THE TURING BENCH

Workbook

Fifteen chapters from the first tape cell to the edge of the computable — exercises at the bench, the theory underneath, and the history around it.

Turing Bench · every number is measured, nothing is decorative

The companion to the 15 lessons, one chapter per lesson. The chapters follow the app's three acts: **Act I · The Instrument** (1–4), **Act II · Judges and Runaways** (5–10), **Act III · The Summit** (11–15). Each chapter has three or four parts:

- **At a glance** — the short version, if you only have a minute.
- **Exercises** — to do *at the bench*. Every exercise states what "done" looks like; machine-building solutions are sketched at the very end, not next to the exercise. Predict first, run second — that is the whole method.
- **In depth** — how the machine really works: its invariants, its cost, and where it sits in the theory.
- **Fun facts & trivia** — history and connections, strictly optional.

Chapter 1 — The Machine

At a glance

Everything on the bench is one of three things: an infinite **tape** of symbols, a **head** that reads/writes one cell and moves one step, and a single **state** — the machine's entire memory. Turing's 1936 claim, still standing: anything any computer can compute, this can compute.

Exercises

1.1 Load BINARY INCREMENT. Without stepping, write down: how many cells does the tape have? *(Careful — trick question. Drag the tape both ways before answering.)*

1.2 Click cells to write your birth year in binary onto the tape. RESET. What comes back — your year or `1011`? Now explain what RESET restores. *(Check: edits made before the first STEP become the saved input; your year should survive RESET if you never stepped.)*

1.3 Exactly one lamp in the STATE REGISTER is lit. Press STEP once. Which lamp is lit now? What, besides the lamp, remembers anything about the step you just

took? *(Answer: the tape and the head position. That's all there is.)*

In depth

Formally a Turing machine is a seven-piece kit: a finite set of states Q, a tape alphabet Γ with a designated blank, an input alphabet, a transition function δ, a start state, and a set of halting states. The bench maps them one-to-one: Q is the STATES rail, Γ the ALPHABET chips, δ the transition table, the start state gets the START tag in the diagram, halting states are the green-lamp rows.

The word to internalize is **configuration**: the complete snapshot (current state, head position, tape contents). A step is a pure function from configuration to configuration — the same configuration always yields the same next one. That determinism is what makes the trace scrubber possible: the bench records each step's tiny difference and can walk the run backwards, because nothing is random and nothing is hidden.

Why an *infinite* tape? Because any fixed bound would smuggle in a second limit besides time. With finite states but unbounded tape, the machine's *program* is finite while its *workspace* is not — exactly the split a real computer has between CPU and (idealized) memory. Exercise 1.1's trick is this definition in disguise: the tape has no cell count, only a growing written region.

Fun facts & trivia

- Turing's 1936 paper is called *"On Computable Numbers, with an Application to the Entscheidungsproblem"* — the machines were a side effect; the goal was settling a question of Hilbert's about mathematical provability.
- Turing called them **a-machines** ("automatic"). The name "Turing machine" was coined by Alonzo Church — in his *review* of Turing's paper.
- The tape was inspired by typewriter ribbon and paper tape; Turing imagined a human "computer" (the word then meant a person doing calculations!) with pencil, eraser, and unlimited paper.
- The claim "this captures everything computable" is the **Church–Turing thesis**. It is not a theorem — it is a definition that has survived every attack

for ninety years, including quantum computers (they are faster, not more capable).

Chapter 2 — Reading the Program

At a glance

A Turing machine has no code and no CPU — the **transition table is the entire machine**. Each row δ(state, read) → (write, move, next) is one rule; determinism means no (state, read) pair appears twice. If you can read the table, you can run the machine in your head.

Exercises

2.1 Open the Transition Table of BINARY INCREMENT. Translate row 3 (`q0, ␣ → ␣, L, q1`) into one German or English sentence.

2.2 The step counter says 0000. Which row is glowing? Why that one? *(Check: the machine is in q0 and the head reads `1` — the glow marks δ(q0, 1).)*

2.3 Cover the bench with your hand and run the whole machine on paper: input `1011`, six rules, head at cell 00. Write the tape after every step until HALT. Then press STEP the same number of times and compare. *(Check: HALT at step 8, tape `1100`.)*

In depth

δ is a *partial* function: it need not cover every (state, read) pair. The bench's table footer counts the uncovered ones, and reaching one halts the machine with "no rule". That is not sloppiness — it is the classical convention, and it gives you a free second way to stop (the Palindrome Check's NO exit works exactly like this in many textbook designs).

A **complete** table for a machine with k working states and an alphabet of m symbols has exactly k·m rows — Binary Increment: 2 × 3 = 6. This is worth sitting with: the machine's entire behavior, on *every possible input*, for *any number of steps*, is fixed by six lines. Everything you will ever watch it do is already in the table; running is just reading it repeatedly.

The table and the state diagram are the same mathematical object — a labeled graph — drawn twice. Rows are edges; states are nodes. The bench keeps the table as the source of truth and derives the diagram, because a table can't be ambiguous about determinism: a duplicate (state, read) key simply can't be stored.

Fun facts & trivia

- Turing called states **m-configurations** and wrote his tables in quintuples, almost exactly the bench's row format — the notation is ninety years old and basically unchanged.
- Emil Post published a strikingly similar formulation independently in 1936, weeks after Turing. "Post machines" write on a two-way infinite tape with an even smaller instruction set.
- With k states you can remember at most $\log_2(k)$ bits between steps. That's why "the state is the memory" is not a metaphor — it's an exact budget.

Chapter 3 — Your First Run

At a glance

Binary Increment in two phases: **SCAN** walks right to the end of the number; **CARRY** walks back left, flipping trailing `1`s to `0` until it can flip a `0` (or a blank) to `1` and halt. `1011` + 1 = `1100`, eight steps, no arithmetic unit anywhere.

Exercises

3.1 RESET, then set the input to `111` (all ones). Predict: how many steps until HALT, and what does the tape say? Run and check. *(Check the carry ripple: `1000`.)*

3.2 Find an input where the machine writes a cell **left of cell 00**. Verify with the HEAD counter showing a negative number. Why does the tape need to be infinite in both directions even for addition?

3.3 Run to HALT, then use the TRACE scrubber to rewind exactly to the step where the first `1` became `0`. What state was the machine in? *(Check: CARRY — q1.)*

In depth

Each phase owns an **invariant** — a sentence that stays true no matter how many steps the phase runs:

- SCAN (q0): "every cell I have passed is unchanged, and the number's end is still somewhere to my right."
- CARRY (q1): "every cell to my right that I have visited was a `1` and is now a `0` — I still owe the number a +1."

Read the rules again with those sentences in mind and the machine stops being six arbitrary lines: q0's rules *preserve* invariant one, q1's rules preserve invariant two, and the two HALT rules are exactly the moments the debt can be

paid. This is how machines are designed (chapter 4) and how they are proven correct — invariant plus termination argument.

Cost: for an n-digit number with t trailing ones, the run takes (n+1) + (t+1) steps — the scan is always full-length, the carry only as long as the ones run. Worst case is all-ones (`111` → `1000`), which also forces the tape to grow a cell leftwards: exercise 3.2's answer is that *numbers grow at the front*, so even simple arithmetic needs room on both sides.

Fun facts & trivia

- Hardware adders have the same worst case: a **ripple-carry** adder is slow for exactly the reason `1111111` is slow here. CPUs use carry-lookahead circuits to cheat; a one-head Turing machine can't.
- The step-8 run you traced is fully reversible — from `(HALT, tape 1100)` plus the trace you can reconstruct every past instant. Thermodynamically, *erasing* information is what costs energy (Landauer's principle); this machine only overwrites what the trace remembers.

Chapter 4 — Write Your Own

At a glance

Design method in three moves: split the job into **phases** (each phase = one state), give each phase a one-sentence invariant, then fill the table row by row asking "in this phase, seeing this symbol — what must I do?". The bit inverter needs exactly one phase.

Exercises

4.1 Build the bit inverter from lesson 4 (CLEAR, three rules). Test on `10110` → expect `01001`. Export it as your first `.turingbench.json`.

4.2 Delete your `(q0, ␣)` rule. Run again. Where does the machine stop, and what does the status chip say? Why is this *honest* rather than an error? *(Check: "Halted · no rule" — δ is simply not defined there.)*

4.3 Parity flag. Build a machine that reads a block of `1`s and halts with a single `0` or `1` on an otherwise blank tape: `1` if the count was odd, `0` if even. Hint: two states that swap on every `1` — the state IS the parity bit. Test on `111` (odd) and `1111` (even).

4.4 Erase. Build a machine that erases the whole input block (any mix of `0`/`1`) and halts on an empty tape. One state suffices. How is "empty tape" even detectable — what symbol tells the machine it's done?

In depth

The three classic beginner bugs, in the order you will meet them:

1. **The forgotten blank rule.** Your machine works on the input and then halts "no rule" one cell past it. Not a crash — the table footer told you the pair was uncovered before you even ran.
2. **The wrong move on the last write.** Off-by-one at the boundary: you wrote the right symbol but drifted one cell before halting. The HEAD counter and

the trace scrubber find these in seconds.
3. **One state doing two jobs.** If you can't state a single invariant for a state, it's two states wearing one name. Split it — states are free.

Exercise 4.3 teaches the deepest principle in small-machine design: **the state register is a variable**. Two states = one stored bit (the parity), four states = two bits, and so on. Whenever you're tempted to "remember" something, the question is only: does it fit in $\log_2$(number of states you are willing to add) bits? If yes, encode it in states; if no — like an unbounded count — it must go on the tape (that trade is chapter 7's whole story).

Fun facts & trivia

- There is a small community sport of **Turing machine golf** — solving a task in the fewest states or rules. Your 3-rule inverter is already optimal; feel free to verify there's no 2-rule version.
- Exporting your machine produces a file another person can import and run — you have just shipped software for a 1936 platform.

Chapter 5 — Down and Under

At a glance

Binary Decrement mirrors the increment: scan right, then **borrow** leftwards, flipping `0`s to `1` until a `1` can be flipped to `0`. New twist: on input `0` the borrow falls off the left edge — the machine halts in a second halt state, **UNDER**, making underflow a visible verdict instead of silent garbage.

Exercises

5.1 Load BINARY DECREMENT. Before running: predict the tape at HALT for input `1100`, and the state the machine borrows in. Run and check. *(Check: `1011`, and the ripple happens in q1 BORROW.)*

5.2 Which 4-digit input makes the borrow travel farthest? Verify with the trace scrubber that every digit gets flipped. *(Check: `1000` → `0111` — three 0s eaten, then the leading 1.)*

5.3 Round trip: run BINARY INCREMENT on `1011` (→ `1100`), then load BINARY DECREMENT, type `1100` onto the tape, and run (→ `1011`). The two machines share their entire SCAN phase — what is the *only* structural difference between their borrow/carry rules?

5.4 Set the tape to `0` and run: UNDER lights, but the borrow left a `1` on the tape before falling off the edge. Is that a bug? Defend the machine's honesty in one sentence. *(Hint: which lamp is the answer — the tape or the state register?)*

In depth

Increment and decrement are **duals**: swap the roles of `0` and `1` in the ripple phase and one becomes the other. Exercise 5.3's answer, spelled out: CARRY says "1→0 keep going, 0→1 stop"; BORROW says "0→1 keep going, 1→0 stop". The scan phase is literally identical — which suggests (correctly) that a single machine with one extra "which job?" state could do both.

The real lesson is UNDER: a machine can have **several halt states, and they are typed results**. HALT means "here is your number", UNDER means "that operation had no answer". The tape after UNDER still holds the rippled `1`s — scratch evidence of the failed borrow — and that's fine, because the *contract* of this machine is: read the answer's kind from the state register, read the value from the tape only if the kind is HALT. Programmers will recognize a sum type / tagged union, invented here with lamps.

Fun facts & trivia

- Real CPUs handle 0 − 1 the same way in spirit: the subtraction happens anyway and a **carry/borrow flag** in the status register lights up. The UNDER lamp *is* a status flag — your laptop has a tiny lamp rack too, it just doesn't glow.
- In two's-complement arithmetic, 0 − 1 = `11111111` (all ones), which is what this machine would produce if the tape were finite and circular. Fixed-width wraparound is not a law of nature — it is what underflow looks like when nobody installs a lamp.

Chapter 6 — Machines That Judge

At a glance

Recognizers compute no number — they read a string and halt in **YES** or **NO**. The Bracket Matcher crosses off pairs: find the *leftmost* unmarked `)`, walk left to the nearest unmarked `(`, mark both `X`, repeat. Leftover brackets of either kind mean NO. The tape's X-litter is scratch work; the verdict is the lamp.

Exercises

6.1 Trace `(())` by hand: write the tape after each pairing round, then STEP through on the bench and compare. How many rounds? *(Check: 2 — inner pair first? No: LEFTMOST `)` first, so the inner `)` pairs before the outer.)*

6.2 Predict YES/NO for `(()(`, `()()`, `)()(` — then run all three. Which one is rejected *fastest*, and why?

6.3 The judge leaves the tape full of `X` scratch marks. On paper, sketch the extra states needed to restore the original input after a YES verdict. What does this tell you about "output" versus "answer"?

6.4 Build your own judge: accept exactly the strings over `0` / `1` that start and end with the same symbol (single characters count). How few working states do you need? *(Sketch at the end.)*

In depth

Why insist on the **leftmost** unmarked `)`? It buys an invariant that keeps the machine tiny: everything left of that `)` is only `(` or `X` — every earlier `)` was already found in an earlier round (it was leftmost then). So the leftward search (q1) needs rules for exactly two symbols, never has to worry about meeting a stray `)`, and one working state suffices for the whole hunt. Choose the *rightmost* `)` instead and the invariant collapses; you'd need more states to survive the mess. Small machines come from strong invariants, not from clever tricks.

Cost: each pair costs a walk of up to the string's length, and there are n/2 pairs — $O(n^2)$ again, the recurring price of one head and one tape.

Vocabulary worth having: a **decider** halts on every input with a verdict (the Bracket Matcher is one). A **recognizer** merely guarantees YES on good inputs — on bad ones it may say NO *or run forever*. The distinction sounds pedantic until chapter 14, where it becomes the sharpest knife in the drawer: the halting problem is recognizable ("run it and see it halt") but *undecidable* (no machine always answers).

Fun facts & trivia

- The language of balanced brackets is the **Dyck language**, after Walther von Dyck (1856–1934) — group theorist, and the first person to formally define an abstract group.
- Balanced brackets are the skeleton of every programming language: your editor's rainbow brackets, the compiler's parser, JSON validation — all descendants of this judge.
- The X-marks-and-shuttle technique is *the* fundamental single-tape idiom. You will see it again in the 0^m1^n Checker, the Copy Machine, and (as field-matching) inside the Universal Machine's MATCH phase.

Chapter 7 — More Than a Pattern

At a glance

The language 0^n1^n ("some zeros, then equally many ones") is the classic example of what **finite memory cannot do**: any left-to-right pattern matcher would need to count unboundedly. The checker pairs each `0` with a `1` by shuttling — the walking *is* the counting, the tape *is* the memory.

Exercises

7.1 Run `000111` and count the head's direction reversals. Repeat for `00001111`. Give the formula for pairing rounds as a function of n — and say in one sentence why the *work* grows like n^2.

7.2 The regex `0*1*` matches "any 0s, then any 1s". Which half of the 0^n1^n job does it do, and which half does it provably miss? *(Answer: order yes, equal count never.)*

7.3 The checker rejects `0101`. Scrub the trace to the exact read where the verdict became inevitable. Which state, which cell?

7.4 (paper) Sketch a checker for $0^n1^n2^n$. How many marker symbols do you need? *(Sketch at the end — and a bonus fact there about why this language is famous.)*

In depth

The impossibility argument is pure pigeonhole and worth owning: suppose a device with k states reads left to right, once, with no tape. Feed it `0^m` for m = 0, 1, …, k. That's k+1 prefixes but only k states, so two different counts — say `0^i` and `0^j` — leave the device in the *same* state. From that state on it cannot tell them apart, so it must give the same verdict to `0^i 1^i` (good) and `0^j 1^i` (bad). Contradiction. No finite-state device decides 0^n1^n — full stop.

The theory ladder this sits on is the **Chomsky hierarchy**: *regular* languages (finite automata, regexes) ⊂ *context-free* (parsers with a stack — 0^n1^n lives here) ⊂ … ⊂ everything Turing machines handle. Each rung is "the previous machine plus more memory discipline". The bench sits on the top rung, which is why one preset library can hold judges from every level below.

Note the price tag though: possible ≠ cheap. Our checker pays $O(n^2)$ head travel for something a stack machine does in one pass. More power lets you *answer more questions*, not answer them faster.

Fun facts & trivia

- Real-world "regexes" (PCRE, JavaScript) secretly exceed regular languages — backreferences like `(a+)\1` are so powerful that matching them is NP-hard. The theory name and the tool name parted ways decades ago.
- There is a genuine lower-bound theorem in the neighborhood: single-tape Turing machines need $\Omega(n^2)$ steps to recognize palindromes (Hennie, 1965, via "crossing sequences") — the shuttle isn't lazy, it's provably necessary on this hardware.
- 0^n1^n is every textbook's first pumping-lemma victim; the argument above is that lemma with the wrapping paper off.

Chapter 8 — Loops

At a glance

INFINITE LOOP bounces between two cells forever, writing nothing. The bench cannot *know* it never halts — after 10,000 steps the **runaway guard** pauses and says exactly that: "still running, cannot know". Raising the guard changes the number, never the situation.

Exercises

8.1 Load INFINITE LOOP. Read its two rules and explain in one sentence why it can never halt — no running needed.

8.2 Press FAST. At what exact step does the bench interrupt? Raise the guard ×10 and FAST again. What changed about the *answer to "does it halt?"* — and what didn't?

8.3 Build the shortest never-halting machine you can. One state, one rule. Which of your rules' fields makes halting impossible? *(Sketch at the end.)*

8.4 Change your loop so that it also *writes* — make it fill the tape rightwards with `1`s forever. FAST for a while, pause, and read the WRITTEN counter. This machine is doing infinite *work*; the previous one was doing infinite *nothing*. Can the bench tell the difference?

In depth

For THIS machine, non-halting is actually provable by a beautiful little argument: its reachable **configurations** (state, head cell, tape contents) are finite — two states × two cells × an unchanging tape = four snapshots at most. A deterministic machine that revisits a configuration is locked in a cycle forever, and with only four snapshots available it must revisit one within four steps. Detector: remember configurations, flag a repeat. Boole Bench's oscillation detector for unstable circuits is this exact idea wearing different clothes.

So why doesn't the bench ship that detector and retire the guard? Because it only works while the reachable configurations stay finite and small. The moment a machine writes on fresh tape (exercise 8.4, or the next chapter's Printer), configurations never repeat and the detector is blind — and *smarter* detectors run into a wall that is not an engineering wall (chapter 14). The guard is the honest general-purpose tool: a bounded budget, loudly announced. Note what the guard's verdict actually is — not "this loops" but "I stopped looking".

Fun facts & trivia

- Embedded systems ship runaway guards in hardware: a **watchdog timer** reboots the device if the software stops checking in. Your dishwasher very likely contains one.
- The guard's philosophy has a name in practice: *timeouts*. Every network request you've ever made was wrapped in a tiny halting-problem workaround.
- Deciding halting for machines with a *bounded* tape is genuinely possible (finite configurations!) — it's "just" PSPACE-complete, i.e. possible and brutally expensive. Unbounded tape is where possible ends.

Chapter 9 — Two Kinds of Forever

At a glance

TURING'S PRINTER runs forever like the Infinite Loop — but writes `0 1 0 1 …` endlessly, never revisiting a configuration, doing honest unbounded work. From the outside both are "a step counter that won't stop"; inside, one is a cycle and the other a straight line to infinity.

Exercises

9.1 Run TURING'S PRINTER with FAST until the guard trips, note WRITTEN and HEAD. Do the same with INFINITE LOOP. Which counters separate the two kinds of forever — and which counter is identical?

9.2 The loop repeats a configuration every 2 steps; the printer never repeats one. Which of the two could a repeat-detector catch? Connect this to your answer from exercise 8.2.

9.3 Build a printer that prints `001 001 001 …` instead. How many states? Run it and verify with the tape. *(Sketch at the end.)*

9.4 Name three programs on your own computer that are Printers, not Loops. For each: what would "halting" even mean, and would you want it?

In depth

Here is the historical surprise: **Turing's 1936 machines were mostly Printers.** His paper is about computable *numbers* — machines that print the infinite decimal expansion of π or 1/3 digit by digit, forever. In his vocabulary a good machine is **circle-free** (keeps producing digits forever) and a broken one is **circular** (gets stuck, stops producing). Halting wasn't the goal; *productive non-halting* was. The "halting problem" formulation we teach today came later — Turing's own undecidable question was "is this machine circle-free?", which is the same knife with a different handle.

The printer also sharpens chapter 8's detector story into a hierarchy: some forevers are *provably* forever (the loop — finite configurations), some are provable with cleverness (the printer — head position strictly increases, an "energy function" argument), and some resist every method, which is chapter 14's theorem. Termination provers used in industry climb exactly this ladder: they hunt for a quantity that always shrinks or always grows, and when they can't find one, they time out — guard-style.

Fun facts & trivia

- Turing's actual Example I printed `0 ␣ 1 ␣ 0 ␣ 1 …` using four m-configurations and every *other* square — he reserved alternate squares for scratch marks, an early memory layout. Our two-state version is the same idea with the scaffolding removed.
- An operating system is *specified* to be a Printer: the POSIX event loop, a web server's `while (true) accept()` — for these, halting is called "crashing".
- The `0 1 0 1 …` tape is the binary expansion of 1/3 ($0.010101\ldots_2$). This preset computes a real number, in the exact 1936 sense.

Chapter 10 — Busy Beavers

At a glance

The busy-beaver game: among all n-state machines that *do* halt on the empty tape, find the champion — most `1`s written (Σ) or most steps taken (S). Σ(3)=6, Σ(4)=13, Σ(5)=4098 with 47,176,870 steps… and the function grows faster than anything computable. Small tables, cosmic runtimes.

Exercises

10.1 Load BUSY BEAVER 3 (empty tape!). Predict nothing — nobody can. Run at 6 Hz and just watch the choreography. Then RESET and answer from the trace: how many steps, how many `1`s? *(Check: 14 and 6 — the 3-state champion score.)*

10.2 Try to beat it: build any 3-state, 2-symbol machine (`0`, `1`, blank = `0`) that halts on the empty tape with **seven** `1`s. Spend ten honest minutes. You are allowed to fail — a theorem says you must.

10.3 Load BUSY BEAVER 5 and press FAST (its guard ships pre-raised). Time it with your watch, then compute: how long would RUN at 30 steps/s have taken? *(Check: ≈18 days.)* When it halts, read the **MARKS** counter: exactly 4,098 — the Σ(5) score. Now compare with WRITTEN (44,908): the machine *changed* a symbol almost 45,000 times to leave 4,098 standing. Σ counts survivors, not operations.

10.4 BB5 was found in 1989 and proven optimal in 2024. What had to be shown about every *other* 5-state machine to finish that proof — and which lesson's problem does that task collide with?

In depth

Two functions, two crowns: **Σ(n)** = most `1`s left by a halting n-state machine, **S(n)** = most steps taken. They usually have *different* champions: our BB3 preset

is the Σ(3) champion (6 ones, in 14 steps), while the S(3) record is 21 steps by a different machine that writes fewer ones. The scoreboard, complete as of today:

n	Σ(n)	S(n)	settled
1	1	1	trivially
2	4	6	1962
3	6	21	Lin & Radó, 1965
4	13	107	Brady, 1983
5	4,098	47,176,870	bbchallenge, **2024**
6	≥ 10↑↑15	≥ 10↑↑15	likely never

Three counters, three different stories — worth keeping apart: **STEP** counts δ applications (S measures this), **MARKS** counts non-blank cells on the tape right now (Σ measures this, at halt), and **WRITTEN** counts symbol *changes* along the way. BB5 makes 44,908 changes across 47M steps to end with 4,098 marks — it builds and demolishes far more than it keeps. A machine can even have WRITTEN > 0 and MARKS = 0 (write a `1`, then blank it again).

Why uncomputability? Suppose a program could compute S(n). Then halting becomes decidable: given any n-state machine, compute S(n), run the machine S(n)+1 steps — if it hasn't halted by then, it never will (the champion bound says so). Chapter 14 proves no such halting decider exists, so no program computes S. Sharper: S must eventually outgrow *every* computable function, else that function would serve as the same universal timeout. "Grows faster than anything computable" is the precise content of the runaway guard's helplessness.

The 2024 proof of BB(5) is a landmark of a new kind: settling it meant classifying **every** 5-state machine — millions of stubborn non-halters each needing its own non-halting proof (chapter 9's ladder, climbed at industrial scale), the whole thing verified in the Coq proof assistant.

Fun facts & trivia

- Tibor Radó, who invented the game in 1962 ("On Non-Computable Functions"), was a Hungarian mathematician who once worked as a railway brakeman after escaping a Siberian POW camp — on foot, across the Arctic.
- The 2024 result came from **bbchallenge.org** — an open online collaboration of hobbyists and researchers. One of the crucial contributors is known only by a pseudonym.
- It gets stranger: there is a specific Turing machine with 745 states whose halting is *independent of ZFC set theory* — standard mathematics can neither prove nor disprove that it halts. The busy-beaver game reaches the cliffs of mathematics itself before n = 1000.
- BB(6)'s current record machine (2024/25) runs more than $10\uparrow\uparrow15$ steps — a tower of exponents fifteen high. The observable universe's particle count ($\sim10^{80}$) is a rounding error of a rounding error.

Chapter 11 — Many Machines, One Power

At a glance

More machinery is not more power. A second tape *track* dissolves into the **alphabet**: every cell holds a pair, drawn on the bench as half-blocks — □ = 0 over 0, ▀ = 1 over 0, ▄ = 0 over 1, █ = 1 over 1. The TWO-TRACK ADDER uses this to add two stacked binary numbers in a single pass: three working states, always 2n+2 steps, bottom track untouched. Multi-tape machines collapse the same way — slower, never stronger. The claim that nothing ever escapes the tape has a name: the **Church–Turing thesis**.

Exercises

11.1 Load TWO-TRACK ADDER. Decode the tape □▀▄ by hand: what number sits on the top track, what on the bottom? Predict the tape at HALT, then run. *(Check: 5 and 3; the tape ends ▀□▄ — top track 1000 = 8, bottom track still 0011; 10 steps.)*

11.2 Set the tape to ▀▀ (7 on top, 1 below) and run. The sum 1000 needs four digits but the number has three cells — watch where the fourth digit comes from. *(Check: ▀□□▄, with the new ▀ written into cell −1; 8 steps.)*

11.3 The step counter reads 2n+2 for every n-cell input — carries or no carries. Explain why, then name a library machine whose step count DOES depend on the input's digits. *(Check: ADD and CARRY walk every cell exactly once; the carry changes which rule fires, never how many cells are walked. Binary Increment, by contrast, halts early at the first 0.)*

11.4 (paper) Three tracks of bits instead of two: how many symbols does the paired alphabet need? k tracks? What grows — and what stays exactly the same shape? *(Answer: $2^3 = 8$ plus blank, in general 2^k. Only the alphabet pays; states, rules-per-state and the single head keep their shape.)*

In depth

The track trick is pure re-lettering, and that is worth sitting with: $\Gamma' = \Gamma \times \Gamma$, and every rule reads and writes pairs instead of symbols. No simulation happens, nothing is encoded — a two-track machine simply *is* a one-tape machine wearing a wider alphabet. That is why the Two-Track Adder sits in the ordinary library between ordinary machines.

Genuinely separate **tapes** with independent heads take real work: keep each tape on its own track, add one marker track per head to remember where that head stands, and let one simulated step be a full sweep — find the marked cells, act on each, sweep back. A t-step multi-tape run costs $O(t^2)$ single-tape steps (Hartmanis–Stearns, 1965). Note exactly what is lost and what is kept: **time is genuinely lost, reach is not.** Chapter 7 cited Hennie's $\Omega(n^2)$ bound for single-tape palindrome checking; a two-tape machine does it in $O(n)$. Tapes provably buy speed. They never buy answers.

That "never" is the chapter's honest edge. The **Church–Turing thesis** says: whatever any effective procedure — any machine you could ever build — can compute, a Turing machine can compute. It is a thesis, not a theorem, and it is *unprovable in principle*: one side of the equation, "whatever a machine could ever do", is not a mathematical object, so no proof can grip it. What can accumulate is evidence. Since 1936 every serious model of computation — Church's lambda calculus, Post's systems, register machines, random-access machines, cellular automata, your laptop, quantum circuits — has been proven to compute exactly the same class. A single physical device computing beyond the tape would break the thesis tomorrow. Ninety years of trying: nothing has.

Fun facts & trivia

- Hartmanis and Stearns priced the multi-tape collapse in "On the Computational Complexity of Algorithms" (1965) — the paper that gave the field of computational complexity its name.
- Turing was two-tracking in 1936: his machines wrote digits on every other square and scratch marks on the squares in between (chapter 9 met this layout) — tracks avant la lettre.

- Your computer is a tower of "more tapes": registers, three levels of cache, RAM, disk. Engineering cares about the difference enormously; computability cannot see it at all.
- The thesis has a sharper modern sibling, the **physical Church–Turing thesis**: no physical device whatsoever computes beyond the tape. Still standing — quantum computers attack *speed* (Shor's algorithm), not reach, and are provably no counterexample to the original thesis.

Chapter 12 — Machines as Data

At a glance

The encoding legend turns any machine into a string of `0`s and `1`s: states and symbols become runs of zeros, `1`s separate fields, `11` separates rules. Lossless both ways — the δ-decoder proves it live. Once a machine is a string, it fits on a tape; once it fits on a tape, another machine can read it.

Exercises

12.1 Switch to UNIVERSAL with BINARY INCREMENT as guest. Using only the ENCODING legend, decode rule R1's bit-string by hand back to δ(q0, 0) → (0, R, q0). Check against the δ-DECODER panel.

12.2 Encode δ(q1, ␣) → (1, —, HALT) by hand, then find it in the decoder list. *(Careful: which index does `␣` have in the alphabet? Which does HALT have among the states?)*

12.3 PROGRAM SIZE says how many tape cells the encoded machine uses. Switch guests (Bit Inverter, Palindrome Check) and note the sizes. What makes a machine's encoding long — states, symbols, or rules?

In depth

What must an encoding guarantee? Two things only: **losslessness** (the decoder recovers exactly the rules — the bench's test suite round-trips every preset) and **parsability** (you can always tell where a field ends — here, unary runs end at a `1`, rules end at `11`, and no third `1` can ever appear inside a rule). Everything else is taste. Unary is gloriously inefficient — a state index i costs i+1 cells — but it makes the structure *visible*, which is worth more on a workbench than compactness.

The idea's pedigree is worth knowing: Gödel numbered *formulas* in 1931 to make mathematics talk about itself; Turing numbered *machines* in 1936 (his "description numbers") to make computation talk about itself. Both proofs of

impossibility — incompleteness and undecidability — begin with exactly the move you practiced in exercise 12.2: flattening the thing that acts into data that can be acted upon.

Exercise 12.3's answer, quantified: each rule costs roughly (state index + read index + write index + move + next index) + separators, so encoding length grows with rule count *and* with how "deep" in the state/alphabet lists the referenced items sit. Palindrome Check (18 rules, 8 states) dwarfs Bit Inverter (3 rules) — program size is a real, measurable thing here, just like a binary's file size.

Fun facts & trivia

- Turing's own encoding also ended in a single number — a machine's **description number**. The machine you built in chapter 4 has one; it just has rather a lot of digits.
- The stored-program computer is this chapter in silicon: von Neumann's 1945 EDVAC report put program and data in one memory. Von Neumann had read Turing's 1936 paper — and said so, loudly, crediting him.
- Biology got there first: a ribosome reads a tape (mRNA) whose codons encode instructions — and part of what the genome encodes is the machinery that reads genomes.

Chapter 13 — The Universal Machine

At a glance

One fixed machine that runs every machine: FETCH the guest's state and symbol, MATCH the encoded rule, WRITE, MOVE-SIM, RETURN — then repeat. Five μ-states of fixed control; everything guest-specific lives on the tape as data. The measured ratio ("1 sim step ≈ N μ-steps") is the eternal price of generality.

Exercises

13.1 RESET, then press μ-STEP slowly until the first sim step completes. Name the five phases in the order the lamps lit. Where was the blue head during each?

13.2 The guest is about to fire a rule stored *late* in the program (say R4 of 6). Why does that sim step cost more μ-steps than one that fires R1? Verify with the ratio readout across several sim steps.

13.3 Every emulator, VM and interpreter pays the μ-overhead. Name the "program δ", "state reg" and "work tape" of a video-game emulator running on your laptop. *(No wrong answers — the mapping is the point.)*

13.4 Click the μ-PROGRAM seal and inspect the script. Find the single μ-op that changes the guest's tape. Which phase owns it?

In depth

Open the STATE DIAGRAMS tab and hold both diagrams in view: the blue ring never changes, the amber diagram swaps with every guest. That picture *is* the fetch–decode–execute cycle — FETCH/MATCH/WRITE/MOVE/RETURN maps almost embarrassingly well onto what a CPU does with instructions in RAM. The UTM is not an ancestor of the computer in a poetic sense; it is the design.

Honesty note (also in BUILD_NOTES #17): the bench's μ-machine is a phase interpreter whose μ-steps are real head-walk operations — every count you see

is measured work — but it is not itself written as a δ-table. A true δ-encoded UTM exists (Turing gave one) and is famously fiddly: dozens of states doing marking, copying and shuttling. The bench trades that purity for a machine you can actually *watch think*.

Exercise 13.2 generalizes into a real phenomenon: rules stored deeper in the program cost more to reach, because MATCH rejects earlier rules cell by cell. Interpreters have paid this "dispatch cost" ever since — bytecode dispatch tables, branch predictors, JIT compilers are all engineering answers to the μ-step bill you watched accumulate.

Fun facts & trivia

- How small can universal be? Shannon proved a 2-*state* UTM exists (you pay with a huge alphabet) and that 1 state can never suffice. Minsky built a 7-state, 4-symbol UTM in 1962; the frontier since has been a zoo of tiny machines with ever-fancier caveats.
- In 2007, 20-year-old Alex Smith won Wolfram's $25,000 prize by proving a particular 2-state, 3-symbol machine universal — with a controversial twist: it needs an infinite, patterned initial tape. Where exactly "universality" begins is still argued about.
- Every emulator ratio you have ever felt — a SNES game jittering inside a browser inside an OS inside a hypervisor — is the COST OF UNIVERSALITY readout, stacked. Each layer multiplies its own N.

Chapter 14 — The Limit

At a glance

No machine H can decide, for every machine and input, "will it halt?" — because a saboteur D could ask H about *D itself* and do the opposite of whatever H predicts. The runaway guard is therefore not a weakness of this bench but the mathematically best posture available: bounded patience, honestly declared.

Exercises

14.1 In UNIVERSAL mode, load INFINITE LOOP as the guest (lesson 14 does this for you). Let it run. The universal machine simulates perfectly — the guard pulses anyway. What *exactly* does the UTM gain you in answering "does it halt?" *(Answer: nothing. Perfect simulation ≠ foresight.)*

14.2 Retell Turing's argument from lesson 14 in four lines: what does the saboteur machine D do when H says "D halts"? When H says "D loops"? Why is that fatal for H — and not just for this bench, but for every possible one?

14.3 Summit exercise. Chapter 8 showed a loop the bench COULD detect (finite configurations must repeat), chapter 9 showed a forever the repeat-trick cannot touch, and lesson 14 proved no general detector exists. Explain to a friend — in speech, no bench — how all three can be true at once. If you can, the course did its job.

In depth

The proof, with every gear visible. Assume H(M, x) always answers "halts" or "loops" correctly and always halts itself. Build D from H's parts — chapter 12 says machines are data, so D can contain a copy of H and can be handed *its own description*:

> D(M): ask H(M, M). If H says "halts" → loop forever. If H says "loops" → halt.

Now run D(D). If it halts, H said "loops" about it — H was wrong. If it loops, H said "halts" — wrong again. H answered *something* (it always answers), and either answer is false. The only unproven assumption was H's existence; it falls. Note which lessons supplied the parts: encoding (12) lets D receive itself, universality (13) lets D simulate what H predicts about it, and the two-kinds-of-forever intuition (8, 9) is why "just run it and see" was never an answer.

The result radiates. **Rice's theorem** (1951) generalizes it: *every* non-trivial question about what a program does — "does it ever print X?", "is it equivalent to this other program?", "is it a virus?" — is undecidable by the same construction; chapter 15 puts that construction on the bench. This is the true summit of the course: not "one weird question is impossible" but "all interesting questions about behavior are". And yet the practical world isn't paralyzed: undecidable means *no method works on all inputs*, not *no method works*. Termination provers, virus scanners and optimizers all live profitably in the gap — with a guard-shaped timeout at the bottom of every one of them.

Fun facts & trivia

- The Entscheidungsproblem that Hilbert posed and Turing demolished asked for an algorithm deciding mathematical truth. Alonzo Church got there months earlier the same year (1936) with the lambda calculus; that the two utterly different approaches captured the same notion is the strongest single piece of evidence for the Church–Turing thesis.
- Turing wrote the paper at 23, before "computer science" existed. Church's system lives on in Lisp, Haskell and every anonymous function you've ever written; Turing's lives on in every machine that runs them.
- Gödel's incompleteness theorem (1931) and undecidability are siblings: both are diagonal arguments, both use self-reference through encoding. Douglas Hofstadter's *Gödel, Escher, Bach* spends 777 pages on this family resemblance — the bench gives you the load-bearing 5%.
- Microsoft's "Terminator" project proves termination of real device drivers, undecidability notwithstanding — by looking for shrinking quantities, and giving up honestly when it finds none. Its guard lamp is an email to the developer.

Chapter 15 — The Domino Chain

At a glance

A **reduction** turns a would-be solver for a new question into a solver for a question already known impossible — so the new question is impossible too. THE WRAPPER makes the move tactile: five states write `1011` onto the empty tape, rewind, then Binary Increment runs unchanged. "Does the wrapper halt on the empty tape?" *is* "does the increment halt on 1011?". The halting problem is not one wall; it is a wall factory, and this chapter is the assembly line.

Exercises

15.1 Load THE WRAPPER — leave the tape empty — and run to HALT. How many steps, and what does the tape read? Then open the transition table: which rows are Binary Increment, unchanged? *(Check: 16 steps — 8 to smuggle the input in, 8 to increment; tape `1100`; every `q0` / `q1` row matches the increment rule for rule — the test suite pins that equality.)*

15.2 Build your own wrapper: BIT INVERTER with the input `10` built in. How many extra states do you need? Run it on the empty tape. *(Check: three — two writers and a rewind; tape `01` at HALT. Sketch at the end.)*

15.3 (paper) A friend sells a detector for "machine M eventually writes a `1`". Build the wrapper that turns it into a halting detector — and mind the trap: M itself may write `1`s while it runs. *(Sketch at the end.)*

15.4 (paper) "Empty-tape halting is undecidable because it is a special case of the halting problem." What is wrong with that sentence? *(Answer: nothing flows in that direction — special cases can be easier, sometimes trivially so. The reduction runs the other way: every instance of the FULL halting problem hides inside some empty-tape instance, via a wrapper. Impossibility travels along the wrapper, not along "is a special case of".)*

In depth

The shape, formally: question A **reduces to** question B (written A ≤ B) when there is a *computable* translation turning any instance of A into an instance of B with the same answer. Then a decider for B would yield one for A. Decidability flows down the arrow; impossibility flows up. All of this chapter is one instance: A = "does M halt on x?", B = "does it halt on the empty tape?", translation = the wrapper.

The word *computable* is the load-bearing beam. The wrapper is assembled by blind text-processing — write-states read off from x symbol by symbol, M's table appended unchanged (exercise 15.1 checked exactly this). A compiler, not a mathematician: it never needs to know whether M halts, which is precisely why the argument doesn't collapse into circularity. The bench's engine needed no new parts for it either — THE WRAPPER is an ordinary preset.

Chapter 14 stated **Rice's theorem**; here is its engine room. Take any non-trivial property S of program *behavior* ("prints a 1", "ever visits cell −5", "is a virus"). Given (M, x), build M' that first simulates M on x — on a scratch region, with M's output symbols renamed so the simulation cannot trigger S by accident (exercise 15.3's trap) — and then, if M halts, behaves like some fixed machine that HAS property S. Now M' has property S exactly when M halts on x: deciding S decides halting. Both ingredients came from this course: the simulation is chapter 13's universality, the self-contained input is this chapter's wrapper. One honest footnote: reductions feel backwards on first contact — you assume the very detector you want to kill and put it to work. That inversion trips everyone exactly once.

Fun facts & trivia

- Emil Post's **correspondence problem** (1946) is literally a domino game: tiles with a word on top and a word below — can copies be lined up so top and bottom read the same? Undecidable, by a chain from halting, and it topples questions about grammars and compilers to this day. This chapter's title is almost its portrait.

- **Hilbert's 10th problem** — does a polynomial equation have integer solutions? — fell in 1970 (Matiyasevich, completing Davis, Putnam and Robinson): a question of pure number theory turned out to contain the halting problem. The domino chain crosses whole fields of mathematics.
- Fred Cohen proved in 1987 that perfect **virus detection** is undecidable — the wrapper argument in a trench coat. Every antivirus since is heuristics plus a guard lamp.
- The chain also runs backwards through history: Turing's own 1936 undecidable question was "is this machine circle-free?" (chapter 9 met it) — the halting problem as taught today is itself a reduction away from Turing's original.

The road on

The course ends where the family's next question begins. The machine runs — but what does the tape actually *carry*? Every cell on this bench held one symbol from a small alphabet: a choice among a few possibilities. Measuring those choices — what information is, how many bits a message really holds, how much noise a channel survives — is **Shannon Bench**: https://shannon.logicbench.net

Solution sketches

4.3 Parity: states EVEN (start) and ODD; on `1`: erase it (`␣`), move R, swap state; on `␣`: write `0` (in EVEN) or `1` (in ODD), stay, → HALT. Erasing as you go keeps the tape clean so the final digit stands alone.

4.4 Erase: one state: `0→␣,R` and `1→␣,R` loop on itself; `␣→␣,−,HALT`. The first blank after the block is the "done" signal — the input block must be contiguous for this to work.

6.4 Same first and last symbol: read the first symbol and erase it — that branches into a "remember 0" and a "remember 1" state (the state IS the memory). Walk right to the last symbol, compare, halt YES/NO. Two remember-states plus a start and two walk states — 4–5 working states depending on how you count; a single-character input should be YES (it is its own first and last symbol).

7.4 $0^n1^n2^n$: same pairing shuttle with one more marker: each round crosses off one `0` (X), one `1` (Y), one `2` (Z) — two extra states for the longer walk. Bonus fact: this language is the standard example of something even context-free grammars cannot do — the tape out-muscles parsers here, not just regexes.

8.3 Shortest loop: one state q0, one rule δ(q0, _) → (_, R, q0), empty tape. The `next = q0` with no halt state reachable and a move that never revisits a written cell — nothing can ever change, so nothing can ever stop.

9.3 The 001-printer: three states in a cycle: p0 prints `0`→p1, p1 prints `0`→p2, p2 prints `1`→p0, all moving R on blank. The period of the output equals the length of the state cycle — the state register is the machine's entire rhythm section.

10.2 Seven ones with 3 states: impossible — Σ(3) = 6 is a proven maximum (Lin & Radó 1965). If you "succeeded", count your states again (HALT does not count as one of the three) or check that your blank is `0`.

13.2 Late rules cost more: MATCH walks the encoded program left to right and must reject R1…R3 cell by cell before it can even reach R4. Deeper rules → longer

walks → more μ-steps. That's why real instruction sets keep hot paths short.

15.2 Bit Inverter wrapper: three states in front: δ(w1, _) → (`1` , R, w2) and δ(w2, _) → (`0` , L, w3) write the input `10` ; w3 rewinds (`0` / `1` → L, stay in w3; _ → R into the inverter's q0). On the empty tape it halts with `01` after 7 steps — 4 to smuggle and rewind, 3 to invert.

15.3 Prints-a-1 detector: rename `1` throughout M to a fresh symbol (alphabet and rules — a mechanical text edit), so the simulation of M never writes a true `1` . Then route every halt of M into one new state that writes a single `1` and stops. The rebuilt machine writes a `1` exactly when M halts on x; feed it to the claimed detector and the halting problem is decided — so the detector cannot exist. The renaming is the honest part: without it, M's own scratch work would set the detector off early and the equivalence would break.