial L/\partial w$ for every weight. The loss is a long composition, and the chain rule (Volume 2, Chapter 17 , and its multivariable form in Volume 2, Chapter 20) says: the derivative of a composition is the product of the derivatives of the links. **Backpropagation** is nothing more than evaluating that product efficiently, starting from the loss and working backwards, reusing each intermediate result.

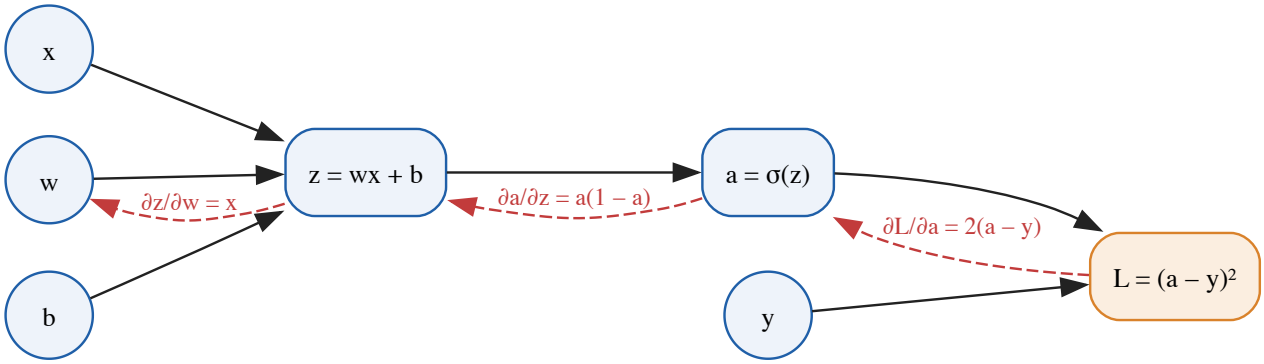


Figure 76. Computation graph for one neuron. Solid arrows: the forward pass computes values left to right. Dashed arrows: the backward pass multiplies local derivatives right to left.

Example 51. Procedure — Backpropagation through a chain of functions

Input: a computation expressed as a sequence of steps $u_1 \to u_2 \to \cdots \to L$, each with a known local derivative; parameter values.

Steps:

1. **Forward pass:** compute and store every intermediate value $u_1, u_2, \ldots, L$.
2. Set $\partial L/\partial L = 1$.
3. **Backward pass:** for each step in reverse order, multiply the incoming gradient by that step's local derivative: $\dfrac{\partial L}{\partial u_k} = \dfrac{\partial L}{\partial u_{k+1}} \cdot \dfrac{\partial u_{k+1}}{\partial u_k}$. Where a value feeds several later steps, *add* the contributions.
4. Read off $\partial L/\partial(\text{parameter})$ for each parameter.

Output: the gradient of the loss with respect to every parameter, at a cost of roughly two forward passes.

33.4.5. Worked example

One neuron, squared-error loss: input $x = 2$, target $y = 1$, weight $w = 0.5$, bias $b = -1$. *Try it:* Backprop through one neuron .

Forward pass.

$$
\begin{aligned}
z &= wx + b = 0.5 \times 2 - 1 = 0 \\
a &= \sigma(0) = 0.5 \\
L &= (a - y)^2 = (0.5 - 1)^2 = 0.25
\end{aligned}
$$

Backward pass, outermost link first:

$$
\begin{aligned}
\frac{\partial L}{\partial a} &= 2(a - y) = 2(0.5 - 1) = -1 \\
\frac{\partial a}{\partial z} &= a(1 - a) = 0.5 \times 0.5 = 0.25 && \text{(the sigmoid derivative)} \\
\frac{\partial z}{\partial w} &= x = 2, \qquad \frac{\partial z}{\partial b} = 1 \\
\frac{\partial L}{\partial w} &= (-1)(0.25)(2) = -0.5, \qquad \frac{\partial L}{\partial b} = (-1)(0.25)(1) = -0.25
\end{aligned}
$$

Both gradients are negative, so gradient descent will *increase* w and b, pushing a up towards the target 1. Now verify against a finite difference (Volume 2, Chapter 32) — the standard "gradient check" every ML engineer learns to do:

```python
import numpy as np

# One neuron: y_hat = sigmoid(w*x + b), squared-error loss
x, y = 2.0, 1.0
w, b = 0.5, -1.0

z = w * x + b                    # forward pass, step 1
a = 1 / (1 + np.exp(-z))         # forward pass, step 2
L = (a - y) ** 2                 # forward pass, step 3

dL_da = 2 * (a - y)              # backward pass: outermost link first
da_dz = a * (1 - a)
dz_dw = x
dL_dw = dL_da * da_dz * dz_dw    # chain rule: multiply the links
dL_db = dL_da * da_dz * 1.0

print(f"z={z:.4f} a={a:.4f} L={L:.4f}")
print(f"dL/dw={dL_dw:.4f}  dL/db={dL_db:.4f}")

# Check against a finite difference
h = 1e-6
L2 = (1 / (1 + np.exp(-((w + h) * x + b))) - y) ** 2
print(f"finite-difference dL/dw = {(L2 - L) / h:.4f}")
```

```
z=0.0000 a=0.5000 L=0.2500
dL/dw=-0.5000  dL/db=-0.2500
finite-difference dL/dw = -0.5000
```

In a real network the same pattern repeats through every layer. The vector form of each link is a **Jacobian** matrix (Volume 2, Chapter 20), and the backward pass is a chain of matrix–vector products, which is why GPUs — machines built for matrix multiplication — train networks so well.

33.4.6. Why backward and not forward?

You could apply the chain rule from the inputs forwards instead. For a function with one input and many outputs that would be efficient. But a loss has *one* output and *millions* of inputs (the parameters). Going backwards, one pass gives the derivative with respect to all of them at once; going forwards would need one pass per parameter. This is **reverse-mode automatic differentiation**, and the asymmetry is the entire reason deep learning is computationally feasible.



Vanishing gradients

The sigmoid's derivative $\sigma(1-\sigma)$ is at most $1/4$ (at $z=0$) and nearly zero when the neuron is saturated near 0 or 1. Backpropagation multiplies one such factor per layer, so through 20 sigmoid layers the gradient can shrink by a factor of $(1/4)^{20} \approx 10^{-12}$ — early layers stop learning. This **vanishing gradient** problem is why modern networks use ReLU (derivative exactly 1 for positive inputs), careful initialisation, and "residual connections" that give gradients a shortcut. The opposite, **exploding gradients**, happens when the factors exceed 1; the fix is gradient clipping.

33.4.7. Where you'd use this

- **Everyday:** face unlock on a phone, speech-to-text, photo search ("show me pictures of dogs"), live translation and the predictive keyboard are all neural networks trained by backpropagation.
- **Science and medicine:** AlphaFold predicts the 3-D shape of proteins from their amino-acid sequence (its designers shared the 2024 Nobel Prize in Chemistry); networks screen retinal photographs for diabetic eye disease; weather services now run neural forecasting models alongside the physics-based ones.
- **Engineering:** a network trained on thousands of slow physics simulations can act as a fast **surrogate model**, letting engineers explore millions of designs in the time one full simulation used to take.

Computer-science connection: autodiff

When you write `loss.backward()` in PyTorch, the framework has recorded every operation of the forward pass into a graph like the one above, and now walks it in reverse, calling each operation's local-derivative rule. You never write a derivative by hand. The same technique, **automatic differentiation**, is used far outside ML: in physics simulators, in financial risk engines (sensitivities of a portfolio to thousands of market inputs), and in differentiable rendering for graphics.

Backpropagation's many inventors

Reverse-mode automatic differentiation was described by Seppo Linnainmaa in his 1970 master's thesis; Paul Werbos applied it to neural networks in 1974; and the 1986 *Nature* paper by Rumelhart, Hinton and Williams made it famous. The perceptron — a single neuron with a step activation — dates from Frank Rosenblatt in 1958, and the idea of a neuron as a weighted threshold unit from McCulloch and Pitts in 1943.

Neuron playground

Build a tiny network (2 inputs, one hidden layer of up to 8 neurons, 1 output) and train it on a 2-D data set: two blobs, a circle inside a ring, a spiral. Watch the decision boundary bend as training proceeds. Remove the activation function and confirm that the boundary stays a straight line no matter how many layers you add. Hover over any weight to see its current gradient.

Try it: Train a tiny network

US curriculum

The chain rule is taught in **AP Calculus AB**. Neural networks, backpropagation and automatic differentiation are university topics (machine learning, typically third year), though many high schools now offer introductory AI electives that show networks without the calculus.

33.4.8. Exercises

1. Explain in one sentence why a network with no activation functions cannot learn the boundary "inside versus outside a circle". *Try it:* Check the exercise .
2. Repeat the worked example with $x = 1, y = 0, w = 1, b = 0$. Find L and $\partial L / \partial w$.
3. What is the derivative of $\mathrm{ReLU}(z)$? Why is it less prone to vanishing gradients?

Answers

1. Without activations the whole network is a single linear map, whose decision boundary $\mathbf{w} \cdot \mathbf{x} + b = 0$ is a straight line (a hyperplane), and no straight line separates the inside of a circle from the outside.
2. $z = 1, a = \sigma(1) \approx 0.7311, L = 0.7311^2 \approx 0.5344$. $\partial L / \partial a = 2(0.7311 - 0) \approx 1.4621$; $\partial a / \partial z = 0.7311 \times 0.2689 \approx 0.1966$; $\partial z / \partial w = x = 1$. So $\partial L / \partial w \approx 1.4621 \times 0.1966 \approx 0.2875$: positive, so gradient descent decreases w, lowering a towards 0.
3. It is 0 for $z < 0$ and 1 for $z > 0$ (undefined at 0; libraries pick 0). For active neurons the local factor is exactly 1, so it does not shrink the gradient layer after layer the way $\sigma' \leq 1/4$ does.

33.5. Embeddings, dot products, and search

33.5.1. The problem

You type "cat on mat" into a search box. Out of millions of documents, which should come first? Computers cannot compare meanings directly; they compare numbers. So the plan is: turn every document and every query into a vector, and measure how well they match with a single arithmetic operation. That operation is the dot product, and this is the $\mathbf{q} \cdot \mathbf{d}$ you will meet in every information-retrieval paper.

33.5.2. Intuition

Give every possible word its own axis. A document becomes a point (a vector) in this enormous space, with coordinate j saying how important word j is to the document. Documents about similar things point in similar directions. The query is a vector too. "How relevant is document $\mathbf{d}$ to query $\mathbf{q}$?" becomes "how closely do the two arrows point the same way?"

Recall from Volume 2, Chapter 26 that

$$\mathbf{q} \cdot \mathbf{d} = \sum_j q_j d_j = \|\mathbf{q}\| \, \|\mathbf{d}\| \cos\theta,$$

so the dot product mixes two things: the *angle* between the vectors (what we care about — topic) and their *lengths* (often an accident of document length). Dividing the lengths out gives **cosine similarity**:

$$\cos\theta = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\| \, \|\mathbf{d}\|}.$$

It is 1 for vectors pointing the same way, 0 for perpendicular (no shared words), and -1 for opposite directions (impossible for word counts, which are non-negative, but common for learned embeddings).

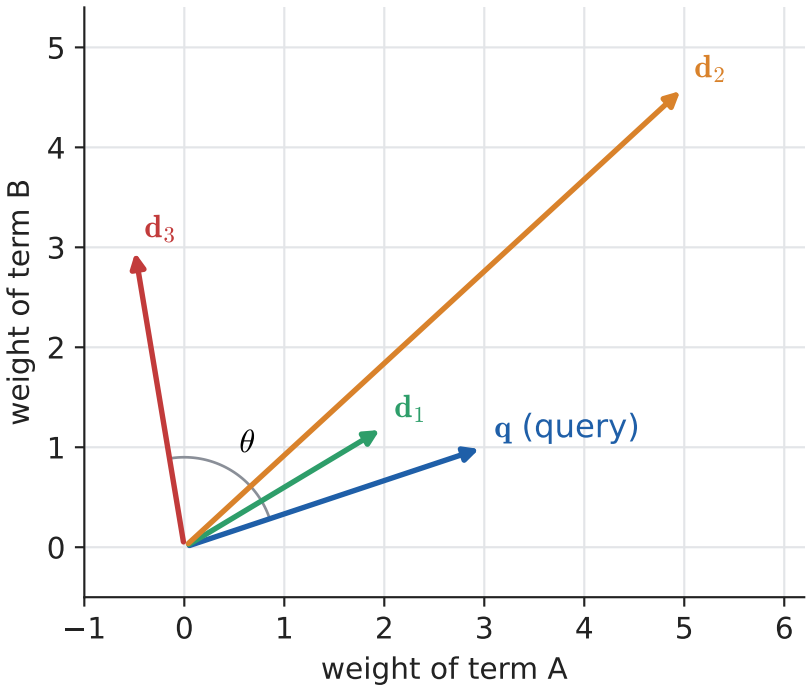


Figure 77. Dot product versus cosine. The long document $\mathbf{d_2}$ has the biggest dot product with $\mathbf{q}$ (19.6, against 7.2 for $\mathbf{d_1}$), simply because it is long. Cosine similarity ranks $\mathbf{d_1}$ first (0.976 against 0.912), because it points most nearly the same way. $\mathbf{d_3}$ is nearly perpendicular: cosine 0.156.



Cosine similarity, norm, unit vector

The **norm** (length) of a vector is $\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{\sum_j v_j^2}$. A **unit vector** has norm 1; **normalising** means dividing by the norm. **Cosine similarity** is the dot product of the two normalised vectors. For unit vectors, dot product and cosine similarity are the same number—which is why systems often normalise everything once at indexing time and then just take dot products.

33.5.3. Turning text into vectors: TF-IDF

Raw counts give too much weight to words like "the". Two adjustments fix this:

- **Term frequency** $\mathrm{tf}(t, d)$: how often term t appears in document d. More mentions, more relevant.
- **Inverse document frequency** $\mathrm{idf}(t) = \ln \dfrac{N}{\mathrm{df}(t)}$, where N is the number of documents and $\mathrm{df}(t)$ the number containing t. A word in every document gets $\ln 1 = 0$: it tells you nothing. A rare word gets a large weight.

The weight of term t in document d is $\mathrm{tf}(t, d) \times \mathrm{idf}(t)$. The logarithm (Volume 1, Chapter 12) is there to compress the scale: a word in 1 document out of a million should count for more than one in 1,000, but not a thousand times more.

33.5.4. Worked example

Three documents and a query: *Try it:* TF-IDF for “cat on mat” .

- d1: "the cat sat on the mat"

- d2: "the dog sat on the log"
- d3: "cats and dogs are pets"
- query: "cat on mat"

With $N = 3$: "cat" and "mat" appear only in d1, so $\text{idf} = \ln 3 \approx 1.0986$. "on", "the" and "sat" appear in two documents: $\text{idf} = \ln(3/2) \approx 0.4055$. The query vector has non-zero entries only for cat, on and mat, so only those coordinates of each document matter for the dot product:

$$\begin{aligned}
\mathbf{q} \cdot \mathbf{d}_1 &= \underbrace{(1.0986)(1.0986)}_{\text{cat}} + \underbrace{(0.4055)(0.4055)}_{\text{on}} + \underbrace{(1.0986)(1.0986)}_{\text{mat}} \\
&= 1.2069 + 0.1644 + 1.2069 = 2.578 \\
\mathbf{q} \cdot \mathbf{d}_2 &= (0.4055)(0.4055) = 0.164 && \text{(only ”on” is shared)} \\
\mathbf{q} \cdot \mathbf{d}_3 &= 0 && \text{(no shared terms)}
\end{aligned}$$

d1 wins decisively. Note that "the" (twice in d1) contributes nothing to the dot product because the query does not contain it — but it does lengthen $\mathbf{d}_1$, which the cosine will account for. The code computes both scores:

```python
import math
from collections import Counter

docs = {
    "d1": "the cat sat on the mat",
    "d2": "the dog sat on the log",
    "d3": "cats and dogs are pets",
}
query = "cat on mat"

tokenised = {name: text.split() for name, text in docs.items()}
vocab = sorted({w for words in tokenised.values() for w in words})
N = len(docs)
df = {w: sum(w in words for words in tokenised.values()) for w in vocab}
idf = {w: math.log(N / df[w]) for w in vocab}

def tfidf(words):
    counts = Counter(words)
    return [counts[w] * idf.get(w, 0.0) for w in vocab]

def dot(u, v):
    return sum(a * b for a, b in zip(u, v))

def cosine(u, v):
    nu, nv = math.sqrt(dot(u, u)), math.sqrt(dot(v, v))
    return dot(u, v) / (nu * nv) if nu and nv else 0.0

q = tfidf(query.split())
for name, words in tokenised.items():
    d = tfidf(words)
    print(f"{name}: q.d = {dot(q, d):.3f}   cos = {cosine(q, d):.3f}")
```

```
d1: q.d = 2.578   cos = 0.871
d2: q.d = 0.164   cos = 0.056
d3: q.d = 0.000   cos = 0.000
```

33.5.5. Alternatives: BM25 and learned embeddings

BM25, the default ranking function in search engines such as Lucene, Elasticsearch and OpenSearch, refines TF-IDF in two ways. Its score is *Try it:* Tune BM25's k_1 and b .

$$\mathrm{BM}\,25(q,d) = \sum_{t \in q} \mathrm{idf}(t) \cdot \frac{f(t,d)\,(k_1+1)}{f(t,d) + k_1\left(1 - b + b\,\dfrac{|d|}{\mathrm{avgdl}}\right)}.$$

Read it piece by piece: for each query term, an IDF weight times a *saturating* function of the term count $f(t,d)$. The fraction approaches $k_1 + 1$ as the count grows, so the tenth mention of "cat" adds much less than the first. The bracket in the denominator normalises for document length $|d|$ relative to the average length "avgdl": $b = 0$ ignores length, $b = 1$ normalises fully. Typical settings are $k_1 \approx 1.2$ and $b = 0.75$.

Dense embeddings replace the hand-built word axes with a few hundred learned dimensions. A neural network (Section 33.4) is trained so that texts with similar meanings get vectors with high cosine similarity — so "feline resting on a rug" lands near "cat on mat" despite sharing no words. The retrieval step is unchanged: $\mathbf{q} \cdot \mathbf{d}$, usually on unit vectors.



Embedding

An **embedding** is a map from discrete objects (words, sentences, images, users, products) to vectors in $\mathbb{R}^k$, chosen so that geometric relationships reflect real ones: similar objects get nearby vectors. **Sparse** embeddings (TF-IDF) have one dimension per vocabulary word and mostly zeros; **dense** embeddings have a few hundred learned dimensions, all typically non-zero.

Choosing. BM25 is fast, needs no training, and is excellent for exact keyword matches (error codes, names, part numbers). Dense embeddings capture meaning and synonyms but can miss rare exact terms and need a model. Modern systems often run both and combine the scores ("hybrid search").



Tokenisation and length traps

- In the example, d3 is about cats but scores zero, because "cats" and "cat" are different strings. Real systems **stem** or **lemmatise** words, or use subword tokens.
- Raw dot products favour long documents ($\mathbf{d_2}$ in the diagram). Use cosine, or normalise vectors, or use BM25's length correction — unless the model was trained for unnormalised dot products, in which case follow its convention.
- In high dimensions, almost all random vectors are nearly perpendicular. A cosine of 0.3 can be meaningful in one embedding model and noise in another; calibrate thresholds per model.

33.5.6. Where you'd use this

- **Everyday: recommendations.** A music service represents each listener and each song as a vector; songs whose vectors have a high cosine with yours are recommended, and "people who liked this also liked" compares item vectors with each other. The same geometry powers "similar products" in online shops and "more like this" on video sites.
- **Chemistry and drug discovery:** a molecule is encoded as a long 0/1 vector ("fingerprint") recording which structural fragments it contains; chemists search libraries of millions of compounds for molecules similar to a promising drug using a cosine-like overlap score (the Tanimoto coefficient).
- **Biology:** DNA and protein sequences are compared by counting shared short substrings, a TF-IDF-like vector that lets search tools find related genes in seconds.

Computer-science connection: vector databases

Ranking a query against a billion document vectors by brute force means a billion dot products per query. **Approximate nearest-neighbour** (ANN) indexes such as HNSW (a layered proximity graph, Chapter 25) and IVF (cluster the vectors, search only the nearest clusters) trade a little accuracy for orders of magnitude in speed. This is what a "vector database" is, and it is the retrieval half of retrieval-augmented generation (RAG) systems built around large language models.



king - man + woman ≈ queen

In the word2vec embeddings published by Mikolov and colleagues in 2013, vector arithmetic captured analogies: the vector $\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}}$ was closer to $\mathbf{v}_{\text{queen}}$ than to any other word. Directions in the space had come to mean things ("royalty", "gender") without anyone telling the model. TF-IDF is much older: Karen Spärck Jones introduced inverse document frequency in 1972.

Search engine in a box

Type a handful of short documents and a query. See the vocabulary, the TF-IDF vectors as a table, the dot product and cosine for each document, and the ranking. Toggle stemming on and off; switch between TF-IDF and BM25 and move the k_1 and b sliders to see how term saturation and length normalisation reorder the results.

Try it: Search engine in a box

US curriculum

Vectors and the dot product appear in **Precalculus** (often only in two dimensions). Cosine similarity, TF-IDF and embeddings are university topics (information retrieval, natural language processing), though the arithmetic is well within reach of a strong high-school student.

33.5.7. Exercises

1. Compute the cosine similarity of $(1, 0)$ and $(0, 1)$, and of $(1, 2)$ and $(2, 4)$.
2. For $\mathbf{q} = (1, 1, 0)$, $\mathbf{d}_1 = (2, 2, 0)$, $\mathbf{d}_2 = (1, 0, 1)$, compute both dot products and both cosines. Which document wins under each?
3. In the worked example, why does adding the word "the" ten more times to d1 leave $\mathbf{q} \cdot \mathbf{d}_1$ unchanged but lower its cosine?

▼ **Answers**

1. $(1, 0) \cdot (0, 1) = 0$, so cosine 0 (perpendicular). $(1, 2) \cdot (2, 4) = 2 + 8 = 10$; norms $\sqrt{5}$ and $\sqrt{20}$, product $\sqrt{100} = 10$; cosine 1 (same direction, different length).
2. $\mathbf{q} \cdot \mathbf{d}_1 = 2 + 2 + 0 = 4$, $\mathbf{q} \cdot \mathbf{d}_2 = 1 + 0 + 0 = 1$. Cosines: $4/(\sqrt{2} \cdot \sqrt{8}) = 4/4 = 1$ and $1/(\sqrt{2} \cdot \sqrt{2}) = 0.5$. $\mathbf{d}_1$ wins under both.
3. The query has weight 0 on "the", so that coordinate contributes $0 \times (\text{anything}) = 0$ to the dot product. But the extra occurrences increase $\|\mathbf{d}_1\|$, the denominator of the cosine, so the cosine falls.

33.6. Information theory: measuring surprise

33.6.1. The problem

How much information is in a message? A file compressor has to answer this concretely: how few bits can it get away with? A language model is graded by how well it predicts the next word — but what number measures "well"? And the logistic-regression loss was called *cross-entropy*. Why?

All three questions have one answer, due to Claude Shannon in 1948.

33.6.2. Intuition: information is surprise

Learning that something *likely* happened tells you little; learning that something *unlikely* happened tells you a lot. "The sun rose this morning" carries almost no information. "It snowed in July in Miami" carries a great deal.

Think of the game of twenty questions. If a number is chosen uniformly from 1 to 8, you can always find it with three yes/no questions ("Is it in the top half?"), because each answer halves the possibilities: $8 \to 4 \to 2 \to 1$. An event of probability $1/8$ needs 3 bits to pin down; $3 = \log_2 8 = -\log_2(1/8)$.

That suggests defining the **surprise** (or **self-information**) of an event with probability p as

$$I(p) = -\log_2 p = \log_2 \frac{1}{p} \quad \text{bits.}$$

Certain events ($p = 1$) have surprise 0. Each halving of probability adds one bit. And because logs turn products into sums (Volume 1, Chapter 12), the surprise of two independent events is the sum of their surprises — exactly what "amount of information" ought to do.

33.6.3. Entropy: average surprise

A random source — a coin, a die, the next letter of English text — produces outcomes with probabilities $p_1, \dots, p_k$. Its **entropy** is the *expected* surprise (expectation from Chapter 29):

$$H(p) = \mathbb{E}\big[-\log_2 p(X)\big] = -\sum_{i=1}^{k} p_i \log_2 p_i.$$

(By convention $0 \log 0 = 0$: impossible outcomes contribute nothing.)



Bit, surprise, entropy

A **bit** is the information in one fair yes/no answer. The **surprise** or **self-information** of an outcome with probability p is $-\log_2 p$ bits. **Entropy** H is average surprise: the uncertainty of a random source before you see the outcome. With natural logarithms the unit is the **nat** ($1 \text{ nat} \approx 1.443$ bits); ML papers often use nats without saying so.

33.6.4. Worked examples

Fair coin. $H = -\left(\tfrac{1}{2}\log_2 \tfrac{1}{2} + \tfrac{1}{2}\log_2 \tfrac{1}{2}\right) = -\left(\tfrac{1}{2}(-1) + \tfrac{1}{2}(-1)\right) = 1$ bit. *Try it:* Entropy and its Huffman code .

Biased coin with $P(\text{heads}) = 0.9$:

$$\begin{aligned} H &= -\big(0.9 \log_2 0.9 + 0.1 \log_2 0.1\big) \\ &= -\big(0.9 \times (-0.152) + 0.1 \times (-3.322)\big) \\ &= 0.137 + 0.332 = 0.469 \text{ bits.} \end{aligned}$$

The biased coin is more predictable, so it carries less than half a bit per flip.

Fair die. Six outcomes of probability $1/6$: $H = -6 \times \frac{1}{6}\log_2 \frac{1}{6} = \log_2 6 \approx 2.585$ bits. In general a uniform distribution over k outcomes has entropy $\log_2 k$, the maximum possible for k outcomes.

A skewed four-symbol source with probabilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}$:

$$H = \tfrac{1}{2}(1) + \tfrac{1}{4}(2) + \tfrac{1}{8}(3) + \tfrac{1}{8}(3) = 0.5 + 0.5 + 0.375 + 0.375 = 1.75 \text{ bits.}$$

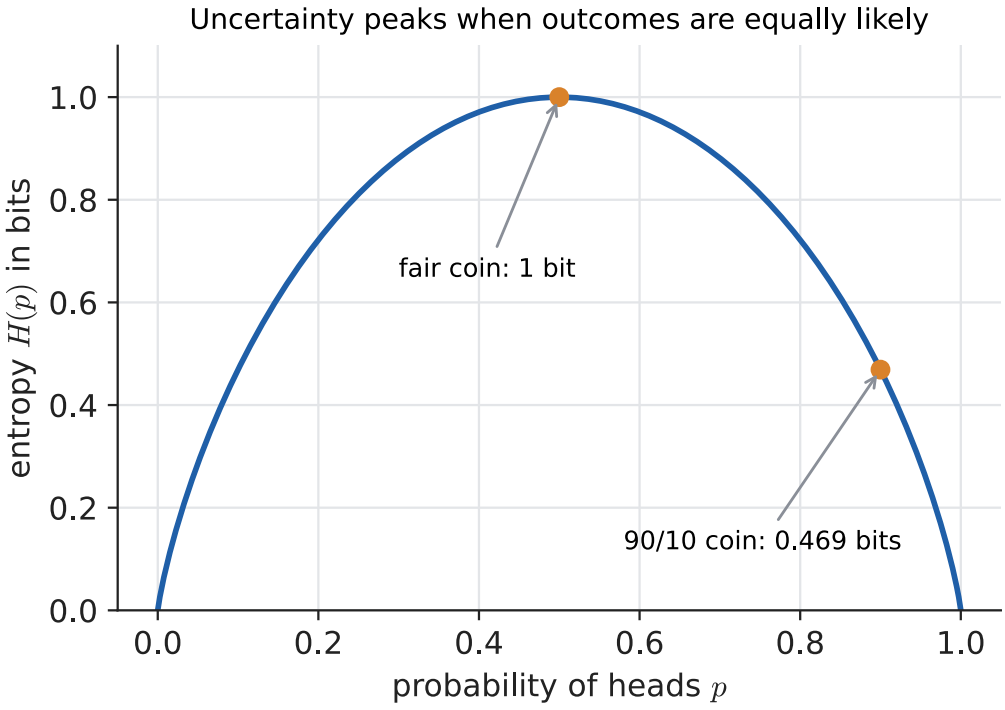


Figure 78. Entropy of a coin as a function of its bias. It is zero for a coin that always lands the same way and peaks at one bit for a fair coin.

33.6.5. Why compression connects to probability

Here is the payoff. For the four-symbol source, use the code

symbol	probability	codeword	length
A	1/2	0	1
B	1/4	10	2
C	1/8	110	3
D	1/8	111	3

No codeword is a prefix of another, so a stream like `0101100` decodes unambiguously as A, B, C, A. The average length is $\frac{1}{2}(1) + \frac{1}{4}(2) + \frac{1}{8}(3) + \frac{1}{8}(3) = 1.75$ bits per symbol—exactly the entropy. A naive fixed-length code needs 2 bits per symbol.

That is no coincidence. Each symbol got a codeword of length $-\log_2 p_i$, its surprise. **Shannon's source coding theorem** says that no lossless code can average fewer than H bits per symbol, and that codes can get arbitrarily close to H. Entropy is the compression limit. Huffman coding (a greedy algorithm on a binary tree, Chapter 25) and arithmetic coding are practical ways to approach it.

The deep moral: **to compress well, you must predict well.** A compressor is a probability model in disguise. Common symbols get short codes only if you know they are common.

33.6.6. Cross-entropy and KL divergence

Suppose the true distribution is p but you designed your code (or your model) for a distribution q. Symbol i then costs $-\log_2 q_i$ bits but occurs with frequency p_i, so the average cost is the **cross-entropy** *Try it:* KL divergence of two coins .

$$H(p, q) = -\sum_i p_i \log_2 q_i.$$

The extra cost over the best possible is the **Kullback–Leibler divergence**:

$$D_{\mathrm{KL}}(p \,\|\, q) = H(p, q) - H(p) = \sum_i p_i \log_2 \frac{p_i}{q_i}.$$

It is always ≥ 0, and zero only when $q = p$: you can never beat the truth. Read $D_{\mathrm{KL}}(p \,\|\, q)$ as "the number of extra bits per symbol you waste by believing q when the truth is p".



Cross-entropy and KL divergence

Cross-entropy $H(p, q)$ is the average surprise when outcomes come from p but you score them with q. **KL divergence** $D_{\mathrm{KL}}(p \,\|\, q)$ is the excess, $H(p, q) - H(p)$. It measures how different q is from p, but it is **not** a distance: in general $D_{\mathrm{KL}}(p \,\|\, q) \neq D_{\mathrm{KL}}(q \,\|\, p)$. The double bar $\|$ separates the two arguments.

Now the logistic-regression loss makes sense. For one example, the "true distribution" puts probability 1 on the observed label y, and the model's distribution is $(p, 1-p)$. Their cross-entropy is $-[y \ln p + (1-y)\ln(1-p)]$ — exactly the loss we derived from maximum likelihood. **Minimising cross-entropy, maximising likelihood, and minimising KL divergence to the data are the same thing.** Large language models are trained with exactly this loss on the next token, and their "perplexity" is just $2^{H(p,q)}$ (or e^{H} in nats).

```python
import math

def entropy(p):
    """Shannon entropy in bits. Terms with p_i = 0 contribute 0."""
    return -sum(pi * math.log2(pi) for pi in p if pi > 0)

def kl(p, q):
    """KL divergence D(p || q) in bits."""
    return sum(pi * math.log2(pi / qi) for pi, qi in zip(p, q) if pi > 0)

fair_coin   = [0.5, 0.5]
biased_coin = [0.9, 0.1]
fair_die    = [1/6] * 6
english_ish = [0.5, 0.25, 0.125, 0.125]   # four symbols, skewed

for name, p in [("fair coin", fair_coin), ("biased coin", biased_coin),
                ("fair die", fair_die), ("skewed 4-symbol", english_ish)]:
    print(f"{name:16s} H = {entropy(p):.3f} bits")

print(f"D(biased || fair) = {kl(biased_coin, fair_coin):.3f} bits")
print(f"D(fair || biased) = {kl(fair_coin, biased_coin):.3f} bits")
```

```
fair coin        H = 1.000 bits
biased coin      H = 0.469 bits
fair die         H = 2.585 bits
skewed 4-symbol  H = 1.750 bits
D(biased || fair) = 0.531 bits
D(fair || biased) = 0.737 bits
```

The last two lines show the asymmetry of KL divergence. By hand: $D(\text{biased} \,\|\, \text{fair}) = 0.9 \log_2 \frac{0.9}{0.5} + 0.1 \log_2 \frac{0.1}{0.5} = 0.9(0.848) + 0.1(-2.322) = 0.763 - 0.232 = 0.531$. The reverse direction is larger, because believing the coin is 90/10 when it is fair makes you very surprised by every tails.



Entropy is not "meaning"

- Shannon information measures *unpredictability*, not importance or truth. A file of random noise has maximal entropy and no meaning.
- Entropy depends on the model. English text has about 4.1 bits per letter if you only use single-letter frequencies, but closer to 1 bit per letter once you exploit context — which is why good compressors (and language models) look at context.
- Check the log base: bits ($\log_2$), nats ($\ln$), or occasionally bans ($\log_{10}$). The formulas are identical up to a constant factor.
- $D_{\mathrm{KL}}(p \,\|\, q)$ is infinite if $q_i = 0$ where $p_i > 0$. A model that assigns zero probability to something that happens is infinitely wrong — which is why models smooth their probabilities.

33.6.7. Where you'd use this

- **Everyday: games of guessing.** In twenty questions or a word-guessing game such as Wordle, the best question is the one whose answers are closest to equally likely, because that maximises the expected information (entropy) of the answer. Twenty perfect yes/no questions distinguish $2^{20} \approx 10^6$ possibilities.
- **Biology:** DNA has four bases, so at most $\log_2 4 = 2$ bits per base. The human genome's 3.1 billion bases are at most 6.2 billion bits, about 775 MB, and it compresses well below that because its letters are far from independent and equally likely. Ecologists use the same formula, the **Shannon diversity index**, to measure how evenly species are spread in a habitat.
- **Engineering: why dial-up modems stopped at about 33.6 kbit/s.** Shannon's capacity formula $C = B \log_2(1 + \mathrm{SNR})$ for a telephone line with bandwidth $B \approx 3000$ Hz and signal-to-noise ratio about 1000 gives $C \approx 3000 \times \log_2 1001 \approx 30\,000$ bits per second. Engineers got close to this limit, not past it; faster broadband needed wider bandwidth, not cleverer coding.

Computer-science connection: information everywhere

Entropy sets the limits for compression (gzip, zstd, and every image and video codec), for channel capacity in networking, and for password strength (a random 12-character password from 64 symbols has $12 \log_2 64 = 72$ bits of entropy). Decision trees choose splits by **information gain**, the drop in entropy. Cross-entropy is the training loss of almost every classifier and language model. KL divergence appears in variational autoencoders, in reinforcement learning (keeping a new policy close to the old one), and in monitoring for data drift.



The word "bit"

Shannon's 1948 paper "A Mathematical Theory of Communication" founded the field in one stroke, and credited John Tukey with coining "bit" as a contraction of "binary digit". Shannon reportedly chose the name "entropy" on John von Neumann's advice, who pointed out that the formula matched Boltzmann's entropy in statistical physics — and that "nobody knows what entropy really is, so in a debate you will always have the advantage."



Build a code

Enter symbol probabilities (or paste text and let the lab count letters). Watch the entropy update, build the Huffman tree step by step, and compare the average code length to H. Then deliberately use the wrong distribution q to build the code and see the average length rise by exactly $D_{\mathrm{KL}}(p \,\|\, q)$ when the probabilities are powers of 2.

Try it: Build a code

US curriculum

Logarithms base 2 are covered in **Algebra II**, and expected value in **AP Statistics**, so the ingredients are all high-school material. Information theory itself is taught at university, in electrical engineering, computer science and statistics departments.

33.6.8. Exercises

1. How many bits of surprise does an event of probability $1/8$ carry? Of probability $1/1024$?
2. Compute the entropy of $(\tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{4})$ and give a prefix code achieving it.
3. What is $D_{\mathrm{KL}}(p \,\|\, p)$? Using $H(p, q) = H(p) + D_{\mathrm{KL}}(p \,\|\, q)$, explain why minimising cross-entropy over q gives $q = p$.
4. A byte chosen uniformly at random has how many bits of entropy? What does that say about compressing encrypted data?

▼ **Answers**

1. $-\log_2(1/8) = 3$ bits; $-\log_2(1/1024) = 10$ bits.
2. $H = \tfrac{1}{2}(1) + \tfrac{1}{4}(2) + \tfrac{1}{4}(2) = 1.5$ bits. Code: A = `0`, B = `10`, C = `11`, with average length $0.5 + 0.5 + 0.5 = 1.5$.
3. $D_{\mathrm{KL}}(p \,\|\, p) = \sum p_i \log_2 1 = 0$. Since $H(p)$ does not depend on q and $D_{\mathrm{KL}} \geq 0$ with equality only at $q = p$, the cross-entropy is smallest exactly when $q = p$.
4. 256 equally likely values: $\log_2 256 = 8$ bits, the maximum for a byte. Good encryption output looks uniformly random, so it is essentially incompressible — which is why you compress *before* encrypting.

33.7. Differential equations: the next horizon

Every model in this chapter so far was *static*: inputs in, prediction out. But much of science is about how things *change*. A **differential equation** relates an unknown function to its own derivatives. The simplest says "the rate of growth is proportional to the current amount":

$$\frac{dy}{dt} = ky.$$

You already know its solution from Volume 1, Chapter 12 and Volume 2, Chapter 17: $y(t) = y_0 e^{kt}$, since differentiating $y_0 e^{kt}$ gives $k\,y_0 e^{kt} = ky$. That one equation describes compound interest, radioactive decay ($k < 0$), and unchecked population growth. Add a ceiling K and you get the **logistic equation**

$$\frac{dy}{dt} = ky\left(1 - \frac{y}{K}\right),$$

whose solution is a scaled sigmoid — the same S-curve as logistic regression, now describing an epidemic or the adoption of a new technology.

Newton's second law, $F = ma$, from your Mechanics days, is a differential equation too: acceleration is the second derivative of position, so a spring obeys $m\,x''(t) = -kx(t)$, whose solutions are sines and cosines (Volume 1, Chapter 13)—simple harmonic motion.

Most differential equations have no formula for their solution. Then you solve them numerically, exactly as in Volume 2, Chapter 32 : Euler's method steps forward with $y_{n+1} = y_n + h\,f(y_n)$, and Runge–Kutta methods do it more accurately. And here is a final connection that ties this chapter together. Gradient descent,

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t),$$

is *exactly* Euler's method with step size η applied to the differential equation $\dfrac{d\theta}{dt} = -\nabla L(\theta)$, called **gradient flow**: a ball sliding downhill in continuous time. Diverging with too large a learning rate is the same phenomenon as Euler's method going unstable with too large a step. Momentum corresponds to giving the ball mass and friction. *Try it:* Descent as Euler on gradient flow .



Differential equation

A **differential equation** is an equation involving an unknown function and its derivatives. An **ordinary** differential equation (ODE) has one independent variable (usually time); a **partial** differential equation (PDE) has several, with partial derivatives (heat flow, waves, fluid dynamics). An **initial-value problem** specifies the starting state, such as $y(0) = y_0$. The **order** is the highest derivative that appears.



Slope fields

Draw the slope field of $dy/dt = f(t, y)$ for a choice of f (growth, logistic, spring). Click anywhere to drop an initial condition and trace the solution curve. Overlay Euler's method with an adjustable step size and watch it peel away from the true solution as the step grows.

Try it: Open the slope-field lab

US curriculum

Slope fields, separable differential equations, exponential growth and (in BC) the logistic equation and Euler's method are part of **AP Calculus AB/BC**. A full course in ordinary differential equations is usually taken in the first or second year of university.

33.8. Where to go next

You have rebuilt a broad foundation. The map below shows how it connects to the subjects that lie just beyond this book. Arrows mean "is the main prerequisite for".

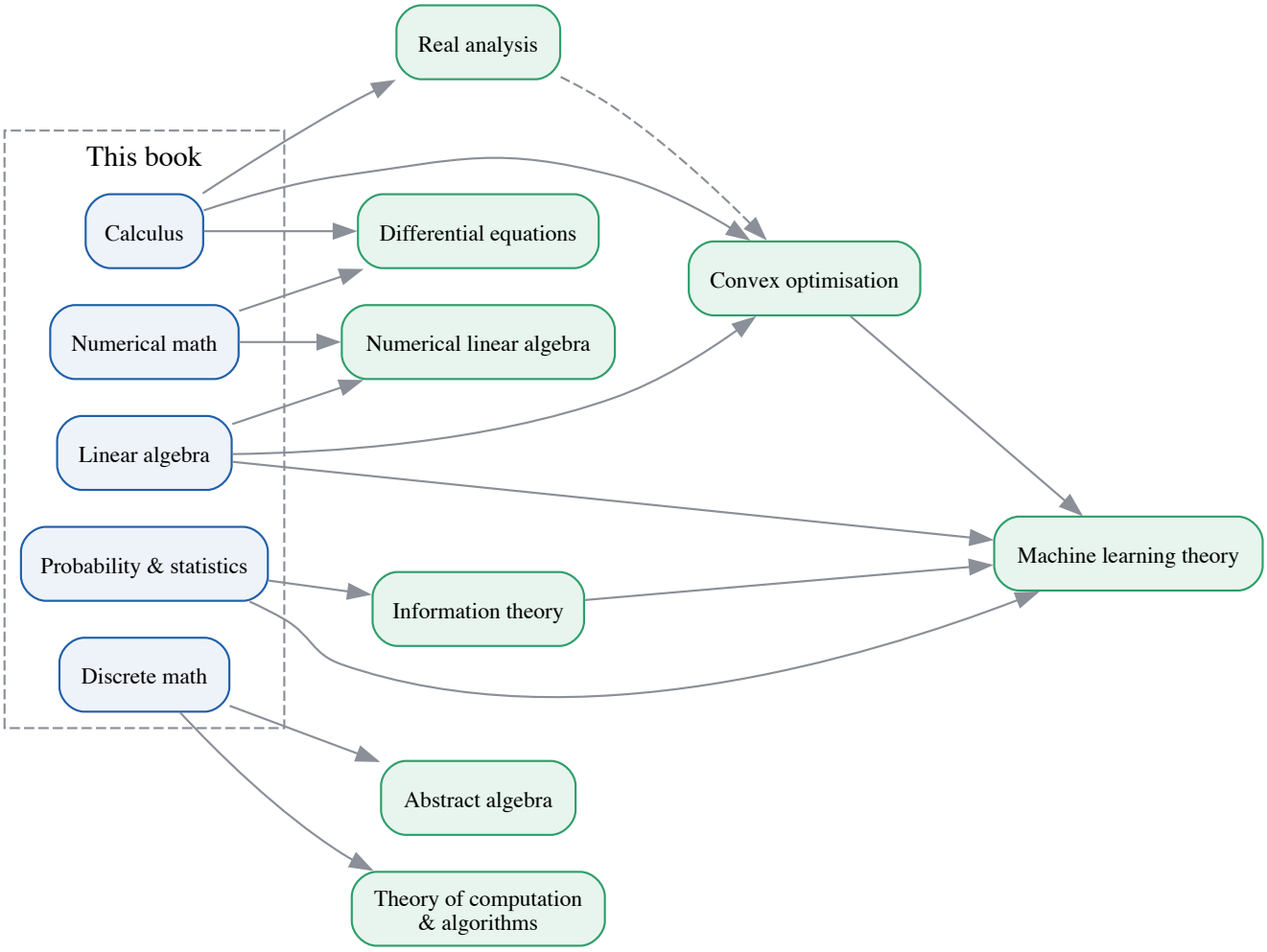


Figure 79. Where to go next. Blue boxes are the branches of this book; green boxes are natural next subjects.

Subject	What it is and why it matters to you	A good next step
Real analysis	Calculus done rigorously: what a limit, a real number, continuity and convergence really are. It is where "prove when proof teaches" becomes the main activity, and it underpins every convergence guarantee in optimisation and probability.	Stephen Abbott, *Understanding Analysis*. Work the exercises; this is a subject learned by proving.
Linear algebra, second pass	Abstract vector spaces, the singular value decomposition, projections and least squares in depth. The SVD alone explains PCA, recommender systems and low-rank approximation of neural networks.	Gilbert Strang, *Introduction to Linear Algebra* (and his recorded MIT lectures).

Subject	What it is and why it matters to you	A good next step
Convex optimisation	When problems are convex they can be solved reliably and fast; recognising and formulating convex problems is a superpower in ML, finance, control and operations research.	Boyd and Vandenberghe, *Convex Optimization* (freely available from the authors).
Numerical linear algebra	How computers actually solve $\boldsymbol{Ax} = \mathbf{b}$, least squares and eigenproblems at scale: QR, SVD, Krylov methods, conditioning and stability. The engine room under NumPy, SciPy and every ML library.	Trefethen and Bau, *Numerical Linear Algebra*.
Probability and statistics, second pass	Measure-free but rigorous probability, Bayesian inference, and statistical learning theory.	Blitzstein and Hwang, *Introduction to Probability*; then Hastie, Tibshirani and Friedman, *The Elements of Statistical Learning* (free online).
Information theory	Entropy, coding, channel capacity, and their deep links to inference and learning.	David MacKay, *Information Theory, Inference, and Learning Algorithms* (free online); Cover and Thomas for depth.
Machine learning	The modern synthesis of everything in this chapter.	Bishop, *Pattern Recognition and Machine Learning*; Goodfellow, Bengio and Courville, *Deep Learning* (free online).
Differential equations and dynamical systems	How systems evolve: physics, biology, epidemics, control, and the continuous-time view of optimisation.	Steven Strogatz, *Nonlinear Dynamics and Chaos* — unusually readable and full of applications.
Abstract algebra	Groups, rings and fields: the mathematics of symmetry, and the foundation of cryptography and error-correcting codes (building on Chapter 23).	Charles Pinter, *A Book of Abstract Algebra*.
Theory of computation and algorithms	Automata, computability, complexity, and the analysis of algorithms, built on Chapter 21, Chapter 22, Chapter 24 and Chapter 25.	Sipser, *Introduction to the Theory of Computation*; Cormen et al., *Introduction to Algorithms*.

How to choose. Follow your projects. If you work on search, go to information theory and numerical linear algebra. If you train models, go to convex optimisation and machine learning. If you found yourself enjoying the proofs in Chapter 22 and Chapter 23, go to real analysis or abstract algebra. Whatever you choose, keep the habit this book has tried to build: **understand → calculate by hand → implement → experiment**, and keep asking *when should I use this, and when shouldn't I?*

33.9. Reading mathematics

The success criterion for this book was never "can execute procedures". It was this: when you see a formula in a paper, you can unpack it — say what each symbol means, what the whole thing computes, why it has that shape, and how you would implement it. This final section practises exactly that on formulas you will genuinely meet in machine-learning and information-retrieval papers.

Example 52. Procedure — Reading a formula

Input: an unfamiliar formula in a paper.

Steps:

1. **Find the type of the result.** Is the left-hand side a number, a vector, a probability, a function, a set? Is this a definition (often $:=$ or $\equiv$), an equation to solve, or an update rule ($\leftarrow$ or a subscript $t+1$)?
2. **List every symbol** and look up any you do not know — most papers define them in the preceding paragraph or in a notation section. Note which are data, which are parameters, and which are dummy (bound) variables such as summation indices.
3. **Identify the big operators** ($\sum$, $\prod$, $\mathbb{E}$, argmin, ∇) and their scope: what they range over and what they apply to.
4. **Read it aloud in words**, from the outside in.
5. **Make it concrete**: plug in tiny numbers (two dimensions, three data points) and compute by hand.
6. **Sketch the code**: loops for sums, `max` / `argmax` for optimisation, matrix products for vectors.

Output: a plain-language explanation and an implementation sketch.

33.9.1. 1. The gradient: $\nabla f(\mathbf{x})$

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}) \quad \cdots \quad \frac{\partial f}{\partial x_n}(\mathbf{x}) \right]^{\mathsf{T}}$$

Symbol	Meaning
f	a scalar-valued function of several variables, for example a loss
$\mathbf{x}$	bold: a vector, the point $(x_1, \ldots, x_n)$ at which we evaluate
∇	"nabla" or "del": the gradient operator
$\partial f / \partial x_i$	partial derivative: rate of change of f when only x_i moves
$^\mathsf{T}$	transpose: write the row as a column

In words: "the gradient of f at $\mathbf{x}$" is the vector of partial derivatives. It points in the direction of steepest increase of f, its length is that steepest rate, and it is perpendicular to the contour through $\mathbf{x}$. Optimisation algorithms step in the direction $-\nabla f(\mathbf{x})$ to decrease f—which is why papers write update rules like $\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$, where the subscript on ∇_θ says "with respect to the parameters θ, not the data". **Concretely:** for $f(x, y) = x^2 + 3y$, $\nabla f = (2x, 3)$, and at $(1, 2)$ it is $(2, 3)$. **Code:** by hand, or `loss.backward()` and read `.grad`.

33.9.2. 2. Relevance as a dot product: $\mathbf{q} \cdot \mathbf{d}$

$$s(\mathbf{q}, \mathbf{d}) = \mathbf{q} \cdot \mathbf{d} = \mathbf{q}^\mathsf{T} \mathbf{d} = \sum_{j=1}^{k} q_j d_j$$

Symbol	Meaning
$s(\cdot, \cdot)$	a score function: two inputs, one number out
$\mathbf{q}, \mathbf{d}$	query and document vectors in $\mathbb{R}^k$ (TF-IDF or learned embeddings)
$\cdot$ or $\mathbf{q}^\mathsf{T}\mathbf{d}$	the dot product; the second form treats vectors as columns and uses a matrix product
q_j, d_j	the j-th components

In words: multiply matching components and add. Geometrically it is $\|\mathbf{q}\| \|\mathbf{d}\| \cos\theta$; if the paper says vectors are normalised, it *is* the cosine similarity. Ranking all documents means computing $D\mathbf{q}$ for a matrix D whose rows are documents, then sorting. **Code:** `scores = D @ q; top = np.argsort(-scores)[:10]`.

33.9.3. 3. Conditional probability and Bayes: $P(A \mid B)$

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B \mid A)\, P(A)}{P(B)}$$

Symbol	Meaning
$P(\cdot)$	probability of an event
$\mid$	"given": restrict attention to outcomes where the right-hand event happened
$A \cap B$	"A and B": both events happen
$P(A)$	the **prior**: belief in A before seeing evidence
$P(B \mid A)$	the **likelihood**: how probable the evidence is if A is true

In words: conditioning on B means shrinking the universe to the outcomes where B happened and re-normalising, which is why we divide by $P(B)$. Bayes' theorem follows from writing $P(A \cap B)$ two ways: $P(A \mid B)P(B) = P(B \mid A)P(A)$. **Concretely:** a disease with prevalence 1% and a test that is 99% sensitive with a 5% false-positive rate gives $P(\text{ill} \mid +) = \frac{0.99 \times 0.01}{0.99 \times 0.01 + 0.05 \times 0.99} = \frac{0.0099}{0.0594} \approx 0.167$ — only one positive in six is truly ill (Chapter 29). In ML papers, $p(y \mid \mathbf{x}; \theta)$ means "the model's probability of label y given input $\mathbf{x}$", with the semicolon separating the parameters θ from the conditioning.

33.9.4. 4. Training as optimisation: **argmin**

$$\hat{\theta} = \operatorname*{argmin}_{\theta \in \Theta} \; \frac{1}{n} \sum_{i=1}^{n} \ell\big(f_\theta(\mathbf{x}_i),\, y_i\big) + \lambda \, \|\theta\|_2^2$$

Symbol	Meaning
$\hat{\theta}$	"theta-hat": the *estimated* parameters (hats mark estimates)
$\operatorname*{argmin}_{\theta \in \Theta}$	the value of θ, ranging over the allowed set Θ, that minimises what follows
f_θ	the model, a function whose behaviour depends on θ
$(\mathbf{x}_i, y_i)$	the i-th training example: input and label
$\ell(\cdot, \cdot)$	the per-example loss, such as squared error or cross-entropy
$\lambda \|\theta\|_2^2$	regularisation: λ times the sum of squared parameters (the subscript 2 means the ordinary Euclidean norm)

In words: "the estimated parameters are those, among the allowed ones, that minimise the average training loss plus a penalty on large weights." That one line describes ridge regression, regularised logistic regression and (with a neural-network f_θ) most of deep learning. The formula says *what* is computed, not *how*: the how is the normal equations, gradient descent or Adam (Section 33.1). **Code:** the training loop.

33.9.5. 5. Expectation: $\mathbb{E}[X]$

$$\mathcal{L}(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\big[-\log p_\theta(y \mid \mathbf{x})\big]$$

Symbol	Meaning
$\mathcal{L}$	calligraphic L: a loss (papers use fancy fonts to distinguish objects)
$\mathbb{E}[\cdot]$	expected value: the probability-weighted average (Chapter 29)
$(\mathbf{x}, y) \sim \mathcal{D}$	"drawn from" the data distribution $\mathcal{D}$; the subscript says what is random
$-\log p_\theta(y \mid \mathbf{x})$	the surprise of the true label under the model: cross-entropy for one example

In words: "the loss is the average surprise of the true label, averaged over all the data the world could give us." We cannot compute an expectation over the whole world, so we estimate it with an average over the training set, $\frac{1}{n}\sum_i -\log p_\theta(y_i \mid \mathbf{x}_i)$, and SGD estimates *that* with a mini-batch. Recognising "expectation, estimated by an average" is half of reading ML theory. For a discrete random variable, $\mathbb{E}[X] = \sum_x x\,P(X = x)$; a fair die has $\mathbb{E}[X] = (1+2+\cdots+6)/6 = 3.5$.

33.9.6. 6. Softmax: turning scores into probabilities

$$\mathbf{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \qquad i = 1, \ldots, K$$

Try it: Softmax with a temperature .

Symbol	Meaning
$\mathbf{z}$	a vector of K real-valued scores ("logits"), one per class
$\mathbf{softmax}(\mathbf{z})_i$	the i-th component of the output vector
e^{z_i}	exponentiate: makes every score positive and exaggerates differences
$\sum_j e^{z_j}$	the normaliser, so the outputs sum to 1

In words: "exponentiate each score and divide by the total." The output is a probability distribution over K classes, with the largest score getting the largest probability — a "soft" version of picking the maximum. With $K = 2$ it reduces to the sigmoid: $\frac{e^{z_1}}{e^{z_1}+e^{z_2}} = \sigma(z_1 - z_2)$. Adding the same constant to every score changes nothing (it cancels top and bottom), which is the key to computing it safely:

```python
import math

def softmax(z):
    m = max(z)                          # subtract the max for numerical safety
    exps = [math.exp(zi - m) for zi in z]
    total = sum(exps)
    return [e / total for e in exps]

scores = [2.0, 1.0, 0.1]
print([round(p, 3) for p in softmax(scores)])
print([round(p, 3) for p in softmax([1000.0, 999.0, 998.1])])  # same gaps, no overflow
```

```
[0.659, 0.242, 0.099]
[0.659, 0.242, 0.099]
```

Without the subtraction, e^{1000} overflows to infinity (Volume 2, Chapter 32). By hand for the first case: $e^{2} \approx 7.389$, $e^{1} \approx 2.718$, $e^{0.1} \approx 1.105$, total ≈ 11.212, giving $7.389/11.212 \approx 0.659$, and so on.

33.9.7. 7. A final boss: attention

Here is the formula at the heart of the transformer architecture behind modern language models. You now have every piece: *Try it:* Attention, step by step .

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathsf{T}}}{\sqrt{d_k}}\right)V$$

- Q, K, V are matrices whose rows are **query**, **key** and **value** vectors — one per token, produced by linear layers.
- QK^{T} is a matrix of dot products: entry (i, j) is $\mathbf{q}_i \cdot \mathbf{k}_j$, "how relevant is token j to token i?" — exactly the retrieval score from Section 33.5.
- Dividing by $\sqrt{d_k}$ (the vector dimension) keeps the dot products from growing with dimension, so the softmax does not saturate and kill the gradients (the vanishing-gradient worry from Section 33.4).
- softmax is applied to each row, turning relevance scores into weights that sum to 1.
- Multiplying by V takes, for each token, a weighted average of the value vectors.

In words: "each token looks up every other token by dot-product similarity, and takes a probability-weighted average of what it finds." A search engine, a softmax and a matrix product — three ideas from this book — make up one line of the most influential formula in current computing.



Notation is a compression scheme

Much of the notation you have learned is surprisingly young. Leibniz introduced $\int$ and dx in the 1670s; Euler popularised $\sum$, $f(x)$, e and π in the 1700s; the nabla ∇ was named in the 1800s after a Hebrew harp of that shape; and blackboard-bold $\mathbb{E}$ for expectation became common only in the 20th century. Good notation is, in Shannon's sense, a compression scheme: it spends few symbols on the ideas that recur most often. That is why it looks dense — and why, once you know the code, it reads fast.



Formula decoder

Paste a LaTeX formula. Hover over any symbol to see its name, its meaning, and the chapter of this book that introduced it. Then switch to "concrete mode": the lab asks for tiny example values, evaluates the formula step by step, and shows equivalent NumPy code.

US curriculum

Function notation, summation notation and probability notation are introduced across **Algebra II**, **Precalculus** and **AP Statistics**; the gradient and partial-derivative notation belong to **multivariable calculus**, which some advanced students take after AP Calculus BC. Reading research papers is a university skill — but, as this section shows, it rests on high-school foundations.

33.10. Interactive tools

Browser tools on the companion website that go with this chapter. Each example link in the chapter opens one of them with that example already set up.

- Surfaces, gradients and gradient descent
- The maths of machine learning
- Statistics
- Graphing calculator
- Neural network designer
- ODE solver and phase portraits

33.11. Mastery check

1. Explain in words what $\nabla f(\mathbf{x})$ is, what direction it points, and how gradient descent uses it. For $f(x, y) = x^2 + xy$, compute $\nabla f(1, 2)$ and one gradient-descent step from $(1, 2)$ with $\eta = 0.1$.
2. Why is it significant that the least-squares and logistic-regression losses are convex, while neural-network losses are not?
3. Fit the least-squares line to $(0, 0), (1, 1), (2, 3)$ using the normal equations.
4. A logistic-regression model has $w = 2, b = -4$. What is the predicted probability at $x = 2$? At $x = 3$? Where is the decision boundary?
5. For one neuron $a = \sigma(wx + b)$ with loss $L = (a - y)^2$, write $\partial L / \partial b$ as a product of three factors from the chain rule.
6. Vectors $\mathbf{q} = (3, 4)$ and $\mathbf{d} = (4, 3)$: compute $\mathbf{q} \cdot \mathbf{d}$ and the cosine similarity.
7. Compute the entropy of a source with probabilities $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ and of $(\frac{1}{2}, \frac{1}{2}, 0, 0)$. Which is more compressible?
8. Decode, in words, $\hat{\theta} = \mathbf{argmin}_\theta \, \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}}[-\log \mathbf{softmax}(f_\theta(\mathbf{x}))_y]$.

Decision rule: All correct without hesitation → you have finished the book; choose a horizon from Section 33.8 and go. Stuck on 1–2 → reread the relevant section (Section 33.1, Section 33.2, Section 33.3, Section 33.4, Section 33.5, Section 33.6, or Section 33.9). Stuck on more → revisit the foundations this chapter leans on: Volume 2, Chapter 20 , Volume 2, Chapter 26 and Chapter 29.

▼ **Mastery check answers**

1. The gradient is the vector of partial derivatives of f at $\mathbf{x}$; it points in the direction of steepest increase, and gradient descent steps the opposite way, $\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x})$. Here $\nabla f = (2x + y,\ x)$, so $\nabla f(1, 2) = (4, 1)$, and the step gives $(1, 2) - 0.1(4, 1) = (0.6,\ 1.9)$.
2. For a convex loss every local minimum is global, so gradient descent (or the normal equations) is guaranteed to find the best parameters regardless of the starting point. For non-convex losses there may be many local minima and saddle points; the result depends on initialisation and the optimiser, and there is no guarantee of optimality.
3. $n = 3$, $\sum x = 3$, $\sum y = 4$, $\sum x^2 = 5$, $\sum xy = 0 + 1 + 6 = 7$. Equations: $4 = 3b + 3w$ and $7 = 3b + 5w$. Subtract: $3 = 2w$, so $w = 1.5$; then $3b = 4 - 4.5 = -0.5$, so $b = -1/6 \approx -0.167$. Line: $\hat{y} = 1.5x - 1/6$.
4. At $x = 2$: $z = 0$, $p = \sigma(0) = 0.5$. At $x = 3$: $z = 2$, $p = \sigma(2) \approx 0.881$. The boundary is where $z = 0$: $x = -b/w = 2$.
5. $\dfrac{\partial L}{\partial b} = \dfrac{\partial L}{\partial a} \cdot \dfrac{\partial a}{\partial z} \cdot \dfrac{\partial z}{\partial b} = 2(a - y) \cdot a(1 - a) \cdot 1$.
6. $\mathbf{q} \cdot \mathbf{d} = 12 + 12 = 24$; both norms are 5, so $\cos\theta = 24/25 = 0.96$.

7. Uniform over four: $\log_2 4 = 2$ bits. $(\frac{1}{2}, \frac{1}{2}, 0, 0)$: 1 bit (the zero-probability symbols contribute nothing). The second has lower entropy and is more compressible: 1 bit per symbol instead of 2.
8. "The estimated parameters are those that minimise the expected cross-entropy: over examples drawn from the data distribution, the model turns its output scores into class probabilities with a softmax, and we penalise the negative log-probability (the surprise) of the true class y." In practice the expectation is replaced by an average over training data and minimised with SGD; this one line is the training objective of essentially every multi-class classifier, including next-token prediction in language models.

Appendix A: Cheat Sheet

This is the "I remember learning this — remind me what it means" appendix. Every row is a reminder, not a lesson: if a formula looks unfamiliar, follow the chapter link and read the explanation in context. Definitions of terms live in the Glossary (Appendix B); symbols and how to read them aloud are in Appendix C.

Conventions used throughout: letters $a, b, c, \ldots$ are real numbers unless stated otherwise; denominators are assumed nonzero; logarithms and square roots are applied only where they are defined; $\ln$ is the natural logarithm (base e); angles in calculus formulas are in radians; C is an arbitrary constant of integration.

A.1. Arithmetic and fractions

Rule	Formula or statement	Ch.
Order of operations	Brackets, then powers, then $\times$/$\div$ left to right, then $+$/$-$ left to right. $2 + 3 \times 4^2 = 2 + 48 = 50$	Volume 1, Chapter 1
Signs	$(-a)(-b) = ab$, $(-a)b = -ab$, $a - (-b) = a + b$	Volume 1, Chapter 1
Division with remainder	$a = qb + r$ with $0 \le r < b$ (quotient q, remainder r)	Volume 1, Chapter 1
Absolute value	$\lvert x \rvert = x$ if $x \ge 0$, $-x$ if $x < 0$; distance between a and b is $\lvert a - b \rvert$	Volume 1, Chapter 1
Place value in base b	$(d_k \dots d_1 d_0)_b = \sum_{i=0}^{k} d_i \, b^i$, e.g. $1011_2 = 8 + 0 + 2 + 1 = 11$	Volume 1, Chapter 1
gcd and lcm	$\gcd(a,b) \cdot \operatorname{lcm}(a,b) = ab$ for positive integers	Volume 1, Chapter 1
Divisibility tests	By 3 or 9: digit sum divisible by 3 or 9. By 11: alternating digit sum divisible by 11. By 4: last two digits divisible by 4.	Volume 1, Chapter 1
Equivalent fractions	$\frac{a}{b} = \frac{ka}{kb}$ for $k \neq 0$	Volume 1, Chapter 2

Rule	Formula or statement	Ch.
Add / subtract fractions	$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd}$ (or use the least common denominator)	Volume 1, Chapter 2
Multiply fractions	$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$	Volume 1, Chapter 2
Divide fractions	$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \cdot \frac{d}{c} = \frac{ad}{bc}$	Volume 1, Chapter 2
Percent of a quantity	$p\%$ of x is $\frac{p}{100}\,x$	Volume 1, Chapter 2
Percentage change	$\frac{\text{new} - \text{old}}{\text{old}} \times 100\%$	Volume 1, Chapter 2
Successive changes	Multiply factors: up 10% then down 10% is $1.1 \times 0.9 = 0.99$, a 1% fall overall	Volume 1, Chapter 2
Ratio split	Sharing T in the ratio $a : b$ gives $\frac{a}{a+b}T$ and $\frac{b}{a+b}T$	Volume 1, Chapter 2
Repeating decimal to fraction	$0.\overline{d_1 \ldots d_k} = \frac{d_1 \ldots d_k}{10^k - 1}$, e.g. $0.\overline{27} = \frac{27}{99} = \frac{3}{11}$	Volume 1, Chapter 2

A.2. Exponents, roots, and logarithms

Rule	Formula	Ch.
Product of powers	$a^m a^n = a^{m+n}$	Volume 1, Chapter 3
Quotient of powers	$\frac{a^m}{a^n} = a^{m-n}$	Volume 1, Chapter 3
Power of a power	$(a^m)^n = a^{mn}$	Volume 1, Chapter 3
Power of a product / quotient	$(ab)^n = a^n b^n$, $\left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}$	Volume 1, Chapter 3
Zero and negative exponents	$a^0 = 1\ (a \neq 0)$, $a^{-n} = \frac{1}{a^n}$	Volume 1, Chapter 3
Fractional exponents	$a^{1/n} = \sqrt[n]{a}$, $a^{m/n} = \left(\sqrt[n]{a}\right)^m = \sqrt[n]{a^m}\ (a > 0)$	Volume 1, Chapter 3
Root rules	$\sqrt{ab} = \sqrt{a}\sqrt{b}$, $\sqrt{a/b} = \sqrt{a}/\sqrt{b}$ for $a, b \geq 0$; but $\sqrt{a+b} \neq \sqrt{a} + \sqrt{b}$	Volume 1, Chapter 3
Square root of a square	$\sqrt{x^2} = \lvert x \rvert$	Volume 1, Chapter 3

Rule	Formula	Ch.
Rationalising	$\dfrac{1}{\sqrt{a}} = \dfrac{\sqrt{a}}{a}$, $\dfrac{1}{\sqrt{a}+\sqrt{b}} = \dfrac{\sqrt{a}-\sqrt{b}}{a-b}$	Volume 1, Chapter 3
Scientific notation	$x = m \times 10^{n}$ with $1 \le \lvert m \rvert < 10$; multiply by multiplying mantissas and adding exponents	Volume 1, Chapter 3
Definition of logarithm	$\log_b x = y \iff b^y = x$ $(b>0, b\neq 1, x>0)$	Volume 1, Chapter 12
Product / quotient laws	$\log_b(xy) = \log_b x + \log_b y$, $\log_b(x/y) = \log_b x - \log_b y$	Volume 1, Chapter 12
Power law	$\log_b(x^k) = k \log_b x$	Volume 1, Chapter 12
Special values	$\log_b 1 = 0$, $\log_b b = 1$, $b^{\log_b x} = x$, $\log_b(b^x) = x$	Volume 1, Chapter 12
Change of base	$\log_b x = \dfrac{\ln x}{\ln b} = \dfrac{\log_{10} x}{\log_{10} b}$; e.g. $\log_2 x \approx 3.32 \log_{10} x$	Volume 1, Chapter 12
Exponential via e	$a^x = e^{x \ln a}$	Volume 1, Chapter 12

Rule	Formula	Ch.
Not a law	$\log(x+y) \neq \log x + \log y$; $(\log x)^2 \neq \log(x^2)$	Volume 1, Chapter 12
Exponential growth / decay	$N(t) = N_0 e^{kt}$ (growth $k>0$, decay $k<0$); half-life form $N_0\left(\frac{1}{2}\right)^{t/T}$	Volume 1, Chapter 12
Doubling time	$T_2 = \frac{\ln 2}{k}$; for $r\%$ per period, roughly $70/r$ periods ("rule of 70")	Volume 1, Chapter 12
Compound interest	$A = P\left(1 + \frac{r}{n}\right)^{nt}$; continuously compounded $A = Pe^{rt}$	Volume 1, Chapter 12
Defining e	$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n \approx 2.71828$	Volume 1, Chapter 12

A.3. Algebraic identities and polynomials

Identity	Formula	Ch.
Distributive law	$a(b+c) = ab + ac$	Volume 1, Chapter 4
Expanding two brackets	$(a+b)(c+d) = ac + ad + bc + bd$	Volume 1, Chapter 4
Square of a sum / difference	$(a \pm b)^2 = a^2 \pm 2ab + b^2$	Volume 1, Chapter 4
Difference of two squares	$a^2 - b^2 = (a+b)(a-b)$	Volume 1, Chapter 4
Cube of a sum / difference	$(a \pm b)^3 = a^3 \pm 3a^2b + 3ab^2 \pm b^3$	Volume 1, Chapter 7
Sum / difference of cubes	$a^3 + b^3 = (a+b)(a^2 - ab + b^2)$, $a^3 - b^3 = (a-b)(a^2 + ab + b^2)$	Volume 1, Chapter 7
Difference of nth powers	$a^n - b^n = (a-b)(a^{n-1} + a^{n-2}b + \cdots + ab^{n-2} + b^{n-1})$	Volume 1, Chapter 7
Binomial theorem	$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k}b^k$	Chapter 24

Identity	Formula	Ch.
Remainder theorem	Dividing $p(x)$ by $(x - r)$ leaves remainder $p(r)$	Volume 1, Chapter 7
Factor theorem	$(x - r)$ is a factor of $p(x) \iff p(r) = 0$	Volume 1, Chapter 7
Rational root theorem	If p/q (lowest terms) is a root of $a_n x^n + \cdots + a_0$ with integer coefficients, then p divides a_0 and q divides a_n	Volume 1, Chapter 7
Number of roots	A degree-n polynomial has at most n real roots, and exactly n complex roots counted with multiplicity	Volume 1, Chapter 7
End behaviour	Determined by the leading term $a_n x^n$: its sign and whether n is even or odd	Volume 1, Chapter 7
Simplifying rational expressions	Cancel common *factors*, never common *terms*: $\frac{x(x+1)}{x} = x + 1$ for $x \neq 0$, but $\frac{x+1}{x} \neq 1$	Volume 1, Chapter 4

A.4. Quadratics

Fact	Formula	Ch.
Quadratic formula	$ax^2+bx+c=0 \implies x=\dfrac{-b\pm\sqrt{b^2-4ac}}{2a}$ ($a\neq 0$)	Volume 1, Chapter 8
Discriminant	$\Delta=b^2-4ac$: $\Delta>0$ two real roots; $\Delta=0$ one repeated real root; $\Delta<0$ two complex-conjugate roots	Volume 1, Chapter 8
Completing the square	$x^2+bx=\left(x+\frac{b}{2}\right)^2-\frac{b^2}{4}$	Volume 1, Chapter 8
Vertex form	$ax^2+bx+c=a(x-h)^2+k$ with $h=-\dfrac{b}{2a}$, $k=c-\dfrac{b^2}{4a}$; vertex (h,k), axis of symmetry $x=h$	Volume 1, Chapter 8
Factored form	$a(x-r_1)(x-r_2)$ where r_1, r_2 are the roots	Volume 1, Chapter 8
Sum and product of roots (Vieta)	$r_1+r_2=-\dfrac{b}{a}$, $r_1 r_2=\dfrac{c}{a}$	Volume 1, Chapter 8
Numerically stable roots	$q=-\frac{1}{2}\left(b+\operatorname{sign}(b)\sqrt{\Delta}\right)$, then $x_1=q/a$, $x_2=c/q$ (avoids cancellation when $b^2\gg 4ac$)	Volume 1, Chapter 8

A.5. Linear equations, lines, and coordinates

Fact	Formula	Ch.
Inequalities	Multiplying or dividing both sides by a negative number reverses the inequality: $-2x < 6 \iff x > -3$	Volume 1, Chapter 5
Absolute-value inequality	$\lvert x - a \rvert < r \iff a - r < x < a + r$	Volume 1, Chapter 5
Slope	$m = \dfrac{y_2 - y_1}{x_2 - x_1} = \dfrac{\text{rise}}{\text{run}}$	Volume 1, Chapter 5
Slope-intercept form	$y = mx + b$ (slope m, y-intercept b)	Volume 1, Chapter 5
Point-slope form	$y - y_1 = m(x - x_1)$	Volume 1, Chapter 5
Standard form	$Ax + By = C$; slope $-A/B$	Volume 1, Chapter 5
Parallel / perpendicular	Parallel: $m_1 = m_2$. Perpendicular: $m_1 m_2 = -1$	Volume 1, Chapter 5
Two equations, two unknowns (Cramer)	$ax + by = e,\ cx + dy = f \implies x = \dfrac{ed - bf}{ad - bc},\ y = \dfrac{af - ec}{ad - bc}$ (if $ad - bc \neq 0$)	Volume 1, Chapter 6

Fact	Formula	Ch.
Number of solutions of a linear system	Exactly one, none (inconsistent), or infinitely many; never exactly two	Volume 1, Chapter 6
Distance	$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$	Volume 1, Chapter 10
Midpoint	$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$	Volume 1, Chapter 10
Circle	Centre (h, k), radius r: $(x - h)^2 + (y - k)^2 = r^2$	Volume 1, Chapter 10
Point-to-line distance	From (x_0, y_0) to $Ax + By + C = 0$: $\frac{\lvert Ax_0 + By_0 + C\rvert}{\sqrt{A^2 + B^2}}$	Volume 1, Chapter 10
Transformations of graphs	$y = f(x - h) + k$ shifts right h, up k; $y = af(x)$ stretches vertically by a; $y = f(bx)$ compresses horizontally by b; $y = -f(x)$ reflects in the x-axis; $y = f(-x)$ in the y-axis	Volume 1, Chapter 11
Composition and inverse	$(f \circ g)(x) = f(g(x))$; $f^{-1}(f(x)) = x$; graph of f^{-1} is the reflection of the graph of f in $y = x$	Volume 1, Chapter 11

A.6. Geometry

Shape or fact	Formula	Ch.
Pythagoras	Right triangle with legs a, b, hypotenuse c: $a^2 + b^2 = c^2$. Triples: 3-4-5, 5-12-13, 8-15-17	Volume 1, Chapter 9
Angle sums	Triangle $180°$; n-sided polygon $(n-2) \times 180°$; exterior angles of a convex polygon total $360°$	Volume 1, Chapter 9
Special right triangles	45-45-90: sides $1 : 1 : \sqrt{2}$. 30-60-90: sides $1 : \sqrt{3} : 2$	Volume 1, Chapter 9
Rectangle / parallelogram	Area lw; parallelogram bh (h perpendicular height)	Volume 1, Chapter 9
Triangle	Area $\frac{1}{2}bh = \frac{1}{2}ab\sin C$; Heron: $\sqrt{s(s-a)(s-b)(s-c)}$ with $s = \frac{a+b+c}{2}$	Volume 1, Chapter 9
Trapezoid (UK: trapezium)	Area $\frac{1}{2}(a+b)h$, a, b the parallel sides	Volume 1, Chapter 9
Circle	Circumference $2\pi r = \pi d$; area πr^2	Volume 1, Chapter 9
Arc and sector (radians)	Arc length $s = r\theta$; sector area $\frac{1}{2}r^2\theta$	Volume 1, Chapter 9

Shape or fact	Formula	Ch.
Degrees and radians	$\pi \text{ rad} = 180°$; $1 \text{ rad} \approx 57.3°$	Volume 1, Chapter 13
Prism / cylinder	Volume Bh (B = base area); cylinder $\pi r^2 h$, surface area $2\pi r^2 + 2\pi r h$	Volume 1, Chapter 9
Pyramid / cone	Volume $\frac{1}{3}Bh$; cone $\frac{1}{3}\pi r^2 h$	Volume 1, Chapter 9
Sphere	Volume $\frac{4}{3}\pi r^3$; surface area $4\pi r^2$	Volume 1, Chapter 9
Similar figures	Scale lengths by k: areas scale by k^2, volumes by k^3	Volume 1, Chapter 9
Circle theorems	Angle in a semicircle is $90°$; inscribed angle is half the central angle on the same arc; tangent is perpendicular to the radius	Volume 1, Chapter 9

A.7. Trigonometry

Fact	Formula	Ch.
SOH-CAH-TOA	$\sin\theta = \frac{\text{opp}}{\text{hyp}}, \cos\theta = \frac{\text{adj}}{\text{hyp}}, \tan\theta = \frac{\text{opp}}{\text{adj}} = \frac{\sin\theta}{\cos\theta}$	Volume 1, Chapter 13
Reciprocal functions	$\sec\theta = \frac{1}{\cos\theta}, \csc\theta = \frac{1}{\sin\theta}, \cot\theta = \frac{1}{\tan\theta}$	Volume 1, Chapter 13
Unit circle	The point at angle θ is $(\cos\theta, \sin\theta)$	Volume 1, Chapter 13
Signs by quadrant ("All Students Take Calculus")	Q1 all positive; Q2 only $\sin$; Q3 only $\tan$; Q4 only $\cos$	Volume 1, Chapter 13
Pythagorean identities	$\sin^2\theta + \cos^2\theta = 1, 1 + \tan^2\theta = \sec^2\theta,$ $1 + \cot^2\theta = \csc^2\theta$	Volume 1, Chapter 13
Symmetry	$\sin(-\theta) = -\sin\theta, \cos(-\theta) = \cos\theta,$ $\tan(-\theta) = -\tan\theta$	Volume 1, Chapter 13
Co-function	$\sin\left(\frac{\pi}{2} - \theta\right) = \cos\theta, \cos\left(\frac{\pi}{2} - \theta\right) = \sin\theta$	Volume 1, Chapter 13
Periodicity	$\sin$ and $\cos$ have period 2π; $\tan$ has period π; $A\sin(bx + c) + d$ has amplitude $\lvert A\rvert$, period $2\pi/\lvert b\rvert$	Volume 1, Chapter 13

Fact	Formula	Ch.
Angle addition	$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$; $\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$	Volume 1, Chapter 13
Tangent addition	$\tan(\alpha \pm \beta) = \dfrac{\tan\alpha \pm \tan\beta}{1 \mp \tan\alpha\tan\beta}$	Volume 1, Chapter 13
Double angle	$\sin 2\theta = 2\sin\theta\cos\theta$; $\cos 2\theta = \cos^2\theta - \sin^2\theta = 2\cos^2\theta - 1 = 1 - 2\sin^2\theta$; $\tan 2\theta = \dfrac{2\tan\theta}{1 - \tan^2\theta}$	Volume 1, Chapter 13
Power reduction	$\sin^2\theta = \frac{1}{2}(1 - \cos 2\theta)$, $\cos^2\theta = \frac{1}{2}(1 + \cos 2\theta)$	Volume 1, Chapter 13
Law of sines	$\dfrac{a}{\sin A} = \dfrac{b}{\sin B} = \dfrac{c}{\sin C}$ ($= 2R$, R the circumradius)	Volume 1, Chapter 13
Law of cosines	$c^2 = a^2 + b^2 - 2ab\cos C$	Volume 1, Chapter 13
Inverse function ranges	$\arcsin : [-\frac{\pi}{2}, \frac{\pi}{2}]$, $\arccos : [0, \pi]$, $\arctan : (-\frac{\pi}{2}, \frac{\pi}{2})$	Volume 1, Chapter 13
Polar coordinates	$x = r\cos\theta$, $y = r\sin\theta$; $r = \sqrt{x^2 + y^2}$, $\theta = \operatorname{atan2}(y, x)$	Volume 1, Chapter 13

Fact	Formula	Ch.
Rotation by θ	$(x, y) \mapsto (x\cos\theta - y\sin\theta,\ x\sin\theta + y\cos\theta)$	Volume 1, Chapter 13

Table 3. Exact values on the unit circle

Degrees	Radians	sin	cos	tan
$0°$	0	0	1	0
$30°$	$\frac{\pi}{6}$	$\frac{1}{2}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{3}} = \frac{\sqrt{3}}{3}$
$45°$	$\frac{\pi}{4}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{2}}{2}$	1
$60°$	$\frac{\pi}{3}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{2}$	$\sqrt{3}$
$90°$	$\frac{\pi}{2}$	1	0	undefined
$120°$	$\frac{2\pi}{3}$	$\frac{\sqrt{3}}{2}$	$-\frac{1}{2}$	$-\sqrt{3}$
$135°$	$\frac{3\pi}{4}$	$\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{2}}{2}$	-1
$150°$	$\frac{5\pi}{6}$	$\frac{1}{2}$	$-\frac{\sqrt{3}}{2}$	$-\frac{\sqrt{3}}{3}$
$180°$	π	0	-1	0
$270°$	$\frac{3\pi}{2}$	-1	0	undefined
$360°$	2π	0	1	0

Memory aid for the first quadrant: $\sin$ at $0°, 30°, 45°, 60°, 90°$ is $\frac{\sqrt{0}}{2}, \frac{\sqrt{1}}{2}, \frac{\sqrt{2}}{2}, \frac{\sqrt{3}}{2}, \frac{\sqrt{4}}{2}$; $\cos$ is the same list backwards. See Volume 1, Chapter 13 .

A.8. Sequences and series

Fact	Formula	Ch.
Arithmetic sequence	$a_n = a_1 + (n-1)d$; sum $S_n = \dfrac{n(a_1 + a_n)}{2} = \dfrac{n}{2}(2a_1 + (n-1)d)$	Volume 1, Chapter 14
Geometric sequence	$a_n = a_1 r^{n-1}$; sum $S_n = a_1 \dfrac{1 - r^n}{1 - r}$ $(r \neq 1)$	Volume 1, Chapter 14
Infinite geometric series	$\sum_{n=0}^{\infty} ar^n = \dfrac{a}{1-r}$ if $\lvert r \rvert < 1$; diverges otherwise (for $a \neq 0$)	Volume 1, Chapter 14
Sum of first n integers	$\sum_{k=1}^{n} k = \dfrac{n(n+1)}{2}$	Volume 1, Chapter 14
Sum of squares / cubes	$\sum_{k=1}^{n} k^2 = \dfrac{n(n+1)(2n+1)}{6}$, $\sum_{k=1}^{n} k^3 = \left(\dfrac{n(n+1)}{2}\right)^2$	Volume 1, Chapter 14
Powers of two	$\sum_{k=0}^{n-1} 2^k = 2^n - 1$	Volume 1, Chapter 14
Harmonic numbers	$H_n = \sum_{k=1}^{n} \dfrac{1}{k} \approx \ln n + 0.5772$; the harmonic series diverges	Volume 1, Chapter 14
p-series	$\sum_{n=1}^{\infty} \dfrac{1}{n^p}$ converges iff $p > 1$; $\sum \dfrac{1}{n^2} = \dfrac{\pi^2}{6}$	Volume 1, Chapter 14

Fact	Formula	Ch.
Sigma rules	$\sum(a_k + b_k) = \sum a_k + \sum b_k$, $\sum c\,a_k = c\sum a_k$, $\sum_{k=1}^{n} c = nc$	Volume 1, Chapter 14
Recurrence (linear, first order)	$a_n = ra_{n-1} + d$ has fixed point $L = \dfrac{d}{1-r}$ and $a_n = L + r^n(a_0 - L)$ $(r \neq 1)$	Volume 1, Chapter 14
Taylor polynomial	$T_n(x) = \sum_{k=0}^{n} \dfrac{f^{(k)}(a)}{k!}(x-a)^k$	Volume 2, Chapter 18
Standard series	$e^x = \sum_{n\geq 0} \dfrac{x^n}{n!}$; $\sin x = x - \dfrac{x^3}{3!} + \dfrac{x^5}{5!} - \cdots$; $\cos x = 1 - \dfrac{x^2}{2!} + \dfrac{x^4}{4!} - \cdots$ (all x)	Volume 2, Chapter 18
More standard series	$\dfrac{1}{1-x} = \sum_{n\geq 0} x^n$ $(\lvert x\rvert < 1)$; $\ln(1+x) = x - \dfrac{x^2}{2} + \dfrac{x^3}{3} - \cdots$ $(-1 < x \leq 1)$	Volume 2, Chapter 18

A.9. Complex numbers

Fact	Formula	Ch.
Imaginary unit	$i^2 = -1$; powers cycle $i, -1, -i, 1$	Volume 1, Chapter 15
Arithmetic	$(a+bi) \pm (c+di) = (a \pm c) + (b \pm d)i$; $(a+bi)(c+di) = (ac - bd) + (ad + bc)i$	Volume 1, Chapter 15
Conjugate	$\overline{a+bi} = a - bi$; $z\bar{z} = a^2 + b^2$; $\overline{zw} = \bar{z}\,\bar{w}$	Volume 1, Chapter 15
Division	$\dfrac{a+bi}{c+di} = \dfrac{(a+bi)(c-di)}{c^2+d^2} = \dfrac{(ac+bd)+(bc-ad)i}{c^2+d^2}$	Volume 1, Chapter 15
Modulus and argument	$\lvert z\rvert = \sqrt{a^2+b^2}$; $\arg z = \operatorname{atan2}(b, a)$	Volume 1, Chapter 15
Polar / exponential form	$z = r(\cos\theta + i\sin\theta) = re^{i\theta}$	Volume 1, Chapter 15
Euler's formula	$e^{i\theta} = \cos\theta + i\sin\theta$; hence $e^{i\pi} + 1 = 0$	Volume 1, Chapter 15
Multiplying in polar form	Moduli multiply, arguments add: $r_1 e^{i\theta_1} \cdot r_2 e^{i\theta_2} = r_1 r_2\, e^{i(\theta_1+\theta_2)}$	Volume 1, Chapter 15

Fact	Formula	Ch.
De Moivre	$(\cos\theta + i\sin\theta)^n = \cos n\theta + i\sin n\theta$	Volume 1, Chapter 15
Roots of unity	Solutions of $z^n = 1$: $z_k = e^{2\pi ik/n}$, $k = 0, 1, \ldots, n-1$	Volume 1, Chapter 15
Real polynomials	Non-real roots come in conjugate pairs $a \pm bi$	Volume 1, Chapter 15

A.10. Limits and continuity

Fact	Formula	Ch.
Limit laws	Limits of sums, differences, products, constant multiples and quotients are the sums (etc.) of the limits, provided those exist and the denominator's limit is nonzero	Volume 2, Chapter 16
Continuity at a	$\lim_{x\to a} f(x) = f(a)$	Volume 2, Chapter 16
Formal definition	$\lim_{x\to a} f(x) = L$ means: for every $\varepsilon > 0$ there is $\delta > 0$ with $0 < \lvert x-a\rvert < \delta \implies \lvert f(x)-L\rvert < \varepsilon$	Volume 2, Chapter 16
Key trigonometric limits	$\lim_{x\to 0} \dfrac{\sin x}{x} = 1$, $\lim_{x\to 0} \dfrac{1-\cos x}{x} = 0$	Volume 2, Chapter 16
Limits defining e	$\lim_{n\to\infty} \left(1+\dfrac{1}{n}\right)^n = e$, $\lim_{x\to 0} \dfrac{e^x-1}{x} = 1$	Volume 2, Chapter 16
Growth hierarchy as $x \to \infty$	$\ln x \ll x^p \ll a^x \ll x!$ (any $p > 0$, $a > 1$); e.g. $\dfrac{x^{100}}{e^x} \to 0$	Volume 2, Chapter 16
Rational functions at ∞	Compare degrees: top lower $\to 0$; equal $\to$ ratio of leading coefficients; top higher $\to \pm\infty$	Volume 2, Chapter 16
Squeeze theorem	If $g \le f \le h$ near a and $g, h \to L$, then $f \to L$	Volume 2, Chapter 16

Fact	Formula	Ch.
L’Hôpital’s rule	For $\frac{0}{0}$ or $\frac{\infty}{\infty}$ forms: $\lim \frac{f(x)}{g(x)} = \lim \frac{f'(x)}{g'(x)}$ if the right side exists	Volume 2, Chapter 18
Intermediate value theorem	If f is continuous on $[a, b]$ and y lies between $f(a)$ and $f(b)$, some c in $[a, b]$ has $f(c) = y$	Volume 2, Chapter 16

A.11. Derivatives

Rule	Formula	Ch.
Definition	$f'(x) = \lim_{h \to 0} \dfrac{f(x+h) - f(x)}{h}$	Volume 2, Chapter 17
Constant / linearity	$(c)' = 0$; $(af + bg)' = af' + bg'$	Volume 2, Chapter 17
Power rule	$\dfrac{d}{dx} x^n = n x^{n-1}$ (any real n)	Volume 2, Chapter 17
Product rule	$(fg)' = f'g + fg'$	Volume 2, Chapter 17
Quotient rule	$\left(\dfrac{f}{g}\right)' = \dfrac{f'g - fg'}{g^2}$	Volume 2, Chapter 17
Chain rule	$\dfrac{d}{dx} f(g(x)) = f'(g(x))\, g'(x)$; in Leibniz form $\dfrac{dy}{dx} = \dfrac{dy}{du}\dfrac{du}{dx}$	Volume 2, Chapter 17
Exponentials	$(e^x)' = e^x$; $(a^x)' = a^x \ln a$	Volume 2, Chapter 17
Logarithms	$(\ln x)' = \dfrac{1}{x}$; $(\log_a x)' = \dfrac{1}{x \ln a}$; $(\ln \lvert x \rvert)' = \dfrac{1}{x}$	Volume 2, Chapter 17

Rule	Formula	Ch.
Trigonometric	$(\sin x)' = \cos x$; $(\cos x)' = -\sin x$; $(\tan x)' = \sec^2 x$	Volume 2, Chapter 17
Inverse trigonometric	$(\arcsin x)' = \dfrac{1}{\sqrt{1-x^2}}$; $(\arccos x)' = -\dfrac{1}{\sqrt{1-x^2}}$; $(\arctan x)' = \dfrac{1}{1+x^2}$	Volume 2, Chapter 17
Logistic sigmoid	$\sigma(x) = \dfrac{1}{1+e^{-x}}$, $\sigma'(x) = \sigma(x)\,(1-\sigma(x))$	Chapter 33
Implicit differentiation	Differentiate both sides with respect to x, treating y as $y(x)$: e.g. $x^2 + y^2 = 1 \implies 2x + 2yy' = 0 \implies y' = -x/y$	Volume 2, Chapter 17
Tangent line / linearisation	$y = f(a) + f'(a)(x-a)$; so $f(a+h) \approx f(a) + hf'(a)$	Volume 2, Chapter 18
Critical points and tests	Critical point: $f'(c) = 0$ or undefined. Second-derivative test: $f''(c) > 0$ local min, $f''(c) < 0$ local max, $= 0$ inconclusive	Volume 2, Chapter 18
Closed-interval extrema	On $[a, b]$ check critical points and both endpoints; the largest/smallest value wins	Volume 2, Chapter 18
Concavity / inflection	$f'' > 0$ concave up, $f'' < 0$ concave down; inflection where f'' changes sign	Volume 2, Chapter 18
Mean value theorem	f continuous on $[a, b]$, differentiable inside $\implies f'(c) = \dfrac{f(b) - f(a)}{b - a}$ for some $c \in (a, b)$	Volume 2, Chapter 18

Rule	Formula	Ch.
Newton’s method	$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$	Volume 2, Chapter 18

A.12. Integrals

Rule	Formula	Ch.
Power rule	$\int x^n\,dx = \frac{x^{n+1}}{n+1} + C\,(n \neq -1)$	Volume 2, Chapter 19
Reciprocal	$\int \frac{1}{x}\,dx = \ln\lvert x\rvert + C$	Volume 2, Chapter 19
Exponentials	$\int e^x\,dx = e^x + C;\int e^{kx}\,dx = \frac{1}{k}e^{kx} + C;$ $\int a^x\,dx = \frac{a^x}{\ln a} + C$	Volume 2, Chapter 19
Trigonometric	$\int \sin x\,dx = -\cos x + C;\int \cos x\,dx = \sin x + C;$ $\int \sec^2 x\,dx = \tan x + C$	Volume 2, Chapter 19
Inverse-trig forms	$\int \frac{dx}{1+x^2} = \arctan x + C;$ $\int \frac{dx}{\sqrt{1-x^2}} = \arcsin x + C$	Volume 2, Chapter 19
Logarithm	$\int \ln x\,dx = x\ln x - x + C$	Volume 2, Chapter 19
Linearity	$\int (af + bg)\,dx = a\int f\,dx + b\int g\,dx$	Volume 2, Chapter 19
Fundamental theorem (part 1)	$\frac{d}{dx}\int_a^x f(t)\,dt = f(x)$	Volume 2, Chapter 19

Rule	Formula	Ch.
Fundamental theorem (part 2)	$\int_a^b f(x)\,dx = F(b) - F(a)$ where $F' = f$	Volume 2, Chapter 19
Definite-integral properties	$\int_a^a f = 0$; $\int_b^a f = -\int_a^b f$; $\int_a^b f + \int_b^c f = \int_a^c f$	Volume 2, Chapter 19
Substitution	$\int f(g(x))\,g'(x)\,dx = \int f(u)\,du$ with $u = g(x)$; for definite integrals change the limits to $g(a), g(b)$	Volume 2, Chapter 19
Integration by parts	$\int u\,dv = uv - \int v\,du$	Volume 2, Chapter 19
Riemann sum	$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\,\Delta x$, $\Delta x = \dfrac{b-a}{n}$	Volume 2, Chapter 19
Area between curves	$\int_a^b (f(x) - g(x))dx$ where $f \geq g$ on $[a, b]$	Volume 2, Chapter 19
Average value	$\bar{f} = \dfrac{1}{b-a}\int_a^b f(x)\,dx$	Volume 2, Chapter 19
Volume of revolution (discs)	$V = \pi \int_a^b f(x)^2\,dx$ about the x-axis	Volume 2, Chapter 19
Gaussian integral	$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}$	Chapter 30

A.13. Multivariable calculus

Fact	Formula	Ch.
Partial derivative	$\frac{\partial f}{\partial x}$: differentiate in x holding the other variables fixed. Example: $f = x^2 y \implies f_x = 2xy,\ f_y = x^2$	Volume 2, Chapter 20
Gradient	$\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$; points in the direction of steepest ascent; its length is that steepest rate	Volume 2, Chapter 20
Directional derivative	$D_{\mathbf{u}} f = \nabla f \cdot \mathbf{u}$ for a unit vector $\mathbf{u}$	Volume 2, Chapter 20
Gradient descent	$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$ (η the learning rate / step size)	Volume 2, Chapter 20
Tangent plane	$z = f(a,b) + f_x(a,b)(x-a) + f_y(a,b)(y-b)$	Volume 2, Chapter 20
Multivariable chain rule	$\frac{d}{dt} f(x(t), y(t)) = f_x \frac{dx}{dt} + f_y \frac{dy}{dt} = \nabla f \cdot \mathbf{r}'(t)$	Volume 2, Chapter 20
Critical points and second-derivative test	$\nabla f = \mathbf{0}$; with $D = f_{xx} f_{yy} - f_{xy}^2$: $D > 0, f_{xx} > 0$ min; $D > 0, f_{xx} < 0$ max; $D < 0$ saddle; $D = 0$ inconclusive	Volume 2, Chapter 20
Hessian	$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$; symmetric for smooth f	Volume 2, Chapter 20

Fact	Formula	Ch.
Lagrange multipliers	Optimise f subject to $g = c$: solve $\nabla f = \lambda \nabla g$, $g = c$	Volume 2, Chapter 20

A.14. Logic

Law	Equivalence	Ch.
Conditional as "or"	$p \to q \equiv \neg p \lor q$; false only when p is true and q false	Chapter 21
Contrapositive (equivalent)	$p \to q \equiv \neg q \to \neg p$	Chapter 21
Converse and inverse (not equivalent to $p \to q$)	Converse $q \to p$; inverse $\neg p \to \neg q$ (these two are equivalent to each other)	Chapter 21
Negating a conditional	$\neg(p \to q) \equiv p \land \neg q$	Chapter 21
Biconditional	$p \leftrightarrow q \equiv (p \to q) \land (q \to p)$	Chapter 21
De Morgan	$\neg(p \land q) \equiv \neg p \lor \neg q$; $\neg(p \lor q) \equiv \neg p \land \neg q$	Chapter 21
Double negation	$\neg\neg p \equiv p$	Chapter 21
Distributive	$p \land (q \lor r) \equiv (p \land q) \lor (p \land r)$; $p \lor (q \land r) \equiv (p \lor q) \land (p \lor r)$	Chapter 21
Absorption	$p \lor (p \land q) \equiv p$; $p \land (p \lor q) \equiv p$	Chapter 21
Excluded middle / contradiction	$p \lor \neg p \equiv \mathbf{T}$; $p \land \neg p \equiv \mathbf{F}$	Chapter 21
Exclusive or	$p \oplus q \equiv (p \lor q) \land \neg(p \land q)$	Chapter 21
Negating quantifiers	$\neg\forall x\, P(x) \equiv \exists x\, \neg P(x)$; $\neg\exists x\, P(x) \equiv \forall x\, \neg P(x)$	Chapter 21
Quantifier order matters	$\forall x\, \exists y\, P(x, y)$ is generally *not* equivalent to $\exists y \forall x\, P(x, y)$	Chapter 21

A.15. Sets

Identity	Formula	Ch.
Basic operations	$A \cup B$ (in either), $A \cap B$ (in both), $A \setminus B$ (in A not B), A^c (not in A)	Chapter 21
Difference via complement	$A \setminus B = A \cap B^c$	Chapter 21
De Morgan for sets	$(A \cup B)^c = A^c \cap B^c$; $(A \cap B)^c = A^c \cup B^c$	Chapter 21
Distributive	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$; $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	Chapter 21
Subset tests	$A \subseteq B \iff A \cap B = A \iff A \cup B = B$	Chapter 21
Inclusion-exclusion (two sets)	$\lvert A \cup B \rvert = \lvert A \rvert + \lvert B \rvert - \lvert A \cap B \rvert$	Chapter 24
Inclusion-exclusion (three sets)	$\lvert A \cup B \cup C \rvert = \lvert A \rvert + \lvert B \rvert + \lvert C \rvert - \lvert A \cap B \rvert - \lvert A \cap C \rvert - \lvert B \cap C \rvert + \lvert A \cap B \cap C \rvert$	Chapter 24
Power set	$\lvert \mathcal{P}(A) \rvert = 2^{\lvert A \rvert}$	Chapter 21
Cartesian product	$\lvert A \times B \rvert = \lvert A \rvert \cdot \lvert B \rvert$	Chapter 21

A.16. Proof techniques

Technique	To prove … do this	Ch.
Direct proof	$p \to q$: assume p, deduce q step by step	Chapter 22
Contrapositive	$p \to q$: assume $\neg q$, deduce $\neg p$	Chapter 22
Contradiction	p: assume $\neg p$, derive something impossible	Chapter 22
Cases	Split into exhaustive cases and prove each	Chapter 22
Counterexample	To *disprove* $\forall x\, P(x)$, exhibit one x with $\neg P(x)$	Chapter 22
Induction	$P(n)$ for all $n \geq n_0$: prove $P(n_0)$ (base case), then $P(k) \implies P(k+1)$ (inductive step)	Chapter 22
Strong induction	Inductive step assumes $P(n_0), \ldots, P(k)$ all hold, then proves $P(k+1)$	Chapter 22
Loop invariant	Induction applied to code: true before the loop, preserved by each iteration, so true at exit	Chapter 22

A.17. Counting

Situation	Count	Ch.
Sum rule	Choose one of m options *or* one of n (disjoint): $m + n$	Chapter 24
Product rule	Make a choice with m options *then* one with n: mn	Chapter 24
Ordered, repetition allowed	Sequences of length r from n symbols: n^r	Chapter 24
Arrangements of n distinct items	$n! = n(n-1)\cdots 2 \cdot 1$, with $0! = 1$	Chapter 24
Permutations (ordered, no repetition)	$P(n, r) = \dfrac{n!}{(n-r)!}$	Chapter 24
Combinations (unordered, no repetition)	$\dbinom{n}{r} = \dfrac{n!}{r!\,(n-r)!}$	Chapter 24
Multisets / stars and bars (unordered, repetition allowed)	Choose r from n types: $\dbinom{n+r-1}{r}$	Chapter 24
Arrangements with repeated items	$\dfrac{n!}{n_1!\,n_2!\cdots n_k!}$ (e.g. MISSISSIPPI: $\frac{11!}{1!\,4!\,4!\,2!} = 34650$)	Chapter 24
Pascal's rule and symmetry	$\dbinom{n}{k} = \dbinom{n-1}{k-1} + \dbinom{n-1}{k}$; $\dbinom{n}{k} = \dbinom{n}{n-k}$	Chapter 24
Row sum	$\sum_{k=0}^{n} \dbinom{n}{k} = 2^n$ (the number of subsets of an n-set)	Chapter 24
Derangements	Permutations with no fixed point: $D_n = n!\sum_{k=0}^{n} \dfrac{(-1)^k}{k!} \approx \dfrac{n!}{e}$	Chapter 24
Pigeonhole principle	n items in $m < n$ boxes: some box holds at least $\lceil n/m \rceil$ items	Chapter 24

A.18. Graphs

Fact	Formula	Ch.
Handshake lemma	$\sum_v \deg(v) = 2\lvert E\rvert$; so the number of odd-degree vertices is even	Chapter 25
Complete graph	K_n has $\binom{n}{2} = \frac{n(n-1)}{2}$ edges	Chapter 25
Trees	A tree on n vertices is connected, acyclic, and has exactly $n - 1$ edges	Chapter 25
Euler circuit	Exists in a connected graph iff every vertex has even degree (path: exactly 0 or 2 odd vertices)	Chapter 25
Euler's formula (planar, connected)	$V - E + F = 2$	Chapter 25
Traversal cost	BFS and DFS run in $O(V + E)$ with adjacency lists	Chapter 25

A.19. Modular arithmetic and number theory

Fact	Formula	Ch.
Congruence	$a \equiv b \pmod{m} \iff m \mid (a-b)$ (same remainder on division by m)	Chapter 23
Arithmetic respects congruence	If $a \equiv b$ and $c \equiv d \pmod{m}$ then $a \pm c \equiv b \pm d$ and $ac \equiv bd \pmod{m}$	Chapter 23
Reduce early	$(a \cdot b) \bmod m = ((a \bmod m)(b \bmod m)) \bmod m$; same for $+$	Chapter 23
Euclidean algorithm	$\gcd(a, b) = \gcd(b, a \bmod b)$, $\gcd(a, 0) = a$	Chapter 23
Bézout's identity	$\gcd(a, b) = ax + by$ for some integers x, y (found by the extended Euclidean algorithm)	Chapter 23
Modular inverse	$a^{-1} \bmod m$ exists iff $\gcd(a, m) = 1$	Chapter 23
Fundamental theorem of arithmetic	Every integer $n \geq 2$ factors into primes uniquely (up to order)	Chapter 23
Fermat's little theorem	p prime, $p \nmid a \implies a^{p-1} \equiv 1 \pmod{p}$	Chapter 23
Euler's totient	$\varphi(n)$ = number of $1 \leq k \leq n$ coprime to n; $\varphi(p) = p - 1$; $\varphi(pq) = (p-1)(q-1)$ for distinct primes	Chapter 23
Euler's theorem	$\gcd(a, n) = 1 \implies a^{\varphi(n)} \equiv 1 \pmod{n}$	Chapter 23
Chinese remainder theorem	If $m_1, \ldots, m_k$ are pairwise coprime, the system $x \equiv a_i \pmod{m_i}$ has a unique solution modulo $m_1 m_2 \cdots m_k$	Chapter 23
RSA	$n = pq$; pick e coprime to $\varphi(n)$; $d \equiv e^{-1} \pmod{\varphi(n)}$. Encrypt $c = m^e \bmod n$; decrypt $m = c^d \bmod n$	Chapter 23
Fast exponentiation	Square-and-multiply computes $a^k \bmod m$ in $O(\log k)$ multiplications	Chapter 23

A.20. Vectors and matrices

Fact	Formula	Ch.
Length (norm)	$\|\mathbf{v}\| = \sqrt{v_1^2 + \cdots + v_n^2}$; unit vector $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$	Volume 2, Chapter 26
Dot product	$\mathbf{u}\cdot\mathbf{v} = \sum_i u_i v_i = \|\mathbf{u}\|\|\mathbf{v}\|\cos\theta$	Volume 2, Chapter 26
Orthogonality	$\mathbf{u}\perp\mathbf{v} \iff \mathbf{u}\cdot\mathbf{v} = 0$	Volume 2, Chapter 26
Cosine similarity	$\cos\theta = \dfrac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$	Volume 2, Chapter 26
Projection of $\mathbf{u}$ onto $\mathbf{v}$	$\operatorname{proj}_{\mathbf{v}}\mathbf{u} = \dfrac{\mathbf{u}\cdot\mathbf{v}}{\mathbf{v}\cdot\mathbf{v}}\,\mathbf{v}$	Volume 2, Chapter 26
Cross product (3D)	$\mathbf{u}\times\mathbf{v} = (u_2v_3 - u_3v_2,\ u_3v_1 - u_1v_3,\ u_1v_2 - u_2v_1)$; length $\|\mathbf{u}\|\|\mathbf{v}\|\sin\theta$, perpendicular to both	Volume 2, Chapter 26
Triangle and Cauchy-Schwarz inequalities	$\|\mathbf{u}+\mathbf{v}\| \le \|\mathbf{u}\| + \|\mathbf{v}\|$; $\|\mathbf{u}\cdot\mathbf{v}\| \le \|\mathbf{u}\|\|\mathbf{v}\|$	Volume 2, Chapter 26
Matrix product	$(AB)_{ij} = \sum_k A_{ik}B_{kj}$; needs (columns of A) = (rows of B); in general $AB \neq BA$	Volume 2, Chapter 27

Fact	Formula	Ch.
Transpose rules	$(A^T)_{ij} = A_{ji}$; $(AB)^T = B^T A^T$	Volume 2, Chapter 27
Inverse rules	$AA^{-1} = A^{-1}A = I$; $(AB)^{-1} = B^{-1}A^{-1}$; $(A^T)^{-1} = (A^{-1})^T$	Volume 2, Chapter 27
2×2 determinant	$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$	Volume 2, Chapter 27
2×2 inverse	$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$ (if $ad - bc \neq 0$)	Volume 2, Chapter 27
3×3 determinant (first-row expansion)	$a(ei - fh) - b(di - fg) + c(dh - eg)$ for rows $(a,b,c), (d,e,f), (g,h,i)$	Volume 2, Chapter 27
Determinant facts	$\det(AB) = \det A \det B$; $\det A^T = \det A$; A invertible $\iff \det A \neq 0$; $\lvert\det A\rvert$ is the area/volume scale factor	Volume 2, Chapter 27
Rotation by θ (2D)	$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$; $R_\theta^{-1} = R_\theta^T = R_{-\theta}$	Volume 2, Chapter 27
Least squares (normal equations)	Minimise $\lVert A\mathbf{x} - \mathbf{b}\rVert$: solve $A^T A\mathbf{x} = A^T\mathbf{b}$	Volume 2, Chapter 27

Fact	Formula	Ch.
Eigenvalues and eigenvectors	$A\mathbf{v} = \lambda\mathbf{v}$, $\mathbf{v} \neq \mathbf{0}$; eigenvalues solve $\det(A - \lambda I) = 0$	Volume 2, Chapter 28
2×2 characteristic equation	$\lambda^2 - (\operatorname{tr} A)\lambda + \det A = 0$; in general $\operatorname{tr} A = \sum \lambda_i$, $\det A = \prod \lambda_i$	Volume 2, Chapter 28
Diagonalisation / spectral theorem	$A = PDP^{-1}$; for symmetric A, $A = Q\Lambda Q^T$ with Q orthogonal and real eigenvalues	Volume 2, Chapter 28
Singular value decomposition	$A = U\Sigma V^T$ (U, V orthogonal, Σ diagonal, non-negative)	Volume 2, Chapter 28

A.21. Probability

Rule	Formula	Ch.
Range and total	$0 \le P(A) \le 1$, $P(\Omega) = 1$	Chapter 29
Equally likely outcomes	$P(A) = \dfrac{\text{number of outcomes in } A}{\text{total number of outcomes}}$	Chapter 29
Complement	$P(A^c) = 1 - P(A)$; "at least one" $= 1 - P(\text{none})$	Chapter 29
Addition rule	$P(A \cup B) = P(A) + P(B) - P(A \cap B)$; just $P(A) + P(B)$ if mutually exclusive	Chapter 29
Conditional probability	$P(A \mid B) = \dfrac{P(A \cap B)}{P(B)}$ $(P(B) > 0)$	Chapter 29
Multiplication rule	$P(A \cap B) = P(A \mid B)\,P(B) = P(B \mid A)\,P(A)$	Chapter 29
Independence	$P(A \cap B) = P(A)\,P(B)$, equivalently $P(A \mid B) = P(A)$	Chapter 29
Law of total probability	$P(A) = \sum_i P(A \mid B_i)\,P(B_i)$ for a partition $B_1, B_2, \ldots$	Chapter 29
Bayes' theorem	$P(H \mid E) = \dfrac{P(E \mid H)\,P(H)}{P(E)} = \dfrac{P(E \mid H)\,P(H)}{P(E \mid H)\,P(H) + P(E \mid H^c)\,P(H^c)}$	Chapter 29
Birthday bound	With N equally likely values and n draws, $P(\text{some repeat}) \approx 1 - e^{-n(n-1)/(2N)}$; likely once $n \approx 1.18\sqrt{N}$	Chapter 29
Expectation	$E[X] = \sum_x x\,P(X = x)$ (discrete) or $\int x\,f(x)\,dx$ (continuous)	Chapter 30
Linearity of expectation	$E[aX + bY + c] = aE[X] + bE[Y] + c$ (always, even if dependent)	Chapter 30
Variance	$\mathrm{Var}(X) = E[(X - \mu)^2] = E[X^2] - (E[X])^2$; $\sigma = \sqrt{\mathrm{Var}(X)}$	Chapter 30
Variance rules	$\mathrm{Var}(aX + b) = a^2\,\mathrm{Var}(X)$; $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X, Y)$	Chapter 30

Rule	Formula	Ch.
Covariance and correlation	$\mathrm{Cov}(X,Y) = E[XY] - E[X]E[Y]$; $\rho = \dfrac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y} \in [-1,1]$; independent $\Longrightarrow$ covariance 0 (not conversely)	Chapter 30

A.22. Common distributions

Distribution	PMF or PDF	Mean	Variance	Ch.
Bernoulli (p)	$P(X=1)=p$, $P(X=0)=1-p$	p	$p(1-p)$	Chapter 30
Binomial (n,p)	$\binom{n}{k} p^k (1-p)^{n-k}$, $k=0,\dots,n$	np	$np(1-p)$	Chapter 30
Geometric (p) (trials until first success)	$(1-p)^{k-1}p$, $k=1,2,\dots$	$\dfrac{1}{p}$	$\dfrac{1-p}{p^2}$	Chapter 30
Poisson (λ)	$\dfrac{e^{-\lambda}\lambda^k}{k!}$, $k=0,1,2,\dots$	λ	λ	Chapter 30
Discrete uniform on $\{1,\dots,n\}$	$\dfrac{1}{n}$	$\dfrac{n+1}{2}$	$\dfrac{n^2-1}{12}$	Chapter 30
Continuous uniform on $[a,b]$	$\dfrac{1}{b-a}$ for $a \le x \le b$	$\dfrac{a+b}{2}$	$\dfrac{(b-a)^2}{12}$	Chapter 30
Exponential (λ)	$\lambda e^{-\lambda x}$ for $x \ge 0$	$\dfrac{1}{\lambda}$	$\dfrac{1}{\lambda^2}$	Chapter 30
Normal (μ, σ^2)	$\dfrac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$	μ	σ^2	Chapter 30

Fact	Formula	Ch.
Standardising	$Z = \frac{X - \mu}{\sigma}$ has mean 0 and standard deviation 1	Chapter 30
68-95-99.7 rule	For a normal distribution, about 68%, 95% and 99.7% of values lie within 1, 2 and 3 standard deviations of the mean	Chapter 30
Central limit theorem	For n independent draws with mean μ and variance σ^2, the sample mean $\bar{X}$ is approximately $N(\mu, \sigma^2/n)$ for large n	Chapter 30
Law of large numbers	$\bar{X} \to \mu$ as $n \to \infty$	Chapter 30

A.23. Statistics

Quantity	Formula	Ch.
Mean	$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$	Chapter 31
Median, quartiles, IQR	Median: middle value of the sorted data (average the two middle values if n is even); $\text{IQR} = Q_3 - Q_1$; outlier rule of thumb: beyond $1.5 \times \text{IQR}$ from the quartiles	Chapter 31
Population variance	$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$	Chapter 31
Sample variance	$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ (divide by $n-1$, Bessel's correction)	Chapter 31
z-score	$z = \frac{x - \bar{x}}{s}$ (or $\frac{x-\mu}{\sigma}$)	Chapter 31
Standard error of the mean	$\text{SE} = \frac{\sigma}{\sqrt{n}} \approx \frac{s}{\sqrt{n}}$	Chapter 31
Confidence interval for a mean	$\bar{x} \pm t^* \frac{s}{\sqrt{n}}$ ($t^* \approx z^* = 1.96$ for 95% with large n)	Chapter 31
Confidence interval for a proportion	$\hat{p} \pm z^* \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$	Chapter 31
Sample size for margin of error E	Proportion: $n \approx \left(\frac{z^*}{E}\right)^2 \hat{p}(1-\hat{p})$ (use $\hat{p} = 0.5$ if unknown); 95%, $\pm 3\%$ needs about 1068	Chapter 31
Correlation coefficient	$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i-\bar{x})^2 \sum(y_i-\bar{y})^2}} \in [-1, 1]$	Chapter 31
Least-squares regression line	$\hat{y} = a + bx$ with $b = \frac{\sum(x_i-\bar{x})(y_i-\bar{y})}{\sum(x_i-\bar{x})^2} = r\frac{s_y}{s_x}$, $a = \bar{y} - b\bar{x}$	Chapter 31
Coefficient of determination	$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i-\bar{y})^2}$; equals r^2 for a straight-line fit	Chapter 31

Quantity	Formula	Ch.
Hypothesis test	p-value = probability, if the null hypothesis were true, of data at least as extreme as observed. Reject H_0 if p-value $< \alpha$ (often 0.05)	Chapter 31
Test statistic for a mean	$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$ ($n - 1$ degrees of freedom)	Chapter 31

A.24. Numerical methods

Method	Formula	Ch.
Newton's method	$x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$; quadratic convergence near a simple root (correct digits roughly double each step); can diverge from a poor start or where $f' \approx 0$	Volume 2, Chapter 32
Square root by Newton (Heron)	$x_{n+1} = \frac{1}{2}\left(x_n + \dfrac{a}{x_n}\right) \to \sqrt{a}$	Volume 2, Chapter 18
Bisection	If $f(a)$ and $f(b)$ differ in sign, test the midpoint $m = \frac{a+b}{2}$ and keep the half where the sign changes. After n halvings the bracket has width $(b-a)/2^n$; need about $\log_2\left((b-a)/\text{tol}\right)$ steps	Volume 2, Chapter 32
Secant method	$x_{n+1} = x_n - f(x_n)\dfrac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$ (Newton without a derivative)	Volume 2, Chapter 32
Fixed-point iteration	$x_{n+1} = g(x_n)$ converges near a fixed point x^* if $\lvert g'(x^*)\rvert < 1$	Volume 2, Chapter 32
Finite differences	Forward $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$ (error $O(h)$); central $f'(x) \approx \dfrac{f(x+h) - f(x-h)}{2h}$ (error $O(h^2)$)	Volume 2, Chapter 32
Trapezoid rule	$\int_a^b f \approx \dfrac{h}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{n-1} + f_n)$, $h = \dfrac{b-a}{n}$, $f_i = f(a+ih)$; error $O(h^2)$	Volume 2, Chapter 32
Simpson's rule (n even)	$\int_a^b f \approx \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n)$; error $O(h^4)$	Volume 2, Chapter 32

Method	Formula	Ch.
Euler's method (ODE $y' = f(t, y)$)	$y_{n+1} = y_n + h\, f(t_n, y_n)$; error $O(h)$	Volume 2, Chapter 32
Floating point (IEEE double)	Machine epsilon $2^{-52} \approx 2.2 \times 10^{-16}$; about 15-16 significant decimal digits; never test floats with $==$, use a tolerance	Volume 2, Chapter 32
Absolute and relative error	$\lvert\hat{x} - x\rvert$ and $\dfrac{\lvert\hat{x} - x\rvert}{\lvert x\rvert}$	Volume 2, Chapter 32

A.25. Optimisation, machine learning, and information

Quantity	Formula	Ch.
Mean squared error	$\mathrm{MSE} = \dfrac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Chapter 33
Softmax	$\mathbf{softmax}(\mathbf{z})_i = \dfrac{e^{z_i}}{\sum_j e^{z_j}}$ (subtract $\max_j z_j$ first for numerical safety)	Chapter 33
Entropy (bits)	$H(p) = -\sum_i p_i \log_2 p_i$; maximal ($\log_2 n$) for the uniform distribution on n outcomes	Chapter 33
Cross-entropy	$H(p, q) = -\sum_i p_i \log q_i$; with a one-hot label it is $-\log q_{\text{true}}$	Chapter 33
Kullback-Leibler divergence	$D_{\mathrm{KL}}(p \,\|\, q) = \sum_i p_i \log \dfrac{p_i}{q_i} = H(p, q) - H(p) \geq 0$	Chapter 33

Appendix B: Glossary

Every Refresher box in Volume 3, alphabetised. This is the book's reference mode: the same definitions you met in context, collected for lookup. Each entry links back to the chapter where it is taught.

B.1. A

argmin and argmax (Chapter 33)

$\min_\theta L(\theta)$ is the smallest value of L. $\operatorname{argmin}_\theta L(\theta)$ is the *argument*—the value of θ—that achieves it. For $L(\theta) = (\theta - 3)^2 + 1$, $\min_\theta L = 1$ but $\operatorname{argmin}_\theta L = 3$. Programmers know this distinction: `min(xs)` versus `xs.index(min(xs))`.

B.2. B

Bayes' theorem: prior, likelihood, posterior (Chapter 29)

$$\underbrace{P(H \mid E)}_{\text{posterior}} = \frac{\overbrace{P(E \mid H)}^{\text{likelihood}}\overbrace{P(H)}^{\text{prior}}}{\underbrace{P(E)}_{\text{evidence}}}$$

H is a **hypothesis** (has the disease), E the **evidence** (tested positive).

- **Prior** $P(H)$: how plausible H was before the evidence (the base rate).
- **Likelihood** $P(E \mid H)$: how probable the evidence is if H is true.
- **Posterior** $P(H \mid E)$: how plausible H is after seeing the evidence.
- **Evidence** or **marginal likelihood** $P(E)$: overall probability of the evidence, from the law of total probability.

For tests: **sensitivity** = $P(+ \mid \text{disease})$ (true-positive rate); **specificity** = $P(- \mid \text{healthy})$; false-positive rate = $1 -$ specificity.

Binomial, the Σ notation (Chapter 24)

A **binomial** is a polynomial with two terms, such as $x + y$ or $2x - 1$.

The **summation sign** $\sum$ (capital Greek sigma, for "sum") packages a long sum. $\sum_{k=0}^{n} a_k$ means "let the **index** k run through the integers from the lower limit 0 to the upper limit n, and add up the terms a_k": $a_0 + a_1 + \cdots + a_n$.

Bipartite graph, matching (Chapter 25)

A graph is **bipartite** if its vertices can be split into two sets L and R (the **parts**) so that every edge joins a vertex of L to a vertex of R. $K_{a,b}$ is the **complete bipartite graph** with $|L| = a$, $|R| = b$ and all ab edges between them. A **matching** is a set of edges no two of which share an endpoint; a **perfect matching** covers every vertex.

Bit, surprise, entropy (Chapter 33)

A **bit** is the information in one fair yes/no answer. The **surprise** or **self-information** of an outcome with probability p is $-\log_2 p$ bits. **Entropy** H is average surprise: the uncertainty of a random source before you see the outcome. With natural logarithms the unit is the **nat** ($1 \text{ nat} \approx 1.443$ bits); ML papers often use nats without saying so.

Bézout's identity (Chapter 23)

For integers a, b, not both zero, there exist integers x, y with

$$ax + by = \gcd(a, b).$$

The x, y are **Bézout coefficients**. They are not unique. A **linear combination** of a and b is any expression $ax + by$. The integer combinations of a and b are exactly the multiples of $\gcd(a, b)$.

B.3. C

Ceiling and floor (Chapter 24)

The **ceiling** $\lceil x \rceil$ is the smallest integer $\geq x$: $\lceil 2.1 \rceil = 3$, $\lceil 5 \rceil = 5$. The **floor** $\lfloor x \rfloor$ is the largest integer $\leq x$: $\lfloor 2.9 \rfloor = 2$. In code, `math.ceil` and `math.floor`; for positive integers, $\lceil N/k \rceil$ is `(N + k - 1) // k` and $\lfloor N/k \rfloor$ is `N // k`.

Central limit theorem (Chapter 30)

If $X_1, \ldots, X_n$ are i.i.d. with mean μ and standard deviation σ, then for large n,

$$\bar{X}_n \approx N\left(\mu, \frac{\sigma^2}{n}\right), \qquad \text{equivalently} \qquad \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} \approx N(0, 1).$$

The sum $X_1 + \cdots + X_n$ is likewise approximately $N(n\mu, n\sigma^2)$. The quantity $\sigma/\sqrt{n}$ is the **standard error** of the mean.

Combination and binomial coefficient (Chapter 24)

A **combination** (or r-combination) of n distinct objects is an *unordered* selection of r of them, that is, a subset of size r. The number of them is the **binomial coefficient**

$$\binom{n}{r} = \frac{n!}{r!\,(n-r)!},$$

read "n choose r". Other notations: $C(n, r)$, ${}_nC_r$, ${}^nC_r$ (common in UK textbooks). By convention $\binom{n}{r} = 0$ if $r < 0$ or $r > n$.

Combinatorics, enumeration, outcome (Chapter 24)

Combinatorics is the mathematics of counting arrangements and selections, and of deciding whether arrangements with given properties exist. To **enumerate** a set is to list or count its elements. An **outcome** is one specific result of a choice process (for example, one particular poker hand). For a finite set S, the notation $|S|$ means the number of elements of S, called its **cardinality**.

Complement rule and addition rule (Chapter 29)

Complement rule: $P(A^c) = 1 - P(A)$.

Addition rule: $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. If A and B are mutually exclusive, the last term is 0.

Monotonicity: if $A \subseteq B$ then $P(A) \leq P(B)$. (A more specific event can't be more probable.)

Complement, union, intersection, mutually exclusive (Chapter 29)

Complement A^c: everything in Ω not in A. **Union** $A \cup B$: "A or B or both" (mathematical "or" is always inclusive). **Intersection** $A \cap B$: "both A and B". Events are **mutually exclusive** (or **disjoint**) if they share no outcomes, $A \cap B = \varnothing$. The symbol $\varnothing$ is the empty set.

Conditional probability (Chapter 29)

For events A and B with $P(B) > 0$, the **conditional probability of A given B** is

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

Read the vertical bar $\mid$ as "given". B is the **conditioning event** (what you know); A is the event you're asking about. $P(A)$, without a condition, is sometimes called the **unconditional** or **marginal** probability.

Meaning: of all the probability sitting on B, what fraction also lies in A? Geometrically: zoom into B and rescale it to have total probability 1.

Confidence interval (Chapter 31)

A **95% confidence interval** for a parameter is an interval computed from the data by a procedure that captures the true parameter in 95% of repeated samples. For a mean:

$$\bar{x} \pm z^* \cdot \frac{s}{\sqrt{n}},$$

where $z^* = 1.96$ for 95% (1.645 for 90%, 2.576 for 99%). The quantity $z^* \cdot \mathrm{SE}$ is the **margin of error**. The **confidence level** (95%) is the long-run success rate of the procedure.

For small samples, replace z^* by a slightly larger t^* from the **t-distribution** (Section 31.6.6) to account for estimating σ by s.

Congruence modulo m (Chapter 23)

Fix a positive integer m, the **modulus**. Integers a and b are **congruent modulo m**, written

$$a \equiv b \pmod{m},$$

if $m \mid (a - b)$. Equivalently, a and b leave the same remainder when divided by m. Read it as "a is congruent to b mod m". For example, $14 \equiv 2 \pmod{12}$, $-10 \equiv 2 \pmod{12}$, and $100 \equiv 4 \pmod{12}$.

The three-bar symbol $\equiv$ was Gauss's choice (1801). It looks like "=" on purpose.

Connected, component (Chapter 25)

An undirected graph is **connected** if there is a path between every pair of vertices. Otherwise it splits into **connected components**: maximal connected pieces. Being "joined by a path" is an equivalence relation (reflexive, symmetric, transitive; see Chapter 21), and the components are its equivalence classes.

A directed graph is **strongly connected** if there is a directed path from every vertex to every other, and **weakly connected** if it is connected when you ignore the arrows.

Connective (Chapter 21)

A **logical connective** (or **operator**) builds a new proposition from one or more existing ones. The five standard connectives are:

Symbol	Name	Read as	In code
$\neg p$	negation	"not p"	`not p`, `!p`
$p \wedge q$	conjunction	"p and q"	`p and q`, `p && q`
$p \vee q$	disjunction	"p or q"	`p or q`, `p || q`
$p \to q$	conditional (implication)	"if p then q"	`(not p) or q`
$p \leftrightarrow q$	biconditional	"p if and only if q"	`p == q`

A proposition built from others using connectives is **compound**; one that cannot be broken down further is **atomic**. Some books write $\sim p$ or $\overline{p}$ for negation, $\Rightarrow$ for $\to$, and $\Leftrightarrow$ or "iff" for $\leftrightarrow$.

Contrapositive, converse, inverse (Chapter 22)

For the implication $P \Rightarrow Q$ ("if P then Q"):

- **Contrapositive:** $\neg Q \Rightarrow \neg P$. Always equivalent to the original.
- **Converse:** $Q \Rightarrow P$. **Not** equivalent in general.
- **Inverse:** $\neg P \Rightarrow \neg Q$. **Not** equivalent in general (it is the contrapositive of the converse).

The symbol $\neg$ means "not". See Chapter 21 for the truth tables.

Convex function (Chapter 33)

A function f is **convex** if the straight line (chord) between any two points on its graph lies on or above the graph:

$$f\big(ta + (1-t)b\big) \le tf(a) + (1-t)f(b) \qquad \text{for all } a, b \text{ and } 0 \le t \le 1.$$

For a twice-differentiable function of one variable this is the same as $f''(x) \geq 0$ everywhere: the slope never decreases. For several variables the condition is that the Hessian matrix (Volume 2, Chapter 20) is positive semidefinite. Examples: x^2, e^x, $|x|$. Non-examples: $\sin x$, x^3, $-x^2$.

Cosine similarity, norm, unit vector (Chapter 33)

The **norm** (length) of a vector is $\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{\sum_j v_j^2}$. A **unit vector** has norm 1; **normalising** means dividing by the norm. **Cosine similarity** is the dot product of the two normalised vectors. For unit vectors, dot product and cosine similarity are the same number—which is why systems often normalise everything once at indexing time and then just take dot products.

Covariance (Chapter 30)

The **covariance** of X and Y, with means μ_X and μ_Y, is

$$\operatorname{Cov}(X, Y) = E\big[(X - \mu_X)(Y - \mu_Y)\big] = E[XY] - \mu_X \mu_Y.$$

Positive when X and Y tend to be above (or below) their means together; negative when one tends to be high as the other is low; zero (**uncorrelated**) when there's no linear tendency.
$\operatorname{Cov}(X, X) = \operatorname{Var}(X)$.

Variance of a sum, in general: $\operatorname{Var}(X + Y) = \operatorname{Var}(X) + \operatorname{Var}(Y) + 2\operatorname{Cov}(X, Y)$.

Cross-entropy and KL divergence (Chapter 33)

Cross-entropy $H(p, q)$ is the average surprise when outcomes come from p but you score them with q. **KL divergence** $D_{\mathrm{KL}}(p \,\|\, q)$ is the excess, $H(p, q) - H(p)$. It measures how different q is from p, but it is **not** a distance: in general $D_{\mathrm{KL}}(p \,\|\, q) \neq D_{\mathrm{KL}}(q \,\|\, p)$. The double bar $\|$ separates the two arguments.

Cross-entropy loss (Chapter 33)

For a true label $y \in \{0, 1\}$ and predicted probability p, the **binary cross-entropy** is $-[y \ln p + (1 - y) \ln(1 - p)]$. If $y = 1$ it is $-\ln p$: zero when $p = 1$, growing without bound as $p \to 0$. Confident wrong answers are punished severely. It is also called **log loss** or **negative log-likelihood**.

Cumulative distribution function (CDF) (Chapter 30)

The **cumulative distribution function** of X is

$$F(x) = P(X \leq x).$$

It rises from 0 (far left) to 1 (far right) and never decreases. Then $P(a < X \leq b) = F(b) - F(a)$. For a discrete variable F is a staircase, jumping by $p(x)$ at each value; for a continuous one it's a smooth curve whose slope is the density ($f = F'$, Volume 2, Chapter 17).

The **median** is the value where $F(x) = 0.5$; more generally the q-th **quantile** (or $100q$-th **percentile**) is where $F(x) = q$. "p99 latency" is the 0.99 quantile.

B.4. D

DAG and topological order (Chapter 25)

A **directed acyclic graph** (**DAG**) is a directed graph with no directed cycle. A **topological order** (or topological sort) of a directed graph is a listing of all its vertices such that for every edge $u \to v$, u comes before v. A vertex with in-degree 0 is a **source**; with out-degree 0, a **sink**.

Degree (Chapter 25)

The **degree** of a vertex v, written $\deg(v)$, is the number of edges incident to it (a loop counts twice). In a directed graph a vertex has an **in-degree** (arrows arriving) and an **out-degree** (arrows leaving). A vertex of degree 0 is **isolated**; a vertex of degree 1 is a **leaf** (or pendant vertex). A graph in which every vertex has the same degree k is **k-regular**.

Differential equation (Chapter 33)

A **differential equation** is an equation involving an unknown function and its derivatives. An **ordinary** differential equation (ODE) has one independent variable (usually time); a **partial** differential equation (PDE) has several, with partial derivatives (heat flow, waves, fluid dynamics). An **initial-value problem** specifies the starting state, such as $y(0) = y_0$. The **order** is the highest derivative that appears.

Dividend, divisor, quotient, remainder (Chapter 23)

In $23 = 4 \cdot 5 + 3$ the number being divided (23) is the **dividend**. The number you divide by (5) is the **divisor**. The number of whole times it fits (4) is the **quotient**, and what is left (3) is the **remainder**. "Quotient" comes from the Latin *quotiens*, "how many times".

Divides, factor, multiple (Chapter 23)

For integers a and b, we say a **divides** b, written $a \mid b$, if $b = ka$ for some integer k. Equivalently, dividing b by a leaves remainder 0. Then a is a **divisor** (or **factor**) of b, and b is a **multiple** of a.

So $3 \mid 12$ because $12 = 4 \cdot 3$, and $5 \nmid 12$ ("5 does not divide 12"). The vertical bar is a *statement* (true or false), not an operation: $3 \mid 12$ is not the number 4. Don't confuse it with the fraction $3/12$.

Division algorithm (quotient–remainder theorem) (Chapter 23)

For any integer a and any positive integer b, there are **unique** integers q (the quotient) and r (the remainder) with

$$a = qb + r, \qquad 0 \le r < b.$$

We write $r = a \bmod b$, read "a mod b".

B.5. E

Embedding (Chapter 33)

An **embedding** is a map from discrete objects (words, sentences, images, users, products) to vectors in $\mathbb{R}^k$, chosen so that geometric relationships reflect real ones: similar objects get nearby vectors. **Sparse** embeddings (TF-IDF) have one dimension per vocabulary word and mostly zeros; **dense** embeddings have a few hundred learned dimensions, all typically non-zero.

Empty set, subset, universal set (Chapter 21)

- The **empty set** $\varnothing$ (or $\{\}$) has no elements; $|\varnothing| = 0$. There is only one empty set.
- A is a **subset** of B, written $A \subseteq B$, if every element of A is also in B: $\forall x\,(x \in A \to x \in B)$. A **proper subset**, $A \subsetneq B$ (often written $A \subset B$), is a subset that is not equal to B.
- $A = B$ exactly when $A \subseteq B$ and $B \subseteq A$. This "two inclusions" approach is the standard way to prove two sets equal.
- The **universal set** U is the set of everything under discussion in a given context (all customers, all integers).

Equivalence relation and equivalence class (Chapter 21)

An **equivalence relation** is a relation that is reflexive, symmetric and transitive. It behaves like "is the same as, in some respect". Common notation is $a \sim b$.

The **equivalence class** of a is $[a] = \{\, x \in A \mid x \sim a \,\}$, everything equivalent to a. The classes form a **partition** of A: they are nonempty, pairwise disjoint, and their union is all of A. Every element lives in exactly one class.

Euclid's lemma (Chapter 23)

If a prime p divides a product ab, then $p \mid a$ or $p \mid b$.

Euler path and Euler circuit (Chapter 25)

An **Euler path** (or Euler trail) is a trail that uses every edge of the graph exactly once. An **Euler circuit** is an Euler path that starts and ends at the same vertex. A graph with an Euler circuit is called **Eulerian**.

Even and odd, parity (Chapter 23)

An integer is **even** if $2 \mid n$ and **odd** otherwise. Whether a number is even or odd is its **parity**. Every odd number can be written as $2k + 1$ for some integer k.

Even, odd, divides (Chapter 22)

- $\mathbb{Z}$ (blackboard-bold Z, from German *Zahlen*, "numbers") is the set of all integers $\ldots, -2, -1, 0, 1, 2, \ldots$. $\mathbb{N}$ is the natural numbers; in this chapter $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- An integer n is **even** if $n = 2k$ for some integer k.
- An integer n is **odd** if $n = 2k + 1$ for some integer k.
- For integers $a \neq 0$ and b, we say a **divides** b, written $a \mid b$, if $b = ak$ for some integer k. So $3 \mid 12$ (because $12 = 3 \cdot 4$) but $3 \nmid 13$. Note that $a \mid b$ is a *statement* (true or false), not a number; do not confuse it with a/b.

- Every integer is either even or odd, and never both. (This follows from division with remainder: dividing by 2 leaves remainder 0 or 1.)

Exclusive or (XOR) (Chapter 21)

$p \oplus q$ ("p xor q") is true when exactly one of p, q is true. It equals $(p \vee q) \wedge \neg(p \wedge q)$, and it is the negation of the biconditional: $p \oplus q \equiv \neg(p \leftrightarrow q)$.

Expected value (Chapter 30)

The **expected value** (or **expectation**, or **mean**) of a discrete random variable X is

$$E[X] = \sum_{x} x\, P(X = x),$$

the sum of each possible value times its probability: a **weighted average** of the values, weighted by probability. It is often written μ (Greek *mu*) or μ_X.

Interpretation: the long-run average of X over many independent repetitions.

B.6. F

Factorial (Chapter 24)

For a positive integer n, **n factorial**, written $n!$, is the product of all integers from 1 to n: $n! = n \times (n-1) \times \cdots \times 2 \times 1$. So $3! = 6$, $5! = 120$, $10! = 3\,628\,800$. By convention $0! = 1$.

$n!$ counts the ways to arrange n distinct objects in a row. The useful identity $n! = n \times (n-1)!$ is the recursive definition you would write in code.

Fermat's little theorem (Chapter 23)

If p is prime and $p \nmid a$, then

$$a^{p-1} \equiv 1 \pmod{p}.$$

Equivalently, $a^p \equiv a \pmod{p}$ for every integer a. ("Little" distinguishes it from Fermat's Last Theorem.)

Floor and truncation (Chapter 23)

The **floor** $\lfloor x \rfloor$ is the greatest integer $\leq x$: $\lfloor 3.5 \rfloor = 3$ and $\lfloor -3.5 \rfloor = -4$. **Truncation** drops the fractional part, rounding toward 0: $-3.5 \to -3$. They agree for nonnegative numbers and differ for negative non-integers.

Function (as a relation) (Chapter 21)

A **function** $f : A \to B$ is a relation from A to B in which each element of the **domain** A is paired with exactly one element of the **codomain** B. "Exactly one" has two halves: **at least one** (defined everywhere on A) and **at most one** (no input has two outputs).

Fundamental theorem of arithmetic (Chapter 23)

Every integer $n > 1$ can be written as a product of primes, and this factorization is **unique** apart from the order of the factors.

B.7. G

GCD, coprime (Chapter 23)

The **greatest common divisor** $\gcd(a, b)$ of two integers, not both zero, is the largest positive integer dividing both. US schools call it the **greatest common factor (GCF)**, and UK schools the **highest common factor (HCF)**. If $\gcd(a, b) = 1$, then a and b are **coprime** (or **relatively prime**). They need not be prime themselves: 8 and 15 are coprime. By convention $\gcd(a, 0) = |a|$.

Graph colouring and chromatic number (Chapter 25)

A (proper) **colouring** of a graph assigns a colour to each vertex so that adjacent vertices get different colours. The **chromatic number** $\chi(G)$ (Greek letter chi) is the fewest colours that suffice. $\chi(G) \le 2$ exactly when G is bipartite; $\chi(K_n) = n$; a graph containing a triangle needs at least 3.

Graph, vertex, edge (Chapter 25)

A **graph** $G = (V, E)$ consists of a set V of **vertices** (also called **nodes**) and a set E of **edges**, each edge joining two vertices. An edge between u and v is written $\{u, v\}$ or simply uv. u and v are its **endpoints**; they are **adjacent** (neighbours), and the edge is **incident** to each of them. The number of vertices $|V|$ is often written n and the number of edges $|E|$ is written m.

The singular of vertices is **vertex**. "Graph" here has nothing to do with the graph of a function (Volume 1, Chapter 11); the word was coined by J. J. Sylvester in 1878 from "chemicograph", a diagram of molecular bonds.

B.8. H

Hamiltonian path and cycle (Chapter 25)

A **Hamiltonian path** visits every vertex exactly once. A **Hamiltonian cycle** is a Hamiltonian path that returns to its start along an edge. The weighted version, "find the cheapest Hamiltonian cycle in a complete weighted graph", is the **travelling salesman problem** (TSP).

Hypotheses, test statistic, p-value, significance level (Chapter 31)

- **Null hypothesis** H_0: the "nothing is going on" claim, precise enough to compute probabilities (the coin is fair: $p = 0.5$; the two pages convert at the same rate).
- **Alternative hypothesis** H_1 (or H_a): what you suspect instead ($p \neq 0.5$: **two-sided**; or $p > 0.5$: **one-sided**).
- **Test statistic:** a number computed from the data that measures how far it departs from H_0, often a z-score: $\frac{\text{estimate} - \text{null value}}{\text{SE}}$.
- **p-value:** the probability, *computed assuming* H_0 *is true*, of getting a test statistic at least as extreme as the one observed.

- **Significance level α**: a threshold chosen in advance (commonly 0.05). If $p \le \alpha$, **reject H_0** ("statistically significant"); otherwise **fail to reject H_0**.

Hypothesis, conclusion, sufficient, necessary (Chapter 21)

In $p \to q$, p is the **hypothesis** (or **antecedent**, or **premise**) and q is the **conclusion** (or **consequent**). All of the following say exactly $p \to q$:

- if p then q; q if p;
- p implies q;
- p only if q;
- p is **sufficient** for q;
- q is **necessary** for p;
- q whenever p; not p unless q.

The biconditional $p \leftrightarrow q$ ("p if and only if q", often "iff") means $(p \to q) \wedge (q \to p)$: each is necessary and sufficient for the other.

B.9. I

Independent events (Chapter 29)

Events A and B are **independent** if

$$P(A \cap B) = P(A)\,P(B).$$

Equivalently (when $P(B) > 0$), $P(A \mid B) = P(A)$: knowing B happened doesn't change the probability of A. Events that aren't independent are **dependent**.

Several events $A_1, \ldots, A_n$ are **mutually independent** if the product rule holds for every sub-collection, e.g. $P(A_1 \cap A_2 \cap A_3) = P(A_1)P(A_2)P(A_3)$.

Indicator random variable (Chapter 30)

The **indicator** of an event A, written I_A or $\mathbf{1}_A$, is the random variable that equals 1 if A happens and 0 otherwise. Its expectation is the probability of the event:

$$E[I_A] = 1 \cdot P(A) + 0 \cdot P(A^c) = P(A).$$

Induction vocabulary (Chapter 22)

- $P(n)$ denotes a **statement** depending on n, true or false for each n; for example $P(n)$: "$\sum_{i=1}^{n}(2i-1) = n^2$". It is a predicate, not a number.
- The **base case** is the proof of $P(n_0)$ for the starting value n_0 (often 0 or 1).
- The **inductive hypothesis** (IH) is the assumption "$P(k)$ is true" for an arbitrary $k \ge n_0$.
- The **inductive step** is the proof that $P(k) \Rightarrow P(k+1)$ for every $k \ge n_0$.

- The **principle of mathematical induction**: if $P(n_0)$ is true and $P(k) \Rightarrow P(k+1)$ for all $k \geq n_0$, then $P(n)$ is true for all integers $n \geq n_0$.

B.10. K

Kinds of graph (Chapter 25)

- **Undirected graph:** edges have no direction; $\{u, v\}$ is the same as $\{v, u\}$. (Friendship, two-way roads, cables.)
- **Directed graph (digraph):** each edge is an ordered pair (u, v), drawn as an arrow from u (the **tail**) to v (the **head**). (Following on social media, hyperlinks, one-way streets, "depends on".)
- **Weighted graph:** each edge carries a number, its **weight** or **cost**: a distance, a latency, a price, a capacity.
- **Simple graph:** undirected, with no **loops** (edges from a vertex to itself) and no **multiple edges** (two edges joining the same pair). A graph that allows multiple edges, like Königsberg, is a **multigraph**.
- **Complete graph K_n**: a simple graph on n vertices in which every pair is adjacent. It has $\binom{n}{2} = n(n-1)/2$ edges (Chapter 24).

Unless stated otherwise, "graph" in this chapter means a simple undirected graph.

Kolmogorov's axioms (Chapter 29)

A **probability** assigns to each event A a number $P(A)$ such that

1. $P(A) \geq 0$ for every event A (no negative probabilities);
2. $P(\Omega) = 1$ (something certainly happens);
3. if A and B are mutually exclusive, $P(A \cup B) = P(A) + P(B)$. (The full axiom says this for any countable list of pairwise disjoint events.)

An **axiom** is a starting rule we accept without proof; everything else is derived from the axioms.

B.11. L

LCM (Chapter 23)

The **least common multiple** $\operatorname{lcm}(a, b)$ is the smallest positive integer that both a and b divide. It is the "lowest common denominator" you used when adding fractions in Volume 1, Chapter 2 .

Learning rate, mini-batch, epoch (Chapter 33)

The **learning rate** η is the step-size multiplier in gradient descent. A **mini-batch** is a small random subset of the training data used to estimate the gradient for one step. An **epoch** is one full pass through the training data — with 1,000 examples and batches of 32, one epoch is 32 steps (the last batch is short). **Momentum** β (Greek *beta*, typically 0.9) controls how much of the previous velocity carries into the next step.

Least squares regression line and residuals (Chapter 31)

The **least-squares regression line** (or **line of best fit**) of y on x is $\hat{y} = a + bx$ with

$$b = \frac{S_{xy}}{S_{xx}} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}, \qquad a = \bar{y} - b\,\bar{x}.$$

b is the **slope** (change in predicted y per unit of x); a is the **intercept**. The line always passes through the point of means $(\bar{x}, \bar{y})$. $\hat{y}$ ("y-hat") denotes a predicted value. A **residual** is $y_i - \hat{y}_i$.

Local and global minimum (Chapter 33)

A **local minimum** is a point lower than everything *nearby*. A **global minimum** is lower than (or equal to) *everything*. A **saddle point** is a critical point (gradient zero) that is a minimum along some directions and a maximum along others — like the middle of a horse saddle.

Logical equivalence, tautology, contradiction (Chapter 21)

Two propositions A and B are **logically equivalent**, written $A \equiv B$, if they have identical truth tables. Equivalently, $A \leftrightarrow B$ is true in every row.

A **tautology** is a proposition that is true in every row (for example $p \vee \neg p$). A **contradiction** is false in every row (for example $p \wedge \neg p$). A proposition that is neither is **contingent**. A proposition is **satisfiable** if at least one row makes it true.

$\equiv$ is a statement *about* two propositions (a "meta" claim), not a connective inside one.

Loop invariant, variant, pre- and postcondition (Chapter 22)

- A **precondition** is what must be true when a piece of code starts (e.g., "`a` is sorted").
- A **postcondition** is what the code guarantees when it ends (e.g., "the return value is an index of `target`, or -1 if `target` is absent").
- A **loop invariant** is a statement about the variables that holds every time execution reaches the top of the loop (before the test).
- A **loop variant** (or **ranking function**, or **termination measure**) is an integer quantity that is always non-negative while the loop runs and strictly decreases with every iteration. Its existence proves the loop terminates.
- A program is **partially correct** if, whenever it terminates, the postcondition holds; **totally correct** if, in addition, it always terminates.

B.12. M

Mean, median, mode (Chapter 31)

For data $x_1, x_2, \ldots, x_n$:

- **Mean** (arithmetic mean, average): $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. The bar over x means "sample mean". It uses every value, so it is sensitive to extreme values.

- **Median**: the middle value of the sorted data (the average of the two middle values if n is even). It is **robust**, meaning a few extreme values can't move it far.
- **Mode**: the most frequent value.

A **statistic** is any number computed from data. Statistics (the subject) studies how to compute and interpret them.

Modular inverse (Chapter 23)

An **inverse of a modulo m** is an integer a^{-1} with

$$a \cdot a^{-1} \equiv 1 \pmod{m}.$$

It exists **if and only if** $\gcd(a, m) = 1$, and it is unique mod m. For example, $3^{-1} \equiv 4 \pmod{11}$ because $3 \cdot 4 = 12 \equiv 1$. The notation a^{-1} here never means the fraction $1/a$.

Multiset and multinomial coefficient (Chapter 24)

A **multiset** is like a set but elements may appear more than once: $\{M, I, I, I, I, S, S, S, S, P, P\}$. The number of distinct orderings of a multiset with n items, of which n_1 are of type 1, n_2 of type 2, ..., n_k of type k, is the **multinomial coefficient**

$$\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{n_1!\, n_2! \cdots n_k!}, \qquad n_1 + \cdots + n_k = n.$$

With two types it is just $\binom{n}{n_1}$: arranging r identical H's and $n - r$ identical T's is choosing which r positions get an H.

B.13. N

Neuron, activation, layer (Chapter 33)

A **neuron** computes $\phi(\mathbf{w} \cdot \mathbf{x} + b)$ for weights $\mathbf{w}$, bias b, and **activation function** ϕ (Greek *phi*). Common activations: the sigmoid σ; $\tanh$; and the **ReLU** (rectified linear unit) $\mathrm{ReLU}(z) = \max(0, z)$. A **layer** applies many neurons to the same input. A **hidden layer** is any layer between input and output. The **forward pass** evaluates the network; the **backward pass** computes gradients.

Normal distribution (Chapter 30)

The **normal** (or **Gaussian**) distribution with mean μ and standard deviation σ, written $N(\mu, \sigma^2)$, has density

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

It is symmetric about μ (mean = median = mode), bell-shaped, and its tails thin out very fast. The **standard normal** is $Z \sim N(0, 1)$; its CDF is written $\Phi(z)$ (capital *phi*). The $\sqrt{2\pi}$ makes the total area 1.

Note: the second parameter in $N(\mu, \sigma^2)$ is the *variance*. $N(100, 225)$ has standard deviation 15.

B.14. O

Objective function, parameters, minimiser (Chapter 33)

The **objective function** (in ML, the **loss** or **cost** function) is the quantity being minimised, often written $L(\theta)$ or $f(\mathbf{x})$. The **parameters** θ (Greek *theta*) are the knobs; when there are many, θ is a vector. A **minimiser** θ^* (read "theta star") is a parameter value where the objective is smallest. The **minimum** is the smallest *value* $L(\theta^*)$; the minimiser is *where* it occurs.

Odds, logit, sigmoid (Chapter 33)

The **odds** of an event with probability p are $p/(1-p)$: a probability of 0.8 is odds of 4 to 1. The **logit** is the log-odds, $\operatorname{logit}(p) = \ln \frac{p}{1-p}$, which runs over all of $\mathbb{R}$. The **sigmoid** $\sigma(z) = 1/(1+e^{-z})$ is the inverse of the logit. So logistic regression says: *the log-odds are a linear function of the input*. Each extra unit of x multiplies the odds by e^{w}.

Optional: expectation as an integral (Chapter 30)

(Uses Volume 2, Chapter 19 ; skip freely.) For a continuous variable the sum $\sum x\,p(x)$ becomes an integral:

$$E[X] = \int_{-\infty}^{\infty} x f(x)\,dx, \qquad \operatorname{Var}(X) = \int_{-\infty}^{\infty} (x-\mu)^2 f(x)\,dx.$$

For the exponential, integrating by parts: $\int_0^\infty t\,\lambda e^{-\lambda t}\,dt = \left[-te^{-\lambda t}\right]_0^\infty + \int_0^\infty e^{-\lambda t}\,dt = 0 + \frac{1}{\lambda}$. Every rule from the discrete case (linearity, the variance formulas) carries over unchanged.

Ordered pair and Cartesian product (Chapter 21)

An **ordered pair** (a, b) is like a two-element set except that order matters: $(1, 2) \neq (2, 1)$, and $(a, b) = (c, d)$ exactly when $a = c$ and $b = d$. More generally an **n-tuple** is $(a_1, \ldots, a_n)$.

The **Cartesian product** of A and B is the set of all ordered pairs with first entry from A and second from B:

$$A \times B = \{\,(a, b) \mid a \in A,\ b \in B\,\}, \qquad |A \times B| = |A| \cdot |B|.$$

$\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ is the coordinate plane of Volume 1, Chapter 10 , which is where the name (after Descartes) comes from.

B.15. P

Partial order (Chapter 21)

A **partial order** is a relation that is reflexive, antisymmetric and transitive. It behaves like "comes before or equals". A set with a partial order is a **poset**. It is **partial** because two elements may be **incomparable**: neither $a \preceq b$ nor $b \preceq a$. If every pair is comparable it is a **total** (or **linear**) order, like $\leq$ on numbers.

Partition and the law of total probability (Chapter 29)

Events $B_1, B_2, \ldots, B_n$ form a **partition** of Ω if they are mutually exclusive and together cover Ω: every outcome lies in exactly one B_i. The **law of total probability** says, for any event A,

$$P(A) = \sum_{i=1}^{n} P(B_i)\, P(A \mid B_i).$$

Here $\sum_{i=1}^{n}$ ("sigma") means "add the following expression for $i = 1, 2, \ldots, n$". In a tree diagram, this is "add up every path that ends in A". Simplest case:
$P(A) = P(B)P(A \mid B) + P(B^c)P(A \mid B^c)$.

Permutation and r-permutation (Chapter 24)

A **permutation** of a set is an arrangement of all its elements in an order. An **r-permutation** of n distinct objects is an ordered selection of r of them (no repeats). The number of r-permutations is written $P(n, r)$, ${}_nP_r$ or $n^{\underline{r}}$ (read "n to the r falling"):

$$P(n, r) = n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!}.$$

Planar graph, face (Chapter 25)

A graph is **planar** if it can be drawn in the plane with no edges crossing. Such a drawing is a **plane embedding**. The edges divide the plane into regions called **faces**, including one unbounded **outer face**. A planar graph can also have non-planar drawings; planarity asks whether *some* crossing-free drawing exists.

Population, sample, parameter, statistic (Chapter 31)

- **Population:** the whole group or process you want to know about.
- **Sample:** the part you actually observe.
- **Parameter:** a number describing the population (true mean μ, true proportion p). Usually unknown.
- **Statistic:** a number computed from the sample ($\bar{x}$, $\hat{p}$). Used to **estimate** a parameter. The "hat" in $\hat{p}$ (read "p-hat") marks an estimate.
- **Simple random sample:** every subset of size n is equally likely to be the sample. This is what makes the probability of Chapter 30 apply.

Power set (Chapter 21)

The **power set** $\mathcal{P}(A)$ is the set of all subsets of A, including $\varnothing$ and A itself. For $A = \{a, b, c\}$:

$$\mathcal{P}(A) = \{\varnothing,\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}\}.$$

If $|A| = n$ then $|\mathcal{P}(A)| = 2^n$, because building a subset means making an independent in-or-out decision for each of the n elements. A subset of an n-element set **is** an n-bit string; that is why bitmasks represent sets.

Precision, recall, F1 (Chapter 31)

- **Precision** $= \dfrac{TP}{TP + FP}$: of the items flagged (or retrieved), what fraction are truly positive (relevant)? "When it says yes, how often is it right?"
- **Recall** $= \dfrac{TP}{TP + FN}$: of the truly positive (relevant) items, what fraction were flagged (retrieved)? "Of everything it should find, how much does it find?" Also called **sensitivity** or **true-positive rate**.
- **F1 score** $= \dfrac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$, the **harmonic mean** of the two. It's high only if both are high.

In probability language (Chapter 29): recall is $P(\text{flagged} \mid \text{positive})$ and precision is $P(\text{positive} \mid \text{flagged})$, the two directions of conditioning. Precision is the medical test's "probability of disease given a positive test", and so it depends on the base rate.

Predicate and domain (Chapter 21)

A **predicate** $P(x)$ is a statement involving a variable x that becomes a proposition when x is given a value. The set of allowed values is the **domain** (or **universe of discourse**). With domain the integers, $P(x)$: "x is even" gives $P(4)$ true and $P(7)$ false. Predicates can have several variables: $L(x, y)$: "$x < y$".

Prime (Chapter 22)

A **prime** is an integer $p > 1$ whose only positive divisors are 1 and p. The first few are $2, 3, 5, 7, 11, 13$. An integer $n > 1$ that is not prime is **composite**. A fact we will prove by strong induction in Section 22.10: every integer $n > 1$ has at least one prime divisor.

Prime, composite, unit (Chapter 23)

A **prime** is an integer $p > 1$ whose only positive divisors are 1 and p. An integer $n > 1$ that is not prime is **composite**: it can be written $n = ab$ with $1 < a, b < n$. The number 1 is **neither**. It is called a **unit**.

The primes start 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … and 2 is the only even one.

Prime factorization (Chapter 23)

A **prime factorization** writes a number as a product of primes, usually grouped with exponents in increasing order: $360 = 2^3 \cdot 3^2 \cdot 5$. The exponent of p is called the **multiplicity** of p.

Probability density function (PDF) (Chapter 30)

A continuous random variable X has a **probability density function** $f(x)$ if

$$P(a \le X \le b) = \text{area under } f \text{ between } a \text{ and } b \quad \left(= \int_a^b f(x)\,dx \right).$$

(The integral notation is from Volume 2, Chapter 19 ; "area under the curve" is all you need here.)

A density satisfies $f(x) \geq 0$ and has total area 1. The height $f(x)$ is **not** a probability (it can exceed 1); it's probability *per unit length*, like population density is people per square mile. For a short interval of width Δx, $P(x \leq X \leq x + \Delta x) \approx f(x)\,\Delta x$.

For a continuous variable, $P(X = a) = 0$ for every single value a, so $P(a \leq X \leq b) = P(a < X < b)$: endpoints don't matter.

Probability, experiment, outcome (Chapter 29)

A **random experiment** is any process whose result is uncertain: rolling a die, sampling a user, running a randomized algorithm. Each possible result is an **outcome**. A **probability** is a number from 0 to 1 attached to a set of outcomes: 0 means impossible, 1 means certain. Probabilities are often written as fractions ($\frac{1}{4}$), decimals (0.25) or percentages (25%); these are the same number in different clothes (Volume 1, Chapter 2).

Odds are a different way to express the same information: odds of $a{:}b$ in favour means probability $\frac{a}{a+b}$. Probability 0.2 is odds $1{:}4$.

Probability mass function (PMF) (Chapter 30)

The **probability mass function** of a discrete random variable X is $p(x) = P(X = x)$. It satisfies $p(x) \geq 0$ for every x, and $\sum_x p(x) = 1$ (the sum over all possible values). "Mass" because each value carries a lump of probability, like beads of different weights on a number line.

The **support** of X is the set of values with $p(x) > 0$.

Proof vocabulary you will see (Chapter 22)

- **"Let ..."** introduces an arbitrary object with stated properties.
- **"Suppose ..."** or **"Assume ..."** introduces a hypothesis, possibly temporary (as in proof by contradiction).
- **"Without loss of generality (WLOG)"** means "the other case is identical by symmetry, so we treat only this one".
- **"It suffices to show ..."** means "if we prove this simpler thing, the original claim follows".
- **"Hence"**, **"thus"**, **"therefore"**, **"it follows that"** mark a step derived from earlier lines.
- **"if and only if"** (iff, $\Leftrightarrow$) requires proving two directions: $P \Rightarrow Q$ and $Q \Rightarrow P$.

Properties of a relation on a set A (Chapter 21)

Property	Definition	Example
Reflexive	$\forall a\ (a\ R\ a)$	$\leq$, $=$, "divides"
Symmetric	$\forall a, b\ (a\ R\ b \to b\ R\ a)$	$=$, "is a sibling of"
Antisymmetric	$\forall a, b\ (a\ R\ b \wedge b\ R\ a \to a = b)$	$\leq$, $\subseteq$
Transitive	$\forall a, b, c\ (a\ R\ b \wedge b\ R\ c \to a\ R\ c)$	$<$, $\leq$, "is an ancestor of"

Antisymmetric is not the opposite of symmetric: a relation can be both (equality) or neither ("follows").

Proposition (Chapter 21)

A **proposition** (or **statement**) is a declarative sentence that is either true or false, but not both. "7 is prime" is a proposition (true). "7 is even" is a proposition (false). "Close the door" (a command) and "$x > 3$" (whose truth depends on an unknown x) are **not** propositions. We name propositions with lowercase letters p, q, r. The **truth value** of a proposition is T (true) or F (false), sometimes written 1 and 0.

B.16. Q

QED and the tombstone (Chapter 22)

QED abbreviates the Latin *quod erat demonstrandum*, "which was to be demonstrated". Modern texts usually end a proof with a small filled square ■, called a **tombstone** or **Halmos symbol** after the mathematician Paul Halmos who popularised it. Either mark says "the proof is complete; you can stop checking here".

Quantifiers (Chapter 21)

- The **universal quantifier** $\forall$ ("for all", an upside-down A): $\forall x\ P(x)$ is true if $P(x)$ is true for **every** x in the domain.
- The **existential quantifier** $\exists$ ("there exists", a backwards E): $\exists x\ P(x)$ is true if $P(x)$ is true for **at least one** x in the domain.
- $\exists!\ x\ P(x)$ means "there exists exactly one x".

The domain is often written into the quantifier: $\forall x \in \mathbb{Z}$ ("for every integer x"). A variable governed by a quantifier is **bound**; one that is not is **free**. Only a statement with no free variables is a proposition.

Quartiles, IQR, percentiles, five-number summary (Chapter 31)

- **First quartile** Q_1: a quarter of the data lie below it (the 25th **percentile**).
- **Median** Q_2 (50th percentile), **third quartile** Q_3 (75th percentile).

- **Interquartile range** $\mathrm{IQR} = Q_3 - Q_1$: the spread of the middle half of the data. Robust to outliers.
- **Five-number summary:** minimum, Q_1, median, Q_3, maximum.
- **The 1.5 × IQR rule:** a value below $Q_1 - 1.5\,\mathrm{IQR}$ or above $Q_3 + 1.5\,\mathrm{IQR}$ is flagged as a possible **outlier**.

Queue and stack (Chapter 25)

A **queue** is a list where items are added at the back and removed from the front (FIFO, "first in, first out"), like people waiting in line. A **stack** adds and removes at the same end (LIFO, "last in, first out"), like a pile of plates. In Python, `collections.deque` is an efficient queue (`append`, `popleft`); a plain list is an efficient stack (`append`, `pop`).

B.17. R

Random variable (Chapter 30)

A **random variable** is a function from the sample space Ω to the real numbers: it assigns a number to every outcome. Random variables are written with capital letters (X, Y, S); their particular values with lower-case letters (x, y, s).

$X = x$ denotes the *event* "X takes the value x", i.e. the set of outcomes $\{\omega \in \Omega : X(\omega) = x\}$. So $P(X = x)$ is an ordinary probability of an event.

A random variable is **discrete** if its possible values can be listed (finitely many, or a list like 0, 1, 2, ...). It is **continuous** if it can take any value in an interval (a time, a length, a temperature).

The name is a historical accident: a random variable is neither random (it's a fixed rule) nor a variable (it's a function). The randomness is in which outcome occurs.

Randomized algorithm, Las Vegas, Monte Carlo (Chapter 29)

A **randomized algorithm** makes random choices during its execution. Two families:

- **Las Vegas** algorithms are always correct; their *running time* is random. Randomized quicksort is Las Vegas: it always sorts correctly, but some runs take longer.
- **Monte Carlo** algorithms run in bounded time; their *answer* may be wrong with small probability. Repeating them independently drives the error probability down exponentially.

Mnemonic: in Las Vegas you always leave with a (correct) answer but you don't know how long you'll be there; in Monte Carlo you leave on time but might have lost.

Rational, irrational, lowest terms (Chapter 22)

- A real number is **rational** if it equals p/q for some integers p, q with $q \neq 0$. The set of rationals is written $\mathbb{Q}$ (for *quotient*).
- A real number that is not rational is **irrational**. Examples: $\sqrt{2}$, π, e.

- A fraction p/q is in **lowest terms** if p and q have no common factor greater than 1. Every rational number can be written in lowest terms (divide top and bottom by their greatest common divisor; see Chapter 23).
- A **contradiction** is a statement that is false by its form, such as $R \wedge \neg R$ ("R and not R").

Recursive definition, full binary tree (Chapter 22)

- A **recursive definition** (or inductive definition) defines a set of objects by (1) some base objects and (2) rules that build new objects from existing ones, with nothing else included.
- A **full binary tree** is defined by: (base) a single node is a full binary tree; (rule) if T_1 and T_2 are full binary trees, then a new root node with left subtree T_1 and right subtree T_2 is a full binary tree. In a full binary tree every node has 0 or 2 children. A node with 0 children is a **leaf**; one with 2 is an **internal node**.

Relation (Chapter 21)

A **(binary) relation** R from set A to set B is a subset of the Cartesian product: $R \subseteq A \times B$. We write $a\ R\ b$ for $(a, b) \in R$. A relation **on** A is a relation from A to itself. Example on $\{1, 2, 3\}$: "less than" is $\{(1, 2), (1, 3), (2, 3)\}$.

Relaxation (Chapter 25)

Relaxing an edge $u \to v$ of weight w means: if $\mathbf{dist}(u) + w < \mathbf{dist}(v)$, set $\mathbf{dist}(v) = \mathbf{dist}(u) + w$ and $\mathbf{parent}(v) = u$. It asks "is going via u better than the best route to v known so far?" The symbol ∞ (infinity) stands for "no route found yet".

Residual, model, design matrix (Chapter 33)

A **model** is a function with adjustable parameters that maps inputs to predictions; here $\hat{y} = b + wx$ with **intercept** b and **slope** (or **weight**) w. A **residual** is $r_i = y_i - \hat{y}_i$, observed minus predicted. The **sum of squared errors** is $\mathrm{SSE} = \sum_i r_i^2$; dividing by n gives the **mean squared error** (MSE). The **design matrix** X has one row per example and one column per parameter; for a line, row i is $[1,\ x_i]$, the 1 multiplying the intercept.

Residue, residue class (Chapter 23)

The **residue** of a mod m is the remainder $a \bmod m$, one of $0, 1, \ldots, m-1$. The **residue class** $[a]_m$ is the set of all integers congruent to a. For example, $[2]_{12} = \{\ldots, -10, 2, 14, 26, \ldots\}$. There are exactly m classes, and the set of them is written $\mathbb{Z}_m$ or $\mathbb{Z}/m\mathbb{Z}$.

B.18. S

Sample space and event (Chapter 29)

The **sample space**, written Ω (capital Greek *omega*) or S, is the set of all possible outcomes of an experiment.

An **event** is a subset of the sample space: a collection of outcomes. We say "event A occurs" if the outcome that actually happens is a member of A.

$P(A)$, read "the probability of A", is the probability that event A occurs.

Examples. One die: $\Omega = \{1, 2, 3, 4, 5, 6\}$; "roll is even" is the event $\{2, 4, 6\}$. One coin tossed twice: $\Omega = \{HH, HT, TH, TT\}$; "at least one head" is $\{HH, HT, TH\}$.

Sample variance and standard deviation (Chapter 31)

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2, \qquad s = \sqrt{s^2}.$$

s is the **sample standard deviation**, roughly the typical distance of a data point from the mean, in the data's units. Greek letters (μ, σ) denote **population** quantities, the true values for the whole population or process; Latin letters ($\bar{x}$, s) denote **sample** quantities computed from data.

Why $n - 1$? The deviations are measured from $\bar{x}$, which was fitted to the same data and so sits closer to the data than the true mean μ does. Dividing by n would systematically underestimate σ^2; dividing by $n - 1$ (**Bessel's correction**) exactly fixes the average bias. For large n it hardly matters.

Set, element, membership (Chapter 21)

A **set** is a well-defined collection of distinct objects, called its **elements** or **members**. We write $x \in A$ for "x is an element of A" and $x \notin A$ for "x is not". Sets are written with braces: $A = \{2, 3, 5, 7\}$. Order and repetition are irrelevant: $\{1, 2, 2, 3\} = \{3, 2, 1\}$. Two sets are **equal** when they have exactly the same elements.

The **cardinality** $|A|$ is the number of elements of a finite set: $|\{2, 3, 5, 7\}| = 4$.

Set-builder notation (Chapter 21)

Set-builder notation describes a set by a rule rather than a list:

$$\{\, x \in \mathbb{Z} \mid x > 0 \text{ and } x \text{ is even} \,\} = \{2, 4, 6, \ldots\}$$

Read the bar $\mid$ (sometimes a colon) as "such that". The left part says where elements come from and what shape they have; the right part is a predicate they must satisfy. You can also put an expression on the left: $\{\, n^2 \mid n \in \mathbb{N} \,\} = \{0, 1, 4, 9, \ldots\}$. Python's set comprehension `{n*n for n in range(10)}` is set-builder notation, almost symbol for symbol.

Sigma notation (Chapter 22)

The symbol Σ (capital Greek sigma, for "sum") abbreviates long sums:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n.$$

Read it as "the sum, for i from 1 to n, of a_i". The **index** i is a dummy variable (like a loop variable: it has no meaning outside the sum), the numbers below and above Σ are its starting and ending values, and a_i is the term, an expression in i. For example $\sum_{i=1}^{4}(2i - 1) = 1 + 3 + 5 + 7 = 16$. In

Python: `sum(2*i - 1 for i in range(1, n + 1))`. A key identity for induction is "peel off the last term":

$$\sum_{i=1}^{k+1} a_i = \left(\sum_{i=1}^{k} a_i\right) + a_{k+1}.$$

Sigma notation is covered in more depth in Volume 1, Chapter 14 .

Standard error (Chapter 31)

The **standard error** (SE) of a statistic is the standard deviation of its sampling distribution: how much the statistic would vary from sample to sample.

- For a sample mean: $\mathrm{SE}(\bar{x}) = \sigma/\sqrt{n}$, estimated by $s/\sqrt{n}$.
- For a sample proportion $\hat{p}$ (a mean of 0/1 values): $\mathrm{SE}(\hat{p}) = \sqrt{p(1-p)/n}$, estimated by $\sqrt{\hat{p}(1-\hat{p})/n}$.

Standard **deviation** describes the spread of individual data points; standard **error** describes the uncertainty of an estimate. Collecting more data doesn't change the first and shrinks the second like $1/\sqrt{n}$.

Standard number sets (Chapter 21)

$\mathbb{N}$ natural numbers $\{0, 1, 2, \ldots\}$ (some authors start at 1; computer scientists usually start at 0); $\mathbb{Z}$ integers (from German *Zahlen*, "numbers"); $\mathbb{Q}$ rationals (*quotient*); $\mathbb{R}$ reals; $\mathbb{C}$ complex numbers. See Volume 1, Chapter 1 .

Sum rule and product rule (Chapter 24)

Sum rule. If a choice can be made from pool A *or* from pool B, and no option is in both (the pools are **disjoint**, $A \cap B = \varnothing$), the number of choices is $|A| + |B|$.

Product rule. If a process has two stages, the first can be done in m ways, and *for each of those* the second can be done in n ways, the whole process can be done in $m \times n$ ways. It extends to any number of stages: $n_1 \times n_2 \times \cdots \times n_k$.

Rule of thumb: **"or" between disjoint cases → add; "and then" between stages → multiply.**

B.19. T

Theorem, lemma, corollary, conjecture (Chapter 22)

- A **theorem** is a statement that has been proved. It is usually an important result.
- A **lemma** is a smaller proved statement used as a stepping stone towards a theorem (Greek *lemma*, "something taken for granted").
- A **corollary** is a statement that follows quickly from a theorem just proved.
- A **conjecture** is a statement believed to be true but not yet proved, such as "every even number greater than 2 is the sum of two primes" (Goldbach's conjecture).

- An **axiom** is a statement accepted without proof as a starting point, such as "for any integers a, b, $a + b = b + a$".
- A **proof** is a chain of statements, each justified by axioms, definitions, earlier results or logical rules, that ends at the claim being proved.

Tree, forest, leaf, rooted tree (Chapter 25)

A **tree** is a connected graph with no cycles. A **forest** is a graph with no cycles (a disjoint union of trees). A **leaf** is a vertex of degree 1. A **rooted tree** singles out one vertex as the **root**; then every other vertex has a unique **parent** (its neighbour on the path to the root) and possibly **children**. The **depth** of a vertex is its distance from the root. A **spanning tree** of a graph G is a tree that includes every vertex of G and uses only edges of G.

Type I and type II errors, power (Chapter 31)

- **Type I error:** rejecting a true H_0 (crying wolf). Its probability is α, which you choose.
- **Type II error:** failing to reject a false H_0 (missing a real effect). Its probability is β.
- **Power** $= 1 - \beta$: the probability of detecting an effect of a given size if it really exists. Conventionally, studies are designed for 80% power.

Power increases with larger samples, larger true effects, less noisy data, and a larger α. For a fixed sample size, lowering α (fewer false alarms) raises β (more misses).

B.20. U

Union, intersection, difference, complement (Chapter 21)

For sets A, B inside a universal set U:

Operation	Definition	Logic behind it
Union $A \cup B$	$\{x \mid x \in A \vee x \in B\}$	or
Intersection $A \cap B$	$\{x \mid x \in A \wedge x \in B\}$	and
Difference $A \setminus B$ (also $A - B$)	$\{x \mid x \in A \wedge x \notin B\}$	and not
Complement $\overline{A}$ (also A^c, A')	$\{x \in U \mid x \notin A\}$	not
Symmetric difference $A \triangle B$	$(A \setminus B) \cup (B \setminus A)$	xor

Sets with no common element, $A \cap B = \varnothing$, are **disjoint**.

Union, intersection, Venn diagram (Chapter 24)

The **union** $A \cup B$ is the set of elements in A or B (or both). The **intersection** $A \cap B$ is the set of elements in both. A **Venn diagram** draws sets as overlapping regions so that each region represents one combination of "in" and "out". See Chapter 21 for the full set vocabulary.

B.21. V

Vacuous truth (Chapter 21)

A conditional whose hypothesis is false is **vacuously true**: true because there is no case that could break it. "Every unicorn in this room is purple" is vacuously true, since there are no unicorns to check. In code, `all(x > 100 for x in [])` returns `True` for the same reason.

Variance and standard deviation (Chapter 30)

The **variance** of X, with $\mu = E[X]$, is the expected squared deviation from the mean:

$$\operatorname{Var}(X) = E\big[(X - \mu)^2\big].$$

It is often written σ^2 (*sigma squared*). The **standard deviation** is its square root, $\sigma = \sqrt{\operatorname{Var}(X)}$, which has the same units as X (milliseconds rather than milliseconds squared), so it's the one to quote.

Rough meaning: σ is a typical distance between X and its mean.

B.22. W

Walk, trail, path, cycle (Chapter 25)

- A **walk** is a sequence of vertices $v_0, v_1, \ldots, v_k$ in which each consecutive pair is joined by an edge. Its **length** is the number of edges, k. Vertices and edges may repeat.
- A **trail** is a walk with no repeated edge.
- A **path** is a walk with no repeated vertex (hence no repeated edge).
- A **closed** walk ends where it starts ($v_k = v_0$). A **circuit** is a closed trail. A **cycle** is a closed walk of length at least 3 with no repeated vertex other than $v_0 = v_k$.
- The **distance** $d(u, v)$ is the length of a shortest path from u to v (or, in a weighted graph, the smallest total weight).

Appendix C: Notation and Symbols

Mathematical notation is a compressed language. This appendix is its phrasebook: for each symbol, how to say it aloud, what it means, a tiny example, and the chapter where it is properly introduced. Saying a formula aloud is half of understanding it, so the "Read aloud" column matters more than it looks.

Where a symbol has more than one common meaning, all are listed; context decides. The section Section C.13 collects the worst offenders, and Section C.14 shows how to take a dense formula apart.

C.1. Arithmetic, comparison, and approximation

Symbol	Read aloud	Meaning	Example	Ch.
$=$	"equals"	Both sides name the same value	$2 + 3 = 5$	Volume 1, Chapter 1
$\neq$	"is not equal to"	The two sides differ	$0.1 + 0.2 \neq 0.3$ in floating point	Volume 1, Chapter 1
$<$, $>$	"is less than", "is greater than"	Strict order; the small end points at the smaller number	$-3 < 2$	Volume 1, Chapter 1
$\leq$, $\geq$	"is less than or equal to", "is greater than or equal to"	Order allowing equality	$x \geq 0$	Volume 1, Chapter 5
$\approx$	"is approximately equal to"	Close in value; how close depends on context	$\pi \approx 3.1416$	Volume 1, Chapter 1
$\equiv$	"is identically equal to"; "is congruent to"; "is logically equivalent to"	(1) true for every value of the variable; (2) same remainder modulo m; (3) same truth value in every case	$(x+1)^2 \equiv x^2 + 2x + 1$; $17 \equiv 2 \pmod 5$	Volume 1, Chapter 4, Chapter 23, Chapter 21
$:=$	"is defined to be"	Defines the left side; not a claim to check	$f(x) := x^2$	Volume 1, Chapter 4

Symbol	Read aloud	Meaning	Example	Ch.
$\propto$	"is proportional to"	Equal up to a constant multiplier	$A \propto r^2$ for circles (constant π)	Volume 1, Chapter 2
$\ll$, $\gg$	"is much less than", "is much greater than"	Smaller (larger) by orders of magnitude	$10^{-16} \ll 1$	Volume 2, Chapter 32
$\pm$, $\mp$	"plus or minus", "minus or plus"	Two cases at once; $\mp$ takes the opposite sign to a $\pm$ in the same formula	$x = 3 \pm 2$ means $x = 5$ or $x = 1$	Volume 1, Chapter 8
$\times$, $\cdot$	"times"	Multiplication (also cross product and dot product for vectors, Cartesian product for sets)	$3 \times 4 = 3 \cdot 4 = 12$	Volume 1, Chapter 1
a/b, $\frac{a}{b}$, $a \div b$	"a over b", "a divided by b"	Division; a fraction	$\frac{6}{4} = 1.5$	Volume 1, Chapter 2
$a : b$	"a to b"	Ratio	$3 : 2$	Volume 1, Chapter 2
$\%$	"percent"	Per hundred: $p\% = p/100$	$15\% = 0.15$	Volume 1, Chapter 2
a^n	"a to the n", "a to the power n" (a^2 "a squared", a^3 "a cubed")	Repeated multiplication, extended to all real n	$2^{-1} = \frac{1}{2}$	Volume 1, Chapter 3

Symbol	Read aloud	Meaning	Example	Ch.
$\sqrt{a}$, $\sqrt[n]{a}$	"root a", "the n-th root of a"	The non-negative number whose square (n-th power) is a	$\sqrt[3]{8} = 2$	Volume 1, Chapter 3
$\lvert x \rvert$	"the absolute value of x", "mod x"	Distance from 0	$\lvert -7 \rvert = 7$	Volume 1, Chapter 1
$\lfloor x \rfloor$	"floor of x"	Largest integer $\leq x$	$\lfloor 2.7 \rfloor = 2$, $\lfloor -2.7 \rfloor = -3$	Volume 1, Chapter 11
$\lceil x \rceil$	"ceiling of x"	Smallest integer $\geq x$	$\lceil 2.1 \rceil = 3$	Volume 1, Chapter 11
$n!$	"n factorial"	$1 \cdot 2 \cdots n$, with $0! = 1$	$4! = 24$	Chapter 24
∞	"infinity"	Not a number: shorthand for "without bound"	$x \to \infty$, $(0, \infty)$	Volume 2, Chapter 16
$\ldots$, $\cdots$	"dot dot dot", "and so on"	Continue the evident pattern	$1 + 2 + \cdots + n$	Volume 1, Chapter 14
$\therefore$	"therefore"	Introduces a conclusion	$2x = 6 \therefore x = 3$	Chapter 22
■ or □	"QED", "end of proof"	The proof is complete		Chapter 22

C.2. Number systems and intervals

Symbol	Read aloud	Meaning	Example	Ch.
$\mathbb{N}$	"the naturals", "N"	Natural numbers $\{0, 1, 2, \ldots\}$ (some authors start at 1)	$5 \in \mathbb{N}$	Volume 1, Chapter 1
$\mathbb{Z}$	"the integers", "Z" (from German *Zahlen*)	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	$-4 \in \mathbb{Z}$	Volume 1, Chapter 1
$\mathbb{Q}$	"the rationals", "Q" (for *quotient*)	Fractions p/q with p, q integers, $q \neq 0$	$0.75 = \frac{3}{4} \in \mathbb{Q}$	Volume 1, Chapter 1
$\mathbb{R}$	"the reals", "R"	All points on the number line	$\sqrt{2} \in \mathbb{R}, \sqrt{2} \notin \mathbb{Q}$	Volume 1, Chapter 1
$\mathbb{C}$	"the complex numbers", "C"	Numbers $a + bi$ with $a, b \in \mathbb{R}$	$3 - 2i \in \mathbb{C}$	Volume 1, Chapter 15
$\mathbb{R}^n$	"R n"	Lists (vectors) of n real numbers	$(1, -2, 0.5) \in \mathbb{R}^3$	Volume 2, Chapter 26
$\mathbb{Z}_m$ or $\mathbb{Z}/m\mathbb{Z}$	"the integers mod m"	The remainders $\{0, 1, \ldots, m-1\}$ with modular arithmetic	In $\mathbb{Z}_{12}$, $9 + 5 = 2$	Chapter 23

Table 4. Interval notation

Notation	Means x satisfies	Endpoints
(a, b)	$a < x < b$	both excluded ("open")
$[a, b]$	$a \le x \le b$	both included ("closed")
$[a, b)$	$a \le x < b$	left included, right excluded ("half-open"; Python's `range`)
$(a, b]$	$a < x \le b$	left excluded, right included
(a, ∞)	$x > a$	∞ is never included
$(-\infty, b]$	$x \le b$	
$(-\infty, \infty)$	any real x	the whole of $\mathbb{R}$

A round bracket means "up to but not including"; a square bracket means "including". Some European texts write $]a, b[$ for (a, b). Do not confuse the open interval (a, b) with the point (a, b) in the plane; context tells them apart. See Volume 1, Chapter 5 .

C.3. Sets

Symbol	Read aloud	Meaning	Example	Ch.
$\{\ \}$	"the set containing ..."	Unordered collection, no repeats	$\{1,2,3\} = \{3,1,2\}$	Chapter 21
$\{x \mid P(x)\}$ or $\{x : P(x)\}$	"the set of x such that P of x"	Set-builder notation: everything satisfying a condition	$\{x \in \mathbb{Z} \mid x^2 < 5\} = \{-2, \ldots, 2\}$	Chapter 21
$\in$	"is in", "is an element of", "belongs to"	Membership	$3 \in \{1,2,3\}$	Chapter 21
$\notin$	"is not in"	Non-membership	$4 \notin \{1,2,3\}$	Chapter 21
$\subseteq$	"is a subset of"	Every element of the left is in the right	$\{1,2\} \subseteq \{1,2,3\}$	Chapter 21
$\subset$ or $\subsetneq$	"is a proper subset of"	Subset but not equal (some books use $\subset$ for $\subseteq$; check the convention)	$\mathbb{N} \subset \mathbb{Z}$	Chapter 21
$\cup$	"union", "A cup B"	In either set (or both)	$\{1,2\} \cup \{2,3\} = \{1,2,3\}$	Chapter 21
$\cap$	"intersection", "A cap B"	In both sets	$\{1,2\} \cap \{2,3\} = \{2\}$	Chapter 21
$A \setminus B$ or $A - B$	"A minus B", "A without B"	In A but not in B	$\{1,2,3\} \setminus \{2\} = \{1,3\}$	Chapter 21
A^c, $\overline{A}$, A'	"A complement"	Everything (in the universe) not in A	In $\{1,\ldots,6\}$, $\{1,2\}^c = \{3,4,5,6\}$	Chapter 21
$\emptyset$ or $\varnothing$	"the empty set"	The set with no elements	$\{1\} \cap \{2\} = \emptyset$	Chapter 21
$\lvert A \rvert$ or $\#A$	"the size of A", "the cardinality of A"	Number of elements	$\lvert\{a,b,c\}\rvert = 3$	Chapter 21

Symbol	Read aloud	Meaning	Example	Ch.
$A \times B$	"A cross B"	Cartesian product: all ordered pairs (a, b)	$\{1, 2\} \times \{x\} = \{(1, x), (2, x)\}$	Chapter 21
$\mathcal{P}(A)$	"the power set of A"	The set of all subsets of A	$\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$	Chapter 21

C.4. Logic

Symbol	Read aloud	Meaning	Example	Ch.
$\neg p$	"not p"	True exactly when p is false (code: `!p`, `not p`)	$\neg(2 > 3)$ is true	Chapter 21
$p \wedge q$	"p and q"	True when both are true (code: `&&`, `and`)	$x > 0 \wedge x < 1$	Chapter 21
$p \vee q$	"p or q"	True when at least one is true; inclusive or (code: `or`)	$x < 0 \vee x > 1$	Chapter 21
$p \oplus q$	"p xor q", "exclusive or"	True when exactly one is true (code: ^ on booleans)	$\mathbf{T} \oplus \mathbf{T}$ is false	Chapter 21
$p \to q$	"if p then q", "p implies q"	The conditional: false only when p is true and q false	$x > 2 \to x > 1$	Chapter 21
$p \leftrightarrow q$	"p if and only if q", "p iff q"	Same truth value	n even $\leftrightarrow$ n^2 even	Chapter 21
$\Rightarrow$	"implies"	Used between statements in an argument: "from this, it follows that". Many authors use $\Rightarrow$ and $\to$ interchangeably	$2x = 6 \Rightarrow x = 3$	Chapter 21
$\Leftrightarrow$	"if and only if", "is equivalent to"	Each side implies the other; an equation-solving step that can be reversed	$x + 1 = 3 \Leftrightarrow x = 2$	Chapter 21
$\forall$	"for all", "for every"	Universal quantifier	$\forall x \in \mathbb{R},\ x^2 \geq 0$	Chapter 21
$\exists$	"there exists", "for some"	Existential quantifier: at least one	$\exists x \in \mathbb{R},\ x^2 = 2$	Chapter 21
$\exists!$	"there exists a unique"	Exactly one	$\exists! x,\ 2x = 6$	Chapter 21
$\mathbf{T},\ \mathbf{F}$	"true", "false"	Truth values (also $1, 0$ or $\top, \bot$)		Chapter 21

C.5. Functions

Symbol	Read aloud	Meaning	Example	Ch.
$f(x)$	"f of x"	The output of function f at input x (not f times x)	$f(x) = x^2,\ f(3) = 9$	Volume 1, Chapter 11
$f : A \to B$	"f from A to B"	f takes inputs in A (domain) to outputs in B (codomain); a type signature	$\sqrt{\cdot} : [0, \infty) \to \mathbb{R}$	Volume 1, Chapter 11
$x \mapsto f(x)$	"x maps to f of x"	Names a function by its rule (like a lambda)	$x \mapsto 2x + 1$	Volume 1, Chapter 11
$f \circ g$	"f composed with g", "f after g", "f of g"	$(f \circ g)(x) = f(g(x))$: apply g first	$f(x) = x^2,\ g(x) = x + 1 \Rightarrow (f \circ g)(2) = 9$	Volume 1, Chapter 11
f^{-1}	"f inverse"	The function that undoes f. **Not** $1/f$	$f(x) = 2x \Rightarrow f^{-1}(x) = x/2$	Volume 1, Chapter 11
$\sin^{-1} x$, $\arcsin x$	"inverse sine of x", "arc sine of x"	The angle whose sine is x. But $\sin^2 x$ means $(\sin x)^2$	$\arcsin \frac{1}{2} = \frac{\pi}{6}$	Volume 1, Chapter 13
$\log_b x$, $\ln x$, $\log x$	"log base b of x", "natural log of x" ("ell-en x", "lin x" or "log x")	The exponent that gives x. $\ln$ is base e; bare $\log$ is base 10 in school, base e in research papers and in Python, often base 2 in CS	$\log_2 8 = 3$	Volume 1, Chapter 12
$\exp(x)$	"exp of x"	e^x, written so the exponent can be large	$\exp(0) = 1$	Volume 1, Chapter 12

Symbol	Read aloud	Meaning	Example	Ch.
Δx	"delta x"	A change (difference) in x	$\Delta x = x_2 - x_1$	Volume 1, Chapter 5
$F(x)\|_a^b$	"F of x evaluated from a to b"	$F(b) - F(a)$	$x^2\|_1^3 = 9 - 1 = 8$	Volume 2, Chapter 19
$O(g)$, $\Theta(g)$, $\Omega(g)$	"big O of g", "big Theta", "big Omega"	Grows at most / exactly / at least as fast as g, up to a constant factor	Binary search is $O(\log n)$	Volume 1, Chapter 14

C.6. Sequences and sums

Symbol	Read aloud	Meaning	Example	Ch.
a_n	"a sub n", "a n"	The n-th term of a sequence (code: `a[n]`)	$a_n = 2n \Rightarrow a_3 = 6$	Volume 1, Chapter 14
$(a_n)_{n\geq 1}$	"the sequence a n"	The whole sequence, not one term	$(1/n)_{n\geq 1}$	Volume 1, Chapter 14
$\sum_{i=1}^{n} a_i$	"the sum from i equals 1 to n of a sub i"	$a_1 + a_2 + \cdots + a_n$; a `for` loop accumulating with `+`	$\sum_{i=1}^{4} i = 10$	Volume 1, Chapter 14
$\prod_{i=1}^{n} a_i$	"the product from i equals 1 to n of a sub i"	$a_1 \cdot a_2 \cdots a_n$; the same loop accumulating with $\times$	$\prod_{i=1}^{4} i = 24$	Volume 1, Chapter 14
$\binom{n}{k}$	"n choose k"	Number of k-element subsets of an n-element set; also written $C(n,k)$, ${}^nC_k$, ${}_nC_k$, nCr	$\binom{5}{2} = 10$	Chapter 24
$P(n,r)$	"the number of permutations of n things r at a time"	Ordered selections: $n!/(n-r)!$; also ${}^nP_r$, nPr	$P(5,2) = 20$	Chapter 24

C.7. Calculus

Symbol	Read aloud	Meaning	Example	Ch.
$\lim_{x \to a} f(x)$	"the limit as x tends to a of f of x"	The value $f(x)$ approaches as x gets close to (but not equal to) a	$\lim_{x \to 0} \frac{\sin x}{x} = 1$	Volume 2, Chapter 16
$x \to a^+$, $x \to a^-$	"x tends to a from above / below"	One-sided limits	$\lim_{x \to 0^+} \frac{1}{x} = \infty$	Volume 2, Chapter 16
$\varepsilon,\ \delta$	"epsilon", "delta"	Small positive tolerances in the formal definition of a limit		Volume 2, Chapter 16
$f'(x)$	"f prime of x"	The derivative of f at x (Lagrange notation)	$f(x) = x^3 \Rightarrow f'(x) = 3x^2$	Volume 2, Chapter 17
$f''(x)$, $f^{(n)}(x)$	"f double prime of x", "the n-th derivative of f"	Second and n-th derivatives; brackets show it is not a power	$(x^3)'' = 6x$	Volume 2, Chapter 17
$\frac{dy}{dx}$	"d y d x", "the derivative of y with respect to x"	Leibniz notation: rate of change of y per unit x. Not a fraction, though it often behaves like one	$y = x^2 \Rightarrow \frac{dy}{dx} = 2x$	Volume 2, Chapter 17
$\frac{d}{dx}$	"d by d x"	The operation "differentiate with respect to x"	$\frac{d}{dx}\sin x = \cos x$	Volume 2, Chapter 17
$\dot{x}$	"x dot"	Newton notation: derivative with respect to time (physics, mechanics)	$\dot{x} = v$	Volume 2, Chapter 17

Symbol	Read aloud	Meaning	Example	Ch.
$\frac{\partial f}{\partial x}$, f_x, $\partial_x f$	"partial f by partial x", "del f del x", "f sub x"	Partial derivative: differentiate in x, holding the other variables fixed	$f = x^2 y \Rightarrow f_x = 2xy$	Volume 2, Chapter 20
∇f	"grad f", "del f", "nabla f"	The gradient: the vector of all partial derivatives	$\nabla(x^2 + y^2) = (2x, 2y)$	Volume 2, Chapter 20
$\int f(x)\,dx$	"the integral of f of x d x"	Indefinite integral: the family of antiderivatives	$\int 2x\,dx = x^2 + C$	Volume 2, Chapter 19
$\int_a^b f(x)\,dx$	"the integral from a to b of f of x d x"	Definite integral: signed area under f between a and b (a number). The $\int$ is an elongated S, for "sum"	$\int_0^1 x\,dx = \frac{1}{2}$	Volume 2, Chapter 19
dx	"d x"	Marks the variable of integration; the infinitesimal width of a strip		Volume 2, Chapter 19

C.8. Vectors and matrices

Symbol	Read aloud	Meaning	Example	Ch.
$\mathbf{v}$, $\vec{v}$, $\underline{v}$	"vector v", "v"	A vector. Print uses bold; handwriting uses an arrow or underline (the UK schools' squiggle)	$\mathbf{v} = (3, 4)$	Volume 2, Chapter 26
v_i	"v sub i"	The i-th component (code: `v[i]`, but mathematics usually counts from 1)	$v_2 = 4$	Volume 2, Chapter 26
$\|\mathbf{v}\|$	"the norm of v", "the length of v"	Length (Euclidean norm unless a subscript says otherwise)	$\|(3, 4)\| = 5$	Volume 2, Chapter 26
$\hat{\mathbf{v}}$	"v hat"	Unit vector in the direction of $\mathbf{v}$; $\hat{\mathbf{i}}, \hat{\mathbf{j}}, \hat{\mathbf{k}}$ are the axis directions	$\hat{\mathbf{v}} = (0.6, 0.8)$	Volume 2, Chapter 26
$\mathbf{u} \cdot \mathbf{v}$	"u dot v"	Dot (scalar, inner) product; also $\langle \mathbf{u}, \mathbf{v} \rangle$	$(1, 2) \cdot (3, 4) = 11$	Volume 2, Chapter 26
$\mathbf{u} \times \mathbf{v}$	"u cross v"	Cross product (3D only): a vector perpendicular to both	$\hat{\mathbf{i}} \times \hat{\mathbf{j}} = \hat{\mathbf{k}}$	Volume 2, Chapter 26
$\perp$	"is perpendicular to", "is orthogonal to"	At right angles (dot product 0)	$(1, 0) \perp (0, 1)$	Volume 2, Chapter 26
A, a_{ij}	"matrix A", "a i j"	Matrices are capital letters; a_{ij} is the entry in row i, column j		Volume 2, Chapter 27

Symbol	Read aloud	Meaning	Example	Ch.
A^T, $A^\top$, A'	"A transpose"	Rows become columns: $(A^T)_{ij} = a_{ji}$	$\begin{pmatrix}1 & 2\end{pmatrix}^T = \begin{pmatrix}1\\2\end{pmatrix}$	Volume 2, Chapter 27
A^{-1}	"A inverse"	The matrix with $AA^{-1} = I$, when it exists		Volume 2, Chapter 27
I, I_n	"the identity", "I n"	Square matrix with 1s on the diagonal, 0s elsewhere	$I\mathbf{x} = \mathbf{x}$	Volume 2, Chapter 27
$\det A$, $\lvert A\rvert$	"the determinant of A", "det A"	Signed area/volume scale factor of A	$\det \begin{pmatrix}2 & 0\\0 & 3\end{pmatrix} = 6$	Volume 2, Chapter 27
$\operatorname{tr} A$	"the trace of A"	Sum of the diagonal entries	$\operatorname{tr} I_3 = 3$	Volume 2, Chapter 28
λ	"lambda"	An eigenvalue: $A\mathbf{v} = \lambda\mathbf{v}$		Volume 2, Chapter 28

C.9. Probability and statistics

Symbol	Read aloud	Meaning	Example	Ch.
Ω	"omega", "the sample space"	The set of all possible outcomes	Coin: $\Omega = \{H, T\}$	Chapter 29
$P(A)$, $\Pr(A)$	"the probability of A"	A number in $[0, 1]$	$P(\text{even die}) = \frac{1}{2}$	Chapter 29
$P(A \mid B)$	"the probability of A given B"	Probability of A once you know B happened	$P(\text{six} \mid \text{even}) = \frac{1}{3}$	Chapter 29
X, x	"capital X", "little x"	Capital: a random variable; lower case: a particular value it takes	$P(X = x)$	Chapter 29
$X \sim D$	"X is distributed as D", "X follows D"	Names the distribution of X	$X \sim N(0, 1)$	Chapter 30
$E[X]$, $\mathbb{E}[X]$	"the expected value of X", "the expectation of X"	Probability-weighted average	Fair die: $E[X] = 3.5$	Chapter 30
$\text{Var}(X)$	"the variance of X"	$E[(X - \mu)^2]$: average squared distance from the mean	Fair coin (0/1): 0.25	Chapter 30
μ	"mu"	Population (true) mean		Chapter 30
σ, σ^2	"sigma", "sigma squared"	Population standard deviation, variance		Chapter 30
$\text{Cov}(X, Y)$, ρ	"the covariance of X and Y", "rho"	How two variables vary together; ρ is the correlation		Chapter 30
$\bar{x}$	"x bar"	Sample mean	$\overline{\{2, 4, 9\}} = 5$	Chapter 31
s, s^2	"s", "s squared"	Sample standard deviation, sample variance		Chapter 31

Symbol	Read aloud	Meaning	Example	Ch.
$\hat{p}, \hat{y}, \hat{\theta}$	"p hat", "y hat", "theta hat"	An estimate or prediction of the un-hatted quantity	45 of 100 say yes: $\hat{p} = 0.45$	Chapter 31
H_0, H_1	"H nought", "H one"	Null and alternative hypotheses	$H_0 : \mu = 100$	Chapter 31
α	"alpha"	Significance level of a test	$\alpha = 0.05$	Chapter 31
z^*, t^*	"z star", "t star"	Critical values for confidence intervals	$z^* = 1.96$ for 95%	Chapter 31

C.10. Number theory and complex numbers

Symbol	Read aloud	Meaning	Example	Ch.
$a \mid b$	"a divides b"	b is a multiple of a (a statement, not a number)	$3 \mid 12, 5 \nmid 12$	Volume 1, Chapter 1
$a \bmod m$	"a mod m"	The remainder on dividing a by m (code: `a % m`)	$17 \bmod 5 = 2$	Chapter 23
$a \equiv b \pmod m$	"a is congruent to b mod m"	$m \mid (a - b)$: same remainder	$38 \equiv 2 \pmod{12}$	Chapter 23
$\gcd(a, b)$, $\operatorname{lcm}(a, b)$	"g c d of a and b", "l c m"	Greatest common divisor (UK: HCF), least common multiple	$\gcd(12, 18) = 6$	Volume 1, Chapter 1
$\varphi(n)$	"phi of n", "Euler's totient of n"	How many of $1, \ldots, n$ are coprime to n	$\varphi(10) = 4$	Chapter 23
i	"i"	$\sqrt{-1}$; engineers write j, as does Python (`1j`)	$i^2 = -1$	Volume 1, Chapter 15
$\bar{z}$, z^*	"z bar", "the conjugate of z"	Flip the sign of the imaginary part	$\overline{3 + 4i} = 3 - 4i$	Volume 1, Chapter 15
$\lvert z \rvert$, $\arg z$	"mod z", "arg z"	Modulus (length) and argument (angle) of z	$\lvert 3 + 4i \rvert = 5$	Volume 1, Chapter 15
$\operatorname{Re} z$, $\operatorname{Im} z$	"real part of z", "imaginary part of z"	a and b in $z = a + bi$	$\operatorname{Im}(3 + 4i) = 4$	Volume 1, Chapter 15

C.11. Optimisation and machine learning

Symbol	Read aloud	Meaning	Example	Ch.
$\min_x f(x)$, $\max_x f(x)$	"the minimum over x of f of x"	The smallest (largest) *value* of f	$\min_x (x-3)^2 + 1 = 1$	Volume 2, Chapter 18
$\operatorname{argmin}_x f(x)$, $\operatorname{argmax}_x f(x)$	"the arg min over x of f of x"	The *input* x that achieves the minimum (maximum)	$\operatorname{argmin}_x (x-3)^2 + 1 = 3$	Chapter 33
θ^*, x^*	"theta star"	The optimal value of a parameter		Chapter 33
η	"eta"	Learning rate (step size) in gradient descent	$\eta = 0.01$	Volume 2, Chapter 20
$\leftarrow$	"becomes", "gets"	Assignment in pseudocode (code: =)	$x \leftarrow x - \eta f'(x)$	Volume 2, Chapter 32

C.12. The Greek alphabet

Greek letters are used because the Latin ones run out, and certain letters have settled into conventional jobs. Knowing the conventions lets you guess what a symbol means before you're told.

Lower	Upper	Name	Common uses
α	A	alpha	Angles; significance level in statistics; coefficients; sometimes learning rate
β	B	beta	Angles; regression coefficients; Beta distribution; momentum in optimisation
γ	Γ	gamma	Angles; Euler-Mascheroni constant $\gamma \approx 0.5772$; discount factor in reinforcement learning. Γ: gamma function ($\Gamma(n) = (n-1)!$)
δ	Δ	delta	δ: small quantity (ε-δ limits), Kronecker delta. Δ: change in (Δx), discriminant
ε, ϵ	E	epsilon	Small positive quantity; error or noise term; machine epsilon
ζ	Z	zeta	Riemann zeta function $\zeta(s) = \sum 1/n^s$; damping ratio
η	H	eta	Learning rate; efficiency
θ, ϑ	Θ	theta	Angles; model parameters. Θ: asymptotically tight bound in algorithm analysis
ι	I	iota	Rare in mathematics; integer ranges in APL and Go
κ	K	kappa	Curvature; condition number of a matrix; Cohen's kappa
λ	Λ	lambda	Eigenvalues; rate of a Poisson or exponential distribution; Lagrange multiplier; wavelength; anonymous functions (lambda calculus). Λ: diagonal matrix of eigenvalues
μ	M	mu	Mean; micro- (10^{-6}); coefficient of friction
ν	N	nu	Degrees of freedom; frequency
ξ	Ξ	xi	Auxiliary or random variables; the point "somewhere in between" in error formulas
o	O	omicron	Almost never used (indistinguishable from Latin o)
π	Π	pi	$3.14159\ldots$; permutations; policy in reinforcement learning; prime-counting function. Π: product
ρ	P	rho	Correlation; density; radius in polar coordinates

Lower	Upper	Name	Common uses
σ	Σ	sigma	Standard deviation; sigmoid function; singular values; permutations. Σ: summation; covariance matrix; diagonal matrix of singular values in the SVD
τ	T	tau	Time constant; torque; $\tau = 2\pi$; Kendall's rank correlation
υ	Υ	upsilon	Rare
φ, ϕ	Φ	phi	Angles (especially in spherical coordinates); Euler's totient; golden ratio $\varphi \approx 1.618$; standard normal density. Φ: standard normal cumulative distribution function
χ	X	chi	Chi-squared distribution χ^2; characteristic (indicator) function
ψ	Ψ	psi	Wave functions in physics; auxiliary angles and functions
ω	Ω	omega	Angular frequency; complex cube root of unity; an outcome. Ω: sample space; asymptotic lower bound; ohms

Pronunciation: most English speakers say "pie" (π), "fie" or "fee" (φ), "kye" (χ), "zay-ta" or "zee-ta" (ζ), "mew" (μ), "new" (ν), "ksee", "zye" or "sigh" (ξ). US and UK speakers differ on "zee-ta"/"zay-ta" and "bay-ta"/"bee-ta"; either is fine.

C.13. Same symbol, different meanings

Notation is reused shamelessly. When a symbol seems to make no sense, check whether it has switched jobs.

Symbol	Meanings
$\lvert\cdot\rvert$ (vertical bars)	Absolute value $\lvert x\rvert$; modulus of a complex number $\lvert z\rvert$; size of a set $\lvert A\rvert$; determinant $\lvert A\rvert$. A single bar $\mid$ means "divides" ($3 \mid 12$), "such that" ($\{x \mid x > 0\}$) or "given" ($P(A \mid B)$)
Prime $'$	Derivative (f'); a related but different object (A', "A prime"); complement (A'); transpose in statistics (X'); feet and minutes ($5'$)
Bar $\bar{x}$	Mean ($\bar{x}$); complex conjugate ($\bar{z}$); complement ($\overline{A}$); repeating decimal ($0.\overline{3}$); logical NOT in circuit diagrams
Hat $\hat{x}$	Estimate ($\hat{\theta}$); prediction ($\hat{y}$); unit vector ($\hat{\mathbf{v}}$)
Superscript -1	Inverse function (f^{-1}); reciprocal ($x^{-1} = 1/x$); inverse matrix (A^{-1}); modular inverse ($3^{-1} \equiv 5 \pmod 7$)
Superscript 2 on a function	$\sin^2 x = (\sin x)^2$, but $f^2(x)$ sometimes means $f(f(x))$; read the author's conventions
Σ	Summation; covariance matrix; matrix of singular values
π	The constant; a permutation; a policy; the prime-counting function
e	Euler's number $2.71828\ldots$; an event; an edge in a graph; an identity element; scientific notation in code (`1e-9` means 10^{-9})
$\times$	Multiplication; cross product; Cartesian product; dimensions (a 3×4 matrix)
$\equiv$	Identity; congruence; logical equivalence
(a, b)	An open interval; a point in the plane; an ordered pair; $\gcd(a, b)$ in older number-theory texts

C.14. How to read a formula

A dense formula is code with one-letter variable names and no comments. Read it the way you would read unfamiliar code: find the overall shape, identify every name, then trace the evaluation from the inside out.

Example 53. Procedure — decoding a formula

Input: a formula you don't immediately understand.

Steps:

1. **Find the top-level shape.** What is being defined or claimed? Is it "name = expression", an equation to solve, an inequality, an update rule?
2. **Name every symbol.** For each letter, decide: constant, variable, index, function, or operator? Use this appendix and the surrounding text. Which symbols are *bound* (indices of a $\sum$, the variable of an $\int$, the x under an argmin) and which are *free* (inputs the result depends on)?
3. **Read from the inside out**, as an interpreter would evaluate it.
4. **Say it aloud** in words, then in plain English.
5. **Translate it into code.**
6. **Try a tiny example** small enough to do by hand.

Output: a sentence you could say to a colleague, and a few lines of code that compute it.

C.14.1. Decoding 1: sample variance

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Shape. A definition: the name s^2 equals (a fraction) times (a sum).

Symbols. n is the number of data values (free). x_i is the i-th data value. $\bar{x}$ ("x bar") is their mean. i is a bound index: it exists only inside the $\sum$, which runs it from 1 to n. s^2 is a single name, "s squared": it is the square of the standard deviation s.

Inside out. $x_i - \bar{x}$ is how far value i is from the mean (its *deviation*). Squaring makes it non-negative and punishes large deviations more. The $\sum$ adds the squared deviations for every data value. Finally multiply by $\frac{1}{n-1}$: an average, except dividing by $n-1$ rather than n (the reason is in Chapter 31).

Aloud. "s squared equals one over n minus one, times the sum from i equals 1 to n of x sub i minus x bar, all squared."

Plain English. "The sample variance is (almost) the average squared distance of the data from its mean."

Code and tiny example. For the data 2, 4, 6: the mean is 4, the deviations are $-2, 0, 2$, their squares are $4, 0, 4$, the sum is 8, and $8/(3-1) = 4$.

PYTHON

```
xs = [2, 4, 6]
n = len(xs)
xbar = sum(xs) / n
s2 = sum((x - xbar) ** 2 for x in xs) / (n - 1)
print(s2)
```

```
4.0
```

C.14.2. Decoding 2: fitting a model by least squares

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \mathbb{R}^d}{\operatorname{argmin}} \; \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \boldsymbol{\theta}^T \mathbf{x}_i \right)^2$$

Shape. A definition: $\hat{\boldsymbol{\theta}}$ is the result of an argmin, that is, the input that makes some expression smallest.

Symbols. The hat says $\hat{\boldsymbol{\theta}}$ is an *estimate*. Bold says $\boldsymbol{\theta}$ and $\mathbf{x}_i$ are vectors. $\boldsymbol{\theta} \in \mathbb{R}^d$ says $\boldsymbol{\theta}$ is a list of d real numbers (the model's parameters) and that the search ranges over all such lists; $\boldsymbol{\theta}$ is bound by the argmin. $\mathbf{x}_i$ is the i-th input (a feature vector) and y_i the matching observed output. $\boldsymbol{\theta}^T \mathbf{x}_i$ is a transpose times a vector, which is just the dot product $\boldsymbol{\theta} \cdot \mathbf{x}_i$: the model's prediction for example i.

Inside out. $y_i - \boldsymbol{\theta}^T \mathbf{x}_i$ is the *residual*, the truth minus the prediction. Square it, sum over all n examples, divide by n: the mean squared error of the model with parameters $\boldsymbol{\theta}$. The argmin then asks for the parameter vector that makes this error smallest.

Aloud. "Theta hat equals the arg min, over theta in R d, of one over n times the sum from i equals 1 to n of y sub i minus theta transpose x sub i, squared."

Plain English. "Our fitted parameters are whichever parameters make the average squared prediction error on the data as small as possible." This is linear regression (Chapter 31); the argmin is found either exactly by linear algebra (Section 33.2) or by gradient descent (Volume 2, Chapter 20 , Chapter 33).

Code and tiny example. With $d = 1$ and two data points $(x, y) = (1, 2)$ and $(2, 4)$, the loss is $\frac{1}{2}\left((2 - \theta)^2 + (4 - 2\theta)^2\right)$, which is zero at $\theta = 2$, so $\hat{\theta} = 2$. A crude argmin by search confirms it:

PYTHON

```
X = [[1.0], [2.0]]
Y = [2.0, 4.0]

def loss(theta):
    n = len(X)
    return sum((y - sum(t * xj for t, xj in zip(theta, x))) ** 2
               for x, y in zip(X, Y)) / n

candidates = [[k / 100] for k in range(0, 501)]
theta_hat = min(candidates, key=loss)   # argmin by brute force
print(theta_hat, loss(theta_hat))
```

```
[2.0] 0.0
```

Notice that Python's `min(…, key=loss)` returns the *argument*, not the minimum value: it is an argmin. `loss(theta_hat)` is the min.

Appendix D: The US Mathematics Curriculum

If you were educated in the UK, the American system can be baffling: maths is called "math", it is taught as a sequence of separately named year-long courses rather than as one two-year syllabus, and a fifteen-year-old can be "doing Algebra II" or "in AP Calc" depending on which track they are on. This appendix is a translation guide. It explains the structure, compares it with GCSE and A-level, and maps each US course to the chapters of this book that cover it.



Read this as a cross-reference, not a rulebook

The United States has no national curriculum. Each of the 50 states sets its own standards, each of roughly 13,000 school districts chooses its own courses, textbooks and pacing, and individual schools offer different tracks. Everything below describes the *typical* pattern. Course names, grade levels and content vary from place to place; when it matters (for example, when choosing a course for next year), check your school's own course catalogue.

D.1. Grades, ages, and school levels

US school years are called **grades**. A child usually starts **kindergarten** (K) in the year they turn 5 or 6 (the cut-off date varies by state) and finishes **12th grade** at 17 or 18. Because the English Reception year comes before Year 1, US grade n corresponds roughly to English Year $n + 1$.

US grade	Typical age during the year	School level	Rough English equivalent
Pre-K	4-5	Preschool	Reception
K	5-6	Elementary school	Year 1
1-5	6-11	Elementary school	Years 2-6 (KS1-KS2)
6	11-12	Middle school (sometimes elementary)	Year 7
7-8	12-14	Middle school or junior high	Years 8-9 (KS3)
9 ("freshman")	14-15	High school	Year 10
10 ("sophomore")	15-16	High school	Year 11 (GCSE exams)
11 ("junior")	16-17	High school	Year 12 (Lower Sixth)
12 ("senior")	17-18	High school	Year 13 (Upper Sixth; A-level exams)

So a US 15-year-old is usually in 9th or 10th grade, the equivalent of the two GCSE years.

D.2. How US high-school mathematics is organised

The biggest difference from the UK is structural.

Courses, not syllabuses. In England, mathematics from 14 to 16 is one two-year GCSE course ending in national exams, and from 16 to 18 one two-year A-level course ending in national exams. In the US, each school year is a separately named **course** (Algebra I, Geometry, Algebra II, …), each lasting one year, each graded by the teacher, and each recorded on the student's **transcript**.

Continuous assessment. The course grade (A to F) is built from homework, quizzes, unit tests and a final exam set by the school. These grades feed into a **grade point average** (GPA), which universities weigh heavily. There is no national exit exam in mathematics, although many states set **end-of-course** (EOC) tests, especially for Algebra I.

Breadth until 18. US students keep studying six or seven subjects until they graduate. There is no equivalent of choosing three or four A-levels at 16. High schools typically require three or four years of mathematics to graduate (commonly including Algebra I and Geometry, often Algebra II), and selective universities expect four years, ideally reaching Precalculus or Calculus.

Tracks. Students in the same grade can be in very different courses. Placement into an **accelerated** track usually happens around grades 6-7 and determines how far a student gets by 12th grade.

D.3. The typical course sequence

Grade	Standard track	Accelerated track	Highly accelerated
6	Math 6	Math 6 (advanced)	Accelerated Math 6
7	Math 7	Accelerated Math 7 / Pre-Algebra	Pre-Algebra or Algebra I
8	Math 8 / Pre-Algebra	Algebra I	Algebra I or Geometry
9	Algebra I	Geometry (Honors)	Geometry or Algebra II (Honors)
10	Geometry	Algebra II (Honors)	Algebra II or Precalculus (Honors)
11	Algebra II	Precalculus (Honors or AP)	AP Calculus BC
12	Precalculus, Statistics, or another elective	AP Calculus AB or BC; AP Statistics	Multivariable Calculus, Linear Algebra (often via dual enrollment); AP Statistics

The "standard" route is the classic sequence

$$\textbf{Pre-Algebra} \to \textbf{Algebra I} \to \textbf{Geometry} \to \textbf{Algebra II} \to \textbf{Precalculus} \to \textbf{Calculus}$$

and the tracks differ mainly in *when* a student starts it. Taking Algebra I in 8th grade (rather than 9th) is the usual gateway to reaching calculus in high school.

Integrated pathways. A minority of states and districts teach **Integrated Math I, II, III** (sometimes called Math 1-3 or Secondary Math I-III) instead of Algebra I / Geometry / Algebra II. Each year mixes algebra, geometry, functions and statistics, which is closer to the UK approach. The total content over three years is roughly the same.



Why does Geometry get a whole year?

The US "layer-cake" sequence, with a year of Euclidean geometry sandwiched between two years of algebra, dates back to curriculum recommendations made in the 1890s, when Euclid was still the model of rigorous thinking. It survives today, and it has an upside: the US Geometry course is where most American students first write formal proofs, often in a distinctive **two-column** format (statement on the left, reason on the right). A typical UK student meets far less formal proof before A-level.

D.4. Common Core, in brief

The **Common Core State Standards for Mathematics** (2010) are a set of grade-by-grade expectations that most states adopted in the early 2010s. A few states never adopted them (Texas and Virginia, for example, have their own standards), and several have since revised or renamed them, but most state standards still closely resemble Common Core.

Key points for a parent:

- They are **standards** (what a student should be able to do by the end of each grade), not a curriculum or a textbook. Districts choose their own materials.
- Grades K-8 are specified year by year. High school is specified by topic area (Number and Quantity, Algebra, Functions, Modeling, Geometry, Statistics and Probability), so the same standards can be delivered as the traditional sequence or as an integrated one.
- Standards marked (+) are extra topics for students heading towards calculus and STEM subjects, such as complex-number arithmetic, vectors and matrices.
- Eight **Standards for Mathematical Practice** run through every grade: make sense of problems and persevere; reason abstractly and quantitatively; construct viable arguments; model with mathematics; use tools strategically; attend to precision; look for and use structure; look for regularity in repeated reasoning.
- What parents most often notice is the emphasis on multiple strategies and representations (number lines, area models for multiplication, "explain your reasoning"). That emphasis is close in spirit to this book's insistence on connecting words, symbols, pictures and computation.

D.5. Honors, AP, and IB

Honors. An Honors course covers the same course name at greater depth and pace, with harder problems. Many schools give Honors and AP grades extra weight in the GPA.

Advanced Placement (AP). AP courses follow a national syllabus set by the College Board and end in a standardised exam in May, scored 1 to 5. Many universities give credit or advanced placement for a 4 or 5 (some for a 3). AP is the closest US analogue to an A-level: a national, externally assessed exam in one subject. The AP mathematics courses are:

Course	Content	Chapters
AP Precalculus	Polynomial and rational functions; exponential and logarithmic functions; trigonometric and polar functions. (Parametric functions, vectors and matrices appear in the course but not on the exam.)	Volume 1, Chapter 11 , Volume 1, Chapter 7 , Volume 1, Chapter 12 , Volume 1, Chapter 13
AP Calculus AB	Limits and continuity; derivatives and their rules; implicit differentiation; related rates; L'Hôpital's rule; optimisation and curve sketching; Riemann sums; definite and indefinite integrals; the Fundamental Theorem; integration by substitution; slope fields and separable differential equations; area and volume. Roughly a first-semester university calculus course.	Volume 2, Chapter 16 , Volume 2, Chapter 17 , Volume 2, Chapter 18 , Volume 2, Chapter 19
AP Calculus BC	Everything in AB, plus integration by parts and partial fractions; improper integrals; Euler's method; logistic growth; parametric, polar and vector-valued functions; infinite series (convergence tests, Taylor and Maclaurin series). Roughly two semesters of university calculus. The BC exam also reports an AB sub-score.	As AB, plus Volume 1, Chapter 14 , Volume 2, Chapter 32
AP Statistics	Exploring and describing data; sampling and experimental design; probability; random variables; binomial and geometric distributions; the normal distribution; sampling distributions; confidence intervals and significance tests (means, proportions, chi-square, regression slope). Roughly an introductory, non-calculus university statistics course.	Chapter 29, Chapter 30, Chapter 31, Chapter 24

Related AP courses that use mathematics heavily: **AP Computer Science A** (Java programming), **AP Computer Science Principles**, **AP Physics 1** (algebra-based mechanics) and **AP Physics C: Mechanics** (calculus-based mechanics).

International Baccalaureate (IB). Some US high schools offer the two-year IB Diploma in grades 11-12. Its mathematics comes in two flavours, **Analysis and Approaches** (more pure, calculus-heavy) and **Applications and Interpretation** (more statistics and modelling), each at Standard Level (SL) or Higher Level (HL). IB Analysis and Approaches HL is roughly comparable to A-level Mathematics plus some Further Mathematics.

Dual enrollment. High-school students can take courses at a local community college or university, earning credit towards both their high-school diploma and a university degree. This is how strong students continue after AP Calculus BC: **Multivariable Calculus**, **Linear Algebra**, **Differential Equations** and **Discrete Mathematics** are the usual next courses.

D.6. SAT and ACT mathematics

Most US universities consider (and some require) a standardised admissions test, the **SAT** or the **ACT**. Both include mathematics, and both test the content of Algebra I, Geometry and parts of Algebra II, with no calculus. Their formats change from time to time, so check the current specifications, but at the time of writing:

- **SAT Math** (College Board): a computer-adaptive section in two modules, 44 questions in 70 minutes, scored 200-800. A calculator (including a built-in graphing calculator) is allowed throughout. Four content areas: Algebra (linear equations, inequalities and systems); Advanced Math (quadratics, polynomials, exponentials, function notation); Problem-Solving and Data Analysis (ratios, percentages, units, statistics, probability); Geometry and Trigonometry. Relevant chapters: Volume 1, Chapter 2 to Volume 1, Chapter 10 , the start of Volume 1, Chapter 13 , and the descriptive parts of Chapter 31.
- **ACT Math**: 45 multiple-choice questions in 50 minutes, scored 1-36, calculator allowed. Slightly broader than the SAT, with occasional logarithms, matrices, complex numbers and trigonometry from Algebra II and Precalculus.
- The **PSAT/NMSQT**, taken in 10th or 11th grade, is a shorter version of the SAT and the qualifying test for National Merit Scholarships.

D.7. The UK and US systems side by side

UK stage	Age	Closest US counterpart	Chapters
Key Stage 3 (Years 7-9)	11-14	Math 6, 7, 8 and Pre-Algebra; Algebra I for accelerated students	Volume 1, Chapter 1 to Volume 1, Chapter 5
GCSE Mathematics, Foundation tier	14-16	Pre-Algebra and Algebra I, with basic geometry and statistics	Volume 1, Chapter 1 to Volume 1, Chapter 6 , Volume 1, Chapter 9
GCSE Mathematics, Higher tier	14-16	Algebra I plus Geometry plus parts of Algebra II: quadratics, simultaneous equations, functions, right-triangle trigonometry, sine and cosine rules, circle theorems, basic vectors, probability. No calculus beyond informal gradients and areas under graphs	Volume 1, Chapter 4 to Volume 1, Chapter 10 , start of Volume 1, Chapter 13 , Chapter 29
Level 2 Further Maths / Additional Maths	15-16	Honors Algebra II and early Precalculus	Volume 1, Chapter 7 , Volume 1, Chapter 11 , Volume 1, Chapter 13
A-level Mathematics (pure)	16-18	Algebra II, Precalculus and most of AP Calculus BC: proof, functions, logs and exponentials, trigonometric identities, sequences and series, differentiation and integration (including by parts and substitution), parametric equations, simple differential equations, numerical methods (Newton-Raphson, trapezium rule)	Volume 1, Chapter 11 to Volume 2, Chapter 19 , Chapter 22, Volume 2, Chapter 32
A-level Mathematics (statistics)	16-18	Much of AP Statistics: sampling, binomial and normal distributions, hypothesis testing, correlation and regression	Chapter 29 to Chapter 31
A-level Mathematics (mechanics)	16-18	Taught in *physics* in the US: AP Physics 1, or AP Physics C: Mechanics for the calculus-based version	Volume 2, Chapter 26 , Volume 2, Chapter 17 , Volume 2, Chapter 19

UK stage	Age	Closest US counterpart	Chapters
A-level Further Mathematics	16-18	Beyond the AP courses: complex numbers (including De Moivre), matrices, proof by induction, polar coordinates, hyperbolic functions, series, further calculus and differential equations. Comparable to first-year university Linear Algebra and Differential Equations, reached in the US by dual enrollment	Volume 1, Chapter 15 , Volume 2, Chapter 27 , Volume 2, Chapter 28 , Chapter 22, Volume 1, Chapter 14

Some broad differences are worth keeping in mind.

- **Specialisation.** An A-level student studies mathematics for a large fraction of the school week from 16. A US student spends one period a day on it. In compensation, the US student usually also studies English, history, a science and a language to 18.
- **Calculus timing.** An A-level student meets calculus at 16-17. A US student on the standard track may not meet it until university; an accelerated student meets it at 16-17, much like the A-level student.
- **Proof and geometry.** US students meet formal two-column proof earlier (in Geometry, typically at 14-15). UK students meet more algebraic manipulation earlier.
- **Statistics.** Both systems now include it, but in the US it is often a separate course (AP Statistics) rather than a strand woven through every year.



What happened to "Maths with Mechanics"?

In the 1980s and early 1990s many English A-level students took **Mathematics with Mechanics**, or **Pure and Applied Mathematics**: pure mathematics (algebra, trigonometry, calculus) alongside applied mathematics, which in practice meant Newtonian mechanics (kinematics, forces, Newton's laws, projectiles, moments, energy). The US has no single course like it. The pure half corresponds to Algebra II, Precalculus and AP Calculus; the mechanics half lives in the physics department, where **AP Physics C: Mechanics** is the closest match because it uses calculus.

D.8. Terminology: UK and US words for the same things

UK	US	Notes
maths	math	
Year 11	10th grade	Subtract one from the English year number
BIDMAS / BODMAS	PEMDAS	US mnemonic "Please Excuse My Dear Aunt Sally": Parentheses, Exponents, Multiplication/Division, Addition/Subtraction
brackets	parentheses	US "brackets" usually means square brackets [ ]
highest common factor (HCF)	greatest common factor (GCF) or greatest common divisor (GCD)	
index, indices, powers	exponent(s)	
surd	radical	
standard form (3.2×10^5)	scientific notation	US "standard form" means a line written as $Ax + By = C$, or a number written out in full
$y = mx + c$	$y = mx + b$	The y-intercept is called b in the US
gradient (of a line)	slope	
simultaneous equations	system of equations	
recurring decimal	repeating decimal	
modulus $\lvert x \rvert$	absolute value	"Modulus" is still used for complex numbers
trapezium	trapezoid	Same shape: a quadrilateral with (at least) one pair of parallel sides. Confusingly, in older usage each country's *other* word meant a quadrilateral with no parallel sides
turning point	vertex (of a parabola), extremum	

UK	US	Notes
stationary point	critical point	US "critical point" also includes points where the derivative is undefined
sine rule, cosine rule	law of sines, law of cosines	
Pythagoras' theorem	the Pythagorean theorem	
Newton-Raphson	Newton’s method	
trapezium rule	trapezoidal rule	
sum to infinity	sum of an infinite (geometric) series	
differentiation from first principles	using the limit definition of the derivative	
nought	zero	
marks	points	

D.9. Course-by-course map to this book

Course	Typical grade (standard / accelerated)	Key topics	Chapters
Elementary math	K-5	Place value, the four operations, multiplication tables, fractions (from grade 3), decimals, measurement, simple geometry, simple data displays	Volume 1, Chapter 1 , Volume 1, Chapter 2 , start of Volume 1, Chapter 9
Math 6 and Math 7	6-7	Ratios, rates and percentages; negative numbers; expressions; one-variable equations and inequalities; area, surface area and volume; introductory statistics and probability	Volume 1, Chapter 1 , Volume 1, Chapter 2 , Volume 1, Chapter 4 , start of Volume 1, Chapter 5 , Volume 1, Chapter 9
Math 8 / Pre-Algebra	8 / 7	Exponents and scientific notation; square roots and irrational numbers; linear equations and functions; introductory systems; the Pythagorean theorem; transformations	Volume 1, Chapter 3 , Volume 1, Chapter 5 , Volume 1, Chapter 6 , Volume 1, Chapter 11 , Volume 1, Chapter 9
Algebra I	9 / 8	Linear equations and inequalities; slope and lines; functions; systems; exponent rules; exponential functions (introduction); adding, multiplying and factoring polynomials; quadratics; data and scatter plots	Volume 1, Chapter 4 , Volume 1, Chapter 5 , Volume 1, Chapter 6 , start of Volume 1, Chapter 7 , Volume 1, Chapter 8 , Volume 1, Chapter 11
Geometry	10 / 9	Congruence, similarity and transformations; two-column proof; right-triangle trigonometry; circles; coordinate geometry; area and volume	Volume 1, Chapter 9 , Volume 1, Chapter 10 , start of Volume 1, Chapter 13 , Chapter 22

Course	Typical grade (standard / accelerated)	Key topics	Chapters
Algebra II	11 / 10	Polynomial and rational functions; radicals; complex numbers; exponential and logarithmic functions; sequences and series; inverse functions and transformations; often the unit circle and trigonometric functions; probability and statistics	Volume 1, Chapter 7 , Volume 1, Chapter 11 , Volume 1, Chapter 12 , Volume 1, Chapter 15 , Volume 1, Chapter 14 , Volume 1, Chapter 13 , Chapter 29
Precalculus (including AP Precalculus)	12 / 11	Functions in depth; trigonometric functions, identities and equations; polar coordinates and complex numbers in polar form; vectors; sometimes matrices and conic sections; sequences, series and an introduction to limits	Volume 1, Chapter 11 , Volume 1, Chapter 13 , Volume 1, Chapter 15 , Volume 1, Chapter 10 , Volume 1, Chapter 14 , Volume 2, Chapter 26 , Volume 2, Chapter 27 , Volume 2, Chapter 16
AP Calculus AB	12 / 11	See the AP table above	Volume 2, Chapter 16 , Volume 2, Chapter 17 , Volume 2, Chapter 18 , Volume 2, Chapter 19
AP Calculus BC	12 / 11	AB plus series, parametric and polar calculus, further integration	As AB, plus Volume 1, Chapter 14 , Volume 2, Chapter 32
AP Statistics (or non-AP Statistics)	11-12	Data, sampling and experiments, probability, distributions, inference	Chapter 24, Chapter 29, Chapter 30, Chapter 31
Discrete Mathematics (occasional high-school elective; standard at university)	12 / university	Logic, sets, proof and induction, number theory, counting, graphs	Chapter 21, Chapter 22, Chapter 23, Chapter 24, Chapter 25

Course	Typical grade (standard / accelerated)	Key topics	Chapters
Multivariable Calculus (Calculus III)	post-BC / university	Partial derivatives, gradients, optimisation in several variables, multiple integrals, vector calculus	Volume 2, Chapter 20 (the book stops before multiple integrals and vector calculus)
Linear Algebra	post-BC / university	Vectors, matrices, systems, determinants, vector spaces, eigenvalues, orthogonality, the SVD	Volume 1, Chapter 6 , Volume 2, Chapter 26 , Volume 2, Chapter 27 , Volume 2, Chapter 28

D.10. Quick reference: what does "He's doing ..." mean?

"She's in Math 7"

Ratios, percentages, negative numbers, simple equations. UK Year 8. Chapters Volume 1, Chapter 2 , Volume 1, Chapter 4 .

"He's doing Pre-Algebra"

The bridge from arithmetic to symbols: exponents, roots, simplifying expressions, solving linear equations, the Pythagorean theorem. UK Years 8-9. Chapters Volume 1, Chapter 3 , Volume 1, Chapter 4 , Volume 1, Chapter 5 .

"She's taking Algebra I in 8th grade"

She is on an accelerated track, one year ahead of the standard sequence, and on course for calculus in high school. Content: lines, systems, functions, factoring, quadratics. Roughly the algebra of GCSE Higher. Chapters Volume 1, Chapter 5 to Volume 1, Chapter 8 , Volume 1, Chapter 11 .

"He's doing Geometry"

A full year of Euclidean geometry with formal proof, right-triangle trigonometry, circles and coordinate geometry. Roughly the geometry of GCSE Higher, plus far more proof. Chapters Volume 1, Chapter 9 , Volume 1, Chapter 10 , Chapter 22.

"He's doing Algebra II"

The big pre-calculus algebra course: polynomials, rational functions, complex numbers, exponentials and logarithms, sequences and series, often trigonometric functions. In UK terms, the top end of GCSE Higher plus a good part of the pure algebra in the first year of A-level. A 15-year-old in Algebra II is a year or two ahead of the standard track. Chapters Volume 1, Chapter 7 , Volume 1, Chapter 11 , Volume 1, Chapter 12 , Volume 1, Chapter 15 , Volume 1, Chapter 14 .

"She's in Honors Algebra II"

The same territory, faster and with harder problems; sometimes includes topics normally saved for Precalculus.

"He's doing Integrated Math II"

A mixed course: typically quadratics, more functions, similarity and right-triangle trigonometry, probability. Roughly the second half of Algebra I combined with much of Geometry.

"She's doing Precalc"

Functions in depth, full trigonometry, polar form, vectors, an introduction to limits. Roughly A-level pure mathematics without the calculus. Chapters Volume 1, Chapter 11 , Volume 1, Chapter 13 , Volume 1, Chapter 14 , Volume 2, Chapter 16 .

"He's in AP Calc AB" / "She's taking BC"

University-level single-variable calculus, ending in a national exam in May. AB is roughly the calculus of A-level Mathematics; BC adds series and polar and parametric calculus, taking it a little beyond. Chapters Volume 2, Chapter 16 to Volume 2, Chapter 19 .

"She got a 5 in BC"

The top score on the AP Calculus BC exam. Many universities will grant credit for one or two semesters of calculus.

"He's doing AP Stats"

Introductory statistics: data, sampling, probability, the normal distribution, confidence intervals and hypothesis tests. Roughly the statistics of A-level Mathematics. Chapters Chapter 29 to Chapter 31.

"She's taking Multivariable at the community college"

She has finished AP Calculus BC early and is taking university-level calculus through dual enrollment. Chapter Volume 2, Chapter 20 , and beyond this book.

"He got a 740 on the SAT Math"

A strong score (out of 800), reflecting secure command of Algebra I, Geometry and Algebra II material. It says nothing about calculus.

US curriculum

Every major topic in this book carries a short **US curriculum** sidebar like this one, naming the course where the topic typically appears. Those sidebars and the tables in this appendix are meant to work together: if your teenager says "we're doing logarithms in Algebra II", the sidebar in Volume 1, Chapter 12 tells you where you are, and this appendix tells you what comes next.

D.11. University mathematics for computer science

US computer-science degrees almost all require a common core of mathematics, usually taken in the first two years. (UK computer-science degrees cover much the same ground in first-year modules with names like "Mathematics for Computer Science" and "Discrete Mathematics".)

Course	Typical content	Chapters
Calculus I and II	Limits, derivatives, integrals, techniques of integration, sequences and series (AP Calculus BC with a 4 or 5 often exempts students from both)	Volume 2, Chapter 16 to Volume 2, Chapter 19 , Volume 1, Chapter 14
Discrete Mathematics (one or two semesters)	Logic, sets, functions and relations, proof techniques and induction, recurrences, counting, number theory, graphs and trees. The most important mathematics course for most programmers	Chapter 21, Chapter 22, Chapter 23, Chapter 24, Chapter 25, Volume 1, Chapter 14
Linear Algebra	Systems of equations, matrices, vector spaces, determinants, eigenvalues, orthogonality, least squares, the SVD. Essential for graphics, machine learning and scientific computing	Volume 1, Chapter 6 , Volume 2, Chapter 26 , Volume 2, Chapter 27 , Volume 2, Chapter 28
Probability and Statistics	Probability, random variables, distributions, expectation, the central limit theorem, estimation and testing; often calculus-based	Chapter 29, Chapter 30, Chapter 31
Multivariable Calculus (often optional for CS, required for machine learning, graphics and robotics specialisms)	Partial derivatives, gradients, optimisation, multiple integrals	Volume 2, Chapter 20 , Chapter 33
Numerical Methods / Scientific Computing (elective)	Floating point, root finding, numerical integration and differential equations, numerical linear algebra	Volume 2, Chapter 32

Algorithms and theory-of-computation courses then draw on all of this, especially discrete mathematics, asymptotic notation and probability. Machine-learning courses lean hardest on linear algebra, multivariable calculus, probability and optimisation, which is exactly the combination brought together in Chapter 33.

Appendix E: Interactive Lab Inventory

Every Interactive Lab opportunity marked in Volume 3, collected in one place. Each entry is a hands-on experiment that makes the idea in its chapter tangible. Most now have a working version on the companion website (https://geekychris.github.io/tools/): follow the entry's _Try it link to open the tool with that example already set up._

There are 35 labs across 9 chapters.

E.1. Chapter 21

Interactive Lab: truth-table playground *(in: Logical equivalence and De Morgan's laws)*

Type two boolean expressions in up to five variables. The lab draws both truth tables side by side, highlights rows where they disagree, and offers a "push negation inward" button that applies De Morgan one step at a time, showing which law fired. A challenge mode gives a messy condition from real code and asks for the shortest equivalent.

Try it: Open the truth-table playground

Interactive Lab: Venn region painter *(in: Sets)*

A three-set Venn diagram has eight regions. Type a set expression such as $(A \setminus B) \cup \overline{C}$ and the lab shades the matching regions; click regions to paint a shape and the lab finds an expression for it. A second panel shows the same regions as rows of a three-variable truth table, making the sets-equal-logic correspondence concrete.

Try it: Regions as truth-table rows · Open the Venn region painter

Interactive Lab: relation property explorer *(in: Relations)*

Draw arrows between up to eight dots. The lab lights up which of reflexive, symmetric, antisymmetric and transitive hold, and for any that fail it highlights a witness (a missing loop, an unreturned arrow, a two-step path with no shortcut). Buttons add the reflexive, symmetric or transitive **closure**; when the result is an equivalence relation the dots recolor by class, and when it is a partial order the lab redraws it as a Hasse diagram.

Try it: Open the relation explorer

E.2. Chapter 22

Counterexample hunter *(in: Counterexamples)*

Type a claim as a Python predicate (for example `lambda n: is_prime(n*n + n + 41)` or `lambda p: not is_prime(p) or is_prime(2**p - 1)`) and a search range. The lab tests every value, plots passes and failures like the Euler-polynomial chart above, and highlights the first counterexample. Try to guess, before running, how far each claim survives.

Try it: Hunt counterexamples in Python

Proof by induction, checked by a script *(in: Mathematical induction)*

An "induction workbench": enter a conjectured closed form such as `n*(n+1)(2*n+1)//6` and a recursive definition such as `S(n) = S(n-1) + n *2`. The lab tabulates both for $n = 1 \ldots 50$ (discovery and checking), then animates the dominoes: the base-case domino turns green when you confirm $P(n_0)$, and the chain falls only after you enter the algebra of the inductive step.

Recursion tree and proof tree *(in: Structural induction and recursion)*

Build a full binary tree by clicking leaves to split them. The lab keeps live counts of leaves and internal nodes (watch $L - I$ stay at 1), and shows the structural-induction proof annotating each split: "this split turns one leaf into an internal node and adds two leaves: L goes up by 1, I goes up by 1".

Invariant tracer *(in: Proving programs correct: loop invariants)*

Step through binary search on an array you choose. At each iteration the lab shades the excluded regions grey (as in the figure), displays the invariant and the variant $\texttt{hi} - \texttt{lo} + 1$, and turns red if you edit the code (for example to `lo = mid`) in a way that breaks maintenance or termination. Try to find an edit that keeps the invariant but loops forever.

Try it: Trace binary search in Python

Spot the bug *(in: Common proof errors)*

A gallery of short "proofs", each with exactly one flaw (division by zero, missing base case, non-exhaustive cases, proving the converse, a broken small case). Click the line you think is wrong; the lab explains the error and shows a counterexample or a corrected proof. A good self-test before the mastery check.

E.3. Chapter 23

Watch the sieve *(in: The sieve of Eratosthenes)*

Animate the sieve on a 20 × 20 grid. Each prime colours its multiples in its own colour, starting at p^2. Let the reader change the grid width. With width 6, the primes after 3 line up in just two columns (the numbers of the form $6k \pm 1$). With width 10, they fall in the columns ending 1, 3, 7, 9. Ask why: the other columns are multiples of 2 or 3, or of 2 or 5.

Try it: Watch the sieve, width 6 · The primes up to 400

Jug puzzles *(in: Extended Euclid and Bézout's identity)*

Build a two-jug simulator: choose capacities a and b and allow fill, empty and pour. Highlight the reachable amounts. The reader should discover that exactly the multiples of $\gcd(a, b)$ (up to the larger capacity) are reachable. This is the *Die Hard 3* puzzle: measure 4 gallons with 3- and 5-gallon jugs. Show the Bézout coefficients the path corresponds to.

Try it: Play the jug puzzle · Bézout for jugs of 3 and 5

Any modulus clock *(in: Modular arithmetic: clock arithmetic made precise)*

A clock face with an adjustable modulus m. The user picks a and b and sees $a + b$ and $a \cdot b$ as walks around the dial. A second mode draws the "times k" map as chords: connect each x to $kx \bmod m$. With $m = 200, k = 2$ you get a cardioid, and other k give nephroids and stranger curves. It is a striking picture of multiplication as rotation.

Try it: Open the any-modulus clock

Exponentiation race *(in: Fast modular exponentiation)*

Let the user enter a, e and m. Show the exponent in binary, then animate the square-and-multiply table, lighting each bit as it is consumed. Beside it, run naive repeated multiplication and keep a step counter. Push e to 10^6 and then 10^{18} to feel the difference between linear and logarithmic.

Try it: Run the exponentiation race · Modular powers, any size

Period explorer *(in: Applications: hashing, check digits, and pseudo-random numbers)*

Sliders for a, c, m and the seed. Draw the state graph as in the figure, so that cycles and tails become visible, and report the period. Let the reader test the Hull–Dobell conditions by breaking them one at a time (for example, set $c = 4$ with $m = 16$). A second panel plots consecutive pairs (x_n, x_{n+1}) so the lattice structure of LCGs becomes visible.

Try it: Open the period explorer

RSA playground *(in: Cryptography: Diffie–Hellman and RSA)*

Let the reader pick small primes p, q and an e. Show $\gcd(e, \varphi)$ by Euclid, flag an invalid e, and derive d with the extended-Euclid table on screen. Encrypt a short word letter by letter (and show why that is a terrible idea: identical letters give identical ciphertexts). Finally, an "attacker" pane factors n by trial division and shows the time growing as p and q get bigger.

Try it: Open the RSA playground · Inverses and powers mod n

E.4. Chapter 24

Pascal's triangle explorer *(in: Binomial coefficients and Pascal's triangle)*

Display Pascal's triangle up to row 64. Let the reader colour entries by their remainder modulo 2, 3 or 5, and watch fractal patterns emerge. Clicking an entry highlights the two entries that sum to it, the hockey-stick diagonal ending at it, and its value as $n!/(r!(n-r)!)$.

Try it: Pascal's triangle mod 3 · Pascal's triangle mod 5 · Pascal's triangle explorer (mod 2)

Stars and bars sliders *(in: Stars and bars: distributing identical objects)*

Choose n (scoops) and k (flavours). The lab shows a random distribution as stars and bars and as a vector $(x_1, \ldots, x_k)$; dragging a bar moves stars between flavours. A counter enumerates all distributions for small n, k and confirms the formula $\binom{n+k-1}{k-1}$.

Try it: Stars and bars sliders

Poker-hand counter *(in: Applications: lotteries, passwords and poker)*

Deal random five-card hands and tally the hand types over thousands of deals, side by side with the exact counts above. A "build a hand" mode walks through the stages (rank choices, suit choices) and lets the reader deliberately make the 13×12 two-pair mistake and see the over-count.

Try it: Poker-hand counter and two-pair mistake · Exact poker hand counts

Birthday simulator *(in: Counting in computer science)*

Add people one at a time to a virtual room, each with a random birthday, and stop at the first collision; repeat thousands of times and plot the histogram of stopping points against the exact curve. Then switch "days" to 2^{16} or 2^{24} hash buckets and see the $\sqrt{N}$ rule hold. A password-keyspace calculator sits alongside: choose alphabet, length and guess rate, and watch the time-to-crack move by orders of magnitude as length grows.

Try it: 32-bit hash collisions · Birthday simulator and √N rule · Birthday bound for hashes

E.5. Chapter 25

Interactive graph sandbox *(in: Walks, paths, cycles and connectivity)*

A canvas where the reader adds vertices and drags edges between them. Live panels show each vertex's degree, the degree sum and $2|E|$ side by side, the connected components in different colours, and whether the current graph has an Euler path, an Euler circuit, or neither, with the odd-degree vertices highlighted.

Try it: Open the graph sandbox

BFS versus DFS race *(in: Depth-first search)*

Draw or load a graph (or a grid maze). Step BFS and DFS side by side from the same start, colouring vertices as discovered, in the queue/stack, and finished, with the queue or stack contents shown live. Toggle "maze mode" to see BFS flood outward in rings while DFS snakes down corridors, and compare the paths each finds to the exit.

Try it: BFS versus DFS race

Dijkstra step-through *(in: Shortest paths: Dijkstra's algorithm)*

Load the example graph or draw your own. Step through Dijkstra's algorithm, with finished vertices shaded, tentative distances shown on each vertex, and each relaxation flashing the edge involved. The trace table above fills in row by row. A "negative edge" toggle lets the reader set one weight below zero and watch the algorithm confidently report a wrong answer.

Try it: Dijkstra step-through

E.6. Chapter 29

Bayes slider *(in: Bayes' theorem)*

Three sliders (prevalence, sensitivity, false-positive rate) drive a live natural-frequency tree and an "icon array" of 1,000 people, colouring true positives, false positives, true negatives and false negatives. Watch the posterior collapse as you slide prevalence from 50% down to 0.1%, and see how much specificity you'd need to make a rare-disease screen useful. A second mode adds a second, independent test.

Try it: Slide the prevalence · Bayes slider lab

Simulate before you believe *(in: Probability at work in computing)*

A sandbox with a two-dice grid where you click cells to define events A and B. The lab shades $A \cap B$, displays $P(A)$, $P(B)$, $P(A \mid B)$ and $P(B \mid A)$, flags whether the events are independent, and animates "conditioning" as the non-B cells fading away. A second panel runs Monty Hall with a "host knows" / "host is random" toggle, so you can see the 2/3 answer turn into 1/2 when the host stops carrying information.

Try it: Two-dice event sandbox · Monty Hall: host knows or random

E.7. Chapter 30

Build your own bell curve *(in: The law of large numbers and the central limit theorem)*

Choose any starting distribution, including a die, an exponential, a two-spike "0 or 100" distribution, or one you draw by hand. Choose n. The lab draws thousands of samples of size n and builds the histogram of their means live, overlaying the predicted normal curve $N(\mu, \sigma^2/n)$. A tail meter shows the fraction outside ± 1.96 standard errors versus the normal's 5%. Then switch to a Cauchy distribution and watch the CLT fail.

Try it: Bell curve from a die · Build your own bell curve · Bell curve from 0 or 100

Balls, bins and two choices *(in: Distributions at work in computing)*

Animate requests falling into a row of servers. Toggle between "one random choice" and "best of two random choices" and watch the tallest stack, the number of idle servers, and a histogram of loads evolve. A second panel collects coupons: watch the long final wait for the last missing coupon, and compare the average against nH_n.

Try it: Balls, bins and two choices · Coupon collector simulation

E.8. Chapter 31

Catch the mean *(in: Confidence intervals)*

Set a population (normal, skewed, or two-humped), a sample size and a confidence level, then press "sample" repeatedly. Each press draws a new sample and adds its interval to a growing stack, coloured by whether it covers the true mean. A counter shows the running coverage rate approaching the stated level, and a slider for n shows the intervals narrowing like $1/\sqrt{n}$. A "biased sampling" switch shifts the population that samples come from, and coverage collapses even as intervals narrow.

Try it: Catch the mean · Biased sampling breaks coverage

Plan an experiment ***(in: A/B testing, end to end)***

Sliders for baseline rate, minimum detectable effect, α and power compute the required sample size live, with a plot of the two sampling distributions (null and alternative) moving apart as n grows. A "run" button simulates the experiment day by day, showing the z-score's random walk and the 1.96 boundary; a "peek" toggle lets you stop at the first crossing, while a counter over many replays shows the false-positive rate climbing.

Try it: Plan an experiment

E.9. Chapter 33

Optimiser race ***(in: Optimisation: the art of going downhill)***

Draw a 2-D loss landscape (a ravine, a bowl, an egg box). Drop a ball and choose plain gradient descent, momentum, or Adam, with sliders for η and β. Watch the paths, count steps to reach the minimum, and find the learning rate where each method first diverges. On the egg box, try ten random starting points and see how many reach the global minimum.

Try it: Ten starts on the egg box · Open the optimiser race

Drag the points ***(in: Linear regression, end to end)***

Plot five to twenty points and the least-squares line. Drag any point and watch the line, the residual squares, and the SSE update live. Drag one point far away to feel how an outlier pulls the line. Toggle between squared and absolute error and compare.

Try it: Drag the points · Least-squares fit and residuals

Logistic curve explorer ***(in: Logistic regression: from lines to probabilities)***

Plot pass/fail data and a sigmoid with sliders for w and b. Show the cross-entropy loss live, and a second panel with the loss surface over (w, b) as a contour plot with a dot for the current setting. Then press "train" and watch gradient descent slide the dot to the bottom of the (convex) bowl.

Try it: Sigmoid with w and b sliders · Logistic curve explorer

Neuron playground ***(in: Neurons, networks, and the chain rule)***

Build a tiny network (2 inputs, one hidden layer of up to 8 neurons, 1 output) and train it on a 2-D data set: two blobs, a circle inside a ring, a spiral. Watch the decision boundary bend as training proceeds. Remove the activation function and confirm that the boundary stays a straight line no matter how many layers you add. Hover over any weight to see its current gradient.

Try it: Train a tiny network

Search engine in a box ***(in: Embeddings, dot products, and search)***

Type a handful of short documents and a query. See the vocabulary, the TF-IDF vectors as a table, the dot product and cosine for each document, and the ranking. Toggle stemming on and off; switch between TF-IDF and BM25 and move the k_1 and b sliders to see how term saturation and length normalisation reorder the results.

Try it: Search engine in a box

Build a code ***(in: Information theory: measuring surprise)***

Enter symbol probabilities (or paste text and let the lab count letters). Watch the entropy update, build the Huffman tree step by step, and compare the average code length to H. Then deliberately use the wrong distribution q to build the code and see the average length rise by exactly $D_{\mathrm{KL}}(p \,\|\, q)$ when the probabilities are powers of 2.

Try it: Build a code

Slope fields ***(in: Differential equations: the next horizon)***

Draw the slope field of $dy/dt = f(t, y)$ for a choice of f (growth, logistic, spring). Click anywhere to drop an initial condition and trace the solution curve. Overlay Euler's method with an adjustable step size and watch it peel away from the true solution as the step grows.

Try it: Open the slope-field lab

Formula decoder ***(in: Reading mathematics)***

Paste a LaTeX formula. Hover over any symbol to see its name, its meaning, and the chapter of this book that introduced it. Then switch to "concrete mode": the lab asks for tiny example values, evaluates the formula step by step, and shows equivalent NumPy code.

Version 0.1
Last updated 2026-10-14 13:05:05 -0700