

COMPANION VOLUME · STAGE 03 OF THE ENGINEER'S SYLLABUS

The Engineer's Compass

The compass is the oldest instrument of proof. With it and a straightedge, Euclid built geometry out of steps nobody could doubt. This volume is the mathematics a working engineer actually uses: logic and proof, numbers and graphs, counting and probability, recurrences, vectors and gradients, and enough statistics to trust an experiment. Every result is proved in plain prose and then checked by a program you can run.

CHAPTERS	CODE EXAMPLES	EXERCISES	STAGE BUDGET
20	20	60	150 h

Contents

How to use this volume	4
Part I · Proof	6
Chapter 01 · Logic: saying exactly what you mean	7
Chapter 02 · Proof techniques, and why examples aren't proofs	13
Chapter 03 · Induction and invariants	19
Chapter 04 · Sets, relations and functions	26
Part II · Numbers & graphs	33
Chapter 05 · Number theory: primes, GCDs and modular arithmetic	34
Chapter 06 · The road to public-key cryptography	41
Chapter 07 · Graphs as mathematics	48
Chapter 08 · Colouring, matching and planarity	56
Part III · Counting	64
Chapter 09 · Counting: sequences, selections and arrangements	65
Chapter 10 · Pigeonhole and inclusion–exclusion	71
Chapter 11 · Sums, asymptotics and recurrences	78
Part IV · Probability	85
Chapter 12 · Conditional probability and Bayes' rule	86
Chapter 13 · Random variables and expectation	93
Chapter 14 · Distributions and tail bounds	100
Chapter 15 · Randomised algorithms	108
Part V · Linear algebra & calculus	116
Chapter 16 · Vectors, matrices and solving systems	117
Chapter 17 · Eigenvectors: what a matrix does in the long run	124
Chapter 18 · Derivatives, gradients and optimisation	131
Part VI · Statistics	139
Chapter 19 · Sampling, estimation and confidence intervals	140

Chapter 20 · A/B tests and Simpson's paradox	147
The deliverable	154
The twenty rules	156
Reading list	158

Run it alongside stage 02, not before it. The syllabus pairs this stage with data structures and algorithms on purpose: induction is the same muscle as recursion, recurrences are how you analyse divide and conquer, and probability is how you analyse anything randomised. Read a part here, then use it on the algorithms in front of you. The table shows where each of the stage's eleven topics is taught here, and what to read alongside it. MCS is *Mathematics for Computer Science* by Lehman, Leighton and Meyer, the free textbook of MIT's 6.042J course that the stage recommends.

STAGE 03 TOPIC	CHAPTERS	READ ALONGSIDE
Logic and proof: implication, quantifiers, contradiction, induction	Ch. 1–3	MCS Part I, Proofs
Sets, relations, functions, equivalence classes, partial orders	Ch. 4	MCS Parts I–II
Number theory and the road to public-key cryptography	Ch. 5–6	MCS Part II, Structures
Graph theory: connectivity, colouring, matchings, planarity	Ch. 7–8	MCS Part II, Structures
Combinatorics: permutations, combinations, pigeonhole, inclusion–exclusion	Ch. 9–10	MCS Part III, Counting
Recurrence relations and the master theorem	Ch. 11	MCS Parts III and V; CLRS ch. 4
Discrete probability: conditional probability, Bayes, expectation, variance	Ch. 12–13	MCS Part IV, Probability
Common distributions and tail bounds; randomised algorithm analysis	Ch. 14–15	MCS Part IV, Probability
Linear algebra: vectors, matrices, transformations, rank, eigenvectors	Ch. 16–17	3Blue1Brown, then 18.06
Calculus to derivatives, gradients and optimisation	Ch. 18	3Blue1Brown, Essence of calculus
Statistics: sampling, confidence intervals, A/B tests, Simpson's paradox	Ch. 19–20	Think Stats

Prove, then run. Each chapter states its results, proves the important ones in prose, and then runs a Python program that checks them on real cases: it counts, enumerates, simulates or solves, and prints what it found. Every program uses only the standard library and a fixed random seed, so it prints the same numbers on every run, and every output shown under a program is what it printed when this volume was built.

Do the exercises on paper. Most exercises here ask for a proof or a derivation, and the stage's deliverable is written proofs. Try each one before opening its solution, and write your answer as you would hand it in: complete sentences, every step justified. The appendix explains the deliverable, two full problem sets from MIT's 6.042J and a simulation of a counter-intuitive probability result, and lists the chapters that prepare you for each.

THE ONE HABIT

Every result in this volume is met twice: once as a proof, which says why it holds in every case, and once as a program, which checks it in the cases you can run.

Neither replaces the other. A program can check a million cases and miss the next one, as chapter 2's counterexamples show, and a proof can contain a mistake that one quick test would expose. So **prove it, then run it, and when the two disagree, find out which one is wrong**. That habit is most of what mathematics does for a programmer.

PART I

Proof

Four chapters on saying exactly what you mean and showing that it's true: the logic of statements and quantifiers, the standard shapes of proof, induction and the invariants that make programs correct, and the vocabulary of sets, relations and functions that the rest of the volume is written in.

Logic: saying exactly what you mean

3 exercises

Logic is the grammar of precise statements. You already use it in every `if`; this chapter makes it exact. It covers what "implies" really means, including why a false premise makes an implication true, which rewritings of a statement are safe and which only look safe, and how to negate a statement full of "for all" and "there exists". For a handful of propositions a truth table settles any question mechanically. The hard part is translating English into logic correctly.

Propositions and connectives

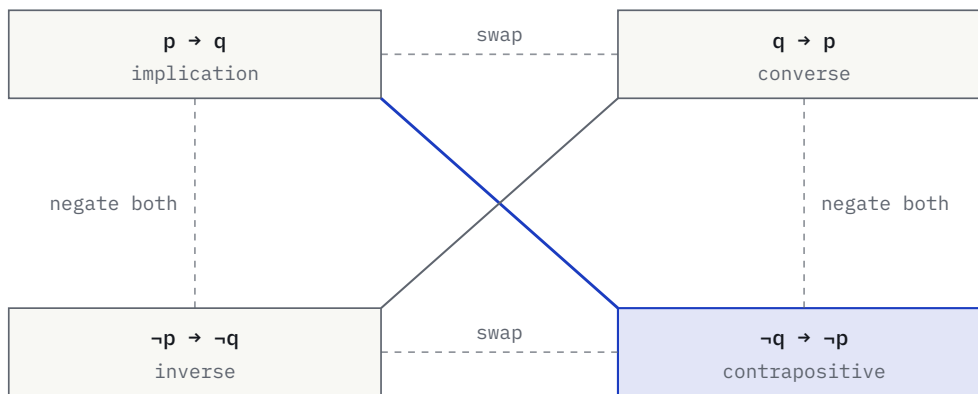
A **proposition** is a statement that is either true or false: "7 is prime", "the build passed", " $x > 3$ " once x has a value. Propositions combine with a few **connectives**, each defined by a truth table: not ($\neg p$), and ($p \wedge q$), or ($p \vee q$), exclusive or ($p \oplus q$), implies ($p \rightarrow q$) and if and only if ($p \leftrightarrow q$). Most map straight onto code: `not`, `and`, `or`, and `!=` on booleans for exclusive or. Mathematical "or" is always inclusive. "p or q" is true when both are true, unlike the "or" of a restaurant menu.

Two formulas are **equivalent** if they have the same truth value in every row of the truth table, that is, for every assignment of true and false to their variables. A formula that is true in every row is **valid**, or a tautology. One that is true in at least one row is **satisfiable**. With n variables there are 2^n rows, so a truth table settles any question about a few variables and becomes useless at a few dozen. Deciding satisfiability in general is the SAT problem, the first problem proved NP-complete, which *The Engineer's Workbench* covers in its last chapter.

Implication, the connective everyone gets wrong

$p \rightarrow q$, read "if p then q" or "p implies q", is false in exactly one case: p is true and q is false. In every other row it's true, including both rows where p is false. That surprises people, but it's the only definition that works. "If a number is divisible by 4, it's even" should be true as a statement about all numbers, so it has to be true for 3 (false premise, false conclusion) and for 6 (false premise, true conclusion) as well as for 8. An implication with a false premise is **vacuously true**. Python follows the same rule: `all([])` is `True`, because a claim about every item of an empty list has no item that could break it.

English hides implications in several phrasings. "p only if q" means $p \rightarrow q$: q is **necessary** for p. "p if q" means $q \rightarrow p$: q is **sufficient** for p. "p unless q" means $\neg q \rightarrow p$. Every implication has three relatives, made by swapping its sides, negating both, or both at once.



solid diagonals: always equivalent · dashed sides: not equivalent in general

An implication and its contrapositive always agree, and so do the converse and the inverse. Across the sides there's no such guarantee: "every square is a rectangle" is true, and its converse, "every rectangle is a square", is false.

The contrapositive is the most useful rewriting in mathematics, because proving $\neg q \rightarrow \neg p$ is often easier than proving $p \rightarrow q$ directly (chapter 2). The converse is the most common error in reasoning: from "if the server is overloaded, requests time out" you can't conclude "requests are timing out, so the server is overloaded".

Checking equivalences by machine

With a few variables you never need to argue about an equivalence. Enumerate every row and compare. The example does that for the four conditionals, De Morgan's laws, and two rewritings of nested implications, then evaluates some quantified statements over a small domain, where "for all" is a loop with `all` and "there exists" is a loop with `any`.

EXAMPLE `logic.py` · truth tables, equivalences and quantifiers checked by enumeration

```
from itertools import product

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

def check(name, f, g, n):
    """Compare two formulas on every row of the truth table; report the first row where they
    differ."""
    for row in product((False, True), repeat=n):
        if f(*row) != g(*row):
            vals = ", ".join(f"{v}={int(b)}" for v, b in zip("pq", row))
            return print(f"    NOT equivalent {name}    (differ at {vals})")
    print(f"    equivalent      {name}    ({2 ** n} rows checked)")

print("p q | p -> q")
for p, q in product((False, True), repeat=2):
    print(f"{int(p)} {int(q)} | {int(implies(p, q))}")

print("\nclaimed equivalences:")
check("p -> q vs not q -> not p    (contrapositive)", implies, lambda p, q: implies(not q, not
p), 2)
check("p -> q vs q -> p            (converse)", implies, lambda p, q: implies(q, p), 2)
```

```

check("p -> q vs not p -> not q (inverse)", implies, lambda p, q: implies(not p, not q), 2)
check("q -> p vs not p -> not q (converse vs inverse)", lambda p, q: implies(q, p), lambda
    p, q: implies(not p, not q), 2)
check("p -> q vs not p or q", implies, lambda p, q: (not p) or q, 2)
check("not (p and q) vs not p or not q (De Morgan)", lambda p, q: not (p and q), lambda p,
    q: (not p) or (not q), 2)
check("not (p -> q) vs p and not q", lambda p, q: not implies(p, q), lambda p, q: p and not
    q, 2)
check("p -> (q -> r) vs (p and q) -> r", lambda p, q, r: implies(p, implies(q, r)), lambda p,
    q, r: implies(p and q, r), 3)
check("(p -> q) -> r vs p -> (q -> r)", lambda p, q, r: implies(implies(p, q), r), lambda p,
    q, r: implies(p, implies(q, r)), 3)

D = range(10) # quantifiers over a finite domain are loops
print("\nover the domain 0..9:")
print("    for all x, exists y: x + y = 9    ", all(any(x + y == 9 for y in D) for x in D))
print("    exists y, for all x: x + y = 9    ", any(all(x + y == 9 for x in D) for y in D))
print("    for all x, exists y: x < y        ", all(any(x < y for y in D) for x in D), "(true
    over all integers)")
P = lambda x, y: x * y % 7 == 1
lhs = not all(any(P(x, y) for y in D) for x in D)
rhs = any(all(not P(x, y) for y in D) for x in D)
print(f"    not (for all x, exists y: xy mod 7 = 1) is {lhs}; exists x, for all y: xy mod 7 != 1
    is {rhs}")
print("    all([]) =", all([]), "    any([]) =", any([]), "    (a claim about no items is vacuously
    true)")

```

OUTPUT

```

p q | p -> q
0 0 | 1
0 1 | 1
1 0 | 0
1 1 | 1

```

claimed equivalences:

```

equivalent      p -> q vs not q -> not p    (contrapositive) (4 rows checked)
NOT equivalent   p -> q vs q -> p            (converse)      (differ at p=0, q=1)
NOT equivalent   p -> q vs not p -> not q     (inverse)         (differ at p=0, q=1)
equivalent      q -> p vs not p -> not q     (converse vs inverse) (4 rows checked)
equivalent      p -> q vs not p or q         (4 rows checked)
equivalent      not (p and q) vs not p or not q (De Morgan) (4 rows checked)
equivalent      not (p -> q) vs p and not q   (4 rows checked)
equivalent      p -> (q -> r) vs (p and q) -> r (8 rows checked)
NOT equivalent   (p -> q) -> r vs p -> (q -> r) (differ at p=0, q=0, r=0)

```

over the domain 0..9:

```

for all x, exists y: x + y = 9    True
exists y, for all x: x + y = 9    False
for all x, exists y: x < y        False (true over all integers)
not (for all x, exists y: xy mod 7 = 1) is True; exists x, for all y: xy mod 7 != 1 is True
all([]) = True    any([]) = False    (a claim about no items is vacuously true)

```

The contrapositive matches in all four rows. The converse and the inverse each fail at $p = 0$, $q = 1$, where the implication is vacuously true and they're false, but they match each other, as the figure says. The negation of an implication, $\neg(p \rightarrow q) \equiv p \wedge \neg q$, is worth memorising: an implication fails only when its premise holds and its conclusion doesn't, which is exactly what a counterexample is. Implication isn't associative. $(p \rightarrow q) \rightarrow r$ and

$p \rightarrow (q \rightarrow r)$ differ when everything is false. The second form equals $(p \wedge q) \rightarrow r$, which is why a chain of conditions in code can be written either as nested `if` s or as one `if` with `and` .

Predicates and quantifiers

A **predicate** is a statement with variables, such as $P(x, y): "x + y = 9"$. It becomes a proposition when you fix the variables or **quantify** them: $\forall x$, "for all x", or $\exists y$, "there exists a y". Quantifiers always range over a **domain**, and the domain changes the truth value. "For every x there's a larger y" is true over the integers and false over 0..9, where nothing is larger than 9, as the example's third line shows.

The order of mixed quantifiers matters. $\forall x \exists y: x + y = 9$ says every x has its own partner, which is true. $\exists y \forall x: x + y = 9$ says one y works for every x at once, which is false. Swapping $\forall$ and $\exists$ changes the claim. Specifications of systems are full of this distinction: "every request is served by some replica" is not "some replica serves every request".

To negate a quantified statement, push the negation inward and flip each quantifier as it passes:

$$\neg \forall x \exists y P(x, y) \equiv \exists x \forall y \neg P(x, y)$$

This is De Morgan's law for quantifiers, since "for all" is a big "and" and "there exists" is a big "or". The example checks one instance: it's false that every x in 0..9 has a y with $xy \bmod 7 = 1$, and correspondingly there's an x, namely 0 (or 7), for which every y fails. Negation is how you find out what would disprove a claim. The negation of the service-level objective "every request gets a response within one second" is "some request doesn't get a response within one second", and that single request is what your monitoring has to catch.

COMMON TRAPS

Affirming the consequent: concluding p from $p \rightarrow q$ and q. That's the converse error.

Negating "all" as "none". The negation of "all tests pass" is "some test fails", not "no test passes".

Swapping quantifiers and turning "everyone has someone" into "someone serves everyone".

Forgetting vacuous truth in code: `all()` of an empty list is `True` , so a check over zero items always passes.

Reading "or" as exclusive when the mathematics, and the programming language, mean inclusive.

Ignoring the domain: a quantified claim that is true over the integers can be false over 32-bit

integers.

EXERCISES

EXERCISE 1.1 Refunds and receipts

A shop's policy says: "You can't get a refund unless you have a receipt." Let r mean "you get a refund" and c mean "you have a receipt". Write the policy as an implication, then state its contrapositive and its converse in English. Which of them does the policy promise?

SOLUTION

"Not r unless c " means $\neg c \rightarrow \neg r$: without a receipt, no refund. Its contrapositive is $r \rightarrow c$: "if you got a refund, you had a receipt", which says the same thing, so the receipt is *necessary*. The converse of $r \rightarrow c$ is $c \rightarrow r$: "if you have a receipt, you get a refund". The policy doesn't promise that. Having a receipt is necessary, not sufficient, and a shop could still refuse a refund to someone holding one.

EXERCISE 1.2 Negating a limit

A sequence a_n converges to L if $\forall \varepsilon > 0 \exists N \forall n > N: |a_n - L| < \varepsilon$. Write the negation, " a_n does not converge to L ", with no $\neg$ left in front of a quantifier, and say in words what it means.

SOLUTION

Push the negation in, flipping each quantifier and negating the inequality at the end:

$$\exists \varepsilon > 0 \forall N \exists n > N: |a_n - L| \geq \varepsilon$$

In words: there's some fixed distance ε such that, however far along the sequence you go, there's always a later term at least ε away from L . For example, $a_n = (-1)^n$ doesn't converge to 1: take $\varepsilon = 1$, and every odd n beyond any N gives $|a_n - 1| = 2$. Note that the condition $\varepsilon > 0$ is not negated. It's part of the domain the quantifier ranges over, not part of the statement.

EXERCISE 1.3 Everything from NAND

NAND is defined by $p \uparrow q = \neg(p \wedge q)$. Express $\neg p$, $p \wedge q$ and $p \vee q$ using only NAND, and check each with a truth table. Why does this matter to hardware designers?

SOLUTION

$\neg p = p \uparrow p$, since $\neg(p \wedge p) = \neg p$. Then
 $p \wedge q = \neg(p \uparrow q) = (p \uparrow q) \uparrow (p \uparrow q)$. By De Morgan,
 $p \vee q = \neg(\neg p \wedge \neg q) = (p \uparrow p) \uparrow (q \uparrow q)$. Each can be checked in four rows, as in the example. Since not, and and or can express every truth table (write one "and" term for each true row and "or" them together), NAND alone can express every boolean function: it's **functionally complete**. That's why chips can be built from one kind of gate. NOR is functionally complete too; and, or and not on their own each are not.

CHECKLIST

- ☐ Can you write the truth table of $p \rightarrow q$ and explain the two vacuous rows?
- ☐ Can you tell an implication's converse, inverse and contrapositive apart, and say which is equivalent to it?
- ☐ Can you translate "only if", "if", "unless", "necessary" and "sufficient" into logic?
- ☐ Can you negate a statement with nested quantifiers mechanically?
- ☐ Do you check the domain before believing a quantified claim?

GO DEEPER

BOOK forall x: Calgary, an introduction to formal logic Magnus, Button et al.

COURSE 6.042J Mathematics for Computer Science MIT OpenCourseWare

BOOK Book of Proof, chapter 2: logic Richard Hammack

Proof techniques, and why examples aren't proofs

3 exercises

A proof is an argument that leaves no case unchecked. A few shapes cover most of the proofs an engineer will ever read or write: direct proof, proof by contrapositive, proof by contradiction, proof by cases, and proofs that something exists. Learning the shapes makes proofs readable. Learning why a thousand confirming examples prove nothing makes them necessary, and this chapter runs famous conjectures that held for every case anyone checked, until they didn't.

What a proof is

A **proof** starts from definitions and facts already established and reaches a conclusion by steps that each follow by logic alone. Definitions do most of the work, so they have to be exact. An integer n is **even** if $n = 2k$ for some integer k , and **odd** if $n = 2k + 1$ for some integer k . A real number is **rational** if it equals p/q for integers p and q with $q \neq 0$. Almost every first proof consists of unpacking a definition, doing algebra, and packing the result back into a definition.

Direct proof

To prove $P \rightarrow Q$ directly, assume P and derive Q . **Claim:** if n is odd, then n^2 is odd. **Proof:** since n is odd, $n = 2k + 1$ for some integer k . Then

$$n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1,$$

which is twice an integer plus one, so n^2 is odd. ■ That's the whole pattern: definition in, algebra, definition out. The square at the end marks where the proof stops.

Contrapositive and contradiction

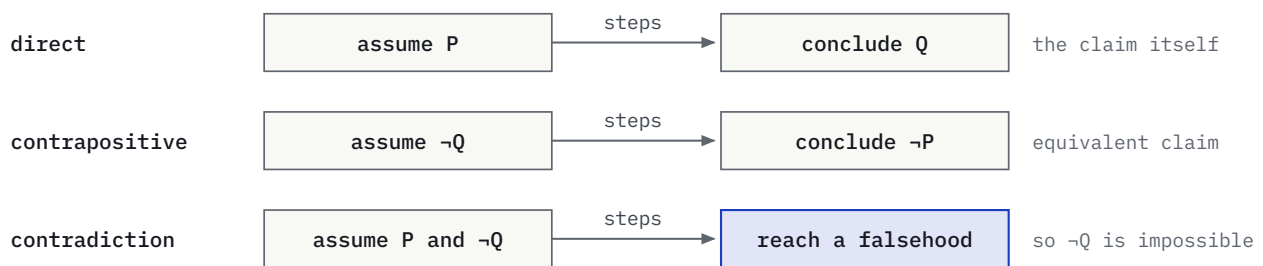
Claim: if n^2 is even, then n is even. A direct proof stalls: knowing $n^2 = 2k$ says nothing obvious about n . The contrapositive, "if n is odd, then n^2 is odd", is exactly what we just proved, and chapter 1 showed that an implication and its contrapositive are equivalent. Done.

Proof by contradiction assumes the statement is false and derives something impossible. The classic example is that $\sqrt{2}$ is irrational. Suppose it were rational, $\sqrt{2} = p/q$ with the fraction in lowest terms. Squaring gives $p^2 = 2q^2$, so p^2 is even, so p is even by the claim

above: $p = 2m$. Then $4m^2 = 2q^2$, so $q^2 = 2m^2$ and q is even too. Both even contradicts lowest terms, so no such fraction exists. ■

Euclid's proof that there are infinitely many primes has the same shape. Suppose $p_1, p_2, \dots, p_k$ were all the primes, and let $N = p_1 p_2 \cdots p_k + 1$. Dividing N by any p_i leaves remainder 1, so no p_i divides N . But $N > 1$ has some prime factor (chapter 3 proves that every integer above 1 does), and it isn't on the list. So the list wasn't complete. ■ Note that N itself needn't be prime. $2 \times 3 \times 5 \times 7 \times 11 \times 13 + 1 = 30,031 = 59 \times 509$. The proof only needs N to have a prime factor that's missing from the list.

TO PROVE $P \rightarrow Q$



all three establish the same implication; pick whichever makes the middle steps easiest

Three routes to the same destination. The contrapositive helps when the negated conclusion gives you something concrete to work with; contradiction helps when the claim is "there is no such thing", because assuming the thing exists gives you an object to reason about.

Cases and existence

Proof by cases splits the situation into possibilities that together cover everything, and proves the claim in each. To show that $n^2 + n$ is always even: if n is even, both terms are even; if n is odd, both terms are odd and their sum is even. Every integer is even or odd, so the claim holds. The danger in a case split is a missing case, which is why "without loss of generality" deserves a second look whenever you see it.

To prove that something **exists**, the most convincing way is to construct it. A **non-constructive** proof shows existence without producing the object. Here's a famous one: there are irrational numbers a and b with a^b rational. Consider $\sqrt{2}^{\sqrt{2}}$. If it's rational, take $a = b = \sqrt{2}$. If it's irrational, take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$, since $a^b = \sqrt{2}^2 = 2$. Either way such a pair exists, and the proof never tells you which case holds. To disprove a universal claim, one **counterexample** is enough, and finding counterexamples is where a computer earns its keep.

Examples aren't proofs

The example tests five claims that each held for every case their first observers checked. It uses nothing more sophisticated than trial division.

EXAMPLE

counterexamples.py · conjectures that held for a while, and one that never can fail

```
from math import isqrt

def smallest_factor(n):
    if n % 2 == 0: return 2
    for d in range(3, isqrt(n) + 1, 2):
        if n % d == 0: return d
    return n
def is_prime(n): return n >= 2 and smallest_factor(n) == n

# 1. Euler's prime-generating polynomial
n = next(n for n in range(10**6) if not is_prime(n * n + n + 41))
print(f"n^2 + n + 41 is prime for n = 0 to {n - 1}, then n = {n} gives {n*n + n + 41:,} = 41 x 41")

# 2. Fermat's conjecture that every 2^(2^k) + 1 is prime
for k in range(6):
    F = 2 ** 2 ** k + 1
    d = smallest_factor(F)
    print(f"F{k} = {F:>13,} " + ("prime" if d == F else f"= {d} x {F // d:,}"))

# 3. "Goldbach's other conjecture": every odd composite is a prime plus twice a square
fails = [n for n in range(9, 100_000, 2) if not is_prime(n)
        and not any(is_prime(n - 2 * k * k) for k in range(1, isqrt(n // 2) + 1))]
print(f"odd composites below 100,000 that are not a prime plus twice a square: {fails}")

# 4. 2^p - 1 for prime p
p = next(p for p in range(2, 100) if is_prime(p) and not is_prime(2 ** p - 1))
print(f"2^p - 1 is prime for p = 2, 3, 5, 7, but 2^{p} - 1 = {2**p - 1:,} = {smallest_factor(2**p - 1)} x {(2**p - 1) // smallest_factor(2**p - 1)}")

# 5. Evidence is not proof, but it can agree with one: no fraction p/q squares to exactly 2
best = []
p, q = 1, 1
while q < 10**6:
    best.append((p, q, p * p - 2 * q * q))
    p, q = p + 2 * q, p + q # each step is the next best approximation to sqrt(2)
print("best fractions for sqrt(2), with p^2 - 2q^2:", ", ".join(f"{p}/{q} ({d:+d})" for p, q, d in best[-4:]))
```

OUTPUT

```
n^2 + n + 41 is prime for n = 0 to 39, then n = 40 gives 1,681 = 41 x 41
F0 =          3  prime
F1 =          5  prime
F2 =         17  prime
F3 =        257  prime
F4 =       65,537  prime
F5 = 4,294,967,297 = 641 x 6,700,417
odd composites below 100,000 that are not a prime plus twice a square: [5777, 5993]
2^p - 1 is prime for p = 2, 3, 5, 7, but 2^11 - 1 = 2,047 = 23 x 89
best fractions for sqrt(2), with p^2 - 2q^2: 47321/33461 (-1), 114243/80782 (+1), 275807/195025 (-1), 665857/470832 (+1)
```

Euler's polynomial $n^2 + n + 41$ produces primes for the forty values 0 to 39, then fails at $n = 40$, where it gives 41^2 . Fermat believed every number of the form $2^{2^k} + 1$ was prime. The first five are, and in 1732 Euler found that the sixth, $4,294,967,297$, is $641 \times 6,700,417$. No

Fermat number beyond the fifth has ever been found to be prime. Goldbach suggested to Euler that every odd composite number is a prime plus twice a square, which holds for every odd composite up to 5,775 and then fails at 5,777 and 5,993. Below 100,000 the example finds no other failures, which is also not a proof that there are none. And $2^p - 1$ for prime p , the form of every Mersenne prime, is prime for the first four primes and fails at $p = 11$.

The last line shows the opposite situation, a search that agrees with a proof. The best fractions for $\sqrt{2}$ come ever closer, but $p^2 - 2q^2$ alternates between -1 and $+1$ and never reaches 0, as it would have to for $p/q = \sqrt{2}$. The proof above explains why it never will. The search alone, however long it ran, could never say so.

Programs are the same. Testing a function on a thousand inputs shows that it works on those thousand; a proof of a loop invariant (chapter 3) shows that it works on all of them. Tests are how you find the counterexample when the "proof" in your head was wrong, so you need both.

COMMON TRAPS

Assuming what you're trying to prove, often by starting from the conclusion and working "backwards" without checking that each step reverses.

Proving the converse instead of the claim.

Dividing by something that might be zero, the usual hidden flaw in "proofs" that $1 = 2$.

A case split with a missing case, especially behind "without loss of generality".

Treating many confirming examples as a proof, or one confirming example as evidence for a "for all" claim.

Contradiction where a direct proof would do. If the contradiction you reach is just the negation of your assumption, you've written a direct or contrapositive proof in disguise.

EXERCISES

EXERCISE 2.1 Rational plus irrational

Prove that if r is rational and x is irrational, then $r + x$ is irrational.
Which technique fits, and why?

SOLUTION

Contradiction fits, because "irrational" means "not rational", a statement that something doesn't exist, which is hard to use directly. Suppose $r + x$ were rational, say $r + x = a/b$, with $r = c/d$ ($b, d \neq 0$). Then

$$x = \frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd},$$

a ratio of integers with nonzero denominator, so x is rational, contradicting the assumption. ■ The same argument does not show that the sum of two irrationals is irrational, and that claim is false:

$$\sqrt{2} + (-\sqrt{2}) = 0.$$

EXERCISE 2.2 Find the flaw

"Let $a = b$. Then $a^2 = ab$, so $a^2 - b^2 = ab - b^2$, so $(a + b)(a - b) = b(a - b)$, so $a + b = b$, so $2b = b$, so $2 = 1$." Which step is wrong, and what general rule does it break?

SOLUTION

The step from $(a + b)(a - b) = b(a - b)$ to $a + b = b$ divides both sides by $a - b$, which is 0 because $a = b$. Cancelling a factor is only valid when the factor is nonzero: from $xz = yz$ you may conclude $x = y$ only if $z \neq 0$. The last step, from $2b = b$ to $2 = 1$, makes the same mistake again when $b = 0$. In code, the same bug is a division whose denominator can be zero on some path, which a proof forces you to rule out explicitly.

EXERCISE 2.3 Infinitely many primes of the form $4k + 3$

Adapt Euclid's argument to prove that there are infinitely many primes that leave remainder 3 when divided by 4. You may use the fact that a product of numbers that are each 1 more than a multiple of 4 is also 1 more than a multiple of 4.

SOLUTION

Suppose $p_1, \dots, p_k$ were all the primes of the form $4k + 3$, and let $N = 4p_1p_2 \cdots p_k - 1$. N is odd and leaves remainder 3 when divided by 4. Its prime factors are all odd, so each has remainder 1 or 3. If every prime factor had remainder 1, then by the given fact N would have remainder 1, which it doesn't. So some prime factor q of N has remainder 3. But q isn't any p_i : each p_i divides $4p_1 \cdots p_k$, so dividing N by it leaves remainder $p_i - 1$, not 0. So q is a prime of the form $4k + 3$

missing from the list, a contradiction. ■ The same trick fails for primes of the form $4k + 1$, because a product of numbers with remainder 3 can have remainder 1. That case needs a different proof.

CHECKLIST

- ☐ Can you write a direct proof that starts from a definition and ends at one?
- ☐ Do you know when to switch to the contrapositive, and when to use contradiction?
- ☐ Can you reproduce the proofs that $\sqrt{2}$ is irrational and that there are infinitely many primes?
- ☐ Can you check a case split for a missing case?
- ☐ Do you search for counterexamples before trying to prove a conjecture?

GO DEEPER

- | | |
|------------------|--|
| BOOK | Book of Proof, part II: how to prove conditional statements Richard Hammack |
| <hr/> | |
| COURSE | 6.042J lectures on proofs, and the MCS text Leighton & Meyer, MIT OpenCourseWare |
| <hr/> | |
| REFERENCE | The Prime Pages Chris Caldwell |

Induction and invariants

3 exercises

Induction proves a statement for every natural number by proving it for the first one and showing that each case carries over to the next. It's recursion read as a proof, which is why the syllabus calls it the same muscle. Invariants apply the same idea to programs and machines: a property that holds at the start and survives every step holds forever. That's how you prove a loop correct, and how you prove that some states can never be reached at all.

The principle

Let $P(n)$ be a statement about a natural number n . If $P(0)$ is true (the **base case**), and $P(n)$ implies $P(n + 1)$ for every n (the **inductive step**), then $P(n)$ is true for every n . Each case knocks over the next, like dominoes. The assumption $P(n)$ inside the step is the **inductive hypothesis**, and the whole skill lies in using it.

Claim: for every $n \geq 1$,

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

Proof. Base case: for $n = 1$ both sides equal 1. Inductive step: assume the formula holds for n . Then

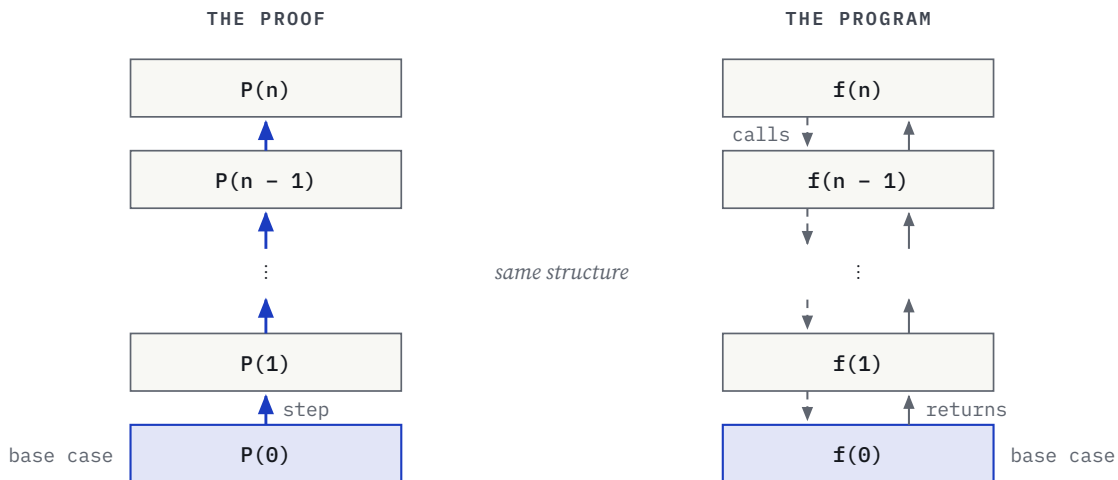
$$\sum_{i=1}^{n+1} i = \underbrace{\frac{n(n+1)}{2}}_{\text{hypothesis}} + (n+1) = \frac{(n+1)(n+2)}{2},$$

which is the formula for $n + 1$. ■ Notice what induction does and doesn't give you. It verifies a formula you already have, but it doesn't find one. Guessing comes first, often from computing small cases, and chapter 11 has methods for finding closed forms directly.

Induction is recursion

A recursive function and an induction proof have the same skeleton. The base case of the recursion is the base case of the proof. The recursive call on a smaller input is the inductive hypothesis, and the work done after the call returns is the inductive step. To prove a recursive function correct, you prove "it returns the right answer for every input of size n "

by induction on n , assuming the recursive calls return the right answers because they're on smaller inputs. If you can't write that proof, the function usually has a bug, most often a missing or wrong base case.



The proof climbs from the base case. The program descends to it with calls, then climbs back with return values. Every correct recursion has an induction proof hiding in it, and a recursion with no base case is an induction with nothing to start from.

Strong induction and well-ordering

Sometimes $P(n+1)$ depends on cases further back than $P(n)$. **Strong induction** lets the step assume $P(k)$ for every k up to n . It proves no more than ordinary induction, but it's often far more convenient. **Claim:** every integer $n \geq 2$ is a product of primes. **Proof.** If n is prime, it's a product of one prime. Otherwise $n = ab$ with $2 \leq a, b < n$, and by the strong hypothesis both a and b are products of primes, so n is too. ■ This is the step chapter 2's proof that there are infinitely many primes relied on, and the example runs it as a recursive function.

A variant: every amount of postage from 12 cents up can be made from 4-cent and 5-cent stamps. Check the four base cases 12, 13, 14 and 15 directly. For any larger amount n , the amount $n - 4$ is at least 12, so by the strong hypothesis it can be made, and one more 4-cent stamp makes n . The step reaches back four cases, which is why it needs four base cases.

An equivalent principle is **well-ordering**: every nonempty set of natural numbers has a smallest element. It gives proofs by **smallest counterexample**. If the claim failed somewhere, there would be a smallest n where it fails; you then show that n 's failure forces a failure at some smaller number, which is impossible.

Induction, run

The example checks two sum formulas on a thousand cases, runs the factorisation and postage proofs as recursive functions, asserts a loop invariant on every iteration of fast exponentiation, and then explores every reachable position of the 8-puzzle to test an

invariant of a state machine.

EXAMPLE

induction.py · formulas, strong induction as recursion, a loop invariant and a puzzle invariant

```
from collections import deque
from math import factorial

# 1. Checking a formula on many cases is evidence; the proof by induction covers the rest.
ok = all(sum(range(1, n + 1)) == n * (n + 1) // 2 for n in range(1, 1001))
ok2 = all(sum(i * i for i in range(1, n + 1)) == n * (n + 1) * (2 * n + 1) // 6 for n in
range(1, 1001))
print(f"1 + ... + n = n(n+1)/2 for n = 1..1000: {ok}; sum of squares = n(n+1)(2n+1)/6: {ok2}")

# 2. Strong induction, run as recursion: every n >= 2 is a product of primes.
calls = 0
def factor(n):
    global calls; calls += 1
    for d in range(2, int(n ** 0.5) + 1):
        if n % d == 0:
            return factor(d) + factor(n // d)    # both parts are smaller: the strong
            hypothesis
    return [n]    # no divisor: n itself is prime
print(f"factor(360) = {factor(360)} in {calls} calls; factor(9973) = {factor(9973)}")

def stamps(n):    # every amount from 12 cents up, with 4c and 5c
    stamps
    base = {12: (3, 0), 13: (2, 1), 14: (1, 2), 15: (0, 3)}
    if n in base: return base[n]
    fours, fives = stamps(n - 4)    # strong induction: the case 4 below, plus one
    stamp
    return fours + 1, fives
print("postage:", ", ".join(f"{n}c = {a}x4 + {b}x5" for n in (16, 23, 31, 100) for a, b in
[stamps(n)]))

# 3. A loop invariant: result * base**exp == x**n holds before and after every iteration.
def power(x, n):
    result, base, exp, mults = 1, x, n, 0
    while exp > 0:
        assert result * base ** exp == x ** n
        if exp & 1: result *= base; mults += 1
        base *= base; exp >>= 1; mults += 1
    assert result * base ** exp == x ** n and exp == 0    # invariant + exit condition =
    correctness
    return result, mults
r, m = power(3, 1000)
print(f"3^1000 by repeated squaring: {m} multiplications instead of 999, invariant held at
every step, "
f"correct: {r == 3 ** 1000}")

# 4. A state-machine invariant: the 8-puzzle. Every slide preserves the parity of inversions.
def inversions(state):
    tiles = [t for t in state if t != 0]
    return sum(tiles[i] > tiles[j] for i in range(len(tiles)) for j in range(i + 1,
len(tiles)))
goal = (1, 2, 3, 4, 5, 6, 7, 8, 0)
seen, queue = {goal}, deque([goal])
while queue:
    s = queue.popleft(); z = s.index(0); r, c = divmod(z, 3)
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        if 0 <= r + dr < 3 and 0 <= c + dc < 3:
            t = list(s); w = (r + dr) * 3 + c + dc; t[z], t[w] = t[w], t[z]; t = tuple(t)
```

```

        if t not in seen: seen.add(t); queue.append(t)
print(f"8-puzzle: {len(seen):,} of {factorial(9):,} arrangements reachable from the goal; "
      f"all with an even number of inversions: {all(inversions(s) % 2 == 0 for s in seen)}")
print(f"swapping tiles 7 and 8 gives {inversions((1, 2, 3, 4, 5, 6, 8, 7, 0))} inversion: "
      f"reachable: "
      f"{(1, 2, 3, 4, 5, 6, 8, 7, 0) in seen}")

```

OUTPUT

```

1 + ... + n = n(n+1)/2 for n = 1..1000: True; sum of squares = n(n+1)(2n+1)/6: True
factor(360) = [2, 2, 2, 3, 3, 5] in 11 calls; factor(9973) = [9973]
postage: 16c = 4x4 + 0x5, 23c = 2x4 + 3x5, 31c = 4x4 + 3x5, 100c = 25x4 + 0x5
3^1000 by repeated squaring: 16 multiplications instead of 999, invariant held at every step,
correct: True
8-puzzle: 181,440 of 362,880 arrangements reachable from the goal; all with an even number of
inversions: True
swapping tiles 7 and 8 gives 1 inversion: reachable: False

```

Invariants

An **invariant** is a property that holds initially and is preserved by every step. By induction on the number of steps, it then holds after any number of steps. This is the main tool for proving programs correct, an idea due to Robert Floyd and Tony Hoare in the late 1960s. For a loop, you show three things: the invariant holds before the first iteration; if it holds before an iteration, it holds after it; and when the loop exits, the invariant plus the exit condition imply the result you wanted.

Fast exponentiation keeps the invariant $r \cdot b^e = x^n$, where r is the result so far, b the base and e the remaining exponent. Initially $r = 1$, $b = x$ and $e = n$. Each iteration either multiplies r by b when e is odd, or squares b and halves e , and either way the product is unchanged. When e reaches 0, the invariant says $r = x^n$. The example asserts the invariant on every iteration and computes 3^{1000} in 16 multiplications instead of 999, because e halves each time. That's chapter 6's modular exponentiation, the operation at the heart of RSA.

Invariants also prove that something is **impossible**. In the 8-puzzle, read the eight tiles row by row and count the **inversions**, pairs that are out of order. Sliding a tile sideways doesn't change the reading order at all. Sliding a tile up or down moves it past exactly two other tiles, which changes the inversion count by -2 , 0 or $+2$. Either way the *parity* of the count never changes. The solved position has 0 inversions, so no position with an odd number can ever reach it. Swapping tiles 7 and 8 creates one inversion, and no sequence of moves will ever undo it. The example's search confirms both halves of the story. Exactly 181,440 positions, half of the $9!$ arrangements, are reachable, and all of them have even parity, so parity is the only obstacle. The 15-puzzle craze of 1880 made a famous challenge of undoing a swap of 14 and 15 on the 4×4 board, and an invariant of the same kind shows that nobody ever could.

When induction goes wrong

The classic fake proof claims that all horses are the same colour. Base case: one horse is the same colour as itself. Step: given $n + 1$ horses, the first n are the same colour by hypothesis, and so are the last n , and since the two groups overlap, all $n + 1$ are. The step is fine for $n \geq 2$, but from $n = 1$ to $n = 2$ the "first one" and the "last one" don't overlap at all, so the chain breaks at its first link. Check that the inductive step works for *every* n , including the smallest, and especially when an argument talks about "the others" or "the overlap".

COMMON TRAPS

Skipping the base case, or proving it for the wrong starting value.

A step that fails for small n , as in the horses proof.

Assuming $P(n + 1)$ inside its own proof, which is circular.

A hypothesis too weak to carry the step. Often the fix is to prove something stronger, which gives you a stronger hypothesis to use.

An invariant that one branch of the loop breaks, such as an early `continue` or a special case.

Proving the invariant but not termination. A loop that never ends also never violates its invariant.

EXERCISES

EXERCISE 3.1 Exponential beats quadratic

Prove by induction that $2^n > n^2$ for every integer $n \geq 5$. Where exactly does the proof use $n \geq 5$, and what goes wrong at $n = 4$?

SOLUTION

Base case: $2^5 = 32 > 25 = 5^2$. Step: assume $2^n > n^2$ for some $n \geq 5$. Then $2^{n+1} = 2 \cdot 2^n > 2n^2$, and it's enough to show $2n^2 \geq (n+1)^2$, that is $n^2 \geq 2n + 1$. That holds for $n \geq 3$, since $n^2 - 2n - 1 = (n-1)^2 - 2 \geq 2$. So $2^{n+1} > (n+1)^2$. ■ The step itself works from $n = 3$, but the base case is what fails below 5: $2^4 = 16 = 4^2$, not greater, and $2^3 = 8 < 9$. Both parts are needed, and the starting point is where both hold.

EXERCISE 3.2 Breaking chocolate

A chocolate bar is a grid of $m \times n$ squares. Each break splits one piece into two along a grid line. Prove that, however you break it, separating the bar into single squares takes exactly $mn - 1$ breaks.

SOLUTION

Use an invariant: *number of pieces* – *number of breaks* = 1. Initially there's 1 piece and 0 breaks. Each break takes one piece and makes two, so pieces go up by 1 and breaks go up by 1, and the difference stays 1. At the end there are mn pieces, so there were $mn - 1$ breaks, whatever order you broke them in. ■ The same invariant says that a binary tree with L leaves has $L - 1$ internal nodes, and that combining n sorted lists by pairwise merges always takes $n - 1$ merges.

EXERCISE 3.3 Binary search, proved

This function returns the first index i with `a[i] >= x` in a sorted list, or `len(a)` if there's none: `lo, hi = 0, len(a); while lo < hi: mid = (lo + hi) // 2; if a[mid] < x: lo = mid + 1, else: hi = mid; then return lo`. State a loop invariant, and prove that the function terminates and returns the right answer.

SOLUTION

Invariant: every element before index `lo` is less than x , and every element from index `hi` onward is at least x . Initially `lo = 0` and `hi = len(a)`, so both parts say nothing and hold vacuously (chapter 1). If `a[mid] < x`, then since the list is sorted every element up to `mid` is less than x , so setting `lo = mid + 1` keeps the first part. Otherwise `a[mid] ≥ x`, and every element from `mid` onward is at least x , so setting `hi = mid` keeps the second. **Termination:** `lo ≤ mid < hi`, so each iteration strictly shrinks `hi - lo`, a natural number, which can't decrease forever. **Correctness:** at exit `lo = hi`, and the invariant says everything before `lo` is less than x and everything from `lo` onward is at least x , so `lo` is exactly the first index with `a[lo] ≥ x`, or `len(a)` if there's none. ■ Writing `hi = mid - 1` or `lo = mid` breaks the invariant or the termination argument, which is where the off-by-one bugs of binary search come from.

CHECKLIST

- ☐ Can you write an induction proof with a clearly stated hypothesis, base case and step?
- ☐ Can you turn a recursive function into an induction proof of its correctness?

- ☐ Do you know when strong induction or a smallest counterexample is the easier route?
- ☐ Can you prove a loop correct with an invariant, an exit condition and a termination argument?
- ☐ Can you use an invariant to prove that a state is unreachable?

GO DEEPER

COURSE 6.042J: induction, strong induction and state machines MIT OpenCourseWare

PAPER An axiomatic basis for computer programming C. A. R. Hoare, 1969

BOOK Book of Proof, chapter 10: mathematical induction Richard Hammack

Sets, relations and functions

3 exercises

Sets, relations and functions are the data types of mathematics, and you already program with all three. Two kinds of relation matter most. Equivalence relations, "is the same as", split things into classes, as hash keys, connected components and arithmetic mod n do. Partial orders, "must come before", describe dependencies, and they're why a build can be scheduled at all. Functions, classified as one-to-one, onto or both, are the main tool for counting. Comparing infinite sets the same way shows that some problems have no program at all.

Sets

A **set** is an unordered collection of distinct elements. Write $x \in A$ for membership, $A \subseteq B$ when every element of A is in B , and use union $A \cup B$, intersection $A \cap B$ and difference $A \setminus B$. The **power set** $\mathcal{P}(A)$ is the set of all subsets of A , and a set of n elements has 2^n of them, since each element is either in or out. The **Cartesian product** $A \times B$ is the set of ordered pairs (a, b) . Set-builder notation, $\{x \in A \mid P(x)\}$, is a set comprehension: Python's `{x for x in A if p(x)}` is the same idea with the same shape.

Relations and their properties

A **relation** R on a set A is a set of ordered pairs from $A \times A$. Write $a R b$ when (a, b) is one of them, like $a \leq b$ or "a divides b". Four properties classify relations:

- **reflexive**: $a R a$ for every a ;
- **symmetric**: $a R b$ implies $b R a$;
- **antisymmetric**: $a R b$ and $b R a$ imply $a = b$;
- **transitive**: $a R b$ and $b R c$ imply $a R c$.

Each property is a "for all" statement over a finite set, so on a small set a program can check it exhaustively, as chapter 1 did with equivalences. Symmetric and antisymmetric aren't opposites. Equality is both, and "differs by at most 1" is symmetric without being transitive: 1 is close to 2 and 2 is close to 3, but 1 isn't close to 3. That relation is the trap behind any "approximately equal" comparison used as if it were equality.

Equivalence relations

A relation that is reflexive, symmetric and transitive is an **equivalence relation**, a precise version of "the same in the respect that matters". Every equivalence relation splits its set into disjoint **equivalence classes**, and every split into disjoint parts comes from an equivalence relation. Congruence mod 3 splits the integers into three classes by remainder,

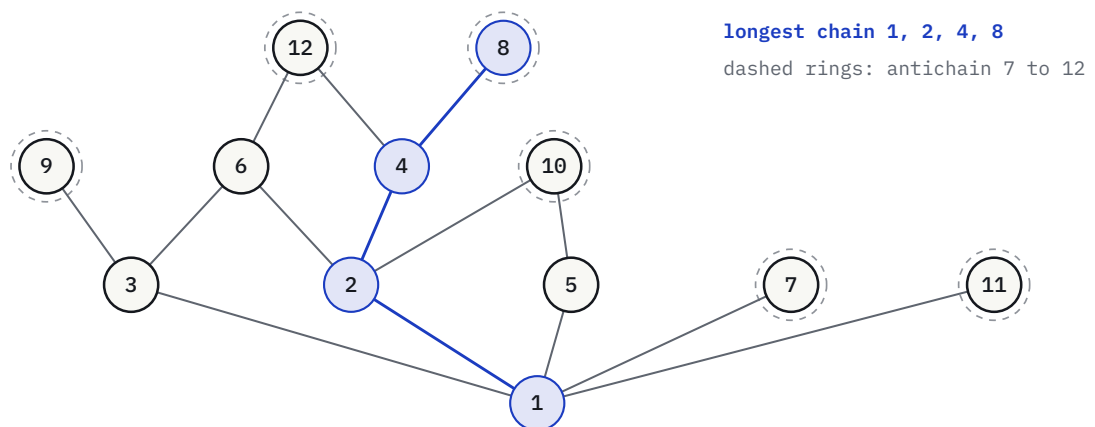
which is the foundation of modular arithmetic in chapter 5. "In the same connected component" is an equivalence relation on the vertices of a graph, and union-find is a data structure for maintaining its classes as edges arrive.

Programming languages quietly require equivalence relations. A hash table assumes that key equality is reflexive, symmetric and transitive, and that equal keys have equal hashes. Floating-point NaN breaks reflexivity, because `nan == nan` is false. In Python, two separately created NaN keys therefore land in a dictionary as two different keys, and you can't look up either one with a fresh NaN. A custom `__eq__` that compares within a tolerance breaks transitivity, with similarly strange results.

Partial orders

A relation that is reflexive, antisymmetric and transitive is a **partial order**. It's "partial" because some pairs may be incomparable: neither 4 divides 6 nor 6 divides 4.

Dependencies form partial orders, such as tasks in a build, courses with prerequisites and package versions, and so does "happened before" between events in a distributed system. A partial order is drawn as a **Hasse diagram**, with smaller elements lower and only the direct relationships shown, so that the rest follow by transitivity.



Divisibility on 1 to 12. Read "a divides b" as "a is below b and joined to it by an upward path". Each level is a set of tasks that can run at once if divisibility were a dependency order, and four levels are needed because the longest chain has four elements.

A **chain** is a set of mutually comparable elements, such as 1, 2, 4, 8. An **antichain** is a set of mutually incomparable ones, such as 7 to 12, none of which divides another. These two ideas answer scheduling questions. If the order is a set of dependencies, tasks in an antichain can run in parallel, and the longest chain is the critical path. You need at least as many rounds as the longest chain has elements, and grouping tasks by the length of the longest chain below them achieves exactly that number (Mirsky's theorem). Every finite partial order can also be listed in an order consistent with it, a **topological sort**, which is how build tools and package managers decide what to do first (exercise 4.2).

Functions

A function $f:A \rightarrow B$ assigns exactly one element of B to each element of A . It is **injective** (one-to-one) if different inputs give different outputs, **surjective** (onto) if every element of B is hit, and **bijective** if both. From an n -set to an m -set there are m^n functions, because each of the n inputs independently picks one of m outputs. There are $m(m-1)\cdots(m-n+1)$ injective ones, because each input must avoid the outputs already used. A bijection between two finite sets proves they have the same size without counting either. That idea, the **bijection rule**, drives much of chapter 9, and the fact that no injection goes from a larger set to a smaller one is the pigeonhole principle of chapter 10.

Checked by enumeration

The example classifies four relations on 1 to 12, lists the classes of congruence mod 3, finds a longest chain and an antichain under divisibility, counts functions between small sets by brute force, and runs Cantor's diagonal construction on a finite list.

EXAMPLE

relations.py · relation properties, classes, chains, counting functions and the diagonal

```
import random
from itertools import product

A = range(1, 13)
relations = {
    "a divides b": lambda a, b: b % a == 0,
    "a = b (mod 3)": lambda a, b: (a - b) % 3 == 0,
    "a <= b": lambda a, b: a <= b,
    "|a - b| <= 1": lambda a, b: abs(a - b) <= 1,
}

def props(R):
    refl = all(R(a, a) for a in A)
    sym = all(R(b, a) for a in A for b in A if R(a, b))
    anti = all(a == b for a in A for b in A if R(a, b) and R(b, a))
    trans = all(R(a, c) for a in A for b in A for c in A if R(a, b) and R(b, c))
    kind = ("equivalence relation" if refl and sym and trans else
           "partial order" if refl and anti and trans else "neither")
    return refl, sym, anti, trans, kind
print(f'{"relation on 1..12":18} reflexive symmetric antisymmetric transitive')
for name, R in relations.items():
    r, s, a, t, kind = props(R)
    print(f'{"name":18} {"r!s":9} {"s!s":9} {"a!s":13} {"t!s":10} -> {kind}')

classes = {}
for a in A: classes.setdefault(a % 3, []).append(a)
print("equivalence classes of mod 3:", list(classes.values()))

# The divisibility order: chains (all comparable) and antichains (none comparable)
div = relations["a divides b"]
longest = {}
for b in A:
    programming
    longest[b] = max((longest[a] + [b] for a in A if a < b and div(a, b)), key=len, default=[b])
chain = max(longest.values(), key=len)
antichain = [a for a in A if a > 6]
7..12
# nothing in 7..12 divides anything else in
```

```

assert not any(div(a, b) for a in antichain for b in antichain if a != b)
print(f"longest chain under 'divides': {chain}; an antichain of size {len(antichain)}:
      {antichain}")

# Counting functions between finite sets by brute force
def count(n, m):
    fs = list(product(range(m), repeat=n)) # a function from n items to m items is a tuple
    inj = sum(len(set(f)) == n for f in fs); sur = sum(len(set(f)) == m for f in fs)
    return len(fs), inj, sur
for n, m in ((3, 4), (4, 3), (4, 4)):
    total, inj, sur = count(n, m)
    print(f"functions from a {n}-set to a {m}-set: {total:3} total, {inj:2} one-to-one, {sur:2}
          onto")

# Cantor's diagonal argument, run on a finite list
random.seed(4)
rows = ["".join(random.choice("01") for _ in range(8)) for _ in range(8)]
diag = "".join("1" if rows[i][i] == "0" else "0" for i in range(8))
print("listed:", " ".join(rows))
print(f"diagonal flipped: {diag} in the list: {diag in rows} (it differs from row i at
      position i)")

```

OUTPUT

```

relation on 1..12  reflexive symmetric antisymmetric transitive
a divides b        True      False      True          True      -> partial order
a = b (mod 3)      True      True      False         True      -> equivalence relation
a <= b             True      False     True          True      -> partial order
|a - b| <= 1       True      True     False         False     -> neither
equivalence classes of mod 3: [[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
longest chain under 'divides': [1, 2, 4, 8]; an antichain of size 6: [7, 8, 9, 10, 11, 12]
functions from a 3-set to a 4-set:  64 total, 24 one-to-one,  0 onto
functions from a 4-set to a 3-set:  81 total,  0 one-to-one, 36 onto
functions from a 4-set to a 4-set: 256 total, 24 one-to-one, 24 onto
listed: 01011000 01100110 01001100 11101100 01101011 01111100 11000111 10001011
diagonal flipped: 10110000 in the list: False (it differs from row i at position i)

```

The table matches the definitions: divisibility and $\leq$ are partial orders, congruence mod 3 is an equivalence relation, and "within 1" is neither, since it fails transitivity. The function counts agree with the formulas: $4^3 = 64$ functions from a 3-set to a 4-set, of which $4 \times 3 \times 2 = 24$ are one-to-one and none are onto, because three inputs can't cover four outputs. From a 4-set to a 4-set, the one-to-one and onto functions are exactly the same 24, the 4! permutations. For finite sets of equal size, injective, surjective and bijective coincide.

Countable and uncountable

Two sets have the same **cardinality** if there's a bijection between them. For infinite sets that has surprising consequences. The even numbers are as numerous as all natural numbers, via $n \leftrightarrow 2n$. The integers are too, listed 0, 1, -1, 2, -2, and so on, and even the rationals can be listed in a zig-zag through the grid of fractions p/q . A set that can be listed this way is **countable**. The set of all finite strings over a finite alphabet is countable: list them by length, then alphabetically. Every program is a finite string, so there are countably many programs.

Georg Cantor showed in 1891 that the infinite binary sequences are **not** countable. Given any list of them, build a new sequence whose i -th bit is the opposite of the i -th bit of the i -th sequence. The new sequence differs from every listed sequence in at least one position, so the list was incomplete. The example runs this on eight rows of eight bits, and the flipped diagonal 10110000 is indeed missing. Now read an infinite binary sequence as a yes-or-no problem about natural numbers, where bit n answers "is n in the set?". There are uncountably many such problems and only countably many programs, so almost every problem has no program that solves it. Turing's proof that the halting problem is undecidable uses the same diagonal trick to name one specific such problem.

COMMON TRAPS

Treating "approximately equal" as an equivalence relation. It isn't transitive.

Confusing symmetric and antisymmetric as if they were opposites.

Calling a relation a total order when some pairs are incomparable, then sorting by it.

Hash keys with a broken equality, such as NaN, mutable objects, or `__eq__` without a matching `__hash__`.

Mixing up one-to-one and onto when counting. Write down which side must be covered.

Assuming all infinite sets are the same size. Cardinality is about bijections, not intuition.

EXERCISES

EXERCISE 4.1 Congruence is an equivalence

Fix an integer $n \geq 1$, and define $a \sim b$ to mean that n divides $a - b$. Prove that $\sim$ is an equivalence relation, and describe its equivalence classes.

SOLUTION

Reflexive: $a - a = 0 = n \cdot 0$. **Symmetric:** if $a - b = nk$, then $b - a = n(-k)$.

Transitive: if $a - b = nk$ and $b - c = nl$, then adding gives $a - c = n(k + l)$. ■

The classes are the n residue classes: all integers with remainder 0 when divided by n , all with remainder 1, and so on up to $n - 1$, because $a \sim b$ exactly when a and b leave the same remainder. Chapter 5 does arithmetic on these classes directly, which works because $\sim$ is also compatible with addition and multiplication.

EXERCISE 4.2 A topological sort always exists

Prove that every finite nonempty partial order has a **minimal** element, one with nothing strictly below it. Then use this to prove, by induction on the number of elements, that every finite partial order can be listed so that whenever a is below b , a comes before b .

SOLUTION

Minimal element: start anywhere. If the current element isn't minimal, move to something strictly below it, and repeat. By transitivity and antisymmetry you can never return to an element already visited (that would put some element strictly below itself), and there are finitely many elements, so the walk stops, at a minimal element. **Topological sort:** a single element is already listed. For $n + 1$ elements, pick a minimal element m and put it first. Nothing is below m , so no constraint is violated by placing it first. Removing m leaves a partial order on n elements, which by the inductive hypothesis can be listed consistently, so list it after m . ■ This proof is an algorithm: repeatedly remove something with no remaining predecessors. That's Kahn's algorithm for topological sorting, and it fails, by finding no minimal element, exactly when the "order" contains a cycle.

EXERCISE 4.3 More problems than programs

Show that the set of all Python programs is countable. Conclude that there is a function from the natural numbers to $\{0, 1\}$ that no Python program computes.

SOLUTION

A Python program is a finite string of Unicode characters, a finite alphabet. List all strings of length 0, then of length 1, then 2, and so on, each length in order of code points. Each length contributes finitely many strings, so every string appears at some finite position, and that's a listing. Discarding the strings that aren't valid programs leaves a listing of the programs, so they're countable. A function from $\mathbb{N}$ to $\{0, 1\}$ is an infinite binary sequence, and Cantor's argument shows those can't be listed. If every such function were computed by some program, listing the programs would list the functions, which is impossible. So some function is computed by no program. ■ The argument shows that uncomputable problems exist, and in fact that almost all problems are uncomputable, without exhibiting one. The halting problem is a concrete example.

CHECKLIST

- ☐ Can you check the four properties of a relation, and classify it as an equivalence or a partial order?
- ☐ Can you explain why equality used by a hash table must be an equivalence relation?
- ☐ Can you draw a Hasse diagram and read chains, antichains and a topological order from it?
- ☐ Can you count all, one-to-one and onto functions between finite sets?
- ☐ Can you run the diagonal argument, and explain what it says about programs?

GO DEEPER

- | | |
|------------------|---|
| BOOK | Book of Proof, chapters 11, 12 and 14: relations, functions, cardinality
Richard Hammack |
| <hr/> | |
| COURSE | 6.042J: relations, partial orders and scheduling MIT OpenCourseWare |
| <hr/> | |
| REFERENCE | Cantor's diagonal argument Wikipedia |

PART II

Numbers & graphs

Two structures that turn up everywhere in computing. First the integers: divisibility, primes, modular arithmetic, and the short path from there to the public-key cryptography behind every HTTPS connection. Then graphs: what can be proved about networks of nodes and edges, from connectivity and Euler tours to colouring, matching and drawing without crossings.

Number theory: primes, GCDs and modular arithmetic

3 exercises

Number theory was once the purest mathematics there was, and it now secures every connection you make. This chapter builds the toolkit. It covers divisibility and primes, Euclid's algorithm, the oldest algorithm still in daily use, and the modular inverses it produces. It treats modular arithmetic as ordinary arithmetic on remainders, with one rule about division, and ends with the Chinese remainder theorem, which turns one big modulus into several small ones. Chapter 6 assembles these pieces into RSA.

Divisibility and primes

An integer a **divides** b , written $a \mid b$, if $b = ak$ for some integer k . For any integer a and positive b , the **division theorem** gives a unique quotient q and remainder r with

$$a = qb + r, \quad 0 \leq r < b,$$

and r is what `a % b` computes. A **prime** is an integer $p > 1$ whose only positive divisors are 1 and p . Chapter 3 proved that every integer above 1 is a product of primes. The **fundamental theorem of arithmetic** adds that the product is unique apart from order, and the proof of uniqueness needs a fact about primes proved below, Euclid's lemma.

Primes thin out, but slowly. The **prime number theorem**, proved in 1896, says that the number of primes up to n , written $\pi(n)$, satisfies $\pi(n) \sim n / \ln n$: the ratio tends to 1. So near n , roughly one number in $\ln n$ is prime. Near 2^{1024} that's one in about 710, or one odd number in about 355, which is why generating a random 1024-bit prime takes a few hundred tries rather than billions. The example computes $\pi(n)$ up to ten million with a sieve, and the ratio creeps towards 1: 1.161 at a thousand, 1.071 at ten million.

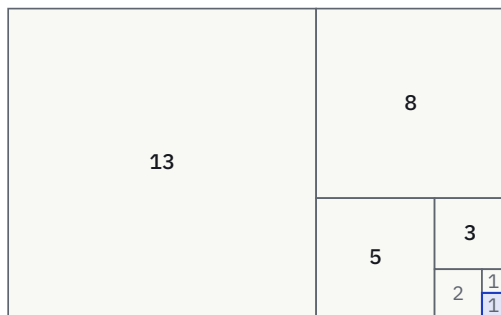
Euclid's algorithm

The **greatest common divisor** $\gcd(a, b)$ is the largest integer dividing both. Euclid's algorithm, from around 300 BC, rests on one identity:

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

Proof. Write $a = qb + r$. Any common divisor of a and b divides $r = a - qb$, and any common divisor of b and r divides $a = qb + r$. So the two pairs have exactly the same common divisors, and in particular the same greatest one. ■ Repeating the identity shrinks the

second number each time, and when it reaches 0, $\gcd(g, 0) = g$ is the answer. Geometrically, each step cuts the largest possible squares off a rectangle, and the side of the last square is the gcd.



$\gcd(21, 13)$

$$21 = 1 \times 13 + 8$$

$$13 = 1 \times 8 + 5$$

$$8 = 1 \times 5 + 3$$

$$5 = 1 \times 3 + 2$$

$$3 = 1 \times 2 + 1$$

$$2 = 2 \times 1 + 0$$

$\gcd = 1$, the last square

one square per
quotient unit

Each division step cuts squares off the remaining rectangle. With consecutive Fibonacci numbers every quotient is 1, so only one square comes off at a time. That makes them the slowest possible input for their size.

How many steps can it take? The remainders shrink fastest when quotients are large and slowest when every quotient is 1, which happens exactly for consecutive Fibonacci numbers. Gabriel Lamé proved in 1844 that the number of steps is at most about five times the number of decimal digits of the smaller number, so Euclid's algorithm is logarithmic. This was one of the first results in the analysis of algorithms. The example runs the worst case, two Fibonacci numbers near a million that take 29 steps, while random pairs of the same size average under 12.

Bézout's identity and modular inverses

Running Euclid's algorithm backwards gives more than the gcd. **Bézout's identity** says there are integers x and y with

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

and the **extended Euclidean algorithm** finds them by carrying the coefficients through each step. For example, $\gcd(240, 46) = 2 = 240 \times (-9) + 46 \times 47$. Two consequences follow at once.

Inverses. If $\gcd(a, n) = 1$, then $ax + ny = 1$, so $ax \equiv 1 \pmod{n}$: x is an **inverse** of a modulo n . If $\gcd(a, n) = d > 1$, then every multiple of a mod n is a multiple of d , so none equals 1 and no inverse exists. The example lists the inverses mod 7, where every nonzero number has one, and mod 8, where only the odd numbers do. Python computes inverses directly with

```
pow(a, -1, n) .
```

Euclid's lemma. If a prime p divides ab , then p divides a or p divides b . **Proof.** Suppose $p \nmid a$. Then $\gcd(a, p) = 1$, so $1 = ax + py$ for some x, y . Multiply by b : $b = abx + pby$. Both terms are multiples of p , so $p \mid b$. ■ Uniqueness of prime factorisation follows: if two factorisations differed, a prime from one would divide the product of the other, and so, by the lemma, one of its primes, which then must be the same prime.

Modular arithmetic

Chapter 4 showed that congruence mod n , $a \equiv b \pmod{n}$, is an equivalence relation. It also respects arithmetic: if $a \equiv a'$ and $b \equiv b'$, then $a + b \equiv a' + b'$ and $ab \equiv a'b'$. So you can reduce mod n at any point in a calculation and the final remainder doesn't change. That's what keeps the numbers in chapter 6 small enough to handle, and it's how hash functions, checksums and clock arithmetic work. Addition, subtraction and multiplication behave as usual. Division is the one difference: you can divide by a only if a has an inverse mod n . When n is a prime p , every nonzero residue has one, so arithmetic mod p supports all four operations, a structure called a **finite field**.

One practical trap: Python's `%` always returns a result with the sign of the divisor, so `-7 % 3` is 2, but C, Java, Go and JavaScript return `-1`. Code that indexes a hash table with `h % size` can produce negative indexes in those languages when h is negative.

The Chinese remainder theorem

A problem in the *Sunzi Suanjing*, a Chinese mathematical text from around the 3rd to 5th century, asks for a number that leaves remainder 2 when divided by 3, remainder 3 when divided by 5, and remainder 2 when divided by 7. The **Chinese remainder theorem** says that if the moduli $m_1, \dots, m_k$ are pairwise coprime, such a system has exactly one solution modulo their product $M = m_1 m_2 \dots m_k$. The proof is a construction. With $M_i = M/m_i$, the number

$$x = \sum_i r_i M_i (M_i^{-1} \bmod m_i)$$

works, because the i -th term is $r_i \bmod m_i$ and 0 mod every other modulus. The inverse exists because M_i is coprime to m_i . Uniqueness holds because two solutions differ by a multiple of every m_i , hence of M . The answer to the old problem is 23. In computing, the theorem lets you replace one computation mod a large M with several mod small factors. RSA implementations decrypt mod p and mod q separately and recombine, which is about four times faster than working mod n directly.

All of it, computed

The example sieves the primes up to ten million, runs Euclid's algorithm on its worst case and on random pairs, finds Bézout coefficients and inverses, and solves the Sunzi problem by the construction above.

EXAMPLE

numbers.py · the prime number theorem, Euclid's algorithm, inverses and the Chinese remainder theorem

```
import math, random

# 1. The sieve of Eratosthenes, and how many primes there are
N = 10 ** 7
sieve = bytearray([1]) * (N + 1); sieve[0] = sieve[1] = 0
for p in range(2, math.isqrt(N) + 1):
    if sieve[p]: sieve[p * p::p] = bytearray(len(range(p * p, N + 1, p)))
print("      n      primes up to n    n / ln n    ratio")
count = 0; marks = {10 ** k for k in range(1, 8)}
for n in range(N + 1):
    count += sieve[n]
    if n in marks: print(f"{n:>10}, {count:>12}, {n / math.log(n):>11,.0f}    {count / (n / math.log(n)):.3f}")

# 2. Euclid's algorithm, counting division steps
def gcd_steps(a, b):
    steps = 0
    while b: a, b = b, a % b; steps += 1
    return a, steps
fib = [1, 1]
while len(fib) < 31: fib.append(fib[-1] + fib[-2])
print(f"gcd({fib[30]:,}, {fib[29]:,}) = {gcd_steps(fib[30], fib[29])[0]} after {gcd_steps(fib[30], fib[29])[1]} steps "
      "(consecutive Fibonacci numbers are the worst case)")
random.seed(5)
pairs = [(random.randrange(1, 10 ** 6), random.randrange(1, 10 ** 6)) for _ in range(100_000)]
avg = sum(gcd_steps(a, b)[1] for a, b in pairs) / len(pairs)
print(f"average steps for 100,000 random pairs below 1,000,000: {avg:.2f}")

# 3. Extended Euclid: Bezout coefficients and modular inverses
def ext_gcd(a, b):
    if b == 0: return a, 1, 0
    g, x, y = ext_gcd(b, a % b)
    return g, y, x - (a // b) * y          # from b*x + (a mod b)*y = g
g, x, y = ext_gcd(240, 46)
print(f"gcd(240, 46) = {g} = {240} x {x} + {46} x {y}")
g, x, _ = ext_gcd(17, 3120)
print(f"inverse of 17 mod 3120: {x % 3120} (check: 17 x {x % 3120} mod 3120 = {17 * (x % 3120) % 3120}); "
      f"Python: pow(17, -1, 3120) = {pow(17, -1, 3120)}")
for m in (7, 8):
    inv = {a: next((b for b in range(1, m) if a * b % m == 1), None) for a in range(1, m)}
    print(f"inverses mod {m}: " + ", ".join(f"{a}->{b} if b else '-' for a, b in inv.items()"))

# 4. The Chinese remainder theorem: x = 2 mod 3, x = 3 mod 5, x = 2 mod 7
def crt(residues, moduli):
    M = math.prod(moduli); x = 0
    for r, m in zip(residues, moduli):
        Mi = M // m
        x += r * Mi * pow(Mi, -1, m)        # this term is r mod m and 0 mod every other modulus
    return x % M, M
x, M = crt([2, 3, 2], [3, 5, 7])
print(f"x = 2 (mod 3), 3 (mod 5), 2 (mod 7) => x = {x} (mod {M}); check: {x % 3}, {x % 5}, {x % 7}")
```

OUTPUT

n	primes up to n	n / ln n	ratio
10	4	4	0.921

100	25	22	1.151
1,000	168	145	1.161
10,000	1,229	1,086	1.132
100,000	9,592	8,686	1.104
1,000,000	78,498	72,382	1.084
10,000,000	664,579	620,421	1.071

$\text{gcd}(1,346,269, 832,040) = 1$ after 29 steps (consecutive Fibonacci numbers are the worst case)
 average steps for 100,000 random pairs below 1,000,000: 11.69
 $\text{gcd}(240, 46) = 2 = 240 \times (-9) + 46 \times 47$
 inverse of 17 mod 3120: 2753 (check: $17 \times 2753 \bmod 3120 = 1$); Python: `pow(17, -1, 3120) = 2753`
 inverses mod 7: 1->1, 2->4, 3->5, 4->2, 5->3, 6->6
 inverses mod 8: 1->1, 2->-, 3->3, 4->-, 5->5, 6->-, 7->7
 $x = 2 \pmod{3}, 3 \pmod{5}, 2 \pmod{7} \Rightarrow x = 23 \pmod{105}$; check: 2, 3, 2

The inverse of 17 mod 3120 is 2753, which is about to become a private key: chapter 6's tiny RSA example uses exactly these numbers. The inverse tables show the rule about division in action. Mod 7, every nonzero number has an inverse. Mod 8, the even numbers have none, because they share the factor 2 with the modulus.

COMMON TRAPS

Negative remainders in languages whose `%` follows the sign of the dividend.

Dividing in modular arithmetic by a number that shares a factor with the modulus.

Overflow before the reduction: computing $a * b \% n$ in fixed-width integers can overflow before the `%` applies.

Applying the Chinese remainder theorem to moduli that aren't coprime, where solutions may not exist or may not be unique.

Trial division for big numbers. Testing primality of a 300-digit number needs chapter 6's methods, not a loop to its square root.

Hash tables sized to a power of two with structured keys (exercise 5.3).

EXERCISES

EXERCISE 5.1 Where Euclid's lemma needs a prime

Show by example that Euclid's lemma fails if p isn't prime. Then find exactly where the proof above uses primality.

SOLUTION

6 divides $4 \times 9 = 36$, but 6 divides neither 4 nor 9. The proof uses primality in its first step: from $p \nmid a$ it concludes $\gcd(a, p) = 1$. For a prime that's true, since the only divisors of p are 1 and p , and p doesn't divide a . For $p = 6$ and $a = 4$, 6 doesn't divide 4, yet $\gcd(4, 6) = 2$, so Bézout gives only $2 = 4x + 6y$, and the argument stalls. The general correct statement is: if $n \mid ab$ and $\gcd(n, a) = 1$, then $n \mid b$.

EXERCISE 5.2 Solving linear congruences

Solve $17x \equiv 5 \pmod{3120}$, $6x \equiv 4 \pmod{8}$ and $6x \equiv 3 \pmod{8}$, or show that a solution doesn't exist.

SOLUTION

First: 17 has inverse 2753 mod 3120, so multiply both sides by it: $x \equiv 5 \times 2753 = 13,765 \equiv 1285 \pmod{3120}$. Check: $17 \times 1285 = 21,845 = 7 \times 3120 + 5$. Second: $\gcd(6, 8) = 2$ divides 4, so divide everything by 2 to get $3x \equiv 2 \pmod{4}$. 3 is its own inverse mod 4, so $x \equiv 6 \equiv 2 \pmod{4}$, which means $x \equiv 2$ or $6 \pmod{8}$, and both check. Third: $6x$ is always even, and $6x \pmod{8}$ is then also even, so it can never be 3. In general, $ax \equiv b \pmod{n}$ has a solution exactly when $\gcd(a, n)$ divides b , and then it has $\gcd(a, n)$ solutions mod n .

EXERCISE 5.3 Why hash tables like prime sizes

A hash table uses bucket $h(k) = k \bmod m$, and the keys that arrive are all multiples of 4: 0, 4, 8, 12, and so on. Prove that only $m / \gcd(4, m)$ of the m buckets can ever be used. How many buckets get used when $m = 64$, and when $m = 61$?

SOLUTION

Let $g = \gcd(4, m)$. Every bucket $4i \bmod m = 4i - qm$ is a multiple of g , since g divides both 4 and m , so at most m/g buckets, the multiples of g , can appear. All of them do appear: by Bézout, $g = 4x + my$, so $4x \equiv g \pmod{m}$, and multiplying by j gives every multiple $4j$ as $4(jx) \bmod m$. So exactly m/g buckets are used. For $m = 64$, $g = 4$ and only 16 buckets are used, a quarter of the table, so chains are four times longer than they should be. For $m = 61$, a prime, $g = 1$ and all 61 buckets are used. This is why textbook hash tables use prime sizes. Tables with power-of-two sizes, which are faster to index, instead scramble the hash bits first so that patterns like this don't survive.

CHECKLIST

- ☐ Can you prove $\gcd(a, b) = \gcd(b, a \bmod b)$ and explain why Euclid's algorithm is logarithmic?
- ☐ Can you run the extended Euclidean algorithm by hand and use it to find an inverse?
- ☐ Do you know exactly when a has an inverse mod n ?
- ☐ Can you prove Euclid's lemma, and explain why it gives unique factorisation?
- ☐ Can you solve a small system with the Chinese remainder theorem?

GO DEEPER

BOOK [A Computational Introduction to Number Theory and Algebra](#) Victor Shoup

COURSE [6.042J: number theory](#) MIT OpenCourseWare

REFERENCE [The Prime Pages](#) Chris Caldwell

The road to public-key cryptography

3 exercises

Public-key cryptography rests on a one-way street. Multiplying two large primes is easy, and recovering them from the product appears to be extraordinarily hard. This chapter proves Fermat's little theorem and Euler's generalisation, builds RSA from them with a complete correctness proof, and generates real 1024-bit primes the way cryptographic libraries do. It then shows why "textbook" RSA, which is mathematically correct, is still insecure. The mathematics is the easy part to get right. The engineering around it is where systems break.

Fermat's little theorem

For a prime p and any integer a not divisible by p ,

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof. Multiply each of the numbers $1, 2, \dots, p-1$ by a , mod p . The results are all nonzero, since p divides neither factor, and all different, since a has an inverse mod p and $ai \equiv aj$ would give $i \equiv j$. So the products are just $1, \dots, p-1$ in some order, and multiplying them all together gives

$$a^{p-1} \cdot (p-1)! \equiv (p-1)! \pmod{p}.$$

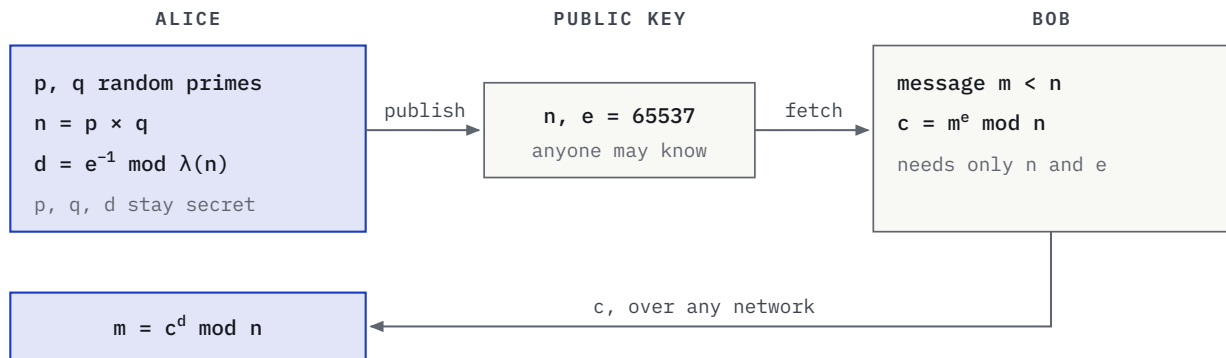
Every factor of $(p-1)!$ is invertible mod p , so cancel it to get $a^{p-1} \equiv 1$. ■ Leonhard Euler generalised the theorem to any modulus. Let $\varphi(n)$ count the numbers from 1 to n that are coprime to n . Then $a^{\varphi(n)} \equiv 1 \pmod{n}$ whenever $\gcd(a, n) = 1$, by the same argument applied to those $\varphi(n)$ numbers. For $n = pq$ with distinct primes, $\varphi(n) = (p-1)(q-1)$, since the numbers not coprime to n are the q multiples of p and the p multiples of q , which share n itself.

RSA

Ron Rivest, Adi Shamir and Leonard Adleman published the scheme in 1977. To make a key pair:

- pick two large random primes p and q , and let $n = pq$;
- pick a public exponent e coprime to $(p-1)(q-1)$, almost always 65537;
- compute the private exponent $d = e^{-1} \pmod{\varphi(n)}$ with the extended Euclidean algorithm. Implementations often use $\lambda(n) = \text{lcm}(p-1, q-1)$ instead of $\varphi(n)$, which gives a smaller d that works the same way.

The public key is (n, e) and the private key is d . To encrypt a message $m < n$, compute $c = m^e \bmod n$. To decrypt, compute $c^d \bmod n$. Both are fast thanks to chapter 3's square-and-multiply with a reduction mod n after each step.



Everything Bob needs is public, and nothing that crosses the network reveals m without d . Recovering d from (n, e) is as hard as factoring n , which is believed to be infeasible for a 2048-bit n with known classical algorithms.

Why decryption works. Since $ed \equiv 1 \pmod{\varphi(n)}$, write $ed = 1 + k\varphi(n)$. If $\gcd(m, n) = 1$, Euler's theorem gives

$$c^d \equiv m^{ed} = m \cdot \left(m^{\varphi(n)}\right)^k \equiv m \cdot 1^k = m \pmod{n}.$$

The rare case where m shares a factor with n needs one more step through the Chinese remainder theorem (exercise 6.2). The same algebra gives **signatures**: Alice computes $s = m^d \bmod n$, and anyone can check $s^e \bmod n = m$. Only the holder of d could have produced s . With chapter 5's numbers, $p = 61$ and $q = 53$ give $n = 3233$, $e = 17$ and $d = 2753$, and the message 65 encrypts to 2790 and decrypts back to 65.

Finding large primes

Key generation needs random primes of 1024 bits or more. By the prime number theorem, about one odd number in 355 of that size is prime, so the generator draws random odd candidates and tests each. It needs a fast test, because trial division up to $\sqrt{n}$ would take longer than the age of the universe.

Fermat's little theorem suggests one: if $a^{n-1} \not\equiv 1 \pmod{n}$ for some a , then n is certainly composite. The converse fails, though. Some composites, the **Carmichael numbers**, satisfy $a^{n-1} \equiv 1 \pmod{n}$ for every a coprime to n . The smallest is $561 = 3 \times 11 \times 17$, and in 1994 Alford, Granville and Pomerance proved there are infinitely many. The **Miller-Rabin test** closes the gap. Write $n - 1 = 2^s d$ with d odd. For a random base a , look at the sequence $a^d, a^{2d}, \dots, a^{2^{s-1}d} \bmod n$. If n is prime, the only square roots of 1 mod n are ± 1 , so the sequence must either

start at 1 or pass through -1 before reaching 1. A composite n violates this for at least three quarters of all bases, so 40 independent rounds leave an error probability below $4^{-40} \approx 10^{-24}$. That's far less likely than a hardware fault during the computation.

Built from scratch

The example finds every Carmichael number below 10,000 and shows Miller–Rabin rejecting each one, repeats the tiny RSA example, generates two random 1024-bit primes, and uses the 2048-bit key to encrypt, decrypt, sign and verify. It then demonstrates two weaknesses of textbook RSA. Its random numbers come from Python's `random` with a fixed seed so that the output is reproducible, which makes these keys worthless. Real keys must come from a cryptographically secure source such as Python's `secrets` module or the operating system.

EXAMPLE

rsa.py · Carmichael numbers, Miller-Rabin, and RSA from key generation to its weaknesses

```
import math, random

def is_probable_prime(n, rounds=40, rng=random):
    """Miller-Rabin: write  $n - 1 = 2^s \cdot d$ ; a witness  $a$  exposes a composite  $n$ ."""
    if n < 4: return n in (2, 3)
    if n % 2 == 0: return False
    s, d = 0, n - 1
    while d % 2 == 0: s += 1; d //= 2
    for _ in range(rounds):
        a = rng.randrange(2, n - 1)
        x = pow(a, d, n)
        if x in (1, n - 1): continue
        for _ in range(s - 1):
            x = x * x % n
            if x == n - 1: break
        else:
            return False
    return True
    # a is a witness: n is certainly composite
    # no witness found: n is prime with high probability

# 1. Fermat's test is fooled by Carmichael numbers; Miller-Rabin is not
def fermat_passes(n): return all(pow(a, n - 1, n) == 1 for a in range(2, n) if math.gcd(a, n) == 1)

def composite(n): return any(n % d == 0 for d in range(2, math.isqrt(n) + 1))
carmichael = [n for n in range(3, 10_000, 2) if composite(n) and fermat_passes(n)]
random.seed(6)
print(f"composites below 10,000 that pass Fermat's test for every coprime base: {carmichael}")
print(f"561 = 3 x 11 x 17; Miller-Rabin verdicts: {[is_probable_prime(n) for n in carmichael]}")

# 2. Textbook RSA with tiny numbers
p, q, e = 61, 53, 17
n, phi = p * q, (p - 1) * (q - 1)
d = pow(e, -1, phi)
c = pow(65, e, n)
print(f"\ntiny RSA: n = {n}, e = {e}, d = {d}; 65 -> {c} -> {pow(c, d, n)}")

# 3. Real-sized keys: random 1024-bit primes
def random_prime(bits):
    tries = 0
    while True:
```

```

        tries += 1
        cand = random.getrandbits(bits) | (1 << (bits - 1)) | 1      # top bit set, odd
        if is_probable_prime(cand): return cand, tries
p, tp = random_prime(1024); q, tq = random_prime(1024)
n, e = p * q, 65537
lam = math.lcm(p - 1, q - 1)
d = pow(e, -1, lam)
print(f"1024-bit primes found after {tp} and {tq} odd candidates (expected about {1024 *
    math.log(2) / 2:.0f} each)")
m = int.from_bytes(b"attack at dawn", "big")
c = pow(m, e, n)
back = pow(c, d, n).to_bytes(14, "big")
print(f"n has {n.bit_length()} bits; decrypt(encrypt('attack at dawn')) = {back.decode()!r}")
sig = pow(m, d, n)
print(f"signature verifies: {pow(sig, e, n) == m}; altered message verifies: {pow(sig, e, n)
    == m + 1}")

# 4. Why textbook RSA is unsafe without padding
c2 = c * pow(2, e, n) % n      # anyone can do this without the private key
print(f"multiplying the ciphertext by 2^e mod n doubles the plaintext: {pow(c2, d, n) == 2 *
    m}")
print(f"encrypting the same message twice gives the same ciphertext: {pow(m, e, n) == c}")

# 5. Cost: square-and-multiply on a 2048-bit exponent
squarings, multiplies = d.bit_length() - 1, bin(d).count("1") - 1
print(f"decryption exponent has {d.bit_length()} bits: about {squarings} squarings +
    {multiplies} multiplications")

```

OUTPUT

```

composites below 10,000 that pass Fermat's test for every coprime base: [561, 1105, 1729, 2465,
    2821, 6601, 8911]
561 = 3 x 11 x 17; Miller-Rabin verdicts: [False, False, False, False, False, False, False]

tiny RSA: n = 3233, e = 17, d = 2753; 65 -> 2790 -> 65
1024-bit primes found after 55 and 275 odd candidates (expected about 355 each)
n has 2048 bits; decrypt(encrypt('attack at dawn')) = 'attack at dawn'
signature verifies: True; altered message verifies: False
multiplying the ciphertext by 2^e mod n doubles the plaintext: True
encrypting the same message twice gives the same ciphertext: True
decryption exponent has 2047 bits: about 2046 squarings + 995 multiplications

```

The two primes took 55 and 275 odd candidates, around the expected 355, because the number of tries before a success is geometrically distributed (chapter 14). Decryption with a 2047-bit exponent costs about 2,046 squarings and 995 multiplications of 2048-bit numbers, which is why RSA private-key operations are expensive and why implementations use the Chinese remainder theorem to speed them up.

Why textbook RSA is broken

Everything above is mathematically correct, and the last two lines of output show that it's still insecure. First, it's **deterministic**. The same message always gives the same ciphertext, so an attacker who can guess the possible messages, "yes" or "no", a salary from a short list, simply encrypts each guess with the public key and compares. Second, it's **malleable**. Multiplying a ciphertext by 2^e doubles the hidden plaintext without anyone knowing it, and real attacks have exploited structure like this. Small exponents with small messages are worse still: with $e = 3$ and $m^3 < n$, the ciphertext is just m^3 and an ordinary cube root

recovers m . Real RSA therefore always uses randomised **padding**, OAEP for encryption and PSS for signatures, and modern protocols mostly use elliptic-curve Diffie–Hellman for key exchange. *The Engineer's Citadel* covers what to use in practice. The rule from this chapter is simpler: understand the mathematics, then never implement it yourself in production.

The security also rests on an assumption. Nobody has proved factoring is hard. The largest RSA challenge number factored so far, RSA-250 with 829 bits, fell in 2020 after thousands of core-years of computation, and 2048-bit keys are the current minimum recommendation. A large quantum computer running Shor's algorithm would factor efficiently, which is why NIST standardised the post-quantum key-encapsulation scheme ML-KEM in 2024.

COMMON TRAPS

Rolling your own RSA, or any cryptography, outside of learning exercises.

Encrypting without padding, which makes ciphertexts deterministic and malleable.

Generating keys from a non-cryptographic random generator, or from too little entropy, as on devices that generate keys at first boot.

Using Fermat's test alone, which every Carmichael number passes.

Exponentiation whose timing depends on the secret exponent, which leaks the key bit by bit to anyone who can time it.

Treating factoring as proved hard. It's an assumption with a quantum expiry date.

EXERCISES

EXERCISE 6.1 A big power by hand

Compute $3^{200} \bmod 13$ without a computer.

SOLUTION

13 is prime and doesn't divide 3, so Fermat's little theorem gives $3^{12} \equiv 1 \pmod{13}$. Since $200 = 16 \times 12 + 8$, we have $3^{200} = (3^{12})^{16} \cdot 3^8 \equiv 3^8$. Now $3^3 = 27 \equiv 1 \pmod{13}$, so $3^8 = (3^3)^2 \cdot 3^2 \equiv 9$. The answer is 9. Reducing exponents mod $p - 1$, or mod the actual order of the base, which here is 3, is the same trick RSA's correctness proof uses.

EXERCISE 6.2 RSA works for every message

The proof above assumed $\gcd(m, n) = 1$. Prove that $m^{ed} \equiv m \pmod{n}$ for every m , including multiples of p or q .

SOLUTION

By the Chinese remainder theorem it's enough to prove the congruence mod p and mod q separately, since p and q are distinct primes. Mod p : if $p \mid m$, both sides are 0. Otherwise Fermat gives $m^{p-1} \equiv 1 \pmod{p}$, and since $ed = 1 + k(p-1)(q-1)$, we get $m^{ed} = m \cdot (m^{p-1})^{k(q-1)} \equiv m$. The same argument works mod q . A number congruent to m mod both p and q is congruent to m mod $pq = n$. ■ The argument only needs $ed \equiv 1$ modulo both $p-1$ and $q-1$, which is why $\lambda(n) = \text{lcm}(p-1, q-1)$ works as well as $\varphi(n)$.

EXERCISE 6.3 Shared primes

Two devices generate RSA keys, and because of a weak random generator they happen to pick the same prime p , with different second primes. Their moduli $n_1 = pq_1$ and $n_2 = pq_2$ are public. How can anyone factor both, and how expensive is it?

SOLUTION

Compute $\gcd(n_1, n_2)$ with Euclid's algorithm. It equals p , since that's the only common factor, and then $q_1 = n_1/p$ and $q_2 = n_2/p$, which reveals both private keys. Euclid's algorithm on 2048-bit numbers takes a few thousand steps, a fraction of a millisecond. This happened in the real world. In 2012 Heninger, Durumeric, Wustrow and Halderman collected millions of public keys from the internet, computed pairwise gcds efficiently, and factored the RSA keys of about 0.5% of the TLS hosts they scanned. Most were embedded devices that generated keys at first boot, before they had gathered enough randomness. The attack needs nothing but chapter 5's oldest algorithm.

CHECKLIST

- ☐ Can you prove Fermat's little theorem, and state Euler's generalisation?
- ☐ Can you generate a toy RSA key by hand and prove decryption correct?
- ☐ Can you explain why Fermat's test is fooled by Carmichael numbers and Miller–Rabin is not?
- ☐ Can you name the two reasons textbook RSA is insecure, and what fixes them?
- ☐ Do you know what the security of RSA rests on, and what threatens it?

GO DEEPER

PAPER A method for obtaining digital signatures and public-key cryptosystems
Rivest, Shamir & Adleman, 1978

PAPER Twenty years of attacks on the RSA cryptosystem Dan Boneh

PAPER Mining your Ps and Qs: detection of widespread weak keys Heninger,
Durumeric, Wustrow & Halderman, 2012

Graphs as mathematics

3 exercises

A graph is just vertices and edges, which is why it can model networks, dependencies, maps, friendships and state machines. Stage 02 teaches the algorithms that run on graphs. This chapter asks what is always true of them, and proves it. It covers the handshake lemma, Euler's theorem on walks that use every edge, the equivalent ways to define a tree, the exact condition for splitting a graph into two sides, and the sudden appearance of a giant component in a random network once the average degree passes one.

Vocabulary

A **graph** $G = (V, E)$ is a set of **vertices** V and a set of **edges** E , each edge joining two vertices. In a simple graph there are no loops and no repeated edges; a **multigraph** allows repeats. The **degree** of a vertex is the number of edges at it. A **walk** is a sequence of vertices joined by edges, a **path** is a walk that repeats no vertex, and a **cycle** is a closed walk that repeats no vertex except the start. A graph is **connected** if there's a path between every pair of vertices, and its maximal connected pieces are its **components**, the classes of the equivalence relation "is connected to" from chapter 4. Directed graphs give each edge a direction, and a vertex then has separate in- and out-degrees.

The handshake lemma

In any graph,

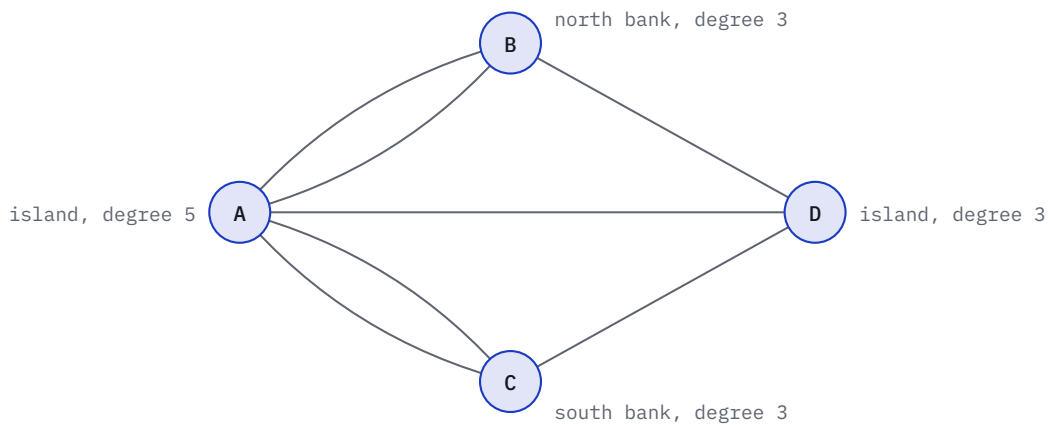
$$\sum_{v \in V} \deg(v) = 2 |E|.$$

Proof. Each edge has two ends and contributes exactly 1 to the degree of each, so summing degrees counts every edge twice. ■ This is **double counting**, counting the same thing in two ways, a technique chapter 9 uses again. One corollary follows at once. The sum of all degrees is even, so the number of vertices with odd degree must be even: in any group, the number of people who have shaken an odd number of hands is even. The example's random graph has 993 edges, degree sum 1986, and 96 odd-degree vertices, and that last number will be even whatever the random seed.

Euler tours

Graph theory began with a puzzle about the city of Königsberg, whose seven bridges joined two river banks and two islands. Could you walk through the city crossing every bridge exactly once? Leonhard Euler showed in 1736 that you couldn't. Model each land mass as a

vertex and each bridge as an edge.



four vertices of odd degree; an Euler walk allows at most two

Every time a walk passes through a land mass it uses two bridges, one in and one out. So every vertex except the start and the end must have even degree.

Königsberg has four odd vertices, two too many.

Euler's theorem. A connected graph has a closed walk that uses every edge exactly once, an **Euler circuit**, if and only if every vertex has even degree. It has an open walk that does so if and only if exactly two vertices have odd degree, and then the walk must start at one and end at the other. Necessity is the argument in the caption. Sufficiency is constructive. Start anywhere and walk along unused edges. Because degrees are even, you can only get stuck back at the start. If unused edges remain, some vertex on your circuit touches one, so walk a detour from there and splice it in. That's **Hierholzer's algorithm**, linear in the number of edges, and the example uses it to find a circuit through all ten edges of K_5 , where every degree is 4.

Change one word and everything changes. A **Hamiltonian cycle** visits every *vertex* exactly once, and deciding whether one exists is NP-complete (*The Engineer's Workbench*, chapter 8). No simple degree condition characterises it. The contrast is a good example of how a problem's statement, not its look, decides whether it's easy.

Trees

A **tree** is a connected graph with no cycles. For a graph with n vertices, the following are equivalent, and each is a useful definition:

- it's connected and has no cycles;
- it's connected and has exactly $n - 1$ edges;
- it has no cycles and has exactly $n - 1$ edges;
- between any two vertices there's exactly one path.

The key step is that every tree with $n \geq 2$ vertices has a **leaf**, a vertex of degree 1: take a longest path, and its end can't have any other neighbour, since that would extend the path or close a cycle. Removing a leaf and its edge leaves a tree on $n - 1$ vertices, so by induction

a tree on n vertices has $n - 1$ edges. The example's spanning-tree test relies on the third form: $n - 1$ edges that never close a cycle must form a tree. It counts the spanning trees of K_n by brute force and matches **Cayley's formula**, n^{n-2} : 16, 125 and 1296. Spanning trees appear in network protocols that switch off redundant links to avoid loops, and in the minimum spanning tree algorithms of stage 02.

Bipartite graphs

A graph is **bipartite** if its vertices split into two sides with every edge crossing between them: workers and jobs, students and courses, users and items. **Theorem:** a graph is bipartite if and only if it has no cycle of odd length. One direction is easy. A cycle alternates sides with each step, so it returns to its start only after an even number of steps. For the other, run breadth-first search from any vertex and colour vertices by the parity of their distance. If some edge joins two vertices of the same colour, the two BFS paths to its ends plus the edge itself form a closed walk of odd length, which contains an odd cycle. So if there's no odd cycle, the colouring is a valid split. The proof is the algorithm: BFS either 2-colours the graph or exposes an odd cycle. In the example, a 6-cycle and an 8×8 grid are bipartite (colour the grid like a chessboard), while a 7-cycle and the Petersen graph, which contains 5-cycles, are not.

Checked, constructed and simulated

The example verifies the handshake lemma, applies Euler's degree test to Königsberg and constructs a circuit in K_5 , counts spanning trees, tests four graphs for bipartiteness, and then builds random graphs on 20,000 vertices with increasing average degree.

EXAMPLE

graphs.py · handshakes, Euler circuits, spanning trees, bipartiteness and the giant component

```
import math, random
from collections import deque, Counter
from itertools import combinations

random.seed(7)
# 1. The handshake lemma on a random graph
n = 200
edges = [(a, b) for a, b in combinations(range(n), 2) if random.random() < 0.05]
deg = Counter(v for e in edges for v in e)
odd = sum(deg[v] % 2 for v in range(n))
print(f"random graph: {n} vertices, {len(edges)} edges; sum of degrees = {sum(deg.values())} =
      2 x {len(edges)}; "
      f"{odd} vertices of odd degree (always even)")

# 2. Euler: the bridges of Königsberg, and a circuit through every edge of K5
bridges = [("A", "B"), ("A", "B"), ("A", "C"), ("A", "C"), ("A", "D"), ("B", "D"), ("C", "D")]
kdeg = Counter(v for e in bridges for v in e)
print(f"Königsberg degrees {dict(sorted(kdeg.items()))}: {sum(d % 2 for d in kdeg.values())}
      odd vertices, "
      "so no walk crosses every bridge exactly once")
def euler_circuit(edge_list, start):
    adj = {}
```

```

for i, (a, b) in enumerate(edge_list):
    adj.setdefault(a, []).append((b, i)); adj.setdefault(b, []).append((a, i))
used, stack, circuit = set(), [start], []
while stack:
    # Hierholzer: walk until stuck, then splice in
    detours
    v = stack[-1]
    while adj[v] and adj[v][-1][1] in used: adj[v].pop()
    if adj[v]:
        w, i = adj[v].pop(); used.add(i); stack.append(w)
    else:
        circuit.append(stack.pop())
    return circuit[::-1]
k5 = list(combinations(range(5), 2))
tour = euler_circuit(k5, 0)
steps = {frozenset(p) for p in zip(tour, tour[1:])}
print(f"K5 (every degree 4): Euler circuit {'-'.join(map(str, tour))} uses all {len(steps)}
edges once")

# 3. Trees: count the spanning trees of K_n by brute force, against Cayley's formula n^(n-2)
def is_tree(n, es):
    parent = list(range(n))
    def find(x):
        while parent[x] != x: parent[x] = parent[parent[x]]; x = parent[x]
        return x
    for a, b in es:
        ra, rb = find(a), find(b)
        if ra == rb: return False
        parent[ra] = rb
    return True
    # n - 1 edges and no cycle: connected, a tree
for m in (4, 5, 6):
    count = sum(is_tree(m, es) for es in combinations(list(combinations(range(m), 2)), m - 1))
    print(f"spanning trees of K{m}: {count:5,} by brute force, {m ** (m - 2):5,} by Cayley's
formula")

# 4. Bipartite or not: BFS 2-colouring, which either succeeds or finds an odd cycle
def two_colour(adj):
    colour = {}
    for s in adj:
        if s in colour: continue
        colour[s] = 0; q = deque([s])
        while q:
            v = q.popleft()
            for w in adj[v]:
                if w not in colour: colour[w] = 1 - colour[v]; q.append(w)
                elif colour[w] == colour[v]: return False
        return True
def cycle(k): return {i: [(i - 1) % k, (i + 1) % k] for i in range(k)}
petersen = {i: [] for i in range(10)}
for i in range(5):
    for a, b in ((i, (i + 1) % 5), (i, i + 5), (i + 5, (i + 2) % 5 + 5)):
        petersen[a].append(b); petersen[b].append(a)
grid = {(r, c): [(r + dr, c + dc) for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)) if 0 <= r +
dr < 8 and 0 <= c + dc < 8]
        for r in range(8) for c in range(8)}
print("2-colourable:", ", ".join(f"{name} {two_colour(g)}" for name, g in
    (("6-cycle", cycle(6)), ("7-cycle", cycle(7)), ("8x8 grid", grid), ("Petersen graph",
    petersen))))

# 5. Random graphs: a giant component appears suddenly when the average degree passes 1
N = 20_000
print(f"\naverage degree c, random graph on {N:,} vertices: share of vertices in the largest
component")
for c in (0.5, 0.9, 1.0, 1.1, 1.5, 2.0, 3.0):
    parent = list(range(N))
    def find(x):
        while parent[x] != x: parent[x] = parent[parent[x]]; x = parent[x]

```

```

    return x
for _ in range(round(c * N / 2)):
    a, b = random.randrange(N), random.randrange(N)
    parent[find(a)] = find(b)
largest = max(Counter(find(v) for v in range(N)).values())
beta = 0.0
if c > 1:
    beta = 0.5
    for _ in range(200): beta = 1 - math.exp(-c * beta)  # fixed point of beta = 1 - e^(-c
    beta)
print(f"    c = {c:3}: {largest / N:6.1%}    (theory for large n: {beta:.1%})")

```

OUTPUT

```

random graph: 200 vertices, 993 edges; sum of degrees = 1986 = 2 x 993; 96 vertices of odd
degree (always even)
Konigsberg degrees {'A': 5, 'B': 3, 'C': 3, 'D': 3}: 4 odd vertices, so no walk crosses every
bridge exactly once
K5 (every degree 4): Euler circuit 0-4-3-2-4-1-3-0-2-1-0 uses all 10 edges once
spanning trees of K4:    16 by brute force,    16 by Cayley's formula
spanning trees of K5:   125 by brute force,   125 by Cayley's formula
spanning trees of K6: 1,296 by brute force, 1,296 by Cayley's formula
2-colourable: 6-cycle True, 7-cycle False, 8x8 grid True, Petersen graph False

average degree c, random graph on 20,000 vertices: share of vertices in the largest component
c = 0.5:    0.1%    (theory for large n: 0.0%)
c = 0.9:    1.2%    (theory for large n: 0.0%)
c = 1.0:    3.9%    (theory for large n: 0.0%)
c = 1.1:   18.1%    (theory for large n: 17.6%)
c = 1.5:   58.3%    (theory for large n: 58.3%)
c = 2.0:   79.4%    (theory for large n: 79.7%)
c = 3.0:   94.1%    (theory for large n: 94.0%)

```

Random graphs and the giant component

The last table shows one of the most striking facts about networks. Paul Erdős and Alfréd Rényi studied graphs whose edges are chosen at random, and found a **phase transition**. When the average degree c is below 1, every component is small, a vanishing fraction of the graph: 0.1% at $c = 0.5$. Just above 1, a **giant component** emerges containing a fixed fraction β of all vertices, where β solves

$$\beta = 1 - e^{-c\beta}.$$

At $c = 1.5$ the theory predicts 58.3% and the simulation gives 58.3%; at $c = 3$, 94.0% against 94.1%. The intuition is a branching process. Explore outward from a vertex: each newly reached vertex brings about c new neighbours on average. Below 1 the exploration dies out quickly. Above 1 it survives with positive probability, and the survivors all join one giant cluster. The same threshold appears in epidemics, where an outbreak becomes large when each case infects more than one other on average, and in the resilience of networks to random failures.

COMMON TRAPS

Confusing Euler and Hamilton: every edge once is easy; every vertex once is NP complete

Confusing Euler and Hamilton. Every edge once is easy, every vertex once is NP-complete.

Forgetting connectivity in Euler's theorem. Two separate cycles have all even degrees and no single circuit.

Checking only one condition for a tree: $n - 1$ edges with a cycle is disconnected, not a tree.

Using undirected degree conditions on directed graphs, where an Euler circuit needs in-degree equal to out-degree at every vertex.

Treating "walk", "path" and "cycle" as interchangeable in proofs, where repeated vertices matter.

Expecting smooth growth in networks. Connectivity can jump at a threshold, as the giant component does.

EXERCISES

EXERCISE 7.1 Three handshakes each

Five people meet. Can each of them shake hands with exactly three of the others? Can six people do so?

SOLUTION

With five people, the degrees would sum to $5 \times 3 = 15$, which is odd, but the handshake lemma says the sum of degrees is twice the number of handshakes, so it must be even. Five people can't do it. Six people can: the degree sum is 18, so there would be 9 handshakes, and one way is to seat them in a ring of six, where each shakes hands with both neighbours, and add the three handshakes between people sitting opposite each other. That graph is $K_{3,3}$, which reappears in chapter 8 as a graph that can't be drawn without crossings.

EXERCISE 7.2 At least two leaves

Prove that every tree with at least two vertices has at least two leaves. Then show that a tree with n vertices has exactly $n - 1$ edges.

SOLUTION

Take a path in the tree with as many edges as possible, with ends u and w . It has at least one edge, since the tree is connected with at least two vertices. If u had a neighbour x other than the next vertex on the path, then either x is on the path, making a cycle, impossible in a tree, or it

isn't, and adding it makes a longer path, contradicting maximality. So u has degree 1, and by the same argument so does w : two leaves. For the edge count, use induction on n . One vertex has 0 edges. For $n \geq 2$, remove a leaf and its single edge. What remains is still connected, since no path between other vertices passed through a leaf, and still has no cycle, so it's a tree on $n - 1$ vertices with $n - 2$ edges by hypothesis. Adding the leaf back gives $n - 1$. ■

EXERCISE 7.3 Cracking a keypad with Euler

A lock opens when the last three binary digits typed match a secret 3-bit code, and it never resets. What's the shortest sequence of key presses guaranteed to open it? Use an Euler circuit to construct one.

SOLUTION

Make a directed graph whose vertices are the four 2-bit strings 00, 01, 10, 11, and add an edge from ab to bc for each 3-bit code abc . Each of the 8 codes is one edge, and every vertex has in-degree 2 and out-degree 2, so the graph has an Euler circuit. Walking it and typing one digit per edge, after an initial two digits for the starting vertex, produces a sequence in which every 3-bit code appears exactly once as a window. That's $8 + 2 = 10$ key presses, for example 0001011100, whose eight windows are 000, 001, 010, 101, 011, 111, 110 and 100. No shorter sequence can work, since 10 presses give exactly 8 windows. This is a **de Bruijn sequence**. For k -digit codes over b symbols it has length $b^k + k - 1$ instead of the $k \cdot b^k$ presses of trying each code separately.

CHECKLIST

- ☐ Can you prove the handshake lemma and its corollary about odd degrees?
- ☐ Can you state Euler's theorem, prove necessity, and describe Hierholzer's construction?
- ☐ Can you list four equivalent definitions of a tree and prove one equivalence?
- ☐ Can you prove that a graph is bipartite exactly when it has no odd cycle?
- ☐ Can you explain why a giant component appears at average degree 1?

GO DEEPER

BOOK Graph Theory, free electronic edition Reinhard Diestel

COURSE 6.042J: graph theory MIT OpenCourseWare

COURSE 6.042J. graph theory MIT OpenCourseWare

BOOK Network Science, chapter 3: random networks Albert-László Barabási

Colouring, matching and planarity

3 exercises

Three classic problems about graphs, each with a theorem that says when it can be solved. Colouring is the mathematics of scheduling without conflicts. Finding the minimum number of colours is NP-hard, but a greedy bound always holds. Matching is the mathematics of pairing up: Hall's theorem says exactly when everyone can be matched, and the Gale–Shapley algorithm finds a stable matching that has been used to assign doctors to hospitals for decades. Planarity is about drawing without crossings, and Euler's formula proves that some graphs can never be drawn flat.

Colouring

A **proper colouring** gives each vertex a colour so that adjacent vertices differ, and the **chromatic number** $\chi(G)$ is the fewest colours that suffice. Colouring models any problem of conflicts. Exams that share a student need different slots, variables that are live at the same time need different registers, and nearby transmitters need different frequencies. Computing $\chi(G)$ is NP-hard, and even deciding whether three colours suffice is NP-complete. One bound is always easy, though.

Theorem: $\chi(G) \leq \Delta + 1$, where Δ is the maximum degree. **Proof.** Colour the vertices one at a time in any order, giving each the smallest colour not used by its already-coloured neighbours. A vertex has at most Δ neighbours, so at most Δ colours are blocked and one of the first $\Delta + 1$ is free. ■ The bound says nothing about how good greedy is on a particular graph, and the order matters enormously. The example colours a **crown graph**, which is bipartite and so needs only 2 colours: in one order greedy uses 8, in another 2. On a random graph with maximum degree 29, greedy used 9 colours, far below the bound of 30, and ordering by degree first saved one more. That's typical. The bound is a guarantee, not a prediction.

Matching and Hall's theorem

A **matching** is a set of edges with no shared vertices. In a bipartite graph of people and jobs, it's an assignment where nobody gets two jobs and no job goes to two people. When can everyone on the left be matched? Write $N(S)$ for the set of jobs that someone in a set S of people would accept. If some set S of people likes fewer than $|S|$ jobs between them, they can't all be matched, by the pigeonhole principle. **Hall's theorem** (1935) says this is the only obstacle: a matching that covers the whole left side exists if and only if

$$|N(S)| \geq |S| \quad \text{for every subset } S \text{ of the left side.}$$

The proof is an induction on the number of people, splitting on whether some proper subset S has $|N(S)|$ exactly equal to $|S|$. Algorithms don't check all 2^n subsets. They grow a matching with **augmenting paths**, paths that alternate between unmatched and matched edges and start and end at unmatched vertices. Flipping such a path adds one edge to the matching, and when none exists the matching is maximum. When it falls short, the search also finds the set S that violates Hall's condition, a certificate of failure. In the example's four-person instance, the maximum matching covers only three people, and the reason is that ann, bob and cat between them accept only db and web, three people for two jobs.

Stable matching

Now suppose both sides have preferences: proposers rank receivers and receivers rank proposers. A matching is **unstable** if some proposer and receiver prefer each other to their assigned partners, because they'd both rather break away. David Gale and Lloyd Shapley showed in 1962 that a stable matching always exists, by an algorithm. Each free proposer proposes to the best receiver they haven't yet proposed to. Each receiver holds on to the best proposal so far and rejects the rest, trading up whenever a better one arrives.

It terminates: each proposer proposes to each receiver at most once, so there are at most n^2 proposals. **It ends with everyone matched:** a receiver, once proposed to, always holds someone, so a proposer rejected by everyone would mean all n receivers are held by the other $n - 1$ proposers, which is impossible. **It's stable:** suppose p prefers r to the partner p ended with. Then p proposed to r earlier and was rejected, or later dropped, in favour of someone r preferred, and since receivers only ever trade up, r prefers her final partner to p . So (p, r) doesn't block. ■

The algorithm isn't neutral. Every proposer gets the best partner they could have in *any* stable matching, and every receiver gets the worst. The example measures it with 1,000 on each side and random preferences. Proposers end up with, on average, their 8th choice and receivers with their 123rd. Swap the roles with the same preferences, and the former proposers fall to an average of 155th. The National Resident Matching Program, which places US medical graduates in residencies, runs a variant of this algorithm, and in 2012 Shapley and Alvin Roth received the Nobel prize in economics for this work. After a long debate about who should propose, the program now runs it with applicants proposing.

Planarity

A graph is **planar** if it can be drawn in the plane with no edges crossing, which matters for laying out circuits on a single layer and drawing readable diagrams. A planar drawing divides the plane into **faces**, counting the unbounded outside, and Euler's formula connects the three counts. For any connected planar drawing,

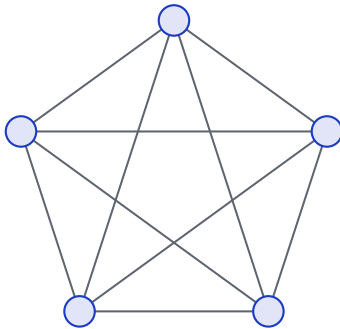
$$V - E + F = 2.$$

Proof by induction on E . A connected graph with the fewest edges is a tree, with $E = V - 1$ and one face, and $V - (V - 1) + 1 = 2$. Adding an edge between two vertices of a connected drawing, without crossings, cuts one face into two: E and F each go up by 1, and the sum is unchanged. ■ The example checks the cube, the octahedron, the icosahedron and a 10×10 grid.

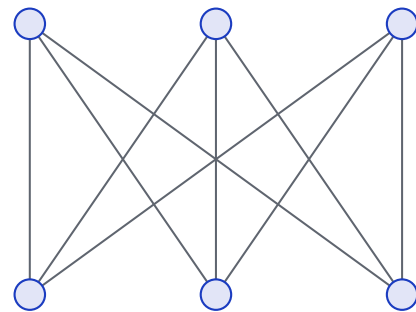
The formula proves impossibility. In a simple planar graph with at least three vertices, every face is bounded by at least 3 edges and every edge borders at most 2 faces, so $3F \leq 2E$. Substituting into Euler's formula gives

$$E \leq 3V - 6.$$

K_5 has 10 edges, but $3 \times 5 - 6 = 9$, so it can't be planar. If the graph has no triangles, every face needs at least 4 edges, which gives $E \leq 2V - 4$. $K_{3,3}$, the "three houses, three utilities" puzzle, has 9 edges against $2 \times 6 - 4 = 8$, so it's not planar either.



K_5 : 10 edges $> 3V - 6 = 9$



$K_{3,3}$: 9 edges $> 2V - 4 = 8$

Every drawing of these two graphs has a crossing, and the edge counts prove it without trying any drawings. Kuratowski's theorem (1930) says they are the only obstacles: a graph is planar exactly when it doesn't contain a subdivided copy of either one.

The same inequality gives planar graphs a vertex of degree at most 5. If every degree were 6 or more, the degree sum $2E$ would be at least $6V$, contradicting $E \leq 3V - 6$. Remove that vertex, colour the rest by induction, and put it back. At most 5 neighbours block at most 5 colours, so **every planar graph is 6-colourable**. The famous **four colour theorem** brings this down to 4. Kenneth Appel and Wolfgang Haken proved it in 1976 with a computer checking nearly two thousand configurations, the first major theorem proved that way, and Georges Gonthier verified it entirely in the Coq proof assistant in 2005.

The algorithms, measured

The example compares greedy colouring orders, runs Gale–Shapley at scale and checks for blocking pairs, finds a maximum matching and the set that violates Hall's condition, and confirms Euler's formula and the edge bounds.

EXAMPLE

matching.py · greedy colouring, stable and maximum matching, and Euler's formula

```
import random
from itertools import combinations

random.seed(8)
# 1. Greedy colouring: never more than (max degree + 1) colours, but the order matters
def greedy(order, adj):
    colour = {}
    for v in order:
        taken = {colour[w] for w in adj[v] if w in colour}
        colour[v] = next(c for c in range(len(adj) + 1) if c not in taken)
    return colour

k = 8 # crown graph: u_i joined to v_j whenever i != j
crown = {("u", i): [("v", j) for j in range(k) if j != i] for i in range(k)}
crown.update({"v", j): [("u", i) for i in range(k) if i != j] for j in range(k)}
bad = [x for i in range(k) for x in (("u", i), ("v", i))]
good = [("u", i) for i in range(k)] + [("v", i) for i in range(k)]
print(f"crowd graph on {2 * k} vertices (bipartite): greedy uses {len(set(greedy(bad,
    crown).values()))} colours "
      f"in one order, {len(set(greedy(good, crown).values()))} in another")

n = 300
adj = {v: [] for v in range(n)}
for a, b in combinations(range(n), 2):
    if random.random() < 0.05: adj[a].append(b); adj[b].append(a)
maxdeg = max(len(x) for x in adj.values())
plain = len(set(greedy(range(n), adj).values()))
by_degree = len(set(greedy(sorted(adj, key=lambda v: -len(adj[v])), adj).values()))
print(f"random graph, max degree {maxdeg}: greedy uses {plain} colours (bound {maxdeg + 1});
      largest-degree-first: {by_degree}")

# 2. Stable matching: Gale-Shapley with 1,000 proposers and 1,000 receivers
def gale_shapley(prop_pref, rcv_pref):
    m = len(prop_pref)
    rank = [{p: r for r, p in enumerate(prefs)} for prefs in rcv_pref]
    nxt, partner, free, proposals = [0] * m, [None] * m, list(range(m)), 0
    while free:
        p = free.pop()
        r = prop_pref[p][nxt[p]]; nxt[p] += 1; proposals += 1
        if partner[r] is None: partner[r] = p
        elif rank[r][p] < rank[r][partner[r]]: free.append(partner[r]); partner[r] = p
        else: free.append(p)
    return {partner[r]: r for r in range(m)}, proposals

m = 1000
A = [random.sample(range(m), m) for _ in range(m)]
B = [random.sample(range(m), m) for _ in range(m)]
match, proposals = gale_shapley(A, B)
rankA = [{x: i for i, x in enumerate(p)} for p in A]; rankB = [{x: i for i, x in enumerate(p)}
    for p in B]
husband = {r: p for p, r in match.items()}
blocking = sum(1 for p in range(m) for r in A[p][:rankA[p][match[p]]] if rankB[r][p] < rankB[r]
    [husband[r]])
print(f"\nGale-Shapley, {m}, a side: {proposals}, proposals (at most {m * m}), blocking
    pairs: {blocking}")
avg_prop = sum(rankA[p][match[p]] for p in range(m)) / m + 1
avg_rcv = sum(rankB[r][husband[r]] for r in range(m)) / m + 1
```

```

print(f"    average rank of partner: proposers {avg_prop:.1f}, receivers {avg_rcv:.1f} (1 =
    first choice)")
match2, _ = gale_shapley(B, A)                                # same preferences, roles swapped
print(f"    roles swapped: the A side now averages {sum(rankA[a][b] for b, a in match2.items())
    / m + 1:.1f}")

# 3. Hall's condition: a perfect matching exists iff every set S of people likes at least |S|
    jobs
likes = {"ann": {"db", "web"}, "bob": {"db"}, "cat": {"db", "web"}, "dan": {"ops", "ml", "db"}}
def max_matching(likes):
    owner = {}
    def augment(p, seen):
        for j in likes[p]:
            if j not in seen:
                seen.add(j)
                if j not in owner or augment(owner[j], seen): owner[j] = p; return True
        return False
    return sum(augment(p, set()) for p in likes), owner
size, owner = max_matching(likes)
worst = min((s for r in range(1, 5) for s in combinations(likes, r)),
    key=lambda s: len(set().union(*(likes[p] for p in s))) - len(s))
print(f"\nmaximum matching: {size} of {len(likes)} people; Hall violation: {'',
    '.join(sorted(worst))} like only "
    f"{'', '.join(sorted(set().union(*(likes[p] for p in worst))))}")

# 4. Euler's formula  $V - E + F = 2$  on planar graphs, and the edge bounds that rule out  $K_5$  and
     $K_{3,3}$ 
g = 10                                # a g x g grid of points: count its parts
    directly
gV = g * g
gE = sum(1 for r in range(g) for c in range(g) for dr, dc in ((0, 1), (1, 0)) if r + dr < g and
    c + dc < g)
gF = (g - 1) ** 2 + 1                # the unit squares, plus the outside
for name, V, E, F in (("cube", 8, 12, 6), ("octahedron", 6, 12, 8), ("icosahedron", 12, 30,
    20), (f"{g}x{g} grid", gV, gE, gF)):
    print(f"{name:12} V - E + F = {V} - {E} + {F} = {V - E + F}")
print(f"K5:    E = {len(list(combinations(range(5), 2)))} > 3V - 6 = {3 * 5 - 6}, so K5 is not
    planar")
print(f"K3,3: E = {3 * 3} > 2V - 4 = {2 * 6 - 4} (no triangles), so K3,3 is not planar")

```

OUTPUT

crown graph on 16 vertices (bipartite): greedy uses 8 colours in one order, 2 in another
 random graph, max degree 29: greedy uses 9 colours (bound 30); largest-degree-first: 8

Gale-Shapley, 1,000 a side: 8,133 proposals (at most 1,000,000), blocking pairs: 0
 average rank of partner: proposers 8.1, receivers 122.9 (1 = first choice)
 roles swapped: the A side now averages 154.7

maximum matching: 3 of 4 people; Hall violation: ann, bob, cat like only db, web
 cube V - E + F = 8 - 12 + 6 = 2
 octahedron V - E + F = 6 - 12 + 8 = 2
 icosahedron V - E + F = 12 - 30 + 20 = 2
 10x10 grid V - E + F = 100 - 180 + 82 = 2
 K5: E = 10 > 3V - 6 = 9, so K5 is not planar
 K3,3: E = 9 > 2V - 4 = 8 (no triangles), so K3,3 is not planar

Gale-Shapley needed 8,133 proposals, against the worst-case bound of a million. With random preferences the expected number is about $n \ln n$, roughly 6,900 for $n = 1,000$, which is the coupon collector's problem of chapter 13 in disguise. The stability check went through every pair a proposer prefers to their partner and found no blocking pair.

COMMON TRAPS

Reporting a greedy colouring as optimal. It gives an upper bound, and the order can make it arbitrarily bad.

Checking Hall's condition one person at a time. The obstacle is usually a group.

Assuming a stable matching is fair to both sides. It's optimal for whoever proposes.

Using $E \leq 3V - 6$ to prove planarity. It's necessary, not sufficient.

Applying Euler's formula to a disconnected drawing, where $V - E + F = 1 + (\text{number of components})$.

Forgetting the outer face when counting faces.

EXERCISES

EXERCISE 8.1 Two stable matchings

Proposers a_1, a_2 and receivers r_1, r_2 have these preferences: a_1 prefers r_1 , a_2 prefers r_2 , r_1 prefers a_2 , r_2 prefers a_1 . Find all stable matchings, and say which one Gale–Shapley produces when the a's propose and when the r's propose.

SOLUTION

There are two matchings. In $\{a_1r_1, a_2r_2\}$, both proposers have their first choice, so neither wants to leave, and it's stable. In $\{a_1r_2, a_2r_1\}$, both receivers have their first choice, so it's stable too. When the a's propose, each proposes to their first choice, nobody is rejected, and the result is $\{a_1r_1, a_2r_2\}$, best for the a's and worst for the r's. When the r's propose, the result is the other matching. Both are stable, and each side's best stable outcome is the other side's worst, which is the general pattern.

EXERCISE 8.2 A card trick from Hall's theorem

Deal a shuffled 52-card deck into 13 piles of 4. Prove that you can always choose one card from each pile so that the 13 chosen cards contain exactly one of each rank, ace to king.

SOLUTION

Build a bipartite graph with the 13 piles on the left and the 13 ranks on the right, joining a pile to a rank if the pile contains a card of that rank. A choice of one card per pile, with all ranks different, is a perfect matching. Check Hall's condition: take any k piles. They hold $4k$ cards. Each rank has only 4 cards in the whole deck, so those $4k$ cards must cover at least k different ranks, so $|N(S)| \geq |S|$. By Hall's theorem a perfect matching exists. ■ The same argument shows that every regular bipartite graph, where every vertex has the same degree, has a perfect matching.

EXERCISE 8.3 The degree-5 vertex, carefully

Prove $E \leq 3V - 6$ for a connected simple planar graph with $V \geq 3$. Where does the proof use "simple", and where does it use $V \geq 3$?

SOLUTION

Count edge-face incidences: for each face, count the edges on its boundary, an edge that touches the face on both sides counting twice. Each edge contributes exactly 2 to the total. Each face has a boundary of length at least 3, because a face bounded by 1 edge would need a loop and a face bounded by 2 would need two edges between the same vertices, both excluded in a simple graph. When $V \geq 3$, the graph isn't a single edge, whose one face has boundary length 2. So $3F \leq 2E$, and $F \leq 2E/3$. Euler's formula gives $2 = V - E + F \leq V - E + 2E/3$, so $E \leq 3V - 6$. ■ A multigraph breaks the bound immediately, since two vertices can be joined by any number of parallel edges without crossings.

CHECKLIST

- ☐ Can you prove the $\Delta + 1$ bound and show that greedy colouring depends on the order?
- ☐ Can you state Hall's theorem, and find a violating set when a matching fails?
- ☐ Can you run Gale-Shapley by hand and prove it terminates in a stable matching?
- ☐ Can you prove Euler's formula and use it to show that K_5 and $K_{3,3}$ aren't planar?
- ☐ Can you prove that every planar graph is 6-colourable?

GO DEEPER

PAPER	College admissions and the stability of marriage Gale & Shapley, 1962
ARTICLE	Stable matching: theory, evidence, and practical design The Nobel Prize, 2012
BOOK	Graph Theory, chapters 2, 4 and 5 Reinhard Diestel

PART III

Counting

Three chapters on counting things without listing them: the rules for sequences, selections and arrangements; the pigeonhole principle and inclusion–exclusion for overlaps and impossibilities; and the sums and recurrences that count the steps an algorithm takes. Probability, in Part IV, is counting divided by counting.

Counting: sequences, selections and arrangements

3 exercises

Counting is the ground floor of probability and of algorithm analysis. A handful of rules covers almost everything: sum, product, complement, division and bijection, plus the binomial coefficients that count selections. The skill is choosing a way to count in which every object is counted exactly once. A computer makes the habit of checking small cases by brute force almost free, and this chapter's example does exactly that, up to classifying every one of the 2,598,960 possible poker hands.

The sum, product and complement rules

The **sum rule**: if a set splits into disjoint parts, its size is the sum of their sizes. The **product rule**: the number of sequences $(a_1, \dots, a_k)$ with a_i drawn from a set of size n_i is $n_1 n_2 \cdots n_k$. It extends to choices that depend on earlier ones, provided the *number* of options at each step doesn't. That's the **generalised product rule**, which counts ordered choices without repetition below. There are $62^8 \approx 2.2 \times 10^{14}$ passwords of 8 characters drawn from letters and digits, because each position independently takes one of 62 values.

The **complement rule** counts what you want by subtracting what you don't. Passwords with at least one digit are hard to count directly, since the digit could be anywhere and there could be several. Passwords with *no* digit are easy, 52^8 , so the answer is $62^8 - 52^8$. Whenever a problem says "at least one", try the complement first.

Permutations, combinations and the division rule

There are $n!$ ways to order n distinct objects, and $P(n, k) = n!/(n - k)!$ ways to choose an ordered sequence of k of them. To count *unordered* selections, use the **division rule**: if every object you want corresponds to exactly d of the things you counted, divide by d . Each set of k elements corresponds to $k!$ ordered sequences, so the number of k -element subsets of an n -element set is

$$\binom{n}{k} = \frac{n!}{k!(n - k)!}.$$

The division rule handles repetition too. The 11 letters of MISSISSIPPI can be ordered in $11!$ ways if you pretend the repeated letters are distinguishable, and every distinct word arises from $4! 4! 2!$ of those orderings, since the four I's, four S's and two P's can be shuffled among themselves. So there are $11!/(4! 4! 2!) = 34,650$ distinct words. People seated around a round table, where only relative positions matter, can be arranged in $n!/n = (n - 1)!$ ways, because each seating corresponds to n rotations.

Bijections

The **bijection rule** (chapter 4) says that two sets in one-to-one correspondence have the same size. It turns a hard count into an easy one. Subsets of an n -element set correspond to bit strings of length n , one bit per element, so there are 2^n of them.

Stars and bars counts the ways to write 10 as $x_1 + x_2 + x_3$ with nonnegative integers.

Represent a solution as 10 stars and 2 bars in a row, x_1 stars, a bar, x_2 stars, a bar, x_3 stars.

Every arrangement of 10 stars and 2 bars is a solution and vice versa, so there are $\binom{12}{2} = 66$ solutions. In general, k nonnegative integers summing to n can be chosen in $\binom{n+k-1}{k-1}$ ways.

Similarly, monotone paths across a 6×6 grid correspond to strings of six R's and six U's, so there are $\binom{12}{6} = 924$.

Binomial coefficients and Pascal's triangle

The name comes from the **binomial theorem**:

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

Expanding the product of n factors $(x + y)$ means choosing x or y from each factor, and the term $x^k y^{n-k}$ arises once for each way of choosing which k factors contribute x . Setting $x = y = 1$ gives $\sum_k \binom{n}{k} = 2^n$, which is another proof that a set has 2^n subsets.

Pascal's rule, $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, has a **combinatorial proof**, an argument that counts one set in two ways. Fix one element of the n -element set. A k -subset either contains it, and then the other $k-1$ elements come from the remaining $n-1$, or it doesn't, and all k come from the remaining $n-1$. Combinatorial proofs are often shorter than algebra and explain why an identity holds.

								ROW SUM
$n = 0$				1				1
$n = 1$			1		1			2
$n = 2$			1		2		1	4
$n = 3$			1		3		3	8
$n = 4$			1		4		6	16
$n = 5$			1		5		10	32
$n = 6$			1		6		15	64

each entry is the sum of the two above it: $C(6, 3) = C(5, 2) + C(5, 3)$

Row n lists $\binom{n}{0}$ to $\binom{n}{n}$. Pascal's rule builds each row from the one above, and the row sums double each time because every subset of an n -set either contains a given element or doesn't.

Every count, checked

The example checks each rule against brute-force enumeration: ordered and unordered choices, the MISSISSIPPI words, stars and bars, the complement rule on a small alphabet, every 5-card poker hand classified one by one, and Pascal's rule and grid paths.

EXAMPLE

counting.py · the counting rules checked against brute force, up to every poker hand

```
from itertools import combinations, permutations, product
from collections import Counter
from math import comb, perm, factorial

# 1. The basic rules, checked against brute force
print(f"ordered choices of 3 from 10: {sum(1 for _ in permutations(range(10), 3))} = P(10,3) = {perm(10, 3)}")
print(f"unordered choices of 3 from 10: {sum(1 for _ in combinations(range(10), 3))} = C(10,3) = {comb(10, 3)}")

# 2. The division rule: arrangements of MISSISSIPPI, generated without duplicates
def arrangements(counts):
    if not any(counts.values()): yield ""; return
    for ch in counts:
        if counts[ch]:
            counts[ch] -= 1
            for rest in arrangements(counts): yield ch + rest
            counts[ch] += 1
n_arr = sum(1 for _ in arrangements(Counter("MISSISSIPPI")))
print(f"arrangements of MISSISSIPPI: {n_arr}, {n_arr} = 11!/(1! 4! 4! 2!) = {factorial(11) // (factorial(4) ** 2 * factorial(2))}, {n_arr}")

# 3. Stars and bars: nonnegative solutions of  $x_1 + x_2 + x_3 = 10$ 
sols = sum(1 for x in range(11) for y in range(11 - x)) # z = 10 - x - y is forced
print(f"solutions of  $x_1 + x_2 + x_3 = 10$  in nonnegative integers: {sols} = C(12,2) = {comb(12, 2)}")

# 4. The complement rule: passwords with at least one digit, brute force on a small alphabet
alpha = "abc12"
with_digit = sum(1 for w in product(alpha, repeat=4) if any(ch.isdigit() for ch in w))
print(f"4-character strings over 'abc12' with a digit: {with_digit} =  $5^4 - 3^4 = {5 ** 4 - 3 ** 4}$ ")
print(f"8-character passwords over 62 symbols: {62 ** 8}, {62 ** 8}; with at least one digit: {62 ** 8 - 52 ** 8}, {62 ** 8 - 52 ** 8}")

# 5. Every 5-card poker hand, classified
deck = range(52) # rank = card // 4, suit = card % 4
kinds = Counter()
for hand in combinations(deck, 5):
    shape = sorted(Counter(c >> 2 for c in hand).values(), reverse=True)
    if shape[0] == 4: kinds["four of a kind"] += 1
    elif shape[:2] == [3, 2]: kinds["full house"] += 1
    elif len({c & 3 for c in hand}) == 1: kinds["flush (any)"] += 1
    elif shape[0] == 3: kinds["three of a kind"] += 1
    elif shape[:2] == [2, 2]: kinds["two pair"] += 1
    elif shape[0] == 2: kinds["one pair"] += 1
print(f"\nall {comb(52, 5)}, {comb(52, 5)} five-card hands, counted one by one, against the formulas:")
formulas = {"four of a kind": 13 * 48, "full house": 13 * comb(4, 3) * 12 * comb(4, 2),
            "flush (any)": 4 * comb(13, 5), "three of a kind": 13 * comb(4, 3) * comb(12, 2) * 4 ** 2,
            "two pair": comb(13, 2) * comb(4, 2) ** 2 * 11 * 4, "one pair": 13 * comb(4, 2) * comb(12, 3) * 4 ** 3}
```

```

for k, f in formulas.items():
    print(f"    {k:16} {kinds[k]:>9,}    formula {f:>9,}")

# 6. Pascal's rule and the binomial theorem
row = [comb(12, k) for k in range(13)]
print(f"\nrow 12 of Pascal's triangle sums to {sum(row):,} = 2^12; C(12,6) = C(11,5) + C(11,6)
      = "
      f"{comb(11, 5)} + {comb(11, 6)} = {comb(12, 6)}")
paths = [[1] * 7 for _ in range(7)]
for i in range(1, 7):
    for j in range(1, 7): paths[i][j] = paths[i - 1][j] + paths[i][j - 1]
print(f"monotone paths across a 6 x 6 grid: {paths[6][6]} = C(12,6) = {comb(12, 6)}")

```

OUTPUT

```

ordered choices of 3 from 10: 720 = P(10,3) = 720
unordered choices of 3 from 10: 120 = C(10,3) = 120
arrangements of MISSISSIPPI: 34,650 = 11!/(1! 4! 4! 2!) = 34,650
solutions of x1 + x2 + x3 = 10 in nonnegative integers: 66 = C(12,2) = 66
4-character strings over 'abc12' with a digit: 544 = 5^4 - 3^4 = 544
8-character passwords over 62 symbols: 218,340,105,584,896; with at least one digit:
    164,880,377,053,440

all 2,598,960 five-card hands, counted one by one, against the formulas:
four of a kind          624    formula    624
full house              3,744    formula    3,744
flush (any)             5,148    formula    5,148
three of a kind        54,912    formula    54,912
two pair               123,552    formula   123,552
one pair              1,098,240    formula 1,098,240

row 12 of Pascal's triangle sums to 4,096 = 2^12; C(12,6) = C(11,5) + C(11,6) = 462 + 462 = 924
monotone paths across a 6 x 6 grid: 924 = C(12,6) = 924

```

The poker table is a good test of the method, because each formula is a product-rule argument that's easy to get subtly wrong. A full house is a rank for the triple (13 ways), three of its four suits ($\binom{4}{3}$), a different rank for the pair (12) and two of its suits ($\binom{4}{2}$): 3,744. Two pair needs care. The two pair ranks form an unordered set, $\binom{13}{2}$, not 13×12 , which would count every hand twice. Then come the suits for each pair and a fifth card from the 11 remaining ranks. Counting all 2,598,960 hands takes the example a few seconds and confirms all six formulas exactly. The "flush" line includes the 40 straight flushes, which a full hand ranking would separate out.

COMMON TRAPS

Counting ordered selections when order doesn't matter, the " 13×12 for two pair" mistake, which overcounts by exactly the number of orderings.

Adding counts of sets that overlap. The sum rule needs disjoint parts; overlaps need chapter 10.

Forgetting the division rule for repeated items, as in words with repeated letters.

Off-by-one errors in stars and bars: "at least one each" and "zero allowed" give different formulas.

Applying the product rule when the number of options changes depending on earlier choices.

Trusting a formula you haven't checked on a small case, when brute force would take seconds.

EXERCISES

EXERCISE 9.1 A combinatorial proof

Prove that $\sum_{k=0}^n \binom{n}{k}^2 = \binom{2n}{n}$ by counting one set in two ways.

SOLUTION

The right side counts the ways to choose a committee of n people from n women and n men. Count the same committees by the number k of women: choose the k women in $\binom{n}{k}$ ways and the $n - k$ men in $\binom{n}{n-k} = \binom{n}{k}$ ways, so committees with k women number $\binom{n}{k}^2$. Summing over k counts every committee exactly once, so both sides are equal. ■
Check with $n = 2$: $1 + 4 + 1 = 6 = \binom{4}{2}$.

EXERCISE 9.2 No two adjacent

How many bit strings of length 10 have exactly four 1s, no two of them adjacent?

SOLUTION

Write down the six 0s first. They leave seven gaps: one before the first 0, five between 0s, and one after the last. Placing the four 1s in four different gaps guarantees no two are adjacent, and every valid string arises from exactly one choice of gaps. That's a bijection with 4-element subsets of the 7 gaps, so the answer is $\binom{7}{4} = 35$. In general, choosing k non-adjacent positions from n can be done in $\binom{n-k+1}{k}$ ways, which counts things like non-overlapping time slots.

EXERCISE 9.3 Jobs on servers

Ten identical jobs are assigned to four servers. How many assignments are there if servers may be idle? If every server must get at least one job? If the jobs are all different?

SOLUTION

Idle allowed: nonnegative solutions of $x_1 + x_2 + x_3 + x_4 = 10$, which stars and bars counts as $\binom{13}{3} = 286$. At least one each: give every server one job first, then distribute the remaining 6 freely, $\binom{9}{3} = 84$. Different jobs, idle allowed: each job independently picks a server, $4^{10} = 1,048,576$ by the product rule. Different jobs with every server busy is the number of onto functions from a 10-set to a 4-set, which needs inclusion–exclusion (chapter 10): $4^{10} - 4 \cdot 3^{10} + 6 \cdot 2^{10} - 4 = 818,520$. Whether the objects are identical changes the answer completely, so decide that first.

CHECKLIST

- ☐ Can you apply the sum, product and complement rules, and say when each applies?
- ☐ Can you derive $\binom{n}{k}$ from the division rule?
- ☐ Can you count words with repeated letters, and solutions by stars and bars?
- ☐ Can you give a combinatorial proof of Pascal's rule?
- ☐ Do you check counting formulas against brute force on small cases?

GO DEEPER

COURSE 6.042J: counting rules and combinatorial proofs MIT OpenCourseWare

BOOK Book of Proof, chapter 3: counting Richard Hammack

BOOK generatingfunctionology Herbert Wilf

Pigeonhole and inclusion–exclusion

3 exercises

Two principles extend the counting rules. The pigeonhole principle says that if you put more objects than boxes, some box gets two. It sounds trivial, yet it proves that no compressor shrinks every file and that every hash function has collisions, and it proves things exist without finding them. Inclusion–exclusion counts overlapping sets exactly by adding and subtracting in alternation. It explains why the chance that nobody gets their own hat back is almost exactly $1/e$, and it counts the passwords that satisfy a policy.

The pigeonhole principle

If $n + 1$ objects go into n boxes, some box holds at least two. More generally, if N objects go into k boxes, some box holds at least $\lceil N/k \rceil$. In the language of chapter 4: there's no one-to-one function from a larger finite set to a smaller one.

The principle proves several facts engineers rely on. **Every hash function has collisions:** a 64-bit hash maps infinitely many inputs to 2^{64} outputs, so some outputs are shared. That's unavoidable, and the design question is only whether collisions are hard to find. **No lossless compressor can shrink every input:** there are 2^n strings of n bits but only $2^n - 1$ strings of all shorter lengths combined, so if a compressor shrank every n -bit string, two would compress to the same output and couldn't both be recovered. A compressor that shrinks some inputs must grow others, and the example shows zlib turning 10,000 random bytes into 10,011 while crushing 10,000 bytes of repetitive text into 99. And with 367 people, two must share a birthday. Chapter 12 shows the far more surprising fact that 23 people give better-than-even odds.

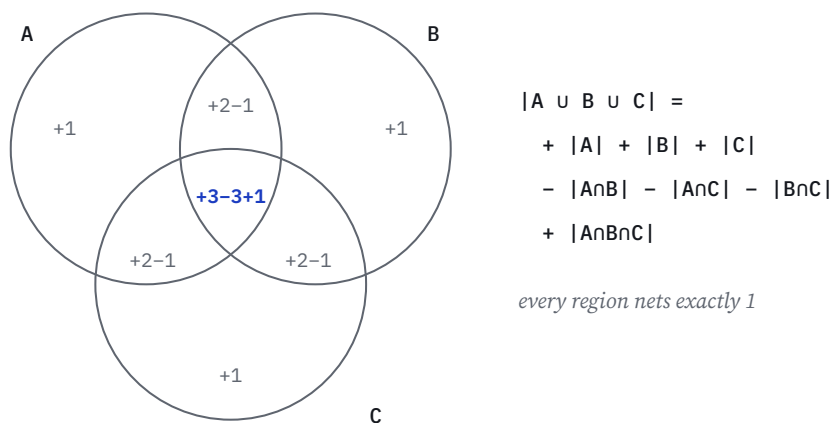
Order in every sequence

Pigeonhole arguments often hide in the choice of boxes. **Erdős–Szekeres theorem:** every sequence of $n^2 + 1$ distinct numbers contains an increasing or a decreasing subsequence of length $n + 1$. **Proof.** Label each position i with a pair (a_i, d_i) : the lengths of the longest increasing and longest decreasing subsequences ending at position i . For two positions $i < j$, if the j -th number is larger, then $a_j > a_i$, since it extends any increasing subsequence ending at i ; if smaller, $d_j > d_i$. So no two positions have the same label. If every a_i and d_i were at most n , there would be only n^2 possible labels for $n^2 + 1$ positions, so two would coincide. Contradiction. ■ The example confirms the theorem on 10,000 random sequences of $10 = 3^2 + 1$ numbers, and shows a sequence of 9 that escapes it, so the bound is tight.

Inclusion–exclusion

The sum rule needs disjoint sets. For overlapping sets, adding the sizes counts the overlap twice, so subtract it: $|A \cup B| = |A| + |B| - |A \cap B|$. With three sets the pairwise overlaps are subtracted, but that removes the triple overlap once too often, so it's added back:

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



Follow any single element. If it lies in exactly two of the sets, it's added twice by the first terms and removed once by the pairwise terms. If it lies in all three, it's added three times, removed three times and added back once. Either way it's counted exactly once.

The general formula alternates over all intersections of 1, 2, 3, ... of the sets. It's correct because an element lying in exactly m of the sets is counted $\binom{m}{1} - \binom{m}{2} + \binom{m}{3} - \dots = 1$ times, by the binomial theorem applied to $(1 - 1)^m = 0$. Inclusion-exclusion usually pairs with the complement rule. It's easier to count the objects that violate at least one condition and subtract them from the total than to count those that satisfy every condition directly.

Four applications

Divisibility. How many numbers from 1 to 1000 are divisible by none of 2, 3 and 5? Let A_2, A_3, A_5 be the multiples of each. Then $|A_2| = 500$, $|A_3| = 333$, $|A_5| = 200$, the pairwise intersections are the multiples of 6, 10 and 15 (166, 100 and 66), and the triple intersection is the multiples of 30 (33). So the answer is $1000 - 1033 + 332 - 33 = 266$. The same reasoning gives **Euler's totient** from chapter 6 as a product over the distinct primes dividing n :

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right), \quad \varphi(360) = 360 \cdot \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{4}{5} = 96.$$

Onto functions. Assign 6 distinct tasks to 3 workers so that nobody is idle. Of the 3^6 assignments, subtract those that miss some particular worker, $3 \cdot 2^6$, and add back those that miss two particular workers, $3 \cdot 1^6$: $729 - 192 + 3 = 540$.

Derangements. n people check their hats and get them back at random. How likely is it that nobody gets their own? A permutation with no fixed point is a **derangement**. Let A_i be the permutations that fix person i . Any j particular people can be fixed in $(n - j)!$ ways, so inclusion–exclusion gives the number of derangements

$$D_n = \sum_{j=0}^n (-1)^j \binom{n}{j} (n - j)! = n! \sum_{j=0}^n \frac{(-1)^j}{j!}.$$

The sum is the start of the series for e^{-1} , so $D_n/n! \rightarrow 1/e \approx 0.368$, and in fact D_n is $n!/e$ rounded to the nearest integer. Remarkably, the probability barely depends on n : about 37% whether 9 people check hats or 9,000.

Password policies. How many 8-character passwords over letters and digits contain at least one lowercase letter, one uppercase letter and one digit? Count the violators. Passwords with no lowercase letter number 36^8 , with no uppercase 36^8 , with no digit 52^8 . Those lacking two kinds at once number 10^8 , 26^8 and 26^8 , and none lack all three. So the policy admits $62^8 - (2 \cdot 36^8 + 52^8) + (10^8 + 2 \cdot 26^8)$ passwords, about 73% of all of them. A policy meant to strengthen passwords actually shrinks the space attackers must search, by about a quarter.

Checked by enumeration

The example verifies the compression count, Erdős–Szekeres, and each inclusion–exclusion formula above against brute force, including every permutation of up to 9 hats.

EXAMPLE

pigeonhole.py · compression, monotone subsequences, and inclusion–exclusion against brute force

```
import math, random, zlib
from itertools import permutations, product

random.seed(10)
# 1. Pigeonhole: no lossless compressor can shrink every input
n = 16
print(f"bit strings of length {n}: {2 ** n:,}; strings of every shorter length combined: {2 ** n - 1:,}")
data = random.randbytes(10_000)
text = (b"the quick brown fox jumps over the lazy dog. " * 223)[:10_000]
print(f"zlib on 10,000 random bytes: {len(zlib.compress(data, 9)):,} bytes; on 10,000 bytes of English-like text: "
      f"{len(zlib.compress(text, 9)):,} bytes")

# 2. Erdos-Szekeres: any sequence of n^2 + 1 distinct numbers has a monotone run of length n + 1
def longest(seq, up):
    best = []
    for i, x in enumerate(seq):
        best.append(1 + max((best[j] for j in range(i) if (seq[j] < x) == up), default=0))
    return max(best)
trials = [random.sample(range(100), 10) for _ in range(10_000)]
print(f"10,000 random sequences of 10 = 3^2 + 1 numbers: every one has a monotone subsequence")
```

```

    of length 4: "
        f"{all(max(longest(s, True), longest(s, False)) >= 4 for s in trials)}")
s9 = [7, 8, 9, 4, 5, 6, 1, 2, 3]
print(f"but 9 numbers can avoid it: {s9} has longest increasing {longest(s9, True)}, decreasing
    {longest(s9, False)}")

# 3. Inclusion-exclusion: numbers up to 1000 divisible by none of 2, 3, 5
brute = sum(1 for k in range(1, 1001) if k % 2 and k % 3 and k % 5)
ie = 1000 - (500 + 333 + 200) + (166 + 100 + 66) - 33
print(f"\n1..1000 divisible by none of 2, 3, 5: {brute} by brute force, {ie} by inclusion-
    exclusion")
phi = sum(1 for k in range(1, 361) if math.gcd(k, 360) == 1)
print(f"phi(360) = {phi} by counting, 360(1 - 1/2)(1 - 1/3)(1 - 1/5) = {360 * 1 * 2 * 4 // (2 *
    3 * 5)}")

# 4. Onto functions: 6 tasks to 3 workers, nobody idle
onto = sum(1 for f in product(range(3), repeat=6) if len(set(f)) == 3)
print(f"assignments of 6 tasks to 3 workers using all 3: {onto} = 3^6 - 3 x 2^6 + 3 x 1^6 =
    {3**6 - 3 * 2**6 + 3}")

# 5. Derangements: nobody gets their own hat back
print("\n n   derangements   n!/e rounded   share of all n! orders")
for k in range(1, 10):
    d = sum(1 for p in permutations(range(k)) if all(p[i] != i for i in range(k)))
    print(f"{k:2}   {d:>12}   {round(math.factorial(k) / math.e):>12}   {d /
        math.factorial(k):.6f}")
print(f"1/e = {1 / math.e:.6f}")

# 6. Passwords of 8 characters with at least one lowercase, one uppercase and one digit
total = 62 ** 8
good = total - (36 ** 8 + 36 ** 8 + 52 ** 8) + (10 ** 8 + 26 ** 8 + 26 ** 8) - 0
small = sum(1 for w in product("aAb1", repeat=5) if any(c.islower() for c in w)
    and any(c.isupper() for c in w) and any(c.isdigit() for c in w))
print(f"\n8-character passwords with all three kinds: {good:}, of {total:}, ({good /
    total:.1%})")
print(f"same rule, alphabet 'aAb1', length 5: {small} by brute force, "
    f"{4**5 - (2**5 + 3**5 + 3**5) + (1**5 + 2**5 + 1**5)} by inclusion-exclusion")

```

OUTPUT

bit strings of length 16: 65,536; strings of every shorter length combined: 65,535
 zlib on 10,000 random bytes: 10,011 bytes; on 10,000 bytes of English-like text: 99 bytes
 10,000 random sequences of 10 = $3^2 + 1$ numbers: every one has a monotone subsequence of length
 4: True
 but 9 numbers can avoid it: [7, 8, 9, 4, 5, 6, 1, 2, 3] has longest increasing 3, decreasing 3

1..1000 divisible by none of 2, 3, 5: 266 by brute force, 266 by inclusion-exclusion
 phi(360) = 96 by counting, $360(1 - 1/2)(1 - 1/3)(1 - 1/5) = 96$
 assignments of 6 tasks to 3 workers using all 3: $540 = 3^6 - 3 \times 2^6 + 3 \times 1^6 = 540$

n	derangements	n!/e rounded	share of all n! orders
1	0	0	0.000000
2	1	1	0.500000
3	2	2	0.333333
4	9	9	0.375000
5	44	44	0.366667
6	265	265	0.368056
7	1,854	1,854	0.367857
8	14,833	14,833	0.367882
9	133,496	133,496	0.367879

1/e = 0.367879

8-character passwords with all three kinds: 159,655,911,367,680 of 218,340,105,584,896 (73.1%)
 same rule, alphabet 'aAb1', length 5: 540 by brute force, 540 by inclusion-exclusion

The derangement table shows how fast the convergence is. The share of permutations with no fixed point is already 0.367882 at $n = 8$, against $1/e = 0.367879$, because the error of a truncated alternating series is smaller than its first omitted term, here $1/9!$. The same series explains why the rounded value $n!/e$ is always exactly right.

COMMON TRAPS

Expecting pigeonhole to say which box. It proves a collision exists, not where it is.

Choosing the wrong boxes. The hard part of a pigeonhole proof is deciding what the boxes are.

Sign errors in inclusion–exclusion, or stopping before the last intersection term.

Using inclusion–exclusion on the conditions you want instead of on the violations, which is usually much messier.

Believing any compressor can always shrink data. Random and already-compressed data grow.

Assuming password rules increase the search space. They remove options, and help only by banning predictable choices.

EXERCISES

EXERCISE 10.1 Five points in a square

Prove that among any five points in a square of side 1, some two are at distance at most $\sqrt{2}/2$ from each other.

SOLUTION

Cut the square into four smaller squares of side $1/2$. Five points in four squares means some small square contains two of them, counting points on a shared edge as belonging to either square. Two points in a square of side $1/2$ are at most its diagonal apart, which is $\sqrt{(1/2)^2 + (1/2)^2} = \sqrt{2}/2$. ■ The proof doesn't find the two points, which is typical of pigeonhole arguments, and the bound is tight: put four points at the corners and one in the centre.

EXERCISE 10.2 A coprime pair

Prove that any $n + 1$ numbers chosen from $\{1, 2, \dots, 2n\}$ include two that are coprime.

SOLUTION

Use the n boxes $\{1, 2\}, \{3, 4\}, \dots, \{2n - 1, 2n\}$. With $n + 1$ numbers, some box contains two of them, which are consecutive integers k and $k + 1$. Any common divisor of k and $k + 1$ divides their difference, 1, so they're coprime. ■ The bound is tight: the n even numbers $2, 4, \dots, 2n$ contain no coprime pair.

EXERCISE 10.3 Four hats

Compute D_4 by inclusion–exclusion, writing out each term, and check it by listing the derangements of 1234 that start with 2.

SOLUTION

$$D_4 = 4! - \binom{4}{1}3! + \binom{4}{2}2! - \binom{4}{3}1! + \binom{4}{4}0! = 24 - 24 + 12 - 4 + 1 = 9.$$

The derangements starting with 2 are 2143, 2341 and 2413, three of them. By symmetry, the same number start with 3 and with 4, and none start with 1, so there are $3 \times 3 = 9$ in total, matching. The share is $9/24 = 0.375$, already close to $1/e \approx 0.368$, as in the example's table.

CHECKLIST

- ☐ Can you state the generalised pigeonhole principle and use it to prove a collision or an impossibility?
- ☐ Can you explain why no compressor shrinks every input?
- ☐ Can you prove the Erdős–Szekeres theorem?
- ☐ Can you write inclusion–exclusion for three sets and explain why each region is counted once?
- ☐ Can you derive the derangement formula and its limit $1/e$?

GO DEEPER

COURSE 6.042J: pigeonhole and inclusion–exclusion MIT OpenCourseWare

BOOK Analytic Combinatorics Flajolet & Sedgewick

Sums, asymptotics and recurrences

3 exercises

Analysing an algorithm comes down to two kinds of mathematics. Loops become sums, and recursion becomes recurrences. This chapter collects the sums that keep appearing, with their closed forms and approximations, including harmonic numbers and Stirling's formula. It defines asymptotic notation precisely enough to use correctly, and solves recurrences three ways: by drawing the recursion tree, by guessing and proving, and by the master theorem, which settles most divide-and-conquer algorithms in one line. It finishes with linear recurrences, and a closed form for Fibonacci numbers that floating point breaks at $n = 71$.

Sums that keep appearing

An **arithmetic** sum, such as the comparisons of insertion sort on reversed input, has the closed form $\sum_{i=1}^n i = n(n+1)/2$, which chapter 3 proved by induction. A **geometric** sum, with ratio $r \neq 1$, is

$$\sum_{k=0}^n r^k = \frac{r^{n+1} - 1}{r - 1}.$$

Multiply the sum by r , subtract the original, and almost everything cancels. When $r > 1$ the largest term dominates. That's why an array that doubles its capacity copies fewer than twice as many elements in total as its final size: $1 + 2 + 4 + \dots + n < 2n$. When $r < 1$ the first term dominates, and the infinite sum converges to $1/(1-r)$.

The **harmonic numbers** $H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$ appear in the expected cost of randomised quicksort, the coupon collector's problem and the analysis of hashing. Comparing the sum with the integral of $1/x$, the area under a staircase versus the area under a curve, gives $\ln(n+1) \leq H_n \leq 1 + \ln n$, and more precisely

$$H_n = \ln n + \gamma + O(1/n), \quad \gamma \approx 0.57722,$$

where γ is Euler's constant. Factorials have **Stirling's approximation**, $n! \approx \sqrt{2\pi n} (n/e)^n$, with relative error about $1/(12n)$. Taking logarithms gives $\log n! = \Theta(n \log n)$, the bound behind every comparison sort's lower limit in *The Engineer's Workbench*.

Asymptotic notation, precisely

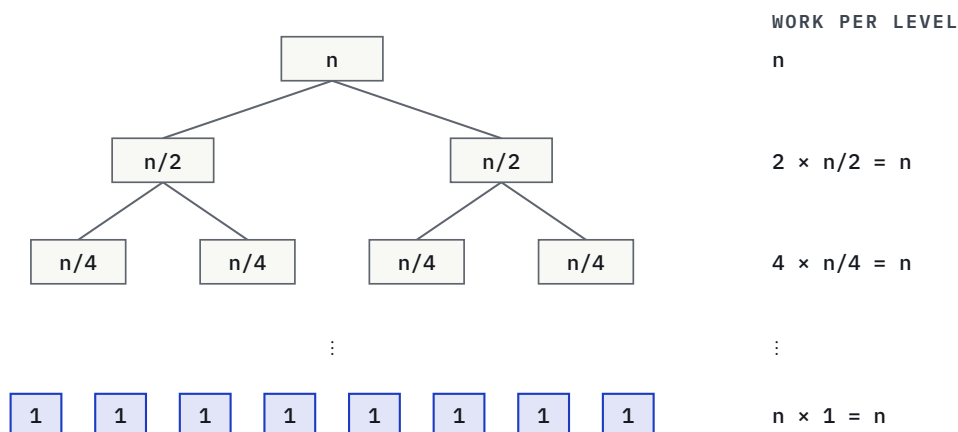
$f(n) = O(g(n))$ means f grows no faster than g up to a constant: there are c and n_0 with $f(n) \leq c g(n)$ for all $n \geq n_0$. Ω is the matching lower bound, and Θ means both. Little- o , $f = o(g)$, means $f/g \rightarrow 0$. A practical test: if $\lim f(n)/g(n)$ exists and is a positive constant, then $f = \Theta(g)$. Three habits avoid most mistakes. O is only an upper bound, so "this algorithm is $O(n^2)$ " is true of a linear algorithm too, and when you mean a tight bound, say Θ . The "=" in $f = O(g)$ is not symmetric, since $O(g)$ is really a set of functions. And the constants hidden by O decide which algorithm wins at the sizes you actually have.

From code to recurrence

A recursive algorithm's cost satisfies the same recursion as the code. Merge sort on n items sorts two halves and merges them with at most n comparisons, so its comparison count satisfies

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, \quad T(1) = 0.$$

To solve a recurrence, draw its **recursion tree**: one node per call, labelled with the work done outside the recursive calls.



$\log_2 n$ levels, each costing n : $n \log_2 n$ in total

The tree for merge sort. Each level splits the same n items into more, smaller pieces, so each level's work adds up to n . Halving reaches pieces of size 1 after $\log_2 n$ levels.

The tree suggests $T(n) = n \log_2 n$ for powers of 2, and a guess can always be proved by induction, which is the **substitution method** (exercise 11.2). For $n = 1,024$ the example computes T exactly as $10,240 = n \log_2 n$. For other n , the floors and ceilings change the answer slightly but not the growth: 9,976 against 9,966 for $n = 1,000$.

The master theorem

Most divide-and-conquer algorithms split a problem of size n into a subproblems of size n/b and do $\Theta(n^c)$ extra work. For

$$T(n) = a T(n/b) + \Theta(n^c),$$

the **master theorem** compares c with $\log_b a$, the exponent that counts the leaves of the recursion tree:

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } c > \log_b a \text{ (the root dominates)} \\ \Theta(n^c \log n) & \text{if } c = \log_b a \text{ (every level costs the same)} \\ \Theta(n^{\log_b a}) & \text{if } c < \log_b a \text{ (the leaves dominate)} \end{cases}$$

The recursion tree explains all three cases. Level j has a^j nodes each doing $(n/b^j)^c$ work, so the level totals form a geometric series with ratio a/b^c . If the ratio is below 1 the first term, the root, dominates. If it's exactly 1, all $\log_b n$ levels contribute equally. If it's above 1 the last level, the leaves, dominates, and there are $a^{\log_b n} = n^{\log_b a}$ of them.

Binary search ($a = 1, b = 2, c = 0$) is the middle case, with $\Theta(\log n)$. Merge sort ($2, 2, 1$) is also the middle case, $\Theta(n \log n)$. Karatsuba's multiplication replaces four half-size products with three ($3, 2, 1$), and the leaves dominate: $\Theta(n^{\log_2 3}) = \Theta(n^{1.585})$, beating the schoolbook n^2 . Strassen's matrix multiplication uses seven half-size products instead of eight, giving $\Theta(n^{\log_2 7}) = \Theta(n^{2.807})$. The theorem doesn't apply when the extra work isn't a polynomial, such as $n/\log n$, or when the subproblems have unequal sizes. Then the Akra–Bazzi method or a direct look at the tree is needed.

Linear recurrences

Recurrences like $F_n = F_{n-1} + F_{n-2}$ have exact solutions. Guess $F_n = x^n$ and substitute, which gives the **characteristic equation** $x^2 = x + 1$, with roots $\varphi = (1 + \sqrt{5})/2 \approx 1.618$ and $\psi = (1 - \sqrt{5})/2 \approx -0.618$. Every solution is a combination $A\varphi^n + B\psi^n$, and the starting values $F_0 = 0, F_1 = 1$ fix the constants, giving **Binet's formula**:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}.$$

Since $|\psi| < 1$, the second term shrinks to nothing and F_n is simply $\varphi^n/\sqrt{5}$ rounded. That's mathematically exact for every n , but the example shows it failing in double-precision floating point at $n = 71$, where F_{71} has 15 digits and the 53-bit significand can no longer carry the error below one half. For exact large Fibonacci numbers, use integer arithmetic with repeated squaring of a 2×2 matrix (chapter 17). The Towers of Hanoi recurrence $M_n = 2M_{n-1} + 1$ is solved by adding 1 to both sides: $M_n + 1 = 2(M_{n-1} + 1)$, so $M_n = 2^n - 1$.

Measured

The example checks the harmonic and Stirling approximations, computes the merge sort recurrence exactly, measures $T(n)$ divided by the master theorem's prediction for six recurrences, finds where Binet's formula fails in floating point, and counts Hanoi moves by

recursion.

EXAMPLE

recurrences.py · harmonic numbers, Stirling, the master theorem and Binet's formula, measured

```
import math
from functools import lru_cache

# 1. Sums: harmonic numbers and Stirling's approximation
for n in (10, 1000, 10 ** 6):
    H = math.fsum(1 / k for k in range(1, n + 1))
    print(f"H({n:>9},) = {H:9.5f}; H - ln n = {H - math.log(n):.5f}")
print(f"Euler's constant gamma = 0.57722 (the limit)")
for n in (10, 100, 1000):
    exact = math.lgamma(n + 1) # ln(n!)
    stirling = n * math.log(n) - n + 0.5 * math.log(2 * math.pi * n)
    print(f"n = {n:4}: n! / Stirling = {math.exp(exact - stirling):.6f} (1 + 1/(12n) = {1 + 1 / (12 * n):.6f})")

# 2. Merge sort: T(n) = T(ceil(n/2)) + T(floor(n/2)) + n, T(1) = 0
@lru_cache(None)
def T(n): return 0 if n <= 1 else T((n + 1) // 2) + T(n // 2) + n
print("\nmerge sort recurrence: " + ", ".join(f"T({n:},) = {T(n):}, vs n log2 n = {n * math.log2(n):,.0f}"
                                             for n in (1024, 1000, 10 ** 6)))

# 3. The master theorem: T(n) = a T(n/b) + n^c, measured at n = 2^k
def solve(a, b, c, n):
    return 1 if n <= 1 else a * solve(a, b, c, n // b) + n ** c
cases = [("binary search", 1, 2, 0, lambda n: math.log2(n) + 1),
         ("merge sort", 2, 2, 1, lambda n: n * math.log2(n)),
         ("Karatsuba", 3, 2, 1, lambda n: n ** math.log2(3)),
         ("4 halves, linear work", 4, 2, 1, lambda n: n ** 2),
         ("2 halves, quadratic work", 2, 2, 2, lambda n: n ** 2),
         ("Strassen", 7, 2, 2, lambda n: n ** math.log2(7))]
print(f"\n{'recurrence':26} {'predicted':>14} T(n) / predicted at n = 2^10, 2^15, 2^20")
for name, a, b, c, f in cases:
    ratios = [solve(a, b, c, 2 ** k) / f(2 ** k) for k in (10, 15, 20)]
    e = math.log(a, b) # compare the leaves' exponent log_b a with c
    pred = ("log n" if (a, c) == (1, 0) else "n log n" if a == b ** c else
            (f"n^{e:.3f}" if abs(e - round(e)) > 1e-9 else f"n^{round(e)}") if a > b ** c else
            f"n^{c}")
    print(f"{'name':26} {'pred':>14} " + " ".join(f"{r:6.3f}" for r in ratios))

# 4. A linear recurrence: Fibonacci, and the closed form with floating point
phi = (1 + math.sqrt(5)) / 2
fib = [0, 1]
while len(fib) < 100: fib.append(fib[-1] + fib[-2])
bad = next(n for n in range(100) if round(phi ** n / math.sqrt(5)) != fib[n])
print(f"\nF(n) = round(phi^n / sqrt 5) holds exactly for n = 0..{bad - 1}; at n = {bad}
      floating point gives "
      f"{round(phi ** bad / math.sqrt(5)):,} instead of {fib[bad]:,}")

# 5. Towers of Hanoi: M(n) = 2 M(n-1) + 1
def hanoi(n, a="A", b="B", c="C"):
    return 0 if n == 0 else hanoi(n - 1, a, c, b) + 1 + hanoi(n - 1, c, b, a)
print("Hanoi moves: " + ", ".join(f"{n} discs {hanoi(n):,}" for n in (3, 10, 20)) + f" (2^n - 1: {2**20 - 1:,} for 20)")
```

OUTPUT

```

H(      10) =   2.92897;  H - ln n = 0.62638
H(    1,000) =   7.48547;  H - ln n = 0.57772
H(1,000,000) =  14.39273;  H - ln n = 0.57722
Euler's constant gamma = 0.57722 (the limit)
n =   10: n! / Stirling = 1.008365  (1 + 1/(12n) = 1.008333)
n =  100: n! / Stirling = 1.000834  (1 + 1/(12n) = 1.000833)
n = 1000: n! / Stirling = 1.000083  (1 + 1/(12n) = 1.000083)

merge sort recurrence: T(1,024) = 10,240 vs n log2 n = 10,240, T(1,000) = 9,976 vs n log2 n =
                      9,966, T(1,000,000) = 19,951,424 vs n log2 n = 19,931,569

recurrence          predicted  T(n) / predicted at n = 2^10, 2^15, 2^20
binary search        log n      1.000   1.000   1.000
merge sort           n log n     1.100   1.067   1.050
Karatsuba            n^1.585     2.965   2.995   2.999
4 halves, linear work      n^2     1.999   2.000   2.000
2 halves, quadratic work   n^2     1.999   2.000   2.000
Strassen             n^2.807     2.328   2.333   2.333

F(n) = round(phi^n / sqrt 5) holds exactly for n = 0..70; at n = 71 floating point gives
      308,061,521,170,130 instead of 308,061,521,170,129
Hanoi moves: 3 discs 7, 10 discs 1,023, 20 discs 1,048,575  (2^n - 1: 1,048,575 for 20)

```

In the master theorem table, every ratio settles to a constant as n grows. That's exactly what Θ promises, and the constants differ: about 3 for Karatsuba, 2.33 for Strassen, 1 in the limit for merge sort. The theorem says nothing about those constants, which is why Strassen's algorithm only beats ordinary matrix multiplication for matrices well into the hundreds or thousands of rows, depending on the implementation.

COMMON TRAPS

Reading O as a tight bound. Say Θ when you mean it.

Applying the master theorem outside its form, with unequal splits or non-polynomial extra work.

Forgetting the base case's cost when you need an exact count rather than a growth rate.

Trusting a closed form computed in floating point, as Binet's formula shows.

Comparing algorithms by exponent alone when the constants and your actual n decide which is faster.

Summing a geometric series in the wrong direction and deciding the smallest term dominates.

EXERCISES

EXERCISE 11.1 Three recurrences

Solve $T(n) = T(n/2) + n$, $T(n) = 4T(n/2) + n^2$ and $T(n) = 9T(n/3) + n$ asymptotically.

SOLUTION

First: $a = 1$, $b = 2$, $c = 1$, and $\log_2 1 = 0 < 1$, so the root dominates and $T(n) = \Theta(n)$. Directly, $n + n/2 + n/4 + \dots < 2n$. Second: $\log_2 4 = 2 = c$, the balanced case, so $T(n) = \Theta(n^2 \log n)$. Third: $\log_3 9 = 2 > 1$, the leaves dominate, so $T(n) = \Theta(n^2)$. The extra linear work is swamped by the 9 subproblems, whose number grows faster than their size shrinks.

EXERCISE 11.2 Merge sort, exactly

Prove by induction that $T(n) = 2T(n/2) + n$ with $T(1) = 0$ gives $T(n) = n \log_2 n$ for every power of 2.

SOLUTION

Write $n = 2^k$ and induct on k . Base case: $k = 0$ gives $T(1) = 0 = 1 \cdot \log_2 1$. Step: assume $T(2^k) = 2^k \cdot k$. Then

$$T(2^{k+1}) = 2T(2^k) + 2^{k+1} = 2 \cdot 2^k k + 2^{k+1} = 2^{k+1}(k+1),$$

which is $n \log_2 n$ for $n = 2^{k+1}$. ■ This is the substitution method: the recursion tree suggested the answer, and induction proved it. If you guess wrongly, the inductive step fails, and how it fails usually suggests a better guess.

EXERCISE 11.3 A characteristic equation

Solve $a_n = 5a_{n-1} - 6a_{n-2}$ with $a_0 = 0$ and $a_1 = 1$, and check your answer for $n = 2$ and 3.

SOLUTION

The characteristic equation is $x^2 = 5x - 6$, that is $(x - 2)(x - 3) = 0$, with roots 2 and 3. So $a_n = A \cdot 2^n + B \cdot 3^n$. From $a_0 = 0$: $A + B = 0$. From $a_1 = 1$: $2A + 3B = 1$. So $B = 1$, $A = -1$, and $a_n = 3^n - 2^n$. Check: the recurrence gives $a_2 = 5 \cdot 1 - 0 = 5 = 9 - 4$ and $a_3 = 5 \cdot 5 - 6 \cdot 1 = 19 = 27 - 8$. The larger root decides the growth rate, $\Theta(3^n)$, as it does for any linear recurrence with distinct roots.

CHECKLIST

- ☐ Can you derive the geometric sum and explain which term dominates?
- ☐ Do you know $H_n \approx \ln n + \gamma$ and Stirling's approximation?
- ☐ Can you define O , Ω and Θ with constants, and use the limit test?
- ☐ Can you draw a recursion tree and prove the guessed solution by induction?
- ☐ Can you apply the master theorem and explain its three cases with the tree?

GO DEEPER

COURSE 6.042J: sums, asymptotics and recurrences MIT OpenCourseWare

BOOK Introduction to Algorithms, chapter 4: divide and conquer Cormen,
Leiserson, Rivest & Stein

BOOK Algorithms, appendix on solving recurrences Jeff Erickson

PART IV

Probability

Four chapters on reasoning about chance precisely. First conditional probability and Bayes' rule, where intuition fails most reliably; then random variables and expectation, with the single most useful trick in the subject; then distributions and the tail bounds that say how unlikely the unlikely really is; and finally randomised algorithms, which use chance on purpose and come with guarantees anyway.

Conditional probability and Bayes' rule

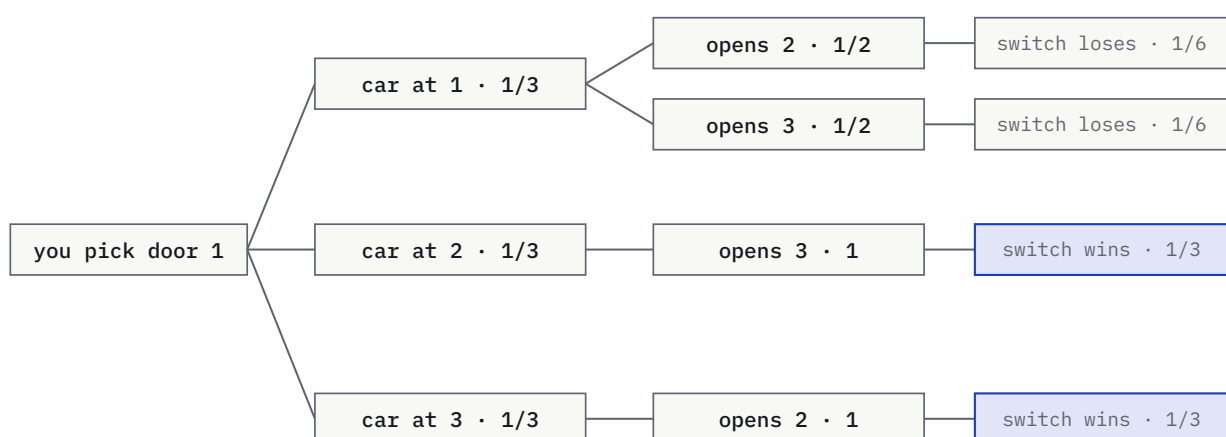
3 exercises

Probability is counting with weights, and conditional probability is how those weights change when you learn something. Intuition fails here in predictable ways. People stick with their first door in Monty Hall, read a positive result from an accurate test as near-certainty, and underestimate shared birthdays. A disciplined method fixes all three: lay out the outcomes in a tree, assign each its probability, and compute. Bayes' rule, which turns $P(\text{evidence} \mid \text{cause})$ into $P(\text{cause} \mid \text{evidence})$, is the formula behind spam filters, diagnostic tests and every alert that pages an engineer at night.

Sample spaces and the tree method

A **sample space** is the set of possible outcomes of an experiment, each with a probability, all summing to 1. An **event** is a set of outcomes, and its probability is the sum of theirs. When the experiment happens in stages, draw a **tree**: each branch is labelled with the probability of that step given everything before it, and an outcome's probability is the product along its path. The 6.042J text calls this the four-step method: find the outcomes, define the event, assign outcome probabilities, add them up. It sounds pedantic, and it settles arguments that intuition can't.

Take the **Monty Hall problem**. A car is behind one of three doors and goats are behind the others. You pick door 1. The host, who knows where the car is, opens a different door that has a goat, and offers you the chance to switch to the remaining door. Should you?



switching wins with probability $1/3 + 1/3 = 2/3$

The tree makes the host's knowledge visible. When your first pick was wrong, which happens two times in three, the host is forced to open the only other goat door, so switching lands on the car. Switching loses only when your first pick was right.

The tree settles it: switching wins with probability 2/3. The common intuition, "two doors left, so 50–50", ignores the fact that the host's choice depends on where the car is. The example confirms 2/3 exactly by walking the tree with fractions, and approximately by playing 100,000 games.

Conditional probability

The probability of A given that B has happened is

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}, \quad \Pr[B] > 0.$$

Knowing B shrinks the sample space to B, and the formula rescales so that B's outcomes sum to 1. Rearranged, it's the **product rule**, $\Pr[A \cap B] = \Pr[B] \Pr[A \mid B]$, which is what multiplying along a tree branch does. If events $B_1, \dots, B_k$ split the sample space, the **law of total probability** adds up the branches: $\Pr[A] = \sum_i \Pr[B_i] \Pr[A \mid B_i]$.

What you condition on must be exactly what you learned, including how you learned it.

With two children, each equally likely to be a boy or a girl,

$\Pr[\text{two boys} \mid \text{at least one boy}] = 1/3$, since three equally likely families remain: BB, BG, GB.

But $\Pr[\text{two boys} \mid \text{the older is a boy}] = 1/2$. The information sounds similar, and it leaves different sample spaces.

Bayes' rule

Writing $\Pr[A \cap B]$ in two ways gives **Bayes' rule**:

$$\Pr[A \mid B] = \frac{\Pr[B \mid A] \Pr[A]}{\Pr[B]}.$$

It converts what you can measure, how often the evidence appears when the cause is present, into what you want, how likely the cause is given the evidence. A diagnostic test detects a condition 95% of the time and gives a false positive 5% of the time. One person in a hundred has the condition. If you test positive, the chance that you have it is

$$\frac{0.95 \times 0.01}{0.95 \times 0.01 + 0.05 \times 0.99} = \frac{0.0095}{0.059} \approx 0.161.$$

Only about 16%. The test is accurate, but the condition is rare, so the healthy 99% produce far more false positives than the sick 1% produce true ones. Ignoring that is the **base rate fallacy**. A second independent positive test raises the probability to about 79%, because the first result's posterior becomes the second one's prior. Monitoring has the same structure. An alert that is 99% accurate, watching for a failure that occurs in 0.1% of minutes, will page you mostly for nothing, and the fix is better specificity or a combination of signals, not a lower threshold.

Independence

A and B are **independent** if $\Pr[A \cap B] = \Pr[A] \Pr[B]$, equivalently if learning B doesn't change the probability of A. Independence is a numerical fact to check, not a matter of whether events "seem related". Roll two dice. "The first die is even" is independent of "the sum is 7", since whatever the first die shows, exactly one value of the second makes 7. It is *not* independent of "the sum is 8". Given an even first die, the second die can always complete 8 (6, 4 or 2), so the sum is 8 with probability 1/6, against 5/36 overall. The example checks both by counting. Assuming independence when it doesn't hold is one of the costliest mistakes in reliability engineering. Disks from the same batch fail together, and replicas in the same rack lose power together, so "two failures at once" is far likelier than the product of the individual probabilities.

Birthdays

How many people must be in a room before a shared birthday is more likely than not? Counting the complement, with 365 equally likely days, n people all have different birthdays with probability

$$\frac{365}{365} \cdot \frac{364}{365} \cdots \frac{365 - n + 1}{365} \approx e^{-n(n-1)/(2 \cdot 365)},$$

and this drops below 1/2 at n = 23. The surprise comes from counting pairs, not people. 23 people form 253 pairs, and each pair has a 1-in-365 chance of matching. Chapter 14 applies the same approximation to random identifiers and hash values, where it decides how many bits you need.

Computed and simulated

The example computes each result exactly with fractions, then simulates it: Monty Hall, the test with its base rate across a million simulated people, the birthday problem, the two-child puzzle, and independence for two pairs of dice events.

EXAMPLE

conditional.py · Monty Hall, Bayes' rule, birthdays and independence, exactly and by simulation

```
import random
from fractions import Fraction as F
from itertools import product

random.seed(12)
# 1. Monty Hall, exactly: walk every branch of the tree with its probability
win_stay = win_switch = F(0)
for car in (1, 2, 3):
    goats = [d for d in (2, 3) if d != car]
    for opened in goats:
        p = F(1, 3) * F(1, len(goats))
        other = next(d for d in (1, 2, 3) if d not in (1, opened))
        win_stay += p * (car == 1); win_switch += p * (car == other)
```

```

print(f"Monty Hall exactly: stay wins {win_stay}, switch wins {win_switch}")
N = 100_000; stay = switch = 0
for _ in range(N):
    car, pick = random.randint(1, 3), random.randint(1, 3)
    opened = random.choice([d for d in (1, 2, 3) if d not in (car, pick)])
    other = next(d for d in (1, 2, 3) if d not in (pick, opened))
    stay += car == pick; switch += car == other
print(f"Monty Hall simulated ({N:,} games): stay {stay / N:.3f}, switch {switch / N:.3f}")

# 2. Bayes' rule and the base rate: a 95%-accurate test for a condition 1 in 100 people have
prior, sens, fpr = F(1, 100), F(95, 100), F(5, 100)
post = sens * prior / (sens * prior + fpr * (1 - prior))
post2 = sens * post / (sens * post + fpr * (1 - post))
print(f"\nP(condition | positive test) = {float(post):.3f}; after a second independent
    positive: {float(post2):.3f}")
people = [(random.random() < 0.01) for _ in range(1_000_000)]
pos = [sick for sick in people if random.random() < (0.95 if sick else 0.05)]
print(f"simulated million people: {len(pos):,} positives, of whom {sum(pos) / len(pos):.3f}
    have the condition")

# 3. The birthday problem
def p_shared(n):
    p_distinct = F(1)
    for i in range(n): p_distinct *= F(365 - i, 365)
    return 1 - p_distinct
print("\nP(some shared birthday): " + ", ".join(f"{n} people {float(p_shared(n)):.3f}" for n in
    (10, 23, 30, 50, 70)))
hits = sum(len(set(random.randrange(365) for _ in range(23))) < 23 for _ in range(N))
print(f"simulated rooms of 23 people: {hits / N:.3f} contain a shared birthday")

# 4. Conditioning on what you're told: two children
families = list(product("BG", repeat=2)) # (older, younger), equally likely
at_least_one = [f for f in families if "B" in f]
older_boy = [f for f in families if f[0] == "B"]
print(f"\nP(two boys | at least one boy) = {F(sum(f == ('B', 'B') for f in at_least_one),
    len(at_least_one))}; "
    f"P(two boys | older is a boy) = {F(sum(f == ('B', 'B') for f in older_boy),
    len(older_boy))}")

# 5. Independence is a numerical fact, not an intuition: two dice
dice = list(product(range(1, 7), repeat=2))
def P(event): return F(sum(1 for d in dice if event(d)), len(dice))
A = lambda d: d[0] % 2 == 0
for s in (7, 8):
    B = lambda d, s=s: sum(d) == s
    both = P(lambda d: A(d) and B(d))
    print(f"first die even, sum = {s}: P(both) = {both}, P(A)P(B) = {P(A) * P(B)} -> "
        f"'independent' if both == P(A) * P(B) else 'dependent'")

```

OUTPUT

```

Monty Hall exactly: stay wins 1/3, switch wins 2/3
Monty Hall simulated (100,000 games): stay 0.332, switch 0.668

P(condition | positive test) = 0.161; after a second independent positive: 0.785
simulated million people: 58,709 positives, of whom 0.160 have the condition

P(some shared birthday): 10 people 0.117, 23 people 0.507, 30 people 0.706, 50 people 0.970, 70
    people 0.999
simulated rooms of 23 people: 0.510 contain a shared birthday

P(two boys | at least one boy) = 1/3; P(two boys | older is a boy) = 1/2
first die even, sum = 7: P(both) = 1/12, P(A)P(B) = 1/12 -> independent
first die even, sum = 8: P(both) = 1/12, P(A)P(B) = 5/72 -> dependent

```

The simulations land close to the exact values, within the sampling error chapter 19 quantifies: 0.668 against $2/3$, 0.160 against 0.161, 0.510 against 0.507. That's the right role for simulation. It checks a calculation, or explores a problem too messy to calculate. It doesn't replace the calculation when one is available, because it can't tell you whether 0.668 means $2/3$ or 0.67.

COMMON TRAPS

Confusing $\Pr[A \mid B]$ with $\Pr[B \mid A]$, known in courtrooms as the prosecutor's fallacy.

Ignoring the base rate when judging an accurate test for a rare condition.

Assuming independence for events that share a cause.

Conditioning on what happened rather than on what you learned and how, as in the two-child puzzle.

The gambler's fallacy: independent trials have no memory, and a run of losses doesn't make a win "due".

Reading a small simulation as exact, without asking how much it could vary.

EXERCISES

EXERCISE 12.1 A one-word spam filter

60% of incoming mail is spam. The word "free" appears in 20% of spam and in 1% of legitimate mail. What's the probability that a message containing "free" is spam?

SOLUTION

By Bayes' rule,

$$\Pr[\text{spam} \mid \text{free}] = \frac{0.2 \times 0.6}{0.2 \times 0.6 + 0.01 \times 0.4} = \frac{0.12}{0.124} \approx 0.968.$$

Here the base rate helps: spam is common, so evidence for it is believed readily. A naive Bayes classifier combines many words this way, treating each word as independent evidence given the class and multiplying their likelihood ratios. The independence assumption is false, since words co-occur, but the classifier works well anyway, because it only needs to rank spam above legitimate mail, not to get the probabilities exactly right.

EXERCISE 12.2 Independence of complements

Prove that if A and B are independent, then so are A and the complement $\bar{B}$.

SOLUTION

A splits into the disjoint events $A \cap B$ and $A \cap \bar{B}$, so

$$\Pr[A \cap \bar{B}] = \Pr[A] - \Pr[A \cap B] = \Pr[A] - \Pr[A] \Pr[B] = \Pr[A](1 - \Pr[B]) = \Pr[A] \Pr[\bar{B}]$$

■ Applying the result twice shows that the complements are independent too. So if two components fail independently, they also *survive* independently, which is what reliability calculations multiply.

EXERCISE 12.3 A host who doesn't know

Change the game: the host doesn't know where the car is and opens one of the other two doors at random. It happens to show a goat. Should you switch now?

SOLUTION

Draw the tree. You pick door 1. The car is behind each door with probability $1/3$, and the host opens door 2 or 3 with probability $1/2$ each, regardless. The outcomes where he reveals a goat are: car at 1 and opens 2 or 3 ($1/6 + 1/6 = 1/3$); car at 2 and opens 3 ($1/6$); car at 3 and opens 2 ($1/6$). The total is $2/3$. Of that, the car is behind your door with probability $1/3$ and behind the other closed door with probability $1/3$. So

$$\Pr[\text{switch wins} \mid \text{goat shown}] = (1/3)/(2/3) = 1/2.$$

Now switching doesn't matter. The doors and the goat look identical in both games. The difference is the process that produced what you saw, and conditioning has to account for it.

CHECKLIST

- ☐ Can you solve a multi-stage problem with a tree, multiplying along branches?
- ☐ Can you state conditional probability, the product rule and the law of total probability?

- ☐ Can you apply Bayes' rule and explain the base rate fallacy with numbers?
- ☐ Do you check independence numerically instead of assuming it?
- ☐ Can you explain why switching wins in Monty Hall, and why it stops winning if the host doesn't know?

GO DEEPER

COURSE Statistics 110: Probability, lectures and the free book Joe Blitzstein, Harvard

NOTEBOOK A concrete introduction to probability, using Python Peter Norvig

INTERACTIVE Seeing Theory: a visual introduction to probability and statistics Brown University

Random variables and expectation

3 exercises

A random variable is a number that depends on chance: the cost of a randomised algorithm, the number of collisions in a hash table, the latency of a request. Its expectation is its long-run average. One fact does most of the work in this chapter, and it's the most useful tool in probability for programmers. Linearity of expectation says the expectation of a sum is the sum of expectations, always, even when the parts depend on each other. Combined with indicator variables, it turns hard problems into a line of algebra. Variance then measures how far reality strays from the average.

Random variables and expectation

A **random variable** X assigns a number to each outcome of an experiment. Its **expectation** is the probability-weighted average of its values:

$$\mathbb{E}[X] = \sum_x x \cdot \Pr[X = x].$$

A fair die has expectation $(1 + 2 + \dots + 6)/6 = 7/2$. The expectation needn't be a possible value: no roll shows 3.5. An **indicator variable** I_A is 1 if event A happens and 0 otherwise, so $\mathbb{E}[I_A] = \Pr[A]$. Indicators are how you turn questions about how many things happen into questions about how likely each one is.

Linearity of expectation

For any random variables X and Y , and constants a and b ,

$$\mathbb{E}[aX + bY] = a \mathbb{E}[X] + b \mathbb{E}[Y].$$

Proof. Write both sides as sums over outcomes ω : $\sum_{\omega} (aX(\omega) + bY(\omega)) \Pr[\omega]$ splits into $a \sum_{\omega} X(\omega) \Pr[\omega] + b \sum_{\omega} Y(\omega) \Pr[\omega]$. ■ The proof never mentions independence, and it isn't needed. That's what makes linearity so powerful: write a complicated count as a sum of indicators, which may depend on each other in tangled ways, and add up their probabilities.

Fixed points. n people get their hats back in a random order. How many get their own, on average? Let I_i indicate that person i does. Each person gets their own hat with probability $1/n$, so the expected number is $n \cdot 1/n = 1$, for every n . The indicators are dependent, since if $n - 1$ people have their own hats the last one must too, and it doesn't matter.

Inversions. How many swaps does insertion sort make on a random permutation? One per **inversion**, a pair of positions in the wrong order. Each of the $\binom{n}{2}$ pairs is inverted with probability $1/2$, so the expected number is $n(n-1)/4$. That makes insertion sort quadratic on average, not just in the worst case.

Running maxima. Scanning an array to find its maximum, how often does the "best so far" change? Position i holds a new maximum exactly when its value is the largest of the first i , which for a random order has probability $1/i$. The expected number of updates is $1 + \frac{1}{2} + \dots + \frac{1}{n} = H_n \approx \ln n$, only about 7.5 for a thousand values. That's also the expected number of times a "keep the best candidate" loop has to write.

Empty bins. Throw n balls into n bins at random. Each bin stays empty with probability $(1 - 1/n)^n \approx 1/e$, so about 37% of the bins are empty, and the same fraction of hash buckets is unused when a table has as many keys as buckets.

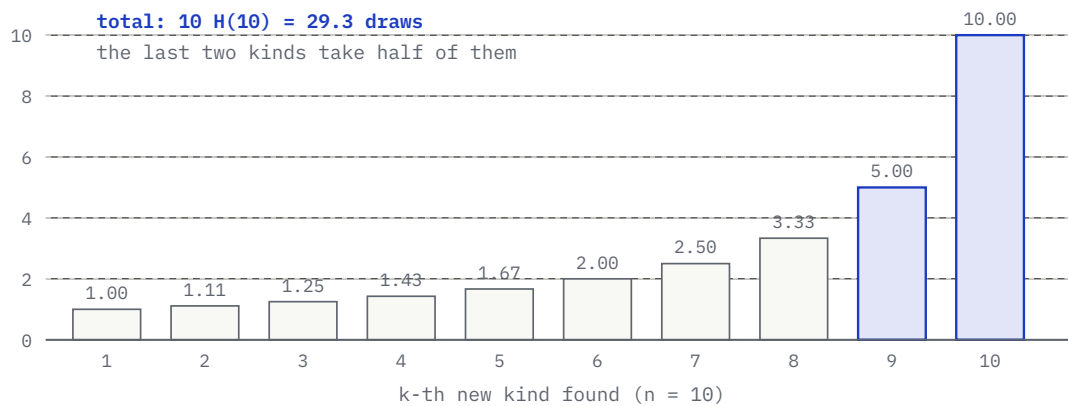
Waiting times and the coupon collector

If each trial succeeds independently with probability p , the number of trials up to and including the first success has a **geometric distribution** with expectation $1/p$. One trial always happens, and with probability $1 - p$ you're back where you started, so $\mathbb{E} = 1 + (1 - p)\mathbb{E}$, which solves to $1/p$.

The **coupon collector's problem** asks how many random draws it takes to collect all n kinds of coupon. Split the process into phases. Once you have i kinds, each draw is new with probability $(n - i)/n$, so that phase takes $n/(n - i)$ draws on average. By linearity the total expectation is

$$\sum_{i=0}^{n-1} \frac{n}{n-i} = n \left(1 + \frac{1}{2} + \dots + \frac{1}{n} \right) = nH_n \approx n \ln n.$$

expected draws



Collecting 10 kinds takes about 29 draws on average, and the last two take 15 of them. Each new kind gets harder to find because most draws repeat something you have. That's the $n \ln n$ shape, and it appears whenever random sampling has to cover everything: probes reaching every server, Gale-Shapley's proposals in chapter 8, or random tests hitting every branch.

Variance

The expectation says nothing about spread. The **variance** measures it as the expected squared distance from the mean, and the standard deviation σ is its square root:

$$\text{Var}[X] = \mathbb{E}[(X - \mu)^2] = \mathbb{E}[X^2] - \mu^2.$$

A die has $\mathbb{E}[X^2] = 91/6$, so $\text{Var}[X] = 91/6 - 49/4 = 35/12$. Unlike expectation, variance adds only for **independent** variables, or more generally uncorrelated ones:

$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y]$. So the sum of 10 dice has variance $350/12 \approx 29.2$ and standard deviation about 5.4, not 10 times the single die's 1.7. The spread of a sum grows like the square root of the number of terms, which is why averages are stable (chapter 19). For dependent variables, add twice their covariance.

Expectations hide the spread, and in engineering the spread is often what matters. The coupon collector's average for 50 kinds is 225 draws, but in the example one run needed 706. Request latency has the same character: a service with a fine mean latency can have a tail that ruins the experience for one request in a hundred, which is why production systems measure percentiles, not just averages (*The Engineer's Watch*).

Simulated

The example estimates each expectation above by repeating the experiment thousands of times and compares the average with the formula.

EXAMPLE

expectation.py · fixed points, coupons, running maxima, inversions, empty bins and variance

```
import random
from fractions import Fraction as F
from statistics import mean, pvariance

random.seed(13)
def H(n): return sum(1 / k for k in range(1, n + 1))

# 1. Fixed points of a random permutation: expected number 1, whatever n (linearity of expectation)
for n in (10, 100):
    counts = []
    for _ in range(20_000):
        p = list(range(n)); random.shuffle(p)
        counts.append(sum(p[i] == i for i in range(n)))
    print(f"fixed points of a random permutation of {n:3}: mean {mean(counts):.3f}, variance {pvariance(counts):.3f}")

# 2. Coupon collector: draws needed to see all 50 kinds
runs = []
for _ in range(10_000):
    seen, draws = set(), 0
    while len(seen) < 50: seen.add(random.randrange(50)); draws += 1
    runs.append(draws)
print(f"coupon collector, 50 kinds: mean {mean(runs):.1f} draws (n H_n = {50 * H(50):.1f}), "
      f"fewest {min(runs)}, most {max(runs)}")

# 3. How often does a running maximum change? Expected H_n times
updates = []
for _ in range(10_000):
    best, k = -1, 0
    for x in random.sample(range(10 ** 6), 1000):
        if x > best: best, k = x, k + 1
    updates.append(k)
print(f"updates to a running maximum over 1,000 random values: mean {mean(updates):.2f} (H_1000 = {H(1000):.2f})")

# 4. Inversions: the swaps insertion sort makes on a random permutation
n = 100
inv = []
for _ in range(2_000):
    p = random.sample(range(n), n)
    inv.append(sum(p[i] > p[j] for i in range(n) for j in range(i + 1, n)))
print(f"inversions in a random permutation of {n}: mean {mean(inv):.0f} (n(n-1)/4 = {n * (n - 1) // 4})")

# 5. Empty bins: n balls into n bins leave about n/e bins empty
n = 1000
empties = [n - len(set(random.randrange(n) for _ in range(n))) for _ in range(2_000)]
print(f"{n} balls into {n} bins: mean empty bins {mean(empties):.1f} (n(1 - 1/n)^n = {n * (1 - 1 / n) ** n:.1f})")

# 6. Variance: exact for one die, additive for independent dice
die = range(1, 7)
EX = F(sum(die), 6); VX = F(sum(x * x for x in die), 6) - EX ** 2
sums = [sum(random.randint(1, 6) for _ in range(10)) for _ in range(50_000)]
print(f"one die: E = {EX}, Var = {VX} = {float(VX):.3f}; sum of 10 dice: mean {mean(sums):.2f} (35), "
      f"variance {pvariance(sums):.2f} ({float(10 * VX):.2f})")
```

OUTPUT

```
fixed points of a random permutation of 10: mean 0.999, variance 1.012
fixed points of a random permutation of 100: mean 0.995, variance 0.999
coupon collector, 50 kinds: mean 224.1 draws (n H_n = 225.0), fewest 96, most 706
updates to a running maximum over 1,000 random values: mean 7.48 (H_1000 = 7.49)
inversions in a random permutation of 100: mean 2478 (n(n-1)/4 = 2475)
1000 balls into 1000 bins: mean empty bins 368.0 (n(1 - 1/n)^n = 367.7)
one die: E = 7/2, Var = 35/12 = 2.917; sum of 10 dice: mean 35.02 (35), variance 29.23 (29.17)
```

Every simulated mean lands within a fraction of a percent of its formula, and the fixed-point experiment adds something the formula didn't promise: the variance is about 1 as well. That's a hint of the Poisson distribution of chapter 14, which the number of fixed points approaches as n grows.

COMMON TRAPS

Thinking linearity needs independence. It never does; that's its power.

Multiplying expectations of dependent variables: $\mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y]$ needs independence.

Assuming $\mathbb{E}[1/X] = 1/\mathbb{E}[X]$, or that a function of the average is the average of the function.

Adding variances of correlated variables, such as replicas that fail together.

Reporting an average without a spread, when the tail is what users feel.

Confusing the mean with the median for skewed quantities such as latency or wealth.

EXERCISES

EXERCISE 13.1 Distinct faces

Roll a fair die six times. What's the expected number of distinct faces that appear?

SOLUTION

Let I_f indicate that face f appears at least once. It fails to appear in all six rolls with probability $(5/6)^6$, so $\mathbb{E}[I_f] = 1 - (5/6)^6 \approx 0.665$. By linearity the expected number of distinct faces is $6(1 - (5/6)^6) \approx 3.99$. Six rolls show only about four different faces on average. It's the empty-bins calculation again: the expected number of missing faces is $6(5/6)^6 \approx 2.01$.

EXERCISE 13.2 Waiting for every face

How many rolls do you expect to need to see a 6? To see every face at least once?

SOLUTION

A 6 appears with probability $1/6$ per roll, so the waiting time is geometric with expectation 6. Seeing every face is the coupon collector with $n = 6$: $6H_6 = 6(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}) = 14.7$. The phases take 1, 1.2, 1.5, 2, 3 and 6 rolls on average, so the last face alone accounts for about 40% of the wait.

EXERCISE 13.3 When products of expectations fail

Prove that $\mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y]$ when X and Y are independent. Then give two dependent variables for which it fails.

SOLUTION

Independence means $\Pr[X = x, Y = y] = \Pr[X = x] \Pr[Y = y]$, so

$$\mathbb{E}[XY] = \sum_{x,y} xy \Pr[X = x] \Pr[Y = y] = \left(\sum_x x \Pr[X = x] \right) \left(\sum_y y \Pr[Y = y] \right) = \mathbb{E}[X] \mathbb{E}[Y]$$

■ For a counterexample, let X be ± 1 with equal probability and $Y = X$. Then $\mathbb{E}[X] = \mathbb{E}[Y] = 0$, but $XY = 1$ always, so $\mathbb{E}[XY] = 1$. The difference $\mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$ is the covariance, which is exactly the term that appears when you add the variances of dependent variables.

CHECKLIST

- ☐ Can you compute an expectation from a distribution, and use indicator variables?
- ☐ Can you prove linearity of expectation, and explain why it needs no independence?
- ☐ Can you derive the expected number of fixed points, inversions and running-maximum updates?
- ☐ Can you derive the coupon collector's nH_n ?
- ☐ Can you compute a variance and say when variances add?

GO DEEPER

COURSE 6.042J: random variables, expectation and deviation MIT OpenCourseWare

COURSE Statistics 110: expectation and indicator random variables Joe Blitzstein,
Harvard

BOOK Probability and Computing, 2nd edition Mitzenmacher & Upfal

Distributions and tail bounds

3 exercises

A handful of distributions cover most random quantities in computing, and each comes from a recognisable mechanism. Recognise the mechanism and you know the distribution. Tail bounds answer the question engineers actually ask: how unlikely is something far from average? Markov's inequality needs only the mean, Chebyshev's adds the variance, and the Chernoff and Hoeffding bounds show that sums of independent pieces almost never stray far. These bounds give randomised algorithms their guarantees, explain why the "power of two choices" balances load so well, and tell you how many bits a random identifier needs.

A field guide to distributions

DISTRIBUTION	MECHANISM	MEAN	VARIANCE	IN COMPUTING
Bernoulli(p)	one trial that succeeds with probability p	p	$p(1 - p)$	a request fails or not
Binomial(n, p)	successes in n independent trials	np	$np(1 - p)$	failed requests out of n
Geometric(p)	trials until the first success	$1/p$	$(1 - p)/p^2$	retries, tries to find a prime
Poisson(λ)	rare events in many independent chances	λ	λ	arrivals per second, keys per bucket
Uniform	every value equally likely	midpoint	$(b - a)^2/12$	hash values, random IDs
Normal(μ, σ^2)	sum of many small independent effects	μ	σ^2	averages, measurement noise

The **Poisson** distribution is the binomial's limit when n is large, p is small and $\lambda = np$ stays fixed: $\Pr[X = k] = e^{-\lambda} \lambda^k / k!$. It models the number of keys in one bucket of a hash table, or requests arriving in one second from many independent users. The **geometric** distribution is **memoryless**: having failed 100 times doesn't change the chance of success on the next try. The **normal** distribution appears whenever many independent contributions add up, a fact chapter 19 makes precise with the central limit theorem. Be wary of using it in the far tails, where it can be wrong by orders of magnitude.

Markov and Chebyshev

Markov's inequality. For a random variable $X \geq 0$ and any $a > 0$,

$$\Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}.$$

Proof. $\mathbb{E}[X] \geq a \cdot \Pr[X \geq a]$, since X contributes at least a whenever $X \geq a$ and never contributes a negative amount. ■ It's weak, but it needs nothing except the mean and non-negativity. If an algorithm's expected running time is 1 second, at most 10% of runs take 10 seconds or more. Restarting runs that exceed twice the expected time therefore gives a guarantee, since each attempt succeeds with probability at least 1/2.

Chebyshev's inequality. For any X with mean μ and variance σ^2 ,

$$\Pr[|X - \mu| \geq t] \leq \frac{\sigma^2}{t^2}.$$

Apply Markov to the non-negative variable $(X - \mu)^2$ with $a = t^2$. It says a variable is rarely more than a few standard deviations from its mean: at most 1/9 of the probability lies beyond 3σ , whatever the distribution. Applied to an average of n independent samples, whose variance is σ^2/n , it proves the **law of large numbers**: averages converge to the mean.

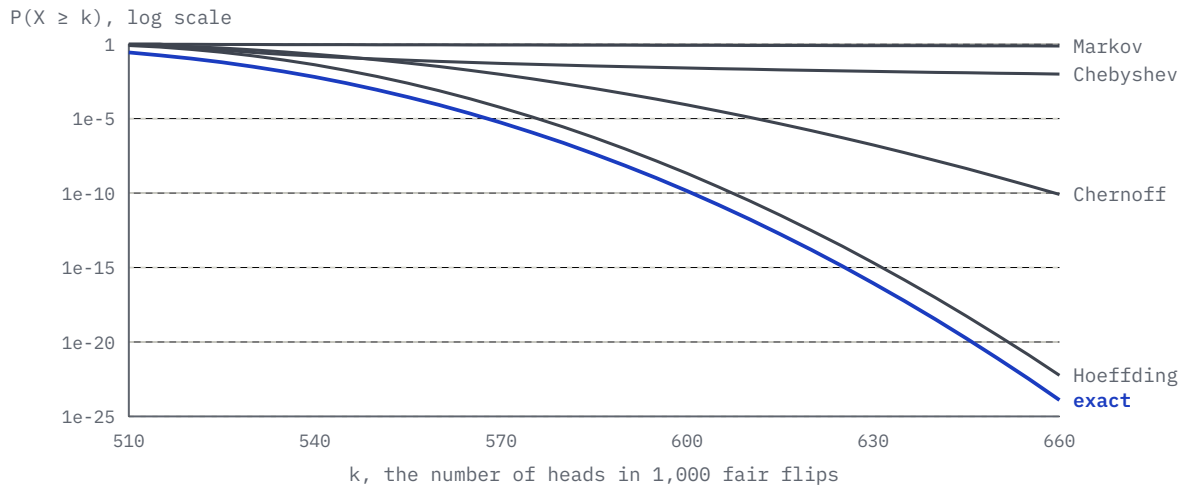
Chernoff and Hoeffding

When X is a **sum of independent** pieces, its tails are far thinner than Chebyshev admits. They're exponentially small. **Hoeffding's inequality**: if X is a sum of n independent variables each between 0 and 1, then

$$\Pr[X - \mathbb{E}[X] \geq t] \leq e^{-2t^2/n}.$$

The **Chernoff bound** measures the deviation relative to the mean μ :

$\Pr[X \geq (1 + \delta)\mu] \leq \left(e^\delta / (1 + \delta)^{1 + \delta}\right)^\mu$, which is at most $e^{-\delta^2\mu/3}$ for $0 < \delta \leq 1$. The proofs apply Markov's inequality to e^{sX} , whose expectation factors into a product because the pieces are independent. That's why independence is essential.



For 1,000 fair flips, the exact chance of at least k heads (highlighted) against four bounds, on a log scale. Markov barely moves. Chebyshev falls polynomially. Chernoff and Hoeffding fall exponentially, like the truth, and at 660 heads Hoeffding's bound is within a factor of about 50 of the exact value.

At 600 heads, 6.3 standard deviations above the mean, the exact tail is 1.36×10^{-10} . Markov says only that it's below 0.83, Chebyshev below 0.025, Chernoff below 8×10^{-5} and Hoeffding below 2×10^{-9} . This is the arithmetic behind "with high probability" in randomised algorithms. Errors that need many independent pieces to go wrong together are exponentially unlikely, so a modest number of repetitions makes failure rarer than a hardware fault.

Balls into bins and the union bound

Throw n balls into n bins at random, the model for hashing n keys or assigning n jobs to n servers. The average load is 1, but the fullest bin holds about $\ln n / \ln \ln n$ balls with high probability. The proof bounds the chance that one particular bin gets k balls, then uses the **union bound**:

$$\Pr \left[\bigcup_i A_i \right] \leq \sum_i \Pr[A_i].$$

If each of the n bins overflows with probability at most $1/n^2$, then some bin overflows with probability at most $1/n$. The union bound needs no independence at all, which makes it the most used inequality in algorithm analysis, and it becomes useless only when the probabilities it adds up exceed 1.

A small change in the process has a dramatic effect. Pick *two* random bins for each ball and put it in the less full one, and the maximum load drops to about $\ln \ln n / \ln 2 + O(1)$. That's an exponential improvement, from a single extra probe. This "**power of two choices**", analysed by Azar, Broder, Karlin and Upfal in 1994, is why load balancers such as NGINX and Envoy

can sample two backends at random and send the request to the less loaded one, instead of tracking every server's load. In the example, 100,000 balls give a fullest bin of 8 with one choice and 3 with two.

The birthday bound for identifiers

Chapter 12's birthday calculation sets the size of random identifiers. Among k values drawn uniformly from N possibilities, the chance of any collision is about

$$1 - e^{-k(k-1)/2N},$$

which reaches $1/2$ near $k \approx 1.18\sqrt{N}$. For b -bit identifiers, $N = 2^b$, so collisions become likely after about $2^{b/2}$ values, not 2^b . A billion random 64-bit IDs collide with probability about 2.7%, which is not safe for a primary key. Version 4 UUIDs carry 122 random bits, and a trillion of them collide with probability around 10^{-13} . The rule of thumb: a random identifier needs about twice as many bits as the logarithm of how many you'll create, plus a margin for the failure probability you can tolerate. The same bound says that a hash with b -bit outputs offers only about $b/2$ bits of collision resistance, which is why cryptographic hashes are 256 bits long.

Exact tails, bounds and simulations

The example computes the exact binomial tail with big integers alongside the four bounds, simulates balls into bins with one and two choices, applies the birthday bound to identifiers and checks it by drawing random 32-bit IDs until one repeats, and measures geometric waiting times.

EXAMPLE

tails.py · exact tails against Markov, Chebyshev, Chernoff and Hoeffding; balls in bins; the birthday bound

```
import math, random
from collections import Counter

random.seed(14)
# 1. How unlikely is 600 or more heads in 1,000 fair flips? The exact tail against four bounds
n, p = 1000, 0.5
mu, var = n * p, n * p * (1 - p)
def tail(k): return sum(math.comb(n, j) for j in range(k, n + 1)) / 2 ** n      # exact, with
    big integers
print(f"X = heads in {n} fair flips: mean {mu:.0f}, standard deviation {math.sqrt(var):.1f}")
print(f"P(X >= k):>10 { 'exact':>11 } { 'Markov':>9 } { 'Chebyshev':>10 } { 'Chernoff':>11 }
    { 'Hoeffding':>11 }")
for k in (550, 600, 650):
    t = k - mu; d = t / mu
    markov = mu / k
    cheb = var / t ** 2                                # bounds the two-sided tail, so the
    one-sided too
    chern = (math.exp(d) / (1 + d) ** (1 + d)) ** mu
    hoeff = math.exp(-2 * t * t / n)
    print(f"{k}>10 {tail(k):>11.2e} {markov:>9.3f} {cheb:>10.4f} {chern:>11.2e}")
```

```

        {hoeff:>11.2e}")

# 2. Balls into bins: the fullest bin, with one random choice and with the better of two
m = 100_000
one = Counter(random.randrange(m) for _ in range(m))
two = [0] * m
for _ in range(m):
    a, b = random.randrange(m), random.randrange(m)
    two[a if two[a] <= two[b] else b] += 1
print(f"\n{m:,} balls into {m:,} bins: fullest bin {max(one.values())} with one random choice "
      f"(ln n / ln ln n = {math.log(m) / math.log(math.log(m)):.1f}); {max(two)} with the less "
      f"full of two")

# 3. The birthday bound for random identifiers
def p_collision(k, bits): return -math.expml(-k * (k - 1) / 2 / 2 ** bits)
print(f"\nP(any collision) among 1 billion random 64-bit IDs: {p_collision(10 ** 9, 64):.2%}")
print(f"among 1 trillion random 122-bit IDs (UUID version 4): {p_collision(10 ** 12,
    122):.1e}")
firsts = []
for _ in range(300):
    seen, k = set(), 0
    while True:
        x = random.getrandbits(32); k += 1
        if x in seen: break
        seen.add(x)
    firsts.append(k)
print(f"random 32-bit IDs: first repeat after {sum(firsts) / len(firsts):.0f} on average over "
      f"300 runs "
      f"(sqrt(pi/2 x 2^32) = {math.sqrt(math.pi / 2 * 2 ** 32):.0f})")

# 4. Geometric waiting: odd candidates until a 1024-bit prime, p = 1/355
q = 1 / 355
waits = []
for _ in range(20_000):
    k = 1
    while random.random() >= q: k += 1
    waits.append(k)
print(f"\nwaiting for a 1-in-355 event: mean {sum(waits) / len(waits):.0f} tries; more than "
      f"1,000 tries in "
      f"{sum(w > 1000 for w in waits) / len(waits):.1%} of runs ((1 - p)^1000 = {(1 - q) ** "
      f"1000:.1%})")

```

OUTPUT

X = heads in 1000 fair flips: mean 500, standard deviation 15.8

P(X >= k)	exact	Markov	Chebyshev	Chernoff	Hoeffding
550	8.65e-04	0.909	0.1000	8.89e-02	6.74e-03
600	1.36e-10	0.833	0.0250	8.33e-05	2.06e-09
650	8.08e-22	0.769	0.0111	1.21e-09	2.86e-20

100,000 balls into 100,000 bins: fullest bin 8 with one random choice (ln n / ln ln n = 4.7); 3 with the less full of two

P(any collision) among 1 billion random 64-bit IDs: 2.67%

among 1 trillion random 122-bit IDs (UUID version 4): 9.4e-14

random 32-bit IDs: first repeat after 82,007 on average over 300 runs (sqrt(pi/2 x 2^32) = 82,137)

waiting for a 1-in-355 event: mean 357 tries; more than 1,000 tries in 6.0% of runs ((1 - p)^1000 = 6.0%)

The last line connects back to chapter 6: waiting for a 1-in-355 event takes 357 tries on average, matching the mean of 355, but more than 1,000 tries in 6% of runs. That's why the example there found one prime after 55 candidates and the other after 275. Geometric waits are memoryless and long-tailed, and a key generator that waits 1,000 candidates isn't broken.

COMMON TRAPS

Applying Markov's inequality to a variable that can be negative.

Using Chernoff or Hoeffding without independence, where correlated failures can make large deviations common.

Trusting a normal approximation in the far tails, where it can be off by orders of magnitude.

A union bound whose terms add up to more than 1, which proves nothing.

Sizing random IDs as if collisions needed 2^b values rather than about $2^{b/2}$.

Modelling bursty traffic as Poisson, which assumes independent arrivals.

EXERCISES

EXERCISE 14.1 Slow runs

A job takes 1 second on average. What fraction of runs can take 10 seconds or more? What if you also know the standard deviation is 0.5 seconds?

SOLUTION

Running time is non-negative, so Markov gives $\Pr[T \geq 10] \leq 1/10$: at most 10% of runs. With the standard deviation, Chebyshev gives

$$\Pr[T \geq 10] \leq \Pr[|T - 1| \geq 9] \leq 0.25/81 \approx 0.3\% .$$

Knowing the spread improves the bound by a factor of 30. Neither bound uses the shape of the distribution, so the true fraction may be far smaller. That's the price of a bound that holds for every distribution with that mean and variance.

EXERCISE 14.2 A table with no collisions

Hash n keys into n^2 buckets with a random hash function. Show that the probability of any collision is less than $1/2$. Why is this useful?

SOLUTION

Each of the $\binom{n}{2}$ pairs of keys collides with probability $1/n^2$. By the union bound, $\Pr[\text{any collision}] \leq \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n-1}{2n} < \frac{1}{2}$. So a random hash function into n^2 buckets is collision-free with probability over $1/2$, and trying fresh random functions until one works takes fewer than 2 tries on average (a geometric wait). This is the second level of **perfect hashing** (Fredman, Komlós and Szemerédi, 1984). Each bucket of an ordinary table gets its own small collision-free table sized to the square of its load, giving worst-case constant-time lookups in linear total space.

EXERCISE 14.3 How many bits?

You'll create 10^{10} random identifiers and want the probability of any collision below 10^{-6} . How many random bits does each need?

SOLUTION

For small probabilities, $1 - e^{-x} \approx x$, so we need $k^2/2N \leq 10^{-6}$ with $k = 10^{10}$: $N \geq 10^{20}/(2 \times 10^{-6}) = 5 \times 10^{25}$. Since $\log_2(5 \times 10^{25}) \approx 85.4$, we need 86 random bits. A 64-bit ID would collide with near certainty at this scale, while 122 bits leave an enormous margin. The general recipe: bits $\approx 2 \log_2 k + \log_2(1/p) - 1$.

CHECKLIST

- ☐ Can you match a random quantity to its distribution from the mechanism that produces it?
- ☐ Can you prove Markov's and Chebyshev's inequalities?
- ☐ Do you know when Chernoff or Hoeffding applies, and how much stronger it is?
- ☐ Can you use the union bound, and explain why the power of two choices works so well?
- ☐ Can you size a random identifier with the birthday bound?

GO DEEPER

BOOK Probability and Computing, chapters 3–5 Mitzenmacher & Upfal

THESIS The power of two choices in randomized load balancing Michael

↑
Mitzenmacher, 1996

REFERENCE RFC 9562: Universally Unique IDentifiers IETF, 2024

Randomised algorithms

3 exercises

Randomness in an algorithm is a tool, not a compromise. A random pivot defeats every bad input for quicksort at once. Contracting random edges finds a graph's minimum cut with almost no machinery. Multiplying by a random vector checks a matrix product faster than recomputing it, and a few random replacements sample a stream of unknown length uniformly. This chapter analyses each one with the tools of the last three chapters, and shows how repetition turns a small chance of success, or a small chance of error, into near-certainty.

Las Vegas and Monte Carlo

Randomised algorithms come in two kinds. A **Las Vegas** algorithm always gives the right answer, and only its running time is random: randomised quicksort and quickselect are examples. A **Monte Carlo** algorithm runs in bounded time but may be wrong with some small probability: the Miller–Rabin test of chapter 6, Karger's algorithm and Freivalds' check below. The two convert into each other. Stop a Las Vegas algorithm after twice its expected time and Markov's inequality bounds the chance it hasn't finished by $1/2$, which gives a Monte Carlo algorithm. A Monte Carlo algorithm whose answers can be checked becomes Las Vegas by rerunning it until the check passes.

Amplification is what makes Monte Carlo algorithms practical. If one run errs with probability at most p , and errors are one-sided, so that a "composite" from Miller–Rabin is always right, then k independent runs all err with probability at most p^k . With two-sided errors, take a majority vote of k runs, and the Chernoff bound of chapter 14 makes the error exponentially small in k .

Randomised quicksort

Quicksort with the first element as pivot takes $n(n-1)/2$ comparisons on sorted input, as *The Engineer's Workbench* showed. With a uniformly random pivot, the expected number of comparisons is about $2n \ln n$ on every input, so no input is bad, only some unlucky runs, and chapter 14's bounds show those are rare.

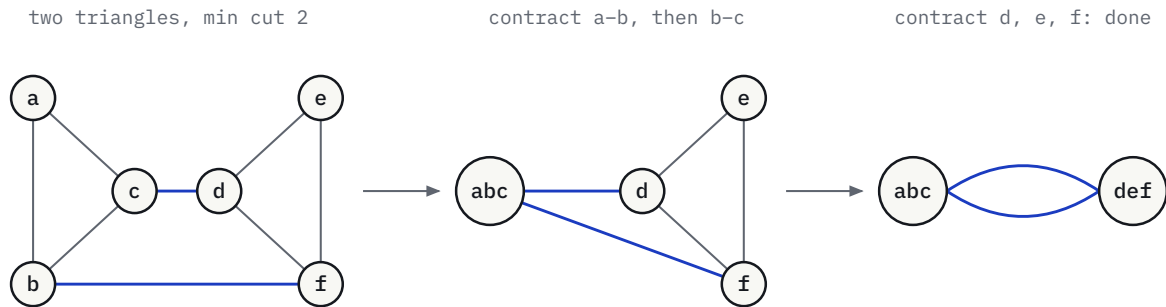
Proof. Name the elements by rank, $z_1 < z_2 < \dots < z_n$. Two elements z_i and z_j , with $i < j$, are compared at most once, and only if one of them is the first pivot chosen from the range $z_i, \dots, z_j$. If a pivot strictly between them comes first, they're split into different sides and never meet. Each of the $j-i+1$ elements in the range is equally likely to be chosen first, so they're compared with probability $2/(j-i+1)$. By linearity of expectation,

$$\mathbb{E}[\text{comparisons}] = \sum_{i < j} \frac{2}{j - i + 1} = 2(n + 1)H_n - 4n \approx 2n \ln n.$$

■ For $n = 2,000$ that's 24,730, and the example's 20 runs on sorted input ranged from 23,291 to 28,530, against 1,999,000 for the fixed pivot. **Quickselect**, which recurses into only the side containing the k -th element, has expected cost linear in n by a similar argument, about $3.39n$ for the median.

Karger's minimum cut

A **cut** of a graph splits its vertices into two non-empty groups, and its size is the number of edges between them. The minimum cut measures how close a network is to falling apart. David Karger's 1993 algorithm is almost absurdly simple: pick a random edge, merge its two endpoints into one vertex, keeping all other edges and discarding the edges that become loops, and repeat until two vertices remain. The edges between them form a cut.



the cut survives if none of its edges is ever chosen for contraction

Contraction never creates a cut that wasn't there, and a particular cut survives exactly as long as none of its edges is contracted. Because the minimum cut has few edges, a random contraction rarely hits one of them.

Analysis. Fix a minimum cut with k edges. Every vertex has degree at least k , since otherwise the edges at that vertex would form a smaller cut. So a graph with m vertices has at least $mk/2$ edges, and a random edge belongs to the cut with probability at most $k/(mk/2) = 2/m$. The cut survives all contractions, from n vertices down to 2, with probability at least

$$\prod_{m=3}^n \left(1 - \frac{2}{m}\right) = \frac{1}{3} \cdot \frac{2}{4} \cdot \frac{3}{5} \cdots \frac{n-2}{n} = \frac{2}{n(n-1)}.$$

That's small, but amplification fixes it. Repeat $\binom{n}{2} \ln n$ times, keep the smallest cut found, and the chance of missing the minimum is at most $(1 - 1/\binom{n}{2})^{\binom{n}{2} \ln n} \leq e^{-\ln n} = 1/n$. The bound is pessimistic. On the example's graph of two 8-vertex cliques joined by 2 edges, the

guarantee is 0.83% per run and the observed rate is 40%, because random edges almost always fall inside the dense cliques.

Freivalds' check

Someone hands you $n \times n$ matrices A , B and C and claims $C = AB$. Recomputing AB costs about n^3 operations, or $n^{2.81}$ with Strassen's algorithm. Rusins Freivalds observed in 1977 that you can check the claim with three matrix-vector products, in $O(n^2)$ time: choose a random vector r of 0s and 1s and test whether $A(Br) = Cr$. If $C = AB$, the test always passes. If not, let $D = AB - C$, which has some nonzero entry d_{ij} . Whatever the other entries of r are, at most one of the two values of r_j makes row i of Dr zero, so the test catches the error with probability at least $1/2$. Twenty independent trials miss it with probability below one in a million. The example plants one wrong entry among 3,600 and catches it in 51% of single trials.

Reservoir sampling

To choose k items uniformly at random from a stream whose length you don't know in advance, such as log lines or network packets, keep the first k . Then, when the i -th item arrives (counting from 1), keep it with probability k/i , replacing a uniformly random member of the sample. **Claim:** after i items, each one is in the sample with probability k/i . **Proof** by induction on i . It holds for $i \leq k$. When item $i + 1$ arrives, it enters with probability $k/(i + 1)$. An item already present, there with probability k/i by hypothesis, is evicted only if the newcomer enters and picks its slot, with probability $\frac{k}{i+1} \cdot \frac{1}{k} = \frac{1}{i+1}$. So it remains with probability $\frac{k}{i} \cdot \frac{i}{i+1} = \frac{k}{i+1}$. ■ Jeffrey Vitter analysed this method in 1985 as Algorithm R, and it needs only k slots of memory, however long the stream.

Measured

The example runs each algorithm many times: quicksort with a fixed and a random pivot on sorted input, Karger's algorithm 2,000 times, Freivalds' check 1,000 times, reservoir sampling 100,000 times, and quickselect for the median of 100,001 values.

EXAMPLE

```
randomised.py · random pivots, Karger's min cut, Freivalds' check, reservoir sampling and quickselect
```

```
import math, random

random.seed(15)
# 1. Randomised quicksort: expected  $2n \ln n$  comparisons on every input, including sorted ones
def quicksort_comparisons(a, pick):
    count, stack = 0, [(0, len(a))]
    a = list(a)
    while stack:
        lo, hi = stack.pop()
        if hi - lo <= 1: continue
        k = pick(lo, hi); a[lo], a[k] = a[k], a[lo]
        pivot, i = a[lo], lo + 1
```

```

        for j in range(lo + 1, hi):
            count += 1
            if a[j] < pivot: a[i], a[j] = a[j], a[i]; i += 1
        a[lo], a[i - 1] = a[i - 1], a[lo]
        stack += [(lo, i - 1), (i, hi)]
    return count
n = 2000
srt = list(range(n))
rand_counts = [quicksort_comparisons(srt, lambda lo, hi: random.randrange(lo, hi)) for _ in
                range(20)]
expected = 2 * (n + 1) * sum(1 / k for k in range(1, n + 1)) - 4 * n
print(f"quicksort on {n:,} sorted items: first-element pivot {quicksort_comparisons(srt, lambda
    lo, hi: lo):,} "
      f"comparisons; random pivot {min(rand_counts):,} to {max(rand_counts):,} (expected
        {expected:,.0f})")

# 2. Karger's min cut: contract random edges until two super-vertices remain
k = 8 # two 8-vertex cliques joined by 2 edges: the
      min cut is 2
edges = [(a, b) for a in range(k) for b in range(a + 1, k)]
edges += [(a + k, b + k) for a, b in edges] + [(0, k), (1, k + 1)]
V = 2 * k
def karger(edges):
    parent = list(range(V))
    def find(x):
        while parent[x] != x: parent[x] = parent[parent[x]]; x = parent[x]
    return x
    groups, pool = V, edges[:]
    random.shuffle(pool) # contracting edges in random order
    for a, b in pool:
        if groups == 2: break
        ra, rb = find(a), find(b)
        if ra != rb: parent[ra] = rb; groups -= 1
    return sum(find(a) != find(b) for a, b in edges)
trials = 2000
cuts = [karger(edges) for _ in range(trials)]
success = cuts.count(2) / trials
bound = 2 / (V * (V - 1))
print(f"\nKarger on a {V}-vertex graph with min cut 2: found it in {success:.1%} of {trials:,}
    runs "
      f"(guaranteed at least {bound:.2%})")
print(f"repetitions for 99% confidence: {math.ceil(math.log(100) / success)} at the observed
    rate, "
      f"{math.ceil(math.log(100) / bound):,} from the guarantee alone")

# 3. Freivalds: check a claimed matrix product in O(n^2) per trial instead of recomputing it
m = 60
A = [[random.randrange(10) for _ in range(m)] for _ in range(m)]
B = [[random.randrange(10) for _ in range(m)] for _ in range(m)]
C = [[sum(A[i][t] * B[t][j] for t in range(m)) for j in range(m)] for i in range(m)]
C[17][42] += 1 # one wrong entry out of 3,600
def mat_vec(M, v): return [sum(M[i][j] * v[j] for j in range(m)) for i in range(m)]
caught = 0
for _ in range(1000):
    r = [random.randrange(2) for _ in range(m)]
    caught += mat_vec(A, mat_vec(B, r)) != mat_vec(C, r)
print(f"\nFreivalds: a product with one wrong entry is caught in {caught / 10:.1f}% of single
    trials "
      f"(at least 50% guaranteed); 20 trials miss it with probability at most {0.5 ** 20:.1e}")

# 4. Reservoir sampling: a uniform sample of k items from a stream of unknown length
def reservoir(stream, k):
    sample = []
    for i, x in enumerate(stream):
        if i < k: sample.append(x)
    else:

```

```

        j = random.randrange(i + 1)
        if j < k: sample[j] = x
    return sample
hits = [0] * 10
for _ in range(100_000):
    for x in reservoir(range(10), 3): hits[x] += 1
print(f"\nreservoir of 3 from a stream of 10, 100,000 runs: each item kept in "
      + " ".join(f"{h / 100_000:.3f}" for h in hits) + " of runs (3/10 = 0.300)")

# 5. Quickselect (a Las Vegas algorithm): always correct, expected linear time
def quickselect(a, kth):
    count = 0
    while True:
        pivot = random.choice(a)
        lo = [x for x in a if x < pivot]; hi = [x for x in a if x > pivot]
        count += len(a)
        if kth < len(lo): a = lo
        elif kth < len(a) - len(hi): return pivot, count
        else: kth -= len(a) - len(hi); a = hi
N = 100_001
data = random.sample(range(10 ** 7), N)
results = [quickselect(data, N // 2) for _ in range(20)]
print(f"\nquickselect for the median of {N:,} values: correct every time: "
      + f"{all(r[0] == sorted(data)[N // 2] for r in results)}; work {min(r[1] for r in results) "
      + f"/ N:.2f}n to "
      + f"{max(r[1] for r in results) / N:.2f}n (expected about {2 + 2 * math.log(2):.2f}n)")

```

OUTPUT

```

quicksort on 2,000 sorted items: first-element pivot 1,999,000 comparisons; random pivot 23,291
to 28,530 (expected 24,730)

Karger on a 16-vertex graph with min cut 2: found it in 40.2% of 2,000 runs (guaranteed at
least 0.83%)
repetitions for 99% confidence: 12 at the observed rate, 553 from the guarantee alone

Freivalds: a product with one wrong entry is caught in 51.3% of single trials (at least 50%
guaranteed); 20 trials miss it with probability at most 9.5e-07

reservoir of 3 from a stream of 10, 100,000 runs: each item kept in 0.300 0.298 0.302 0.299
0.300 0.301 0.300 0.301 0.299 0.300 of runs (3/10 = 0.300)

quickselect for the median of 100,001 values: correct every time: True; work 1.79n to 5.08n
(expected about 3.39n)

```

Quickselect was correct all 20 times, as a Las Vegas algorithm must be, while its cost varied from 1.79n to 5.08n around the expected 3.39n. The reservoir frequencies all sit within a few thousandths of 3/10, which is the uniformity the induction proved.

COMMON TRAPS

Predictable randomness when an adversary chooses the input. If attackers can predict your pivots or hash seeds, the worst case comes back.

Running a Monte Carlo algorithm once and trusting it without amplification.

Confusing expected time with worst-case time in a deadline or SLA.

Biased sampling: `rand() % n` is not uniform unless n divides the generator's range, and shuffling by sorting with a random comparator isn't uniform either.

Assuming the input is random instead of randomising the algorithm, which is what makes the guarantee hold for every input.

Repeating runs that aren't independent, for example by reseeding with the same value, which makes amplification useless.

EXERCISES

EXERCISE 15.1 How many minimum cuts?

Use the analysis of Karger's algorithm to prove that a graph with n vertices has at most $\binom{n}{2}$ minimum cuts. Show that the cycle on n vertices has exactly that many.

SOLUTION

For each particular minimum cut, one run of the algorithm outputs exactly that cut with probability at least $2/(n(n-1)) = 1/\binom{n}{2}$. A run outputs only one cut, so these events are disjoint for different cuts, and their probabilities sum to at most 1. So there are at most $\binom{n}{2}$ minimum cuts. ■ In a cycle every cut has at least 2 edges, and removing any 2 of the n edges splits the cycle into two paths, so every pair of edges is a minimum cut: $\binom{n}{2}$ of them. On a cycle, Karger's algorithm always succeeds, because it always ends with some minimum cut, which shows that the bound for one particular cut can be far below the chance of finding a minimum cut.

EXERCISE 15.2 Amplifying a primality test

A Miller–Rabin round declares a composite number "probably prime" with probability at most $1/4$. How many rounds make the error at most 2^{-128} ? Why can't you do the same with a test whose errors go both ways?

SOLUTION

The errors are one-sided: a "composite" verdict comes with a witness and is always right, so the test is fooled only if every round is fooled, with probability at most $(1/4)^k = 2^{-2k}$. For 2^{-128} , $k = 64$ rounds. With two-sided errors, k runs can't simply be "anded" together, because a single wrong "yes" and a single wrong "no" are both possible. Instead take the majority of k runs. If each run is right with probability at least

$2/3$, the Chernoff bound shows the majority is wrong with probability e^{-ck} for some constant c . That's still exponential in k , with a worse constant.

EXERCISE 15.3 A shuffle that isn't uniform

This shuffle swaps each position with a random position anywhere in the list: `for i in range(n): j = randrange(n); a[i], a[j] = a[j], a[i]`. Show that it can't produce all $n!$ orders with equal probability, and give the correct version.

SOLUTION

The loop makes n independent choices of n values each, so there are n^n equally likely execution paths. If every one of the $n!$ orders were equally likely, each would arise from $n^n/n!$ paths, which must be a whole number. For $n = 3$ that's $27/6$, not an integer, so some orders are more likely than others. The correct Fisher-Yates shuffle draws j from the not-yet-fixed part only: `for i in range(n - 1, 0, -1): j = randrange(i + 1)`, then swap. That gives $n \times (n - 1) \times \dots \times 1 = n!$ paths, one per order, and it's what Python's `random.shuffle` does. Counting execution paths is a quick way to show that a randomised procedure can't be uniform.

CHECKLIST

- ☐ Can you tell Las Vegas from Monte Carlo algorithms, and convert one into the other?
- ☐ Can you derive the expected comparisons of randomised quicksort with indicator variables?
- ☐ Can you prove Karger's success probability and say how many repetitions it needs?
- ☐ Can you explain why Freivalds' check catches a wrong product with probability at least $1/2$?
- ☐ Can you prove reservoir sampling uniform by induction?

GO DEEPER

BOOK Probability and Computing, chapters 1–2 Mitzenmacher & Upfal

PAPER Random sampling with a reservoir Jeffrey Vitter, 1985

PART V

Linear algebra & calculus

Three chapters of continuous mathematics, the part of the stage that machine learning, graphics and scientific computing are built on. First vectors and matrices as transformations, and how to solve systems and fit models with them; then eigenvectors, which explain long-run behaviour from Markov chains to PageRank; and finally derivatives, gradients and the optimisation methods that follow them downhill.

Vectors, matrices and solving systems

3 exercises

Linear algebra is the mathematics of doing the same thing to many numbers at once. Vectors represent data: embeddings, pixels, features, word counts. Matrices represent linear transformations, and multiplying matrices composes transformations. Gaussian elimination solves systems of equations and reveals their rank, least squares fits a model to noisy data, and conditioning tells you when floating point will quietly ruin the answer. Watch 3Blue1Brown's Essence of Linear Algebra first for the geometry; this chapter adds the computation.

Vectors and the dot product

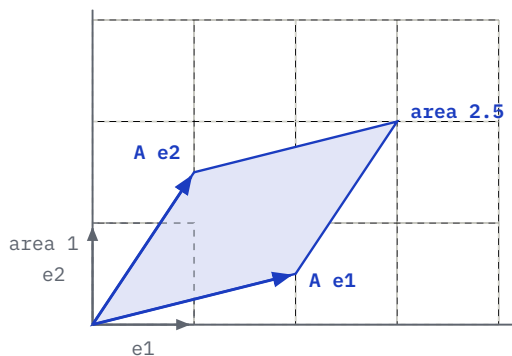
A vector in $\mathbb{R}^n$ is a list of n numbers, added and scaled entry by entry. The **dot product** multiplies matching entries and adds them up, and it has a geometric meaning:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^n u_i v_i = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta,$$

where $\|\mathbf{u}\| = \sqrt{\mathbf{u} \cdot \mathbf{u}}$ is the length and θ the angle between the vectors. So the **cosine similarity** $\mathbf{u} \cdot \mathbf{v} / (\|\mathbf{u}\| \|\mathbf{v}\|)$ measures how closely two vectors point the same way, ignoring their lengths. It's how search engines compare documents and how retrieval systems compare embeddings (*The Engineer's Foundry*). The example compares word-count vectors of three sentences. "The cat sat on the mat" and "the cat lay on the rug" score 0.75. A sentence about stocks still scores 0.47 against each, purely because all three share "the" and "on". That's why real systems discount common words, for example with tf-idf weighting, or use learned embeddings.

Matrices are transformations

A matrix A with n columns turns a vector $\mathbf{x}$ into $A\mathbf{x}$, and this transformation is **linear**: $A(a\mathbf{x} + b\mathbf{y}) = aA\mathbf{x} + bA\mathbf{y}$. Linearity means a matrix is completely determined by what it does to the basis vectors $\mathbf{e}_1, \dots, \mathbf{e}_n$, and those images are exactly its columns. $A\mathbf{x}$ is the combination of A 's columns with weights $x_1, \dots, x_n$.



columns of A are the images:

$$A e_1 = (2, 0.5)$$

$$A e_2 = (1, 1.5)$$

$$\det A = 2 \times 1.5 - 1 \times 0.5 = 2.5$$

every area is multiplied by 2.5

The unit square, dashed, is carried to the shaded parallelogram spanned by A 's columns. Every grid square goes to a copy of that parallelogram, so every area in the plane is multiplied by the same factor, the determinant.

Applying B and then A is the single transformation AB , and that's the definition of matrix multiplication. Note the order: the matrix written on the right acts first. Because order matters for transformations, it matters for matrices. Rotating by 90° then stretching horizontally is not the same as stretching then rotating, and the example shows $SR \neq RS$. The **determinant** is the factor by which a transformation scales areas, or volumes in higher dimensions, negative if it flips orientation. It multiplies, $\det(AB) = \det A \det B$, and it's zero exactly when the transformation squashes space into a lower dimension, losing information that can't be recovered.

Solving systems: Gaussian elimination and rank

A system of linear equations is a matrix equation $Ax = b$. **Gaussian elimination** solves it using row operations that don't change the solutions: swap two rows, scale a row by a nonzero number, add a multiple of one row to another. Each column in turn gets a **pivot**, and the entries below it are cleared. The number of pivots is the **rank** of A , the number of genuinely independent equations, which is also the dimension of the space of A 's outputs. There are exactly three possible outcomes.

- Rank equals the number of unknowns: a **unique** solution.
- A row reduces to " $0 = \text{nonzero}$ ": **no** solution, because the equations contradict each other.
- Otherwise: **infinitely many** solutions, forming a space of dimension $n - \text{rank}$.

The example produces all three with exact fractions. In the last two systems the third row is the sum of the first two, so the rank is 2, and whether the right-hand side respects the same relationship decides between infinitely many solutions and none. A square matrix has an inverse exactly when its rank is full, equivalently when its determinant is nonzero.

Elimination costs about $n^3/3$ multiplications. To solve a system, eliminate. Don't compute A^{-1} and multiply, which costs more and loses more accuracy.

Least squares

Data rarely satisfies a model exactly. Fitting a line $y = a + bx$ to 50 noisy points gives 50 equations in 2 unknowns, with no exact solution. **Least squares** chooses the parameters that minimise the sum of squared errors $\|A\beta - \mathbf{y}\|^2$. Geometrically, it projects $\mathbf{y}$ onto the space spanned by A 's columns, and the residual is then perpendicular to every column, which gives the **normal equations**:

$$A^T A \beta = A^T \mathbf{y}.$$

From points generated by $y = 2 + 0.5x$ plus noise, the example recovers $y = 1.998 + 0.517x$. Every linear regression is this computation. Libraries usually solve it with a QR decomposition instead of forming $A^T A$, because forming it squares the condition number discussed next.

Conditioning: when floating point betrays you

Some systems are inherently sensitive: a tiny change in the data causes a large change in the solution. The **condition number** $\kappa(A)$ measures this. In floating point, with relative precision about 10^{-16} , you can lose up to $\log_{10} \kappa$ digits of the answer, whatever algorithm you use. The classic example is the **Hilbert matrix** $H_{ij} = 1/(i + j - 1)$, whose rows are nearly parallel. The example solves $H\mathbf{x} = H\mathbf{1}$, whose exact answer is all ones, with careful floating-point elimination. The worst error is 4.6×10^{-13} for $n = 4$, 1.4×10^{-7} for $n = 8$, and 6×10^{-3} for $n = 12$, where κ is about 10^{16} and nearly every digit is gone. Exact fractions get every entry exactly 1, at a cost that grows quickly with n . The practical lessons are to use partial pivoting, which the example does, prefer well-conditioned formulations, and check the error, not just the residual: a solution can satisfy the equations to many digits and still be wrong.

All of it, from scratch

The example builds everything from lists and the standard library: cosine similarity, composition and determinants, exact Gaussian elimination, a least-squares fit, the Hilbert matrix in floating point and in fractions, and the cost of schoolbook multiplication.

EXAMPLE

linear.py · vectors, transformations, elimination, least squares and conditioning, in pure Python

```
import math, random
from fractions import Fraction as F

def dot(u, v): return sum(a * b for a, b in zip(u, v))
def norm(u): return math.sqrt(dot(u, u))
def matmul(A, B): return [[sum(A[i][k] * B[k][j] for k in range(len(B))) for j in
    range(len(B[0]))] for i in range(len(A))]
def matvec(A, v): return [dot(row, v) for row in A]
```

1. Vectors: cosine similarity of word-count vectors

```

docs = ["the cat sat on the mat", "the cat lay on the rug", "stocks fell on the news"]
vocab = sorted(set(" ".join(docs).split()))
vecs = [[d.split().count(w) for w in vocab] for d in docs]
print("cosine similarity of word counts:")
for i in range(3):
    for j in range(i + 1, 3):
        print(f"doc {i + 1} vs doc {j + 1}: {dot(vecs[i], vecs[j]) / (norm(vecs[i]) *
            norm(vecs[j])):.3f}")

# 2. Matrices are transformations, and composing them is multiplying them (in order)
R = [[0, -1], [1, 0]] # rotate 90 degrees
S = [[2, 0], [0, 1]] # stretch x by 2
def det2(M): return M[0][0] * M[1][1] - M[0][1] * M[1][0]
print(f"\nrotate then stretch, S R = {matmul(S, R)}; stretch then rotate, R S = {matmul(R,
    S)}")
print(f"det R = {det2(R)}, det S = {det2(S)}, det(S R) = {det2(matmul(S, R))}: areas scale by
    the product")

# 3. Gaussian elimination with exact fractions: rank and the three possible outcomes
def solve(A, b):
    M = [[F(x) for x in row] + [F(y)] for row, y in zip(A, b)]
    n, m = len(M), len(M[0]) - 1
    r = 0
    for c in range(m):
        p = next((i for i in range(r, n) if M[i][c] != 0), None)
        if p is None: continue
        M[r], M[p] = M[p], M[r]
        M[r] = [x / M[r][c] for x in M[r]]
        for i in range(n):
            if i != r and M[i][c] != 0: M[i] = [a - M[i][c] * b2 for a, b2 in zip(M[i], M[r])]
        r += 1
    inconsistent = any(all(x == 0 for x in row[:-1]) and row[-1] != 0 for row in M)
    if inconsistent: return r, "no solution"
    if r < m: return r, "infinitely many solutions"
    return r, "unique solution x = (" + ", ".join(str(M[i][-1]) for i in range(m)) + ")"
systems = {
    "independent rows": ([[2, 1, -1], [-3, -1, 2], [-2, 1, 2]], [8, -11, -3]),
    "row 3 = row 1 + row 2, consistent": ([[1, 2, 3], [4, 5, 6], [5, 7, 9]], [6, 15, 21]),
    "row 3 = row 1 + row 2, inconsistent": ([[1, 2, 3], [4, 5, 6], [5, 7, 9]], [6, 15, 20]),
}
for name, (A, b) in systems.items():
    rank, outcome = solve(A, b)
    print(f"{name:37} rank {rank}: {outcome}")

# 4. Least squares: fit a line to noisy data through the normal equations ( $A^T A$ )  $\beta = A^T y$ 
random.seed(16)
xs = [i / 5 for i in range(50)]
ys = [2 + 0.5 * x + random.gauss(0, 1) for x in xs]
A = [[1, x] for x in xs]
At = [list(col) for col in zip(*A)]
AtA, Aty = matmul(At, A), matvec(At, ys)
d = det2(AtA)
a = (AtA[1][1] * Aty[0] - AtA[0][1] * Aty[1]) / d
b = (AtA[0][0] * Aty[1] - AtA[1][0] * Aty[0]) / d
print(f"\nleast squares on 50 noisy points from y = 2 + 0.5x: fitted y = {a:.3f} + {b:.3f}x")

# 5. Conditioning: the Hilbert matrix, solved in floating point and exactly
def gauss_float(A, b):
    n = len(A); M = [row[:] + [y] for row, y in zip(A, b)]
    for c in range(n):
        p = max(range(c, n), key=lambda i: abs(M[i][c])); M[c], M[p] = M[p], M[c] # partial
        pivoting
        for i in range(c + 1, n):
            f = M[i][c] / M[c][c]
            M[i] = [x - f * y for x, y in zip(M[i], M[c])]
    x = [0.0] * n

```

```

    for i in range(n - 1, -1, -1):
        x[i] = (M[i][n] - sum(M[i][j] * x[j] for j in range(i + 1, n))) / M[i][i]
    return x
print("\nHilbert matrix H[i][j] = 1/(i + j + 1), solving H x = H 1 (true answer: all ones):")
for n in (4, 8, 12):
    H = [[1 / (i + j + 1) for j in range(n)] for i in range(n)]
    b = [sum(row) for row in H]
    xf = gauss_float(H, b)
    HF = [[F(1, i + j + 1) for j in range(n)] for i in range(n)]
    xe = solve(HF, [sum(row) for row in HF])[1]
    exact_ok = xe == "unique solution x = (" + ", ".join(["1"] * n) + ")"
    print(f"    n = {n:2}: worst floating-point error {max(abs(v - 1) for v in xf):.1e};    "
          f"with exact fractions every entry is exactly 1: {exact_ok}")

# 6. The cost of multiplying n x n matrices the schoolbook way
for n in (10, 100, 1000):
    print(f"n = {n:4}: {n ** 3}>13,} multiplications" + (f"    (n^2.807, Strassen's growth: {n
    ** math.log2(7):, .0f})" if n == 1000 else ""))

```

OUTPUT

```

cosine similarity of word counts:
  doc 1 vs doc 2: 0.750
  doc 1 vs doc 3: 0.474
  doc 2 vs doc 3: 0.474

rotate then stretch, S R = [[0, -2], [1, 0]]; stretch then rotate, R S = [[0, -1], [2, 0]]
det R = 1, det S = 2, det(S R) = 2: areas scale by the product
independent rows                rank 3: unique solution x = (2, 3, -1)
row 3 = row 1 + row 2, consistent  rank 2: infinitely many solutions
row 3 = row 1 + row 2, inconsistent  rank 2: no solution

least squares on 50 noisy points from y = 2 + 0.5x: fitted y = 1.998 + 0.517x

Hilbert matrix H[i][j] = 1/(i + j + 1), solving H x = H 1 (true answer: all ones):
  n = 4: worst floating-point error 4.6e-13; with exact fractions every entry is exactly 1:
    True
  n = 8: worst floating-point error 1.4e-07; with exact fractions every entry is exactly 1:
    True
  n = 12: worst floating-point error 6.0e-03; with exact fractions every entry is exactly 1:
    True
n = 10:          1,000 multiplications
n = 100:         1,000,000 multiplications
n = 1000: 1,000,000,000 multiplications    (n^2.807, Strassen's growth: 264,280,285)

```

Multiplying two $n \times n$ matrices the schoolbook way takes n^3 multiplications, a billion for $n = 1000$. Strassen's algorithm (chapter 11) reduces the exponent to 2.807, and optimised libraries such as BLAS, and GPUs, make dense linear algebra fast by arranging those multiplications to suit caches and parallel hardware. That's why numerical Python code hands its matrices to NumPy rather than looping over lists as this example does.

COMMON TRAPS

Assuming $AB = BA$. Matrix multiplication composes transformations, and order matters.

Computing an inverse to solve a system, instead of eliminating or factorising.

Deciding rank with floating-point equality. Use a tolerance, or exact arithmetic for small problems.

Ignoring conditioning and trusting a small residual as proof of an accurate answer.

Solving least squares by the normal equations on ill-conditioned data, which squares the condition number.

Mixing up rows and columns, or vectors of the wrong shape, the most common bug in matrix code.

EXERCISES

EXERCISE 16.1 Rotations compose

Write the matrix R_θ that rotates the plane by angle θ . Multiply $R_\alpha R_\beta$, and explain what the result proves about sines and cosines.

SOLUTION

The columns are the images of the basis vectors: $\mathbf{e}_1$ goes to $(\cos \theta, \sin \theta)$ and $\mathbf{e}_2$ to $(-\sin \theta, \cos \theta)$. So

$$R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}.$$

Rotating by β then α is rotating by $\alpha + \beta$, so $R_\alpha R_\beta = R_{\alpha + \beta}$. Multiplying out the left side, the top-left entry is $\cos \alpha \cos \beta - \sin \alpha \sin \beta$ and the bottom-left is $\sin \alpha \cos \beta + \cos \alpha \sin \beta$. Comparing with the right side proves the addition formulas $\cos(\alpha + \beta) = \cos \alpha \cos \beta - \sin \alpha \sin \beta$ and $\sin(\alpha + \beta) = \sin \alpha \cos \beta + \cos \alpha \sin \beta$, with no geometry beyond "rotations compose". Rotations have determinant $\cos^2 \theta + \sin^2 \theta = 1$: they preserve area.

EXERCISE 16.2 Two solutions means infinitely many

Prove that if $A\mathbf{x} = \mathbf{b}$ has two different solutions, it has infinitely many.

SOLUTION

Let $\mathbf{x}$ and $\mathbf{x}'$ be different solutions, and let $\mathbf{d} = \mathbf{x}' - \mathbf{x} \neq \mathbf{0}$. Then $A\mathbf{d} = \mathbf{b} - \mathbf{b} = \mathbf{0}$. For every real t , by linearity $A(\mathbf{x} + t\mathbf{d}) = A\mathbf{x} + tA\mathbf{d} = \mathbf{b}$, and different values of t give different vectors because $\mathbf{d} \neq \mathbf{0}$. ■ This is

why a linear system has 0, 1 or infinitely many solutions and never exactly 2. The set of all solutions is one particular solution plus every vector that A sends to zero, the **null space**.

EXERCISE 16.3 A least-squares fit by hand

Fit $y = a + bx$ to the points (0, 1), (1, 3) and (2, 4) by solving the normal equations.

SOLUTION

The rows of A are (1, 0), (1, 1) and (1, 2), and $\mathbf{y} = (1, 3, 4)$. Then

$$A^T A = \begin{pmatrix} 3 & 3 \\ 3 & 5 \end{pmatrix}, \quad A^T \mathbf{y} = \begin{pmatrix} 8 \\ 11 \end{pmatrix}.$$

The determinant is $15 - 9 = 6$, so $a = (5 \times 8 - 3 \times 11)/6 = 7/6$ and $b = (3 \times 11 - 3 \times 8)/6 = 3/2$. The line is $y = 7/6 + 1.5x$. Its residuals at $x = 0, 1, 2$ are $-1/6, +1/3$ and $-1/6$. They sum to zero, as they must, since the residual is perpendicular to the column of ones, and that's a useful check on any regression you compute by hand.

CHECKLIST

- ☐ Can you compute a dot product and a cosine similarity, and say what they measure?
- ☐ Can you write a matrix from what it does to the basis vectors, and compose transformations?
- ☐ Can you explain the determinant as an area factor?
- ☐ Can you run Gaussian elimination and read off the rank and the kind of solution?
- ☐ Can you set up least squares, and explain what conditioning does to floating-point answers?

GO DEEPER

VIDEO [Essence of Linear Algebra 3Blue1Brown](#)

COURSE [18.06 Linear Algebra Gilbert Strang, MIT OpenCourseWare](#)

BOOK [Mathematics for Machine Learning, part I Deisenroth, Faisal & Ong](#)

Eigenvectors: what a matrix does in the long run

3 exercises

Most vectors change direction when a matrix acts on them. Eigenvectors don't: they're only stretched, by a factor called the eigenvalue. Finding them simplifies the matrix, turning repeated application into repeated multiplication by numbers, which is why eigenvectors explain long-run behaviour. A Markov chain settles into its eigenvector for eigenvalue 1. PageRank is the dominant eigenvector of the web's link matrix. Fibonacci numbers grow like the golden ratio because that's an eigenvalue. Power iteration finds the dominant eigenvector by nothing more than multiplying repeatedly.

Eigenvalues and eigenvectors

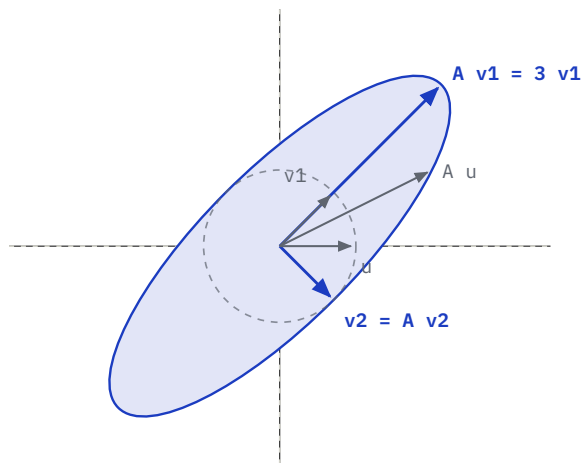
A nonzero vector $\mathbf{v}$ is an **eigenvector** of a square matrix A , with **eigenvalue** λ , if

$$A\mathbf{v} = \lambda\mathbf{v}.$$

Rewriting this as $(A - \lambda I)\mathbf{v} = \mathbf{0}$ shows that $A - \lambda I$ sends a nonzero vector to zero, so it must squash space, and chapter 16 says that happens exactly when its determinant is zero. The eigenvalues are therefore the roots of the **characteristic polynomial** $\det(A - \lambda I) = 0$. For

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \det(A - \lambda I) = (2 - \lambda)^2 - 1 = \lambda^2 - 4\lambda + 3,$$

with roots 3 and 1. Solving $(A - 3I)\mathbf{v} = \mathbf{0}$ gives $\mathbf{v} = (1, 1)$, and $(A - I)\mathbf{v} = \mathbf{0}$ gives $(1, -1)$.



$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

eigenvalues 3 and 1

$v_1 = (1, 1)$ is stretched by 3

$v_2 = (1, -1)$ is left alone

$u = (1, 0)$ turns: $A u = (2, 1)$

*the circle becomes an ellipse
whose axes are the eigenvectors*

The dashed unit circle is carried to the shaded ellipse. Vectors along the eigenvectors stay on their own lines, one stretched by 3 and one unchanged. Every other vector, like u , is turned towards the dominant eigenvector, and repeating A turns it further each time.

Diagonalisation and powers

If an $n \times n$ matrix has n independent eigenvectors, put them in the columns of V and the eigenvalues on the diagonal of Λ . Then $A = V\Lambda V^{-1}$, and the middle factors cancel in a power:

$$A^k = V\Lambda^k V^{-1}.$$

Raising a diagonal matrix to a power just raises each eigenvalue to that power. In the eigenvector coordinates, A is nothing but independent stretches, and in the long run the largest eigenvalue dominates everything. Fibonacci numbers are the standard example.

The step $(F_{n+1}, F_n) \mapsto (F_{n+2}, F_{n+1})$ is multiplication by $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, whose characteristic polynomial $\lambda^2 - \lambda - 1$ has roots φ and ψ , exactly chapter 11's characteristic equation. Diagonalising gives Binet's formula. For exact large Fibonacci numbers, raise the matrix to the n -th power by repeated squaring in integer arithmetic: the example gets all 209 digits of F_{1000} from 16 matrix multiplications, with none of the floating-point trouble chapter 11 found at $n = 71$. Not every matrix can be diagonalised. $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ has only one eigenvector direction, and rotations of the plane have no real eigenvectors at all (exercise 17.2).

Power iteration

Start with almost any vector, multiply by A , rescale to length 1, and repeat. Write the starting vector in eigenvector coordinates as $c_1 v_1 + c_2 v_2 + \dots$. After k steps the components are $c_i \lambda_i^k$, so the one with the largest $|\lambda|$ wins, and the vector turns towards v_1 with error shrinking like $|\lambda_2/\lambda_1|^k$. That's **power iteration**, the simplest eigenvector algorithm, and it needs only matrix-vector products, so it works on enormous sparse matrices. On the

matrix above, with ratio 1/3, it reaches 12 digits in 26 steps. The ratio is everything, though. The example's last lines need 27 iterations at ratio 0.5, 175 at 0.9 and 1,833 at 0.99, matching $\log(10^{-8})/\log(\text{ratio})$.

Markov chains

A **Markov chain** moves between states at random, with the next state depending only on the current one. Its **transition matrix** P has P_{ij} as the probability of moving from state i to state j , so each row sums to 1. If π_t is the row vector of probabilities at time t , then $\pi_{t+1} = \pi_t P$. A **stationary distribution** satisfies $\pi P = \pi$: it's an eigenvector, on the left, for eigenvalue 1. Take a weather model: a sunny day is followed by sun with probability 0.9, and a rainy day by sun with probability 0.5. Solving $\pi P = \pi$ gives $\pi = (5/6, 1/6)$, so 5 days in 6 are sunny in the long run, whatever the starting day. Starting from rain, the example reaches 0.5 on day 1, 0.825 on day 5 and 0.833333 by day 30. If the chain can reach every state from every other, and doesn't cycle with a fixed period, the stationary distribution is unique and every starting point converges to it.

PageRank

In 1998 Sergey Brin and Larry Page ranked web pages with a Markov chain. Imagine a **random surfer** who, at each step, follows a random link on the current page with probability $d = 0.85$, and otherwise jumps to a page chosen uniformly at random. A page's **PageRank** is the long-run fraction of time the surfer spends there, the stationary distribution of this chain. That definition is recursive in a useful way. A page is important if important pages link to it, and the eigenvector equation is exactly what resolves the circularity. The random jumps make every page reachable from every other, which guarantees a unique answer, and they bound the second eigenvalue by d , so power iteration converges at a geometric rate. Real implementations also have to handle pages with no outgoing links, which the example's graph avoids.

The example's nine pages show why counting links isn't enough. The page "post2" has five incoming links, more than any other, but ranks only third. Three of its links come from spam pages that nobody links to, and a link from an unimportant page passes on little importance. "Home" and "blog", with three links each from pages that matter, rank first.

Computed

The example runs power iteration and checks it against the characteristic polynomial, computes F_{1000} by repeated squaring, iterates the weather chain, ranks nine pages with PageRank, and measures how the eigenvalue ratio controls convergence.

EXAMPLE

eigen.py · power iteration, Fibonacci by matrix powers, a Markov chain and PageRank

```
import math
```

```
def matvec(A, v): return [sum(a * x for a, x in zip(row, v)) for row in A]
```

```

def matmul(A, B): return [[sum(A[i][k] * B[k][j] for k in range(len(B))) for j in
    range(len(B[0]))] for i in range(len(A))]
def normalise(v):
    s = math.sqrt(sum(x * x for x in v)); return [x / s for x in v]

# 1. Power iteration finds the dominant eigenvector
A = [[2, 1], [1, 2]] # eigenvalues 3 and 1, eigenvectors (1,1) and
    (1,-1)
v, steps = [1.0, 0.0], 0
while True:
    w = normalise(matvec(A, v)); steps += 1
    if max(abs(a - b) for a, b in zip(v, w)) < 1e-12: break
    v = w
lam = sum(a * b for a, b in zip(matvec(A, w), w))
print(f"power iteration on [[2,1],[1,2]]: eigenvector ({w[0]:.6f}, {w[1]:.6f}), eigenvalue
    {lam:.9f}, {steps} steps")
print(f"characteristic polynomial: lambda^2 - 4 lambda + 3 = 0, roots {(4 + math.sqrt(4)) /
    2:g} and {(4 - math.sqrt(4)) / 2:g}")
u = [1, 0]
print(f"A(1,0) = {matvec(A, u)}: turned off its line; A(1,1) = {matvec(A, [1, 1])} = 3 x
    (1,1); "
    f"A(1,-1) = {matvec(A, [1, -1])} = 1 x (1,-1)")

# 2. Fibonacci numbers by repeated squaring of [[1,1],[1,0]], in exact integers
def mat_pow(M, n):
    result, mults = [[1, 0], [0, 1]], 0
    while n:
        if n & 1: result = matmul(result, M); mults += 1
        M = matmul(M, M); n >>= 1; mults += 1
    return result, mults
Fm, mults = mat_pow([[1, 1], [1, 0]], 1000)
print(f"\nF(1000) has {len(str(Fm[0][1]))} digits, found with {mults} 2x2 matrix
    multiplications; "
    f"starts {str(Fm[0][1][:12]}...")

# 3. A Markov chain settles into its stationary distribution, the eigenvector for eigenvalue 1
P = [[0.9, 0.1], [0.5, 0.5]] # sunny -> sunny 0.9; rainy -> sunny 0.5
pi = [0.0, 1.0] # start on a rainy day
for day in range(1, 31):
    pi = [sum(pi[i] * P[i][j] for i in range(2)) for j in range(2)]
    if day in (1, 2, 5, 30): print(f"day {day:2}: P(sunny) = {pi[0]:.6f}")
print("stationary distribution solving pi P = pi: (5/6, 1/6) = (0.833333, 0.166667)")

# 4. PageRank by power iteration, on six pages
links = {"home": ["about", "blog", "shop"], "about": ["home"], "blog": ["home", "post1",
    "post2"],
    "post1": ["post2", "blog"], "post2": ["blog"], "shop": ["home"],
    "spam1": ["post2"], "spam2": ["post2"], "spam3": ["post2"]} # pages nobody links
    to
pages = list(links); n = len(pages); d = 0.85
rank = {p: 1 / n for p in pages}
for it in range(1, 200):
    new = {p: (1 - d) / n for p in pages}
    for p, outs in links.items():
        for q in outs: new[q] += d * rank[p] / len(outs)
    delta = max(abs(new[p] - rank[p]) for p in pages); rank = new
    if delta < 1e-10: break
indeg = {p: sum(p in outs for outs in links.values()) for p in pages}
print(f"\nPageRank on {n} pages (damping 0.85) converged in {it} iterations:")
for p in sorted(pages, key=rank.get, reverse=True):
    print(f"    {p:6} {rank[p]:.4f}    incoming links: {indeg[p]}")

# 5. Convergence speed depends on the ratio of the top two eigenvalues
for ratio in (0.5, 0.9, 0.99):
    B = [[1, 0], [0, ratio]]
    v, k = normalise([1.0, 1.0]), 0

```

```
while abs(v[1]) > 1e-8: v = normalise(matvec(B, v)); k += 1
print(f"eigenvalue ratio {ratio}: {k:5} iterations to reach 1e-8 (log(1e-8)/log(ratio) =
      {math.log(1e-8) / math.log(ratio):.0f})")
```

OUTPUT

power iteration on $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$: eigenvector (0.707107, 0.707107), eigenvalue 3.000000000, 26 steps

characteristic polynomial: $\lambda^2 - 4\lambda + 3 = 0$, roots 3 and 1

$A(1,0) = [2, 1]$: turned off its line; $A(1,1) = [3, 3] = 3 \times (1,1)$; $A(1,-1) = [1, -1] = 1 \times (1,-1)$

F(1000) has 209 digits, found with 16 2x2 matrix multiplications; starts 434665576869...

day 1: P(sunny) = 0.500000

day 2: P(sunny) = 0.700000

day 5: P(sunny) = 0.824800

day 30: P(sunny) = 0.833333

stationary distribution solving $\pi P = \pi$: $(5/6, 1/6) = (0.833333, 0.166667)$

PageRank on 9 pages (damping 0.85) converged in 89 iterations:

blog 0.2743 incoming links: 3

home 0.2368 incoming links: 3

post2 0.1770 incoming links: 5

post1 0.0944 incoming links: 1

about 0.0838 incoming links: 1

shop 0.0838 incoming links: 1

spam1 0.0167 incoming links: 0

spam2 0.0167 incoming links: 0

spam3 0.0167 incoming links: 0

eigenvalue ratio 0.5: 27 iterations to reach 1e-8 ($\log(1e-8)/\log(\text{ratio}) = 27$)

eigenvalue ratio 0.9: 175 iterations to reach 1e-8 ($\log(1e-8)/\log(\text{ratio}) = 175$)

eigenvalue ratio 0.99: 1833 iterations to reach 1e-8 ($\log(1e-8)/\log(\text{ratio}) = 1833$)

Symmetric matrices, like the one in the figure, are especially well behaved. The **spectral theorem** says their eigenvalues are real and their eigenvectors can be chosen perpendicular, so they're pure stretches along perpendicular axes. Covariance matrices are symmetric, and their eigenvectors are the **principal components** of a dataset, the directions of greatest variation, which is the idea behind principal component analysis and, more generally, the singular value decomposition.

COMMON TRAPS

Assuming every matrix can be diagonalised. Some lack enough eigenvectors.

Expecting real eigenvalues from every real matrix. Rotations have complex ones.

Running power iteration when the top two eigenvalues are close, where convergence crawls.

Forgetting to normalise during power iteration, so the numbers overflow or vanish.

Using right eigenvectors for a Markov chain whose rows are the "from" states. The stationary distribution is a left eigenvector.

Ignoring dangling nodes in PageRank, pages without outgoing links, which leak probability out of the chain.

EXERCISES

EXERCISE 17.1 Eigenvalues by hand

Find the eigenvalues and eigenvectors of $\begin{pmatrix} 4 & 1 \\ 2 & 3 \end{pmatrix}$, and check one eigenvector by multiplying.

SOLUTION

$$\det(A - \lambda I) = (4 - \lambda)(3 - \lambda) - 2 = \lambda^2 - 7\lambda + 10 = (\lambda - 5)(\lambda - 2),$$

so the eigenvalues are 5 and 2. For $\lambda = 5$, $(A - 5I)\mathbf{v} = \mathbf{0}$ reads $-v_1 + v_2 = 0$, so $\mathbf{v} = (1, 1)$, and indeed $A(1, 1) = (5, 5)$. For $\lambda = 2$, $2v_1 + v_2 = 0$ gives $\mathbf{v} = (1, -2)$, and $A(1, -2) = (2, -4)$. Two quick checks for any 2×2 matrix: the eigenvalues sum to the trace ($4 + 3 = 7 = 5 + 2$) and multiply to the determinant ($12 - 2 = 10 = 5 \times 2$).

EXERCISE 17.2 A matrix with no real eigenvectors

Show that the rotation by 90° , $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$, has no real eigenvalues. Why does that make sense geometrically, and what does power iteration do with it?

SOLUTION

$\det(A - \lambda I) = \lambda^2 + 1$, which has no real roots; the eigenvalues are $\pm i$. Geometrically, a rotation by 90° turns every nonzero vector off its line, so no real vector is merely stretched. Power iteration never settles: the vector just rotates by 90° each step and returns to where it started every four steps. Both eigenvalues have absolute value 1, so neither dominates. This is the general failure case of power iteration, two or more eigenvalues tied for the largest absolute value.

EXERCISE 17.3 Every two-state chain

A two-state chain moves from state 1 to state 2 with probability a and from state 2 to state 1 with probability b , with $0 < a, b < 1$. Find its stationary distribution and its second eigenvalue, and check both against the weather example.

SOLUTION

$P = \begin{pmatrix} 1-a & a \\ b & 1-b \end{pmatrix}$. In the stationary state the flows balance:
 $\pi_1 a = \pi_2 b$, so $\pi = \left(\frac{b}{a+b}, \frac{a}{a+b} \right)$. The eigenvalues sum to the trace, $2 - a - b$, and one of them is 1, so the other is $1 - a - b$. The distance from stationarity shrinks by that factor every step. For the weather, $a = 0.1$ and $b = 0.5$, so $\pi = (5/6, 1/6)$ and the second eigenvalue is 0.4. Starting from rain, the gap to $5/6$ is 0.833, then 0.333, then 0.133: it shrinks by a factor of 0.4 each day, exactly the example's 0.5 and 0.7 for days 1 and 2.

CHECKLIST

- ☐ Can you find eigenvalues from the characteristic polynomial and solve for eigenvectors?
- ☐ Can you explain why $A^k = V\Lambda^k V^{-1}$ makes the largest eigenvalue dominate?
- ☐ Can you run power iteration and predict its speed from the eigenvalue ratio?
- ☐ Can you find the stationary distribution of a Markov chain?
- ☐ Can you explain PageRank as a random surfer and as an eigenvector?

GO DEEPER

VIDEO Essence of Linear Algebra: eigenvectors and eigenvalues 3Blue1Brown

PAPER The PageRank citation ranking: bringing order to the web Page, Brin, Motwani & Winograd

COURSE 18.06: eigenvalues, Markov matrices and the SVD Gilbert Strang, MIT OpenCourseWare

Derivatives, gradients and optimisation

3 exercises

For an engineer, calculus is mostly about rates and optimisation. The derivative is the best linear approximation to a function, the gradient points uphill, and gradient descent walks downhill to minimise a loss, which is the engine inside machine learning. This chapter covers three ways to compute derivatives. By hand, with the rules. Numerically, with finite differences that fight a losing battle against floating point. And automatically, with dual numbers and backpropagation, which is how every machine learning framework does it. It ends with the step size that makes gradient descent diverge, and Newton's method, which doubles its correct digits at every step.

The derivative

The **derivative** of f at x is the limit of the slope between nearby points,

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h},$$

and it gives the best linear approximation near x : $f(x+h) \approx f(x) + f'(x)h$. The rules for computing derivatives follow from the definition: sums differentiate term by term, the **product rule** gives $(fg)' = f'g + fg'$, and the **chain rule** gives

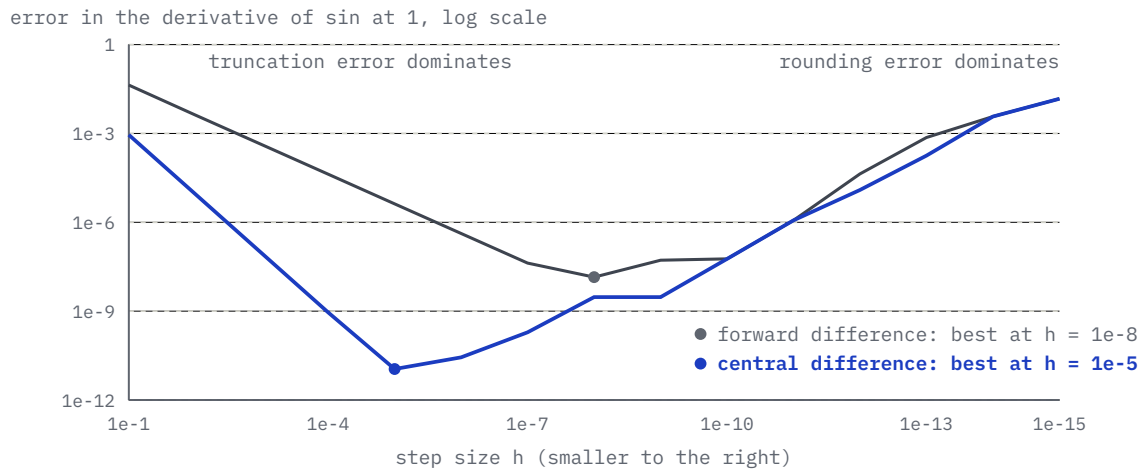
$$(f(g(x)))' = f'(g(x)) \cdot g'(x).$$

The chain rule is the one that matters most for computing. Any program built from elementary operations is a long composition of functions, and the chain rule says its derivative is the product of the derivatives of the steps, which is exactly what automatic differentiation exploits.

Numerical derivatives and their two errors

The obvious way to differentiate a function you can only evaluate is to pick a small h and compute the **forward difference** $(f(x+h) - f(x))/h$. It has two errors that pull in opposite directions. The **truncation error** comes from the curvature the straight line ignores, about $h|f''(x)|/2$, so it shrinks as h shrinks. The **rounding error** comes from subtracting two nearly equal floating-point numbers, each correct to a relative precision $\epsilon \approx 10^{-16}$, and it's about $\epsilon|f(x)|/h$, so it grows as h shrinks. Their sum is smallest when $h \approx \sqrt{\epsilon} \approx 10^{-8}$, and even

then the derivative is good to only about 8 digits. The **central difference** $(f(x + h) - f(x - h))/2h$ cancels the curvature term, leaving a truncation error proportional to h^2 . Its best h is about $\epsilon^{1/3} \approx 10^{-5}$, with an error near 10^{-11} .



Making h smaller helps until rounding error takes over, and then it hurts. The forward difference bottoms out near $h = 10^{-8}$ with about 8 correct digits, and the central difference near $h = 10^{-5}$ with about 11. By $h = 10^{-15}$ the central difference is about 16 times worse than it was at $h = 0.1$.

Automatic differentiation

Automatic differentiation computes derivatives exactly, up to ordinary rounding, by applying the chain rule to the program itself. The neatest version uses **dual numbers**, numbers of the form $a + b\epsilon$ with the rule $\epsilon^2 = 0$. Expanding a function at $a + \epsilon$, every term beyond the linear one vanishes:

$$f(a + \epsilon) = f(a) + f'(a)\epsilon.$$

So if you implement addition, multiplication and the elementary functions on dual numbers, where $\sin(a + b\epsilon) = \sin a + b \cos a \epsilon$ and so on, then running any program on the input $x + 1\epsilon$ returns the function's value and its derivative at once. The example implements this in a dozen lines and differentiates $x \sin x e^x$ at $x = 1$, matching the derivative worked out by hand with the product rule to all 12 printed digits.

Dual numbers are **forward mode**, one pass per input variable. A neural network has millions of parameters and one loss, and it needs the derivative of the loss with respect to every parameter. **Reverse mode** does that in a single backward pass. It records the computation, then applies the chain rule from the output back towards the inputs, accumulating each intermediate's contribution. Reverse mode applied to neural networks is **backpropagation**, and it's what PyTorch and JAX compute when you ask for a gradient.

Gradients

For a function of several variables, the **partial derivative** $\partial f/\partial x$ is the derivative with respect to one variable with the others held fixed. Collected into a vector, the partials form the **gradient**:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right).$$

The gradient points in the direction of steepest increase, and its length is the rate of increase in that direction, because the linear approximation $f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \nabla f \cdot \mathbf{h}$ is a dot product, largest when $\mathbf{h}$ points along ∇f (chapter 16). At a minimum of a smooth function the gradient is zero. The converse fails: a zero gradient can also mark a maximum or a **saddle point**, rising in one direction and falling in another.

Gradient descent

To minimise f , repeatedly step against the gradient:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k),$$

where η is the **step size**, called the learning rate in machine learning. Its choice is critical. On a quadratic bowl whose steepest curvature is L , each step multiplies the error along that direction by $1 - \eta L$, so the iteration converges only if $0 < \eta < 2/L$ (exercise 18.2). The example minimises $x^2 + 10y^2$, where $L = 20$, so the limit is 0.1. A tiny step of 0.001 needs 6,901 iterations, 0.05 needs 132, and 0.09 needs 70. At 0.099 the y -coordinate overshoots and oscillates, taking 627 iterations, and at 0.101 it diverges. The ratio between the steepest and the flattest curvature, here $20/2 = 10$, is the problem's **condition number**. The step must suit the steepest direction, so progress along the flattest direction is slow, and that's why badly scaled features slow down model training. A function is **convex** if the segment between any two points on its graph lies above the graph. For convex functions every local minimum is global, and gradient descent with a suitable step finds it. Neural network losses aren't convex, and practice relies on variants such as stochastic gradient descent, momentum and Adam that cope with noise, saddles and poor conditioning.

Newton's method

To solve $f(x) = 0$, replace f by its linear approximation at the current guess and solve that instead:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Near a simple root, Newton's method converges **quadratically**: the error is roughly squared at each step, so the number of correct digits doubles. For $\sqrt{2}$, with $f(x) = x^2 - 2$, the iteration is $x \leftarrow (x + 2/x)/2$, the Babylonian method, and from $x = 1$ the example's correct digits go 1, 2, 5, 11, 24, 48. Applied to the gradient instead of f , Newton's method minimises

using curvature as well as slope, converging in very few steps. In high dimensions the matrix of second derivatives is too large to form and invert, which is why first-order methods like gradient descent dominate machine learning.

Computed

The example tabulates forward and central difference errors for every h from 10^{-1} to 10^{-15} , differentiates a function with dual numbers, runs gradient descent at six step sizes, and runs Newton's method for $\sqrt{2}$ at 60-digit precision.

EXAMPLE

gradients.py · finite differences, dual numbers, gradient descent and Newton's method

```
import math
from decimal import Decimal, getcontext

# 1. Numerical derivatives: truncation error against rounding error
x, true = 1.0, math.cos(1.0)
print("derivative of sin at 1 (exact: cos 1):")
print("      h      forward error    central error")
for k in range(1, 16):
    h = 10.0 ** -k
    fwd = (math.sin(x + h) - math.sin(x)) / h
    cen = (math.sin(x + h) - math.sin(x - h)) / (2 * h)
    print(f" 1e-{{k:<3}}    {{abs(fwd - true):12.2e}}    {{abs(cen - true):12.2e}}")

# 2. Forward-mode automatic differentiation with dual numbers a + b*eps, where eps^2 = 0
class Dual:
    def __init__(self, a, b=0.0): self.a, self.b = a, b
    def __add__(self, o): o = o if isinstance(o, Dual) else Dual(o); return Dual(self.a + o.a, self.b + o.b)
    def __mul__(self, o): o = o if isinstance(o, Dual) else Dual(o); return Dual(self.a * o.a, self.a * o.b + self.b * o.a)
    __radd__, __rmul__ = __add__, __mul__
def sin(d): return Dual(math.sin(d.a), math.cos(d.a) * d.b)      # chain rule, one function at a time
def exp(d): return Dual(math.exp(d.a), math.exp(d.a) * d.b)
f = lambda t: t * sin(t) * exp(t)
out = f(Dual(1.0, 1.0))
by_hand = math.sin(1) * math.e + math.cos(1) * math.e + math.sin(1) * math.e # product rule, worked out
print(f"\nf(x) = x sin(x) e^x at x = 1: value {{out.a:.12f}}, derivative by dual numbers {{out.b:.12f}}, by hand {{by_hand:.12f}}")

# 3. Gradient descent on f(x, y) = x^2 + 10 y^2, whose largest curvature is L = 20
def descend(lr, steps=10_000):
    px, py = 10.0, 1.0
    for k in range(1, steps + 1):
        px, py = px - lr * 2 * px, py - lr * 20 * py
        val = px * px + 10 * py * py
        if val < 1e-10: return f"converged in {{k:5}} steps"
        if val > 1e12: return f"diverged after {{k}} steps"
    return f"not converged after {{steps}} steps (f = {{val:.2e})"
print("\ngradient descent on x^2 + 10y^2 from (10, 1); stability needs a step size below 2/L = 0.1:")
for lr in (0.001, 0.01, 0.05, 0.09, 0.099, 0.101):
    print(f"    step size {{lr:<6}} {{descend(lr)}}")

# 4. Newton's method: the number of correct digits roughly doubles each step
```

```

getcontext().prec = 60
two = Decimal(2); root = two.sqrt(); g = Decimal(1)
print("\nNewton's method for sqrt(2), x <- (x + 2/x) / 2:")
for k in range(1, 7):
    g = (g + two / g) / 2
    err = abs(g - root)
    print(f"    step {k}: error {err:.1e}, correct digits: about {-int(err.log10()) if err else 60}")

```

OUTPUT

derivative of sin at 1 (exact: cos 1):

h	forward error	central error
1e-1	4.29e-02	9.00e-04
1e-2	4.22e-03	9.00e-06
1e-3	4.21e-04	9.01e-08
1e-4	4.21e-05	9.00e-10
1e-5	4.21e-06	1.11e-11
1e-6	4.21e-07	2.77e-11
1e-7	4.18e-08	1.94e-10
1e-8	1.41e-08	2.97e-09
1e-9	5.25e-08	2.97e-09
1e-10	5.85e-08	5.85e-08
1e-11	1.17e-06	1.17e-06
1e-12	4.32e-05	1.23e-05
1e-13	7.34e-04	1.79e-04
1e-14	3.71e-03	3.71e-03
1e-15	1.48e-02	1.48e-02

$f(x) = x \sin(x) e^x$ at $x = 1$: value 2.287355287179, derivative by dual numbers 6.043404514274, by hand 6.043404514274

gradient descent on $x^2 + 10y^2$ from (10, 1); stability needs a step size below $2/L = 0.1$:

```

step size 0.001 converged in 6901 steps
step size 0.01  converged in 684 steps
step size 0.05  converged in 132 steps
step size 0.09  converged in 70 steps
step size 0.099 converged in 627 steps
step size 0.101 diverged after 640 steps

```

Newton's method for $\sqrt{2}$, $x \leftarrow (x + 2/x) / 2$:

```

step 1: error 8.6e-2, correct digits: about 1
step 2: error 2.5e-3, correct digits: about 2
step 3: error 2.1e-6, correct digits: about 5
step 4: error 1.6e-12, correct digits: about 11
step 5: error 9.0e-25, correct digits: about 24
step 6: error 2.9e-49, correct digits: about 48

```

COMMON TRAPS

Making h as small as possible in a finite difference, past the point where rounding error dominates.

Dropping the inner derivative of the chain rule, the most common hand-differentiation mistake.

A learning rate above $2/L$, which diverges, or far below it, which crawls.

Treating a zero gradient as a minimum. It may be a maximum or a saddle.

Differentiating at a kink, such as $|x|$ or ReLU at 0, where the derivative doesn't exist and libraries pick a convention.

Checking a gradient implementation only on easy points. Compare it with central differences at random points.

EXERCISES

EXERCISE 18.1 The gradient of a regression loss

For a line $y = a + bx$ fitted to points (x_i, y_i) , the mean squared error is $L(a, b) = \frac{1}{n} \sum_i (a + bx_i - y_i)^2$. Compute ∇L , and show that setting it to zero gives chapter 16's normal equations.

SOLUTION

Write $r_i = a + bx_i - y_i$ for the residuals. By the chain rule, each squared term differentiates to $2r_i$ times the derivative of r_i , which is 1 with respect to a and x_i with respect to b . So

$$\frac{\partial L}{\partial a} = \frac{2}{n} \sum_i r_i, \quad \frac{\partial L}{\partial b} = \frac{2}{n} \sum_i r_i x_i.$$

Setting both to zero says the residual vector is perpendicular to the column of ones and to the column of x values, which is exactly $A^T(A\beta - \mathbf{y}) = \mathbf{0}$, the normal equations. Least squares can be solved directly, and gradient descent on the same loss reaches the same answer step by step. For models without a closed form, gradient descent is the only option.

EXERCISE 18.2 The 2/L limit

Run gradient descent on $f(x) = \frac{L}{2}x^2$ with step size η . For which η does it converge, for which does it oscillate, and for which does it diverge?

SOLUTION

$f'(x) = Lx$, so each step is $x_{k+1} = x_k - \eta Lx_k = (1 - \eta L)x_k$, and $x_k = (1 - \eta L)^k x_0$. This converges to 0 exactly when $|1 - \eta L| < 1$, that is $0 < \eta < 2/L$. If $\eta < 1/L$ the factor is positive and x approaches 0 from one side; at exactly $1/L$ it gets there in one step; between $1/L$ and $2/L$ the factor is negative, so x overshoots and oscillates around 0 while

shrinking; above $2/L$, $|1 - \eta L| > 1$ and the oscillation grows without bound. The example's y-coordinate, with $L = 20$, oscillated slowly at 0.099 because the factor was -0.98 , and diverged at 0.101 with factor -1.02 .

EXERCISE 18.3 Dividing without division

Apply Newton's method to $f(x) = 1/x - a$ and show that the iteration computes $1/a$ using only multiplication and subtraction. Why would hardware want this?

SOLUTION

$f'(x) = -1/x^2$, so

$$x_{k+1} = x_k - \frac{1/x_k - a}{-1/x_k^2} = x_k + x_k - ax_k^2 = x_k(2 - ax_k).$$

No division appears. Writing $x_k = (1 - e_k)/a$ gives $e_{k+1} = e_k^2$, so the error squares each step: starting from a guess with a relative error of $1/16$, the steps give errors 2^{-8} , 2^{-16} , 2^{-32} , 2^{-64} . Multiplication is much faster in hardware than division, so some processors and many software libraries compute reciprocals by taking a rough first guess from a small lookup table and refining it with a few such steps.

CHECKLIST

- ☐ Can you differentiate with the product and chain rules?
- ☐ Can you explain the two errors of a finite difference and the best step size for each formula?
- ☐ Can you implement forward-mode differentiation with dual numbers, and say when reverse mode is better?
- ☐ Can you compute a gradient and explain why it points uphill?
- ☐ Can you predict when gradient descent diverges, and why Newton's method converges so fast?

GO DEEPER

VIDEO [Essence of Calculus 3Blue1Brown](#)

PAPER [Automatic differentiation in machine learning: a survey Baydin,](#)

PART VI

Statistics

Two chapters on learning from data without fooling yourself. First sampling: how a sample estimates a population, why averages behave so well, and what a confidence interval does and doesn't promise. Then experiments: A/B tests, the process mistakes that ruin them, and Simpson's paradox, where every subgroup says one thing and the total says the opposite.

Sampling, estimation and confidence intervals

3 exercises

Statistics turns a sample into a statement about a population, with an honest account of the uncertainty. This chapter covers estimators and bias, which is why the sample variance divides by $n - 1$, and the square-root law for standard errors. The central limit theorem explains why averages are approximately normal even when the data isn't, and confidence intervals are built on it. A 95% interval promises something more modest than most people think. The bootstrap gives an interval when no formula exists. The chapter ends with sampling bias, the error that no amount of data reduces.

Populations, samples and estimators

A **parameter** describes a population, such as the true conversion rate of every visitor or the mean latency of every request. A **statistic** is computed from a sample, and an **estimator** is a statistic used to guess a parameter. An estimator is **unbiased** if its expectation equals the parameter. By linearity of expectation (chapter 13), the sample mean $\bar{x}$ is an unbiased estimator of the population mean μ .

The obvious estimator of the variance, the average squared deviation from $\bar{x}$, is biased low. The sample mean sits in the middle of its own sample, so deviations from it are systematically smaller than deviations from the true mean. Working through the expectation gives $\mathbb{E}\left[\sum_i (x_i - \bar{x})^2\right] = (n - 1)\sigma^2$, so dividing by $n - 1$ instead of n makes the estimate unbiased. That's **Bessel's correction**, and it's why `statistics.stdev` and `statistics.pstdev` give different answers. The example estimates the variance of a die, $35/12 \approx 2.917$, from samples of 5: dividing by n averages 2.340, a fifth too low, and dividing by $n - 1$ averages 2.925.

Standard error and the square-root law

The mean of n independent observations, each with variance σ^2 , has variance σ^2/n (chapter 13: variances of independent variables add, and dividing by n divides the variance by n^2). Its standard deviation, the **standard error**, is

$$\text{SE}(\bar{x}) = \frac{\sigma}{\sqrt{n}}.$$

Precision grows with the square root of the sample size: four times the data halves the error, and a hundred times the data gains one decimal digit. That square root governs the cost of every measurement and every experiment. The example's means of n exponential

values have standard deviations 0.445, 0.183 and 0.101 for $n = 5, 30$ and 100 , matching $1/\sqrt{n}$.

The central limit theorem

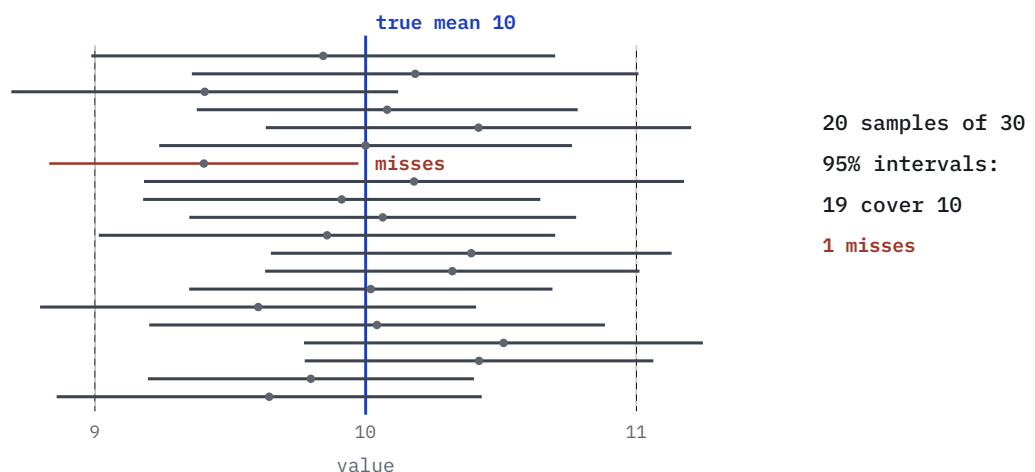
The **central limit theorem** says that the mean of n independent, identically distributed observations with finite variance is approximately normal for large n , with mean μ and standard deviation $\sigma/\sqrt{n}$, *whatever the shape of the original distribution*. That's why normal-based methods work on averages of data that is anything but normal. "Large" depends on the data. The exponential distribution is strongly skewed, with skewness 2, and the skewness of a mean of n values shrinks like $2/\sqrt{n}$. The example measures 0.87 at $n = 5$, 0.35 at $n = 30$ and 0.21 at $n = 100$. For very skewed or heavy-tailed data, such as request latencies or revenue per user, much larger samples are needed, and for distributions with infinite variance the theorem doesn't apply at all.

Confidence intervals

Since $\bar{x}$ is approximately normal with standard deviation $\sigma/\sqrt{n}$, it lies within 1.96 standard errors of μ with probability 95%. Turning that around gives the 95% **confidence interval**

$$\bar{x} \pm 1.96 \frac{s}{\sqrt{n}},$$

with the sample standard deviation s in place of the unknown σ . What "95%" means is subtle. The *procedure* produces intervals that contain the true value in 95% of repeated samples. Any particular interval either contains it or doesn't. The figure shows 20 such intervals from the same population.



Twenty samples of 30 from a population whose mean is 10, each with its 95% interval. The intervals move around with their samples. Nineteen contain the true mean and one doesn't, about what "95%" leads you to expect. Nothing about the missing interval looks different from the others, so you can't tell from one interval whether it's one of the unlucky ones.

Replacing σ by s adds uncertainty that matters for small samples. William Gosset, a chemist at the Guinness brewery publishing as "Student" in 1908, worked out the exact distribution for normal data, the **t distribution**, whose critical values exceed 1.96: 2.776 for samples of 5, 2.045 for samples of 30. The example checks coverage over 10,000 samples. With $n = 5$ from a normal population, the 1.96 interval covers only 87.6% of the time and the t interval 95.1%. With skewed data, even the t interval falls short, 88.4% at $n = 5$ and 92.9% at $n = 30$, because its derivation assumed normality. The method's assumptions are part of the method.

The bootstrap

For a median, a 90th percentile or a ratio of two means, there's often no convenient formula for the standard error. Bradley Efron's **bootstrap** (1979) estimates it by simulation. Treat the sample as a stand-in for the population, draw many new samples of the same size from it *with replacement*, compute the statistic on each, and take the middle 95% of the results as the interval. The example does this for the median of 50 skewed values with 5,000 resamples and gets (0.688, 1.123), which just contains the true median $\ln 2 \approx 0.693$. The bootstrap is easy to apply to almost any statistic, and it inherits whatever is wrong with the sample, so it can't fix a biased one.

Bias you can't compute away

Every formula above assumes the sample is random. When it isn't, more data only makes you more confident in the wrong answer, because a larger sample shrinks the variance and leaves the bias untouched. In 1936 the *Literary Digest* polled over two million readers, drawn largely from telephone directories and car registrations, and predicted a landslide for Alf Landon. George Gallup, with a far smaller but more representative sample, correctly predicted Franklin Roosevelt's victory. In software the same mistake takes quieter forms. A study of the users who answered a survey describes the people who answer surveys. Latency measured only on successful requests misses the ones that timed out, and an analysis of active accounts ignores everyone who churned, which is **survivorship bias**.

Simulated

The example measures Bessel's correction, the central limit theorem's shrinking skewness, the real coverage of z and t intervals on normal and skewed data, and a bootstrap interval for a median.

EXAMPLE

sampling.py · Bessel's correction, the central limit theorem, interval coverage and the bootstrap

```
import math, random
from statistics import mean, median, stdev, pstdev

random.seed(19)
# 1. Bessel's correction: dividing by n - 1 makes the sample variance unbiased
```

```

die_var = 35 / 12
biased, unbiased = [], []
for _ in range(100_000):
    s = [random.randint(1, 6) for _ in range(5)]
    m = sum(s) / 5; ss = sum((x - m) ** 2 for x in s)
    biased.append(ss / 5); unbiased.append(ss / 4)
print(f"variance of a die is {die_var:.3f}; samples of 5 estimate it on average as
      {mean(biased):.3f} "
      f"(divide by n) or {mean(unbiased):.3f} (divide by n - 1)")

# 2. The central limit theorem: means of a skewed distribution become symmetric and normal
def skew(xs):
    m, s = mean(xs), pstdev(xs)
    return sum((x - m) ** 3 for x in xs) / len(xs) / s ** 3
print("\nmeans of n exponential values (skewness 2): standard deviation and skewness of 20,000
      means")
for n in (1, 5, 30, 100):
    means = [mean(random.expovariate(1) for _ in range(n)) for _ in range(20_000)]
    print(f"    n = {n:3}: sd {pstdev(means):.3f} (1/sqrt(n) = {1 / math.sqrt(n):.3f}), skewness
          {skew(means):.2f} "
          f"(2/sqrt(n) = {2 / math.sqrt(n):.2f})")

# 3. Do 95% confidence intervals contain the truth 95% of the time?
def coverage(draw, true_mean, n, crit, trials=10_000):
    hits = 0
    for _ in range(trials):
        s = [draw() for _ in range(n)]
        half = crit * stdev(s) / math.sqrt(n)
        hits += abs(mean(s) - true_mean) <= half
    return hits / trials
normal = lambda: random.gauss(10, 2)
expo = lambda: random.expovariate(1)
T975 = {5: 2.776, 30: 2.045} # Student's t quantiles for 4 and 29 degrees of
                             freedom
print("\ncoverage of nominal 95% intervals, 10,000 trials each:")
print(f"    normal data, n = 5:    mean +- 1.96 s/sqrt(n) covers {coverage(normal, 10, 5,
    1.96):.1%}; "
      f"with t = 2.776 covers {coverage(normal, 10, 5, T975[5]):.1%}")
print(f"    normal data, n = 30:   z covers {coverage(normal, 10, 30, 1.96):.1%}; t covers
      {coverage(normal, 10, 30, T975[30]):.1%}")
print(f"    skewed data, n = 5:    t covers {coverage(expo, 1, 5, T975[5]):.1%}; n = 30: t
      covers {coverage(expo, 1, 30, T975[30]):.1%}")

# 4. The bootstrap: an interval for a median, with no formula
data = [random.expovariate(1) for _ in range(50)]
boots = sorted(median(random.choices(data, k=len(data))) for _ in range(5_000))
lo, hi = boots[124], boots[4874] # the middle 95% of 5,000 resampled medians
print(f"\nsample of 50 skewed values: median {median(data):.3f}; bootstrap 95% interval
      ({lo:.3f}, {hi:.3f}); "
      f"true median ln 2 = {math.log(2):.3f}")

```

OUTPUT

```

variance of a die is 2.917; samples of 5 estimate it on average as 2.340 (divide by n) or 2.925
(divide by n - 1)

```

```

means of n exponential values (skewness 2): standard deviation and skewness of 20,000 means

```

```

n =   1: sd 0.983 (1/sqrt(n) = 1.000), skewness 1.99 (2/sqrt(n) = 2.00)
n =   5: sd 0.445 (1/sqrt(n) = 0.447), skewness 0.87 (2/sqrt(n) = 0.89)
n =  30: sd 0.183 (1/sqrt(n) = 0.183), skewness 0.35 (2/sqrt(n) = 0.37)
n = 100: sd 0.101 (1/sqrt(n) = 0.100), skewness 0.21 (2/sqrt(n) = 0.20)

```

```

coverage of nominal 95% intervals, 10,000 trials each:

```

```

normal data, n = 5:    mean +- 1.96 s/sqrt(n) covers 87.6%; with t = 2.776 covers 95.1%
normal data, n = 30:   z covers 94.2%; t covers 95.1%
skewed data, n = 5:    t covers 88.4%; n = 30: t covers 92.9%

```

sample of 50 skewed values: median 0.805; bootstrap 95% interval (0.688, 1.123); true median $\ln 2 = 0.693$

COMMON TRAPS

Reading a 95% interval as "95% probability the true value is inside". The 95% describes the procedure, not this interval.

Using 1.96 with tiny samples, where the t distribution is needed, and normality matters.

Applying normal-based intervals to heavy-tailed data such as latencies, without checking.

Dividing by n when estimating a variance from a sample.

Believing a bigger sample cures bias. It only makes a biased answer look more precise.

Concluding two groups differ, or don't, from whether their intervals overlap. Compute the interval for the difference instead.

EXERCISES

EXERCISE 19.1 The mean's mean and variance

For independent observations $x_1, \dots, x_n$ with mean μ and variance σ^2 , prove that $\mathbb{E}[\bar{x}] = \mu$ and $\text{Var}[\bar{x}] = \sigma^2/n$.

SOLUTION

By linearity, $\mathbb{E}[\bar{x}] = \frac{1}{n} \sum_i \mathbb{E}[x_i] = \frac{1}{n} \cdot n\mu = \mu$. For the variance, scaling a variable by c scales its variance by c^2 , and variances of independent variables add, so $\text{Var}[\bar{x}] = \frac{1}{n^2} \sum_i \text{Var}[x_i] = \frac{1}{n^2} \cdot n\sigma^2 = \sigma^2/n$. ■ The first result needs no independence. The second does. With positively correlated observations, such as repeated measurements of the same user, the variance of the mean is larger, and treating them as independent produces intervals that are too narrow.

EXERCISE 19.2 How big a sample?

You want to estimate a conversion rate that's around 10% to within ± 0.5 percentage points, with 95% confidence. How many visitors do you need?

SOLUTION

A proportion p estimated from n visitors has standard error $\sqrt{p(1-p)/n}$. We need $1.96\sqrt{0.1 \times 0.9/n} \leq 0.005$, so $n \geq 1.96^2 \times 0.09/0.005^2 \approx 13,830$. Halving the margin to ± 0.25 points would need four times as many, about 55,000, which is the square-root law again. If you don't know p in advance, use $p = 0.5$, which maximises $p(1-p)$ and so gives a sample size that's always enough.

EXERCISE 19.3 An interval for latency

A sample of 1,000 request latencies has mean 120 ms and standard deviation 300 ms. Is $120 \pm 1.96 \times 300/\sqrt{1000}$, about 120 ± 19 ms, a trustworthy 95% interval for the mean latency? What would you report instead?

SOLUTION

A standard deviation 2.5 times the mean signals a heavily skewed, long-tailed distribution: most requests are fast and a few are very slow. With 1,000 observations the central limit theorem helps, but convergence is slow for such data, so the interval's real coverage may fall short of 95%, and a few extreme requests can move the mean a lot. A bootstrap interval is a useful check. More importantly, the mean may be the wrong summary. Users experience the distribution, so report percentiles such as the median, p95 and p99, each with a bootstrap interval if precision matters. A mean of 120 ms is compatible with one request in a hundred taking several seconds.

CHECKLIST

- ☐ Can you explain bias and why the sample variance divides by $n - 1$?
- ☐ Can you derive the standard error $\sigma/\sqrt{n}$ and use it to size a sample?
- ☐ Can you state the central limit theorem and its conditions?
- ☐ Can you explain exactly what a 95% confidence interval promises?
- ☐ Can you compute a bootstrap interval, and name biases that no sample size fixes?

GO DEEPER

BOOK

Think Stats: exploratory data analysis in Python Allen Downey

BOOK Statistics Done Wrong Alex Reinhart

PAPER Bootstrap methods: another look at the jackknife Bradley Efron, 1979

A/B tests and Simpson's paradox

3 exercises

An A/B test is a randomised controlled experiment, the most reliable way to learn whether a change causes an improvement. The statistics fits on a page: a two-proportion test, a p-value and a sample-size formula. Most failed experiments fail on process instead. Tests are too small to detect the effect they're looking for. People check every morning and stop at the first good day, or test twenty metrics and report the one that moved. And they compare groups that weren't randomised, where Simpson's paradox can make every subgroup say one thing and the total say the opposite.

Why randomise

Observational data shows correlations, and correlations have many possible causes. Users who enable a feature may convert more because the feature helps, or because the kind of user who explores settings was going to convert anyway. Assigning users to versions A and B **at random** breaks every such link. Any difference in who ends up in each group, motivation, device, country, is due to chance alone, and chance is exactly what statistics can quantify. So a difference between the groups beyond what chance explains is caused by the change itself. That's why randomised experiments are the standard of evidence, from drug trials to product changes.

Hypothesis tests and p-values

Version A converts 1,000 of 10,000 users, 10.0%, and version B converts 1,080 of 10,000, 10.8%. Is B better, or was it luck? Start from the **null hypothesis** that there's no difference, and ask how surprising the data would be if it were true. Under the null, both groups share one rate, estimated by pooling them as $\hat{p} = 2080/20000 = 0.104$, and the difference in observed rates has standard error $\sqrt{\hat{p}(1 - \hat{p})(1/n_A + 1/n_B)} \approx 0.0043$. The test statistic is

$$z = \frac{\hat{p}_B - \hat{p}_A}{\sqrt{\hat{p}(1 - \hat{p})\left(\frac{1}{n_A} + \frac{1}{n_B}\right)}} = \frac{0.008}{0.0043} \approx 1.85.$$

The **p-value** is the probability, if the null hypothesis were true, of a difference at least this large in either direction. Here it's 0.064. By the usual convention, a result is **significant** if $p < 0.05$, so this one isn't. Three readings of that number are wrong. A p-value isn't the probability that the null hypothesis is true. It isn't the probability that B is no better. And "not significant" doesn't mean "no effect". The 95% interval for the lift, from -0.05 to $+1.65$ percentage points, includes zero, but it also includes improvements that would be well

worth having. The experiment was simply too small to tell. The significance level, 5%, is the false-positive rate you accept: if there's truly no effect, 5% of experiments will still come out "significant".

Power and sample size

The **power** of a test is the probability of detecting an effect that's really there. To detect a lift from rate p_1 to p_2 with significance level α and power $1 - \beta$, each group needs about

$$n = \frac{(z_{1-\alpha/2} + z_{1-\beta})^2 (p_1(1-p_1) + p_2(1-p_2))}{(p_2 - p_1)^2}$$

users, where the z values are standard normal quantiles, 1.96 and 0.84 for the conventional 5% and 80%. Detecting 10% $\rightarrow$ 11% takes 14,749 users per group. The effect size sits squared in the denominator, so halving the lift you want to detect quadruples the sample: 57,760 per group for 10% $\rightarrow$ 10.5%, against 3,839 for 10% $\rightarrow$ 12%. Decide the sample size before the experiment starts. An underpowered test mostly returns "not significant", and when it does come out significant, the estimated effect is likely to be exaggerated, because only the lucky overestimates crossed the threshold.

Peeking

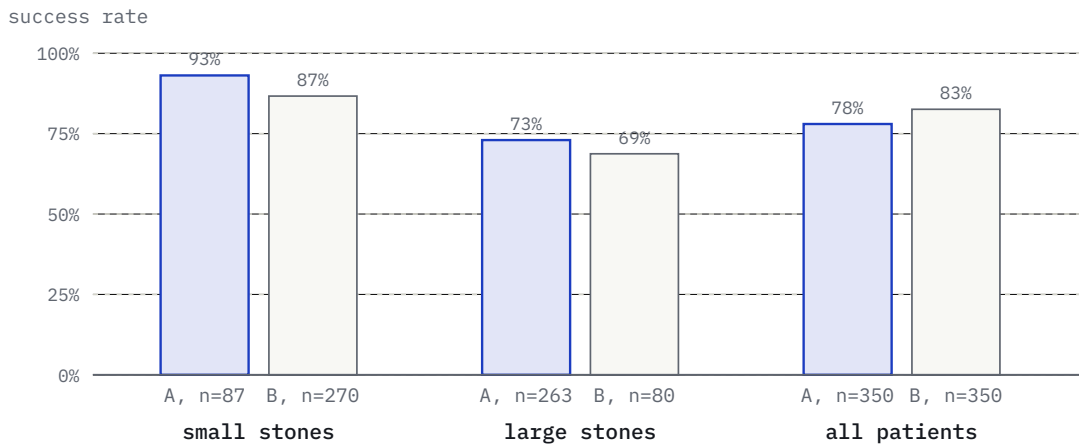
The most common way to ruin an experiment is to check it repeatedly and stop as soon as it looks significant. Each check is another chance for noise to cross the threshold, and stopping at the first crossing keeps the false alarms and discards the corrections. The example runs 2,000 A/A tests, comparisons of two identical versions where any "win" is false by construction. Checked once at the end, 4.7% come out significant, close to the 5% promised. Checked after every 500 users and stopped at the first $p < 0.05$, 25.6% do. If you need to monitor an experiment as it runs, fix the number of looks in advance and spend the 5% across them, or use sequential methods designed for continuous monitoring, which several experimentation platforms provide.

Many comparisons

Test 20 independent metrics on a change that does nothing, and the chance that at least one is "significant" at 5% is $1 - 0.95^{20} \approx 64\%$. The example's 5,000 simulated experiments give 63.6%. Report the one metric that moved, and you've reported noise. The **Bonferroni correction** tests each of m metrics at α/m , here 0.0025, which brings the chance of any false positive back to about 5%. Less conservative procedures control the expected share of false discoveries instead. The simplest defence is procedural: choose one primary metric before the experiment, and treat the rest as exploration that needs its own follow-up experiment.

Simpson's paradox

A 1986 study in the *British Medical Journal* compared treatments for kidney stones. Open surgery (A) succeeded in 78% of its 350 patients and a less invasive percutaneous procedure (B) in 83% of its 350. B looks better. But split by stone size, A was better for small stones, 93% against 87%, and better for large stones, 73% against 69%.



A wins in both groups and loses overall. The groups have very different sizes: A was given mostly to large stones, the harder cases, and B mostly to small ones. Each overall rate is a weighted average of the group rates, and A's average is weighted towards the harder group.

This is **Simpson's paradox**. There's no arithmetic error. Each overall rate is a weighted average of its group rates, A's weighted 75% towards large stones and B's 77% towards small ones, so the overall comparison mostly compares case mixes, not treatments. Stone size is a **confounder**: it influenced which treatment doctors chose, and it influenced the outcome. Here the subgroup comparison is the right one, because stone size was known before treatment and affects both. That depends on the causal story, though, not on the arithmetic, and the same numbers could call for the overall comparison if the subgroup variable were itself a result of the treatment.

Randomisation prevents the paradox in a properly run A/B test, because it balances confounders across the groups. It returns whenever you compare groups that weren't randomised: users who chose a feature against users who didn't, this month against last month when the traffic mix has shifted between platforms or countries, or segments sliced from an experiment after the fact.

Simulated and computed

The example runs the z-test and its interval, computes sample sizes for three effect sizes, simulates peeking with 2,000 A/A tests, simulates 20 metrics with and without Bonferroni, and tabulates the kidney stone data. The peeking simulation uses

`random.binomialvariate`, which needs Python 3.12 or later.

EXAMPLE `ab.py` · z-tests, sample sizes, peeking, multiple metrics and Simpson's paradox

```

import math, random
from statistics import NormalDist

Z = NormalDist()
random.seed(20)
# 1. A two-proportion z-test
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 = math.sqrt(pooled * (1 - pooled) * (1 / n_a + 1 / n_b)) # standard error if
        there's no difference
    z = (pb - pa) / se0
    p = 2 * (1 - Z.cdf(abs(z)))
    se = math.sqrt(pa * (1 - pa) / n_a + pb * (1 - pb) / n_b)
    return pa, pb, z, p, (pb - pa - 1.96 * se, pb - pa + 1.96 * se)
pa, pb, z, p, ci = ab_test(1000, 10_000, 1080, 10_000)
print(f"A: 1,000 / 10,000 = {pa:.1%} B: 1,080 / 10,000 = {pb:.1%}")
print(f"z = {z:.2f}, two-sided p = {p:.3f}; 95% interval for the lift: ({ci[0]:+.2%},
    {ci[1]:+.2%})")

# 2. How many users to detect 10% -> 11% with 80% power at the 5% level?
p1, p2, alpha, power = 0.10, 0.11, 0.05, 0.80
za, zb = Z.inv_cdf(1 - alpha / 2), Z.inv_cdf(power)
n = (za + zb) ** 2 * (p1 * (1 - p1) + p2 * (1 - p2)) / (p2 - p1) ** 2
print(f"\nusers per group to detect 10% -> 11% (power 80%, alpha 5%): {math.ceil(n):,}")
for lift in (0.02, 0.005):
    q = p1 + lift
    print(f"    for 10% -> {q:.1%}: {math.ceil((za + zb) ** 2 * (p1 * (1 - p1) + q * (1 - q)) /
        lift ** 2):,}")

# 3. Peeking: A/A tests (no real difference) checked after every 500 users per group
def aa_experiment(looks, step, rate=0.10):
    ca = cb = n = 0
    for _ in range(looks):
        ca += random.binomialvariate(step, rate); cb += random.binomialvariate(step, rate); n
            += step # Python 3.12+
        if ab_test(ca, n, cb, n)[3] < 0.05: return True # stop and "ship" at the first
            significant look
    return False
runs = 2000
once = sum(aa_experiment(1, 10_000) for _ in range(runs)) / runs
peek = sum(aa_experiment(20, 500) for _ in range(runs)) / runs
print(f"\nA/A tests, 10,000 users per group, {runs:,} experiments: false positives {once:.1%}
    with one look at the end, "
    f"{peek:.1%} when checking after every 500 users")

# 4. Many metrics at once
metrics, exps = 20, 5000
any_hit = bonf = 0
for _ in range(exps):
    ps = [random.random() for _ in range(metrics)] # p-values are uniform when nothing
        is going on
    any_hit += min(ps) < 0.05; bonf += min(ps) < 0.05 / metrics
print(f"\n{metrics} metrics, no real effect: some metric 'significant' in {any_hit / exps:.1%}
    of experiments "
    f"(1 - 0.95^20 = {1 - 0.95 ** 20:.1%}); with Bonferroni (p < 0.0025): {bonf / exps:.1%}")

# 5. Simpson's paradox: kidney stone treatments (Charig et al., BMJ 1986)
data = {"A (open surgery)": {"small": (81, 87), "large": (192, 263)},
        "B (percutaneous)": {"small": (234, 270), "large": (55, 80)}}
print(f"\n{treatment':18} {'small stones':>16} {'large stones':>16} {'all patients':>16}")
for t, g in data.items():
    s, l = g["small"], g["large"]
    tot = (s[0] + l[0], s[1] + l[1])
    print(f"{t:18} " + " ".join(f"{a:>4}/{b:<4} {a / b:5.0%}" for a, b in (s, l, tot)))

```

OUTPUT

A: 1,000 / 10,000 = 10.0% B: 1,080 / 10,000 = 10.8%

z = 1.85, two-sided p = 0.064; 95% interval for the lift: (-0.05%, +1.65%)

users per group to detect 10% -> 11% (power 80%, alpha 5%): 14,749

for 10% -> 12.0%: 3,839

for 10% -> 10.5%: 57,760

A/A tests, 10,000 users per group, 2,000 experiments: false positives 4.7% with one look at the end, 25.6% when checking after every 500 users

20 metrics, no real effect: some metric 'significant' in 63.6% of experiments ($1 - 0.95^{20} = 64.2\%$); with Bonferroni ($p < 0.0025$): 4.8%

treatment	small stones		large stones		all patients	
A (open surgery)	81/87	93%	192/263	73%	273/350	78%
B (percutaneous)	234/270	87%	55/80	69%	289/350	83%

COMMON TRAPS

Peeking and stopping at the first significant result, which multiplies the false-positive rate.

Reading a p-value as the probability that the null hypothesis is true.

Reading "not significant" as "no effect", especially from an underpowered test.

Testing many metrics and reporting the winner without correcting for the number of tests.

Comparing groups that weren't randomised, where confounders and Simpson's paradox take over.

Ignoring a sample ratio mismatch. Groups meant to be equal that aren't usually mean a bug in the assignment (exercise 20.3).

EXERCISES

EXERCISE 20.1 A borderline test

A has 500 conversions from 5,000 users and B has 560 from 5,000.

Compute the z statistic and the two-sided p-value, and say what you'd do next.

SOLUTION

The rates are 10.0% and 11.2%, and the pooled rate is $1060/10000 = 0.106$. The standard error is $\sqrt{0.106 \times 0.894 \times (2/5000)} \approx 0.00616$, so $z = 0.012/0.00616 \approx 1.95$, and the two-sided p-value is about 0.051, just

above 0.05. Don't round it down, and don't extend the test until it crosses the line, which is peeking. The honest summary is that the data is suggestive but inconclusive, with a 95% interval for the lift of roughly 0 to 2.4 points. If the decision matters, run a new experiment sized in advance for the smallest lift worth detecting.

EXERCISE 20.2 Simpson's paradox as weighted averages

Write each treatment's overall success rate in the kidney stone study as a weighted average of its two group rates, and explain the reversal from the weights.

SOLUTION

Each overall rate is a weighted average of the group rates, weighted by how many patients were in each group:

$$\text{A: } \frac{87}{350}(0.931) + \frac{263}{350}(0.730) = 0.249 \times 0.931 + 0.751 \times 0.730 = 0.78,$$

$$\text{B: } \frac{270}{350}(0.867) + \frac{80}{350}(0.688) = 0.771 \times 0.867 + 0.229 \times 0.688 = 0.826.$$

A's group rates are higher in both groups, but three quarters of A's weight sits on the large-stone rate, the lower one, while three quarters of B's weight sits on the small-stone rate, the higher one. A weighted average can reverse a comparison whenever the weights differ this much. A fair overall comparison applies the same weights to both treatments, for example the share of small and large stones among all 700 patients, which ranks A first.

EXERCISE 20.3 A suspicious split

An experiment meant to assign users 50/50 ends with 50,600 users in A and 49,400 in B. Could that be chance? What should you do with its results?

SOLUTION

Under a fair 50/50 split of 100,000 users, the count in A is binomial with mean 50,000 and standard deviation $\sqrt{100,000 \times 0.25} \approx 158$. An excess of 600 is 3.8 standard deviations, with a two-sided p-value of about 0.00015. That's almost certainly not chance. A **sample ratio mismatch** like this usually means users are being lost or duplicated unequally between the arms: a redirect that drops some B users, a bot filter

applied to one arm, a crash in the new version that keeps some users from being logged. The missing users are unlikely to be typical, so the comparison is no longer between randomised groups. Find the cause before trusting any metric from the experiment. Experimentation platforms check for this automatically for exactly this reason.

CHECKLIST

- ☐ Can you explain why randomisation lets an experiment establish cause?
- ☐ Can you run a two-proportion z-test and state correctly what the p-value means?
- ☐ Can you size an experiment for a given effect, significance level and power?
- ☐ Can you explain why peeking and many metrics inflate false positives, and how to prevent both?
- ☐ Can you explain Simpson's paradox as weighted averages and spot a confounder?

GO DEEPER

BOOK Trustworthy Online Controlled Experiments Kohavi, Tang & Xu

ARTICLE How not to run an A/B test Evan Miller

PAPER Understanding Simpson's paradox Judea Pearl

The stage asks for two things: written solutions to two full problem sets from MIT's 6.042J, including the proofs, and a short simulation that demonstrates a counter-intuitive probability result, with the mathematics written up next to the code. Every chapter here prepares for one or both.

Two problem sets, written as proofs

Take the problem sets from the 6.042J course on MIT OpenCourseWare. Choose one from the first half of the course, on proofs, induction, number theory or graphs (chapters 1–8 of this volume), and one from the second half, on counting and probability (chapters 9–15). Do every problem in the set, not just the ones that look easy. Before you hand a solution to anyone, yourself included, check it against this list:

- 01 State exactly what you're proving, with every variable and its domain.
 - 02 Name the method at the start: direct, contrapositive, contradiction, cases, or induction on what.
 - 03 For induction, write the hypothesis $P(n)$ as a sentence, prove the base case, and mark where the hypothesis is used.
 - 04 Justify every step with a definition, an earlier result, or algebra a reader can check.
 - 05 Check the edge cases: zero, one, the empty set, equal elements, the smallest n .
 - 06 Try to break your own proof: test the claim on small cases by hand or by program.
 - 07 Write in complete sentences. A proof is an argument for a reader, not a list of equations.
 - 08 Mark the end, and reread the whole proof once as a sceptic before calling it done.
-

The simulation

Pick a result that surprises people, derive it, simulate it, and write the two up side by side. The best write-ups explain *why* intuition fails, not just that it does. Any of these works, and each chapter's example is a starting point:

RESULT	THE SURPRISE	CHAPTER
Monty Hall	switching wins two times in three, unless the host doesn't know	Ch. 12
The base rate fallacy	a positive result from a 95%-accurate test means only a 16% chance	Ch. 12
The birthday problem	23 people are enough for even odds; 64-bit IDs collide at a billion	Ch. 12, 14
Derangements	the chance nobody gets their own hat is $1/e$, for 9 people or 9,000	Ch. 10
The coupon collector	the last few coupons take half of all the draws	Ch. 13
The giant component	a network connects suddenly once the average degree passes 1	Ch. 7
The power of two choices	one extra random probe shrinks the fullest bin exponentially	Ch. 14
Peeking at an A/B test	checking 20 times turns a 5% false-positive rate into 25%	Ch. 20
Simpson's paradox	better in every group, worse overall	Ch. 20

Structure the write-up in five parts: the claim, stated precisely; the intuition it defies; the derivation, in the style of this volume's proofs; the simulation, with a fixed random seed so anyone can reproduce it; and the comparison, with the simulated value, the exact value and the sampling error you'd expect from the number of trials (chapter 19). If the simulation and the mathematics disagree, find out which is wrong before you publish. That's the habit this whole volume is built on.

THE TWENTY RULES

One per chapter, each linked back to it. Read them before you write a proof, analyse an algorithm, or trust a number someone hands you.

-
- 01 An implication fails only when its premise holds and its conclusion doesn't; never trade it for its converse.
-
- 02 Examples can disprove a claim but never prove one; search for counterexamples first, then prove.
-
- 03 Every correct loop has an invariant and every correct recursion has an induction proof; write them down.
-
- 04 Equality you hash on must be an equivalence relation, and dependencies must form a partial order.
-
- 05 Divide in modular arithmetic only by numbers with an inverse, and find inverses with Euclid's algorithm.
-
- 06 Understand the mathematics of cryptography, and never deploy your own implementation of it.
-
- 07 Know which graph problems are easy and which merely look easy: every edge once is easy, every vertex once is NP-complete.
-
- 08 A greedy colouring is an upper bound, a stable matching favours whoever proposes, and an edge count can prove a graph isn't planar.
-
- 09 Count each object exactly once, and check every counting formula against brute force on a small case.
-
- 10 For "at least one", count the complement; for overlaps, include and exclude.
-
- 11 Turn loops into sums and recursion into recurrences, and say Θ when you mean a tight bound.
-
- 12 Draw the tree, respect the base rate, and condition on exactly what you learned.
-

- 13 Break a count into indicators and add their probabilities; linearity of expectation needs no independence.

- 14 Bound the tail, not just the average, and remember that collisions arrive after about the square root of the space.

- 15 Randomise the algorithm rather than hoping for random input, and amplify a small chance of success by repetition.

- 16 Read a matrix as a transformation, solve systems by elimination, and check conditioning before trusting floating point.

- 17 Long-run behaviour belongs to the dominant eigenvector, and the eigenvalue ratio decides how long the run is.

- 18 Differentiate automatically when you can, keep the step size below $2/L$, and never push a finite difference's h towards zero.

- 19 A sample shrinks error like one over the square root of its size, and no size cures a biased sample.

- 20 Randomise, size the test in advance, don't peek, choose one primary metric, and beware comparisons that weren't randomised.

READING LIST

The books, courses and papers behind this volume, and the chapters each one deepens. If you read only one, make it *Mathematics for Computer Science*, the 6.042J text, which is free and covers Parts I to IV. If you do only one thing, do the two problem sets from the appendix, because writing proofs that someone else could check is the skill this stage is really about.

WORK	AUTHOR	DEEPENS
Mathematics for Computer Science, and 6.042J	Lehman, Leighton & Meyer	Ch. 1–15
Book of Proof	Richard Hammack	Ch. 1–4, 9–10
A Computational Introduction to Number Theory and Algebra	Victor Shoup	Ch. 5–6
Graph Theory	Reinhard Diestel	Ch. 7–8
Concrete Mathematics	Graham, Knuth & Patashnik	Ch. 9–11
Introduction to Probability (Statistics 110)	Blitzstein & Hwang	Ch. 12–14
Probability and Computing	Mitzenmacher & Upfal	Ch. 13–15
Essence of Linear Algebra and Essence of Calculus	3Blue1Brown	Ch. 16–18
18.06 Linear Algebra	Gilbert Strang	Ch. 16–17
Mathematics for Machine Learning	Deisenroth, Faisal & Ong	Ch. 16–18
Think Stats	Allen Downey	Ch. 19–20
Trustworthy Online Controlled Experiments	Kohavi, Tang & Xu	Ch. 20

Compiled October 2026 as the companion to stage 03, Mathematics for computing, of The Engineer's Syllabus, alongside *The Engineer's Workbench*, *Exchange*, *Ledger*, *Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel* and *Foundry*. It follows the order of topics in the stage and draws on the works in the reading list, and on the original sources for its named results: Euclid's *Elements* and Lamé (1844) on the gcd; Rivest, Shamir and Adleman (1978); Miller (1976) and Rabin (1980) on primality; Heninger, Durumeric, Wustrow and Halderman (2012) on shared primes; Euler (1736) on Königsberg; Erdős and Rényi (1959–60) on random graphs; Hall (1935); Gale and Shapley (1962); Appel and Haken (1976); Karger (1993) on minimum cuts; Freivalds (1977); Vitter (1985) on reservoir sampling; Azar, Broder, Karlin and Upfal (1994) on two choices; Page, Brin, Motwani and Winograd on PageRank; "Student" (1908) on the t distribution; Efron (1979) on the bootstrap; and Charig, Webb, Payne and Wickham (1986) for the kidney stone data. Historical details and figures such as factoring records are late-2026 facts used to teach method, not facts to rely on.

Formulas are written in TeX and converted at build time to MathML, which the browser renders natively, so the page loads no mathematics library. Every example was executed before publication, in pure-standard-library Python with fixed random seeds, so it prints the same output on every run; the A/B example needs Python 3.12 or later. Re-run in October 2026 with Python 3.14.7, every example printed exactly the output shown. Tested here: truth tables and quantifiers; counterexamples to five famous conjectures; induction as recursion, a loop invariant and the 8-puzzle's parity; relation properties, chains and the diagonal argument; the prime number theorem, Euclid's worst case, inverses and the Chinese remainder theorem; Carmichael numbers, Miller-Rabin and RSA at 2048 bits; handshakes, Euler circuits, Cayley's formula, bipartiteness and the giant component; greedy colouring, Gale-Shapley, Hall's condition and Euler's formula; every poker hand; derangements and inclusion-exclusion; the master theorem and Binet's formula in floating point; Monty Hall, Bayes and birthdays; expectations by linearity; tail bounds, balls in bins and the birthday bound; randomised quicksort, Karger, Freivalds, reservoir sampling and quickselect; elimination, least squares and the Hilbert matrix; power iteration, Markov chains and PageRank; finite differences, dual numbers, gradient descent and Newton's method; Bessel's correction, the central limit theorem, interval coverage and the bootstrap; and z-tests, sample sizes, peeking, multiple metrics and Simpson's paradox.