Course Companion

University Computer Science

Every topic handout in one volume

exercises.lol

Contents

1. **Programming Fundamentals** . . . 3
2. **Topic 2** . . . 16
3. **Topic 3** . . . 24
4. **Topic 4** . . . 32
5. **Topic 5** . . . 42
6. **Topic 6** . . . 51
7. **Topic 7** . . . 59
8. **Topic 8** . . . 68
9. **Topic 9** . . . 76
10. **Topic 10** . . . 85
11. **Topic 11** . . . 93
12. **Topic 12** . . . 101
13. **Topic 13** . . . 110

Programming Fundamentals

University Computer Science

What this course is, and how to use it

This is the first course in University Computer Science: **programming fundamentals**, taught with Python 3. It assumes no prior programming. Every program in this handout is complete and runnable – type it, run it, change one line, and predict what changes before running again. That predict-then-run loop is the entire skill.

The course is a ladder:

- Units 1.1 to 1.3 build the mechanics: values and types, selection, iteration.
- Unit 1.4 packages work into functions.
- Units 1.5 and 1.6 add the standard collections and files with error handling.
- Unit 1.7 is the discipline on top: decomposing problems, testing, debugging.

How to read each unit: a short story says why it matters; a picture or a rule of thumb comes before any syntax; then the precise rules; then worked examples that get harder, each with a trace you can check by running the code; then the mistakes people actually make; then a three-line summary. Every example here can be checked a second way – running it. Reading code without running it is like reading sheet music without hearing it.

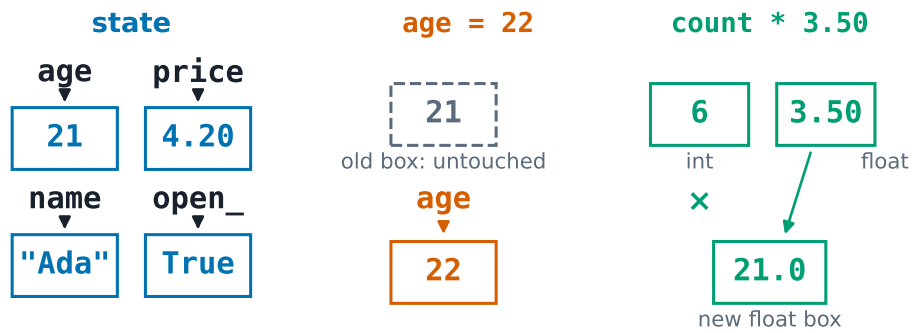
Values, types and input/output

A shopping app shows "4.20" – is that text or a number? If it is text, adding two prices gives "4.203.50"; if it is a number, it gives 8.7. Which one happens is decided by the **type**, and the whole of unit 1.1 is learning to see types before you run anything.

A **value** has a type, and the type decides what you can do with it: `int` (whole numbers), `float` (decimals), `str` (text), `bool` (`True`/`False`). The `type(...)` function names any value's type – ask it freely while learning.

```
age = 21              # int
price = 4.20          # float
name = "Ada"          # str
open_ = True          # bool
print(type(age))      # <class 'int'>
```

Variables are names bound to values; assignment `age = 22` rebinds the name – it never edits the old value. The mental picture, worth keeping for the whole degree:



Names are tags on boxes; expressions read boxes and make new boxes

Reading an expression is "open the boxes, compute, close a new box": `count * 3.50` with `count = 6` opens the `int` box 6 and the `float` box 3.50, and Python's mixing rule promotes the `int` – the result is the `float` `21.0`, never the `int` 21.

Conversion is explicit: `int("21")` is 21, but `"4" + 2` is a `TypeError` – Python refuses to guess whether you meant `"42"` or `6`. Output with `print(...)`; input with `input(...)`, which always returns a **string**, so numbers need converting:

```
born = int(input("Birth year? "))
print("About", 2026 - born, "years old")
```

> **Worked example 1 (the café till).** `count = int(input("Coffees? "))`, then `print("Total:", count * 3.50)`. At 6 coffees this prints `Total: 21.0` – the `.0` marks a float, because `int * float` promotes to float. Watching which type comes out of an expression is unit 1.1's whole habit. Trace it the state way: `input` returns the string `"6"`; `int(...)` opens a new box holding the `int` 6; the multiplication makes a fresh `float` box `21.0`; `print` shows it.

Worked example 2 (the classic accident, repaired). A banner printer joins a count onto a message:

```
count = int(input("Tickets? "))
print("Sold " + count + " today")
```

Run it and Python raises `TypeError: can only concatenate str (not "int") to str` – the rule refuses to guess. The repair names the conversion explicitly: `print("Sold " + str(count) + " today")`, or the cleaner `print("Sold", count, "today")`, which inserts the spaces for you. The lesson to carry: `+` means addition for

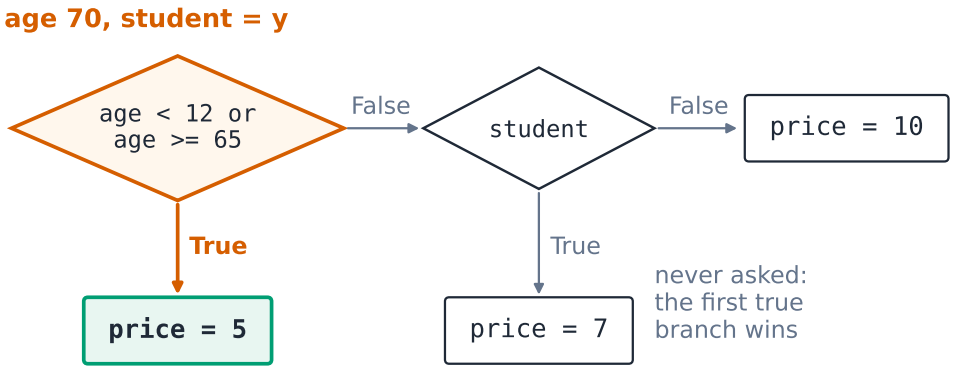
numbers, joining for strings, and never a silent mix.

Common mistakes. Forgetting that `input()` always gives a string, so `input() * 2` repeats text rather than doubling a number. Writing `4.20 + 6` and expecting an `int` answer – any float in the expression makes a float out. Trusting `0.1 + 0.2 == 0.3` (it is `False`; binary rounding – unit 1.2 shows the fix).

Summary: four basic types decide every operation; names are tags, assignment rebinds; expressions mix types by promotion with floats; conversions are explicit and `input()` needs one.

Selection: choosing between paths

A theme park turnstile must charge children, adults and seniors differently – one program, several paths. **Selection** is how a program chooses: `if`, `elif`, `else` run a block only when a condition holds.



Age 70, student: the first true branch decides the price

The condition is any boolean expression; comparisons (`==`, `!=`, `<`, `>=`) and logic (`and`, `or`, `not`) build it. Two rules make every decision readable:

1. Branches are tested top to bottom; the **first true branch wins** and the rest are skipped.
2. Indentation is not decoration – it IS the block. One extra space changes which lines belong to the branch.

```
temp = 37.5
if temp >= 38:
    print("Fever")
elif temp >= 36 and temp < 38:
    print("Normal")
else:
    print("Low")
```

Worked example 1 (the cinema price). Under 12 or 65+ pay 5, students pay 7, everyone else 10:

```
age = int(input("Age? "))
student = input("Student? y/n ") == "y"
if age < 12 or age >= 65:
    price = 5
elif student:
    price = 7
else:
    price = 10
print("Pay", price)
```

Order matters: the first true branch wins, which is why the age check precedes the student check – a 70-year-old student should pay 5, and this ordering says so. Run it with `70` and `y` to see the flowchart's leftmost path taken.

Worked example 2 (grade bands, escalated). The same skeleton, now with a chain where each `elif` only needs one comparison, because the tests above it have already failed:

```
score = int(input("Score? "))
if score >= 80:
    band = "A"
elif score >= 70:
    band = "B"
elif score >= 60:
    band = "C"
else:
    band = "F"
print("Band:", band)
```

At `79` the first test fails, the second succeeds: `Band: B`. The escalation to notice: the `elif score >= 70` does not also need `and score < 80` – reaching that line already proves the score is below 80. Trusting the fall-through like this is what makes chains short; re-testing everything is the beginner's longer, wronger version.

Floating-point comparisons need care: `0.1 + 0.2 == 0.3` is `False` (binary rounding), so compare against a tolerance: `abs(x - y) < 1e-9`.

Common mistakes. Writing `=` where `==` is meant – the classic bug, and Python refuses it in conditions (`SyntaxError`) precisely because it is so common. Testing the special case after the general one, so the general one always wins. Using `elif` with a condition that can never be reached because an earlier branch already caught it.

Summary: conditions choose paths, first-true-wins, indentation is the block; order tests specific-to-general; floats compare by tolerance, never by `==`.

Iteration: doing it again

Paying 5,000 invoices by hand is a career; with a loop it is a program. **Iteration** repeats work: a `for` loop walks a known sequence; a `while` loop runs while a condition holds. The design question is always: do you know how many trips, or does a condition decide?

```
total = 0
for i in range(1, 101):      # 1..100 - range stops BEFORE its end
    total += i
print(total)                 # 5050
```

Use `for` when the trip count is known; use `while` when a condition decides. Every `while` needs three things visible in the code: the variable's start, the condition, and something inside that moves toward exiting – otherwise it is an **infinite loop** .

```
n = 5
factorial = 1
while n > 1:
    factorial *= n
    n -= 1
print(factorial)
```

after	n	factorial	n > 1 ?
start	5	1	True
round 1	4	5	True
round 2	3	20	True
round 3	2	60	True
round 4	1	**120**	False

False: the loop ends and 120 is printed

Trace table for the factorial loop

```
n = 5
factorial = 1
while n > 1:
    factorial *= n
    n -= 1                   # the step toward exiting
print(factorial)             # 120
```


The **trace table** is the course's checking habit for loops: follow the variables round by round, on paper first. For the factorial loop: `(n=5, f=1) → (4, 5) → (3, 20) → (2, 60) → (1, 120)`. Build the table by hand, then add a `print(n, factorial)` inside the loop and let Python grade you.

Worked example 2 (a search with `break`, escalated). Find the first multiple of 7 that is greater than 100 – the trip count is unknown, the condition decides:

```
n = 100
while True:
    if n % 7 == 0:
        break                # found it - leave the loop now
    n += 1
print(n)                     # 105
```

The trace is short: 100, 101, …, 104 all fail the test; 105 passes and `break` exits. `break` leaves the loop early; `continue` skips to the next round. Both are readability tools – a loop that must scan until it finds something, then stop, reads best with `break`. The further escalation, doubling money: at 5% a year, how long does 1000 take to reach 2000?

```
balance = 1000.0
years = 0
while balance < 2000.0:      # the condition itself is the goal
    balance *= 1.05
    years += 1
print(years, round(balance, 2))   # 15 2078.93
```

Fifteen years, and the loop stops the moment the condition flips – the trace table would show balance crossing the line at 2078.93, not landing on 2000 exactly.

Common mistakes. Editing the list (unit 1.5) or the loop variable you are `for`-looping over. `range(1, 100)` when 100 was meant – the end is excluded. A `while` whose exit step is missing or commented out: the condition never flips, and the program freezes.

Summary: `for` for known trip counts, `while` for conditions; every loop shows start, condition and step; trace on paper, then print-grade; `break` and `continue` for clean early exits.

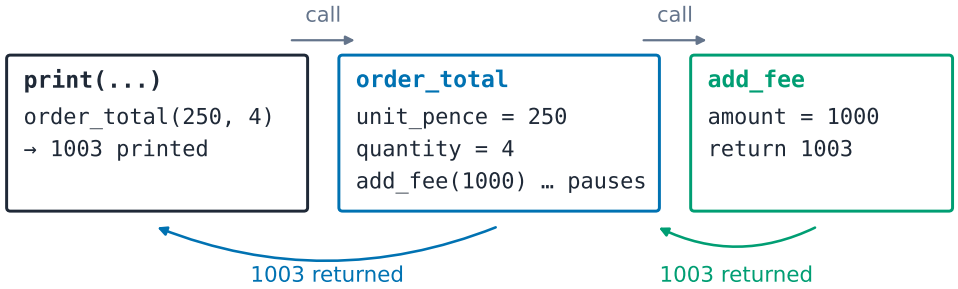
Functions: naming a job once

The same price calculation appears in five places; the tax rule changes; four of the five get updated. A **function** packages a job behind a name so it is written once and trusted everywhere: parameters in, a **return value** out. `def` defines; calling runs.

```
def total_price(count, unit=3.50):
    """Return the total for count items at unit price."""
    return count * unit

print(total_price(6))            # 21.0
print(total_price(6, unit=4.0))  # 24.0
```

The definition gives a **parameter** (`count`, `unit`); the call supplies an **argument** (6). A default value (`unit=3.50`) makes an argument optional. `return` hands a value back and ends the call; **printing is not returning** – a function without `return` returns `None`, and the caller that tries arithmetic on it gets a `TypeError`.



Nested calls: the caller pauses until the value comes back

Each call gets its own **local scope**: names inside are invisible outside, and disappear when the call ends. A call trace of nested calls reads: caller pauses → callee runs → return value lands in the caller.

Worked example 1 (the fee-inclusive order).

```
def add_fee(amount):
    """Return amount plus a fixed 3 pence fee."""
    return amount + 3

def order_total(unit_pence, quantity):
    """Return the order total: subtotal plus the fixed fee."""
    return add_fee(unit_pence * quantity)

print(order_total(250, 4))   # 1003
```


The contract in the docstring matters: whole pence in, whole pence out. Tracing `order_total(250, 4)`: subtotal `1000`, `add_fee` returns `1003` – and the frame diagram above shows the pause and the hand-back.

Worked example 2 (a real algorithm inside a small function, escalated). `is_prime(n)` returns `True` when nothing from 2 up to $\sqrt{n}$ divides `n`:

```
def is_prime(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:        # only test up to the square root
        if n % d == 0:
            return False     # found a divisor - answer known, stop
        d += 1
    return True

print([p for p in range(2, 30) if is_prime(p)])
# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Two function habits meet here: an early `return` ends the call the moment the answer is known (no flag variable needed), and the loop bound `d * d <= n` comes from the derivation – if $n = a \times b$ with $a \le b$, then $a \le \sqrt{n}$, so testing beyond $\sqrt{n}$ re-tests divisors already seen. Run it: the list is exactly the primes under 30.

Common mistakes. Printing inside a function instead of returning, so the result cannot be reused. Forgetting the `return` on one branch – that path quietly yields `None`. Reaching into a function's locals from outside; they are gone when the call ends.

Summary: `def` once, call anywhere; parameters in, one value out via `return`; early returns end the call; locals live and die with the call; docstrings state the contract.

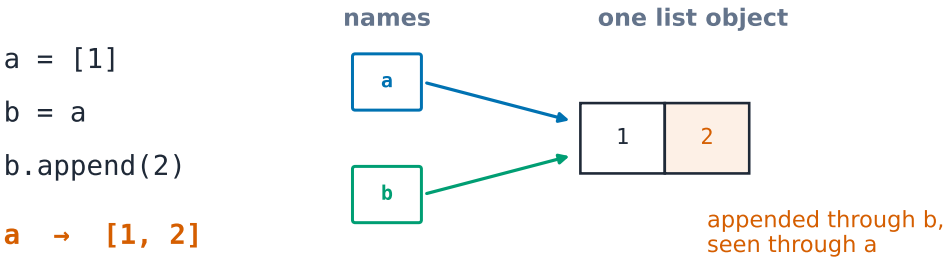
Collections: lists, dictionaries, tuples, sets

A café's till needs the day's orders in order, the menu by name, a location that must not drift, and today's unique drinks. Four jobs, four containers – choosing among them is the design decision of unit 1.5.

A **list** is an ordered, changeable sequence: `marks = [72, 85, 61]`, indexed from 0, sliced with `marks[1:3]`. A **dictionary** maps keys to values: `price = {"flat white": 3.5, "tea": 2.8}`, read and written by key. A **tuple** is a fixed sequence `(53.1, -1.5)` – good for coordinates that should not change. A **set**

holds unique items with no order: `{"a", "b"}`.

```
marks = [72, 85, 61]
marks.append(90)                # [72, 85, 61, 90]
best = max(marks)               # 90
average = sum(marks) / len(marks)   # 77.0
```



Aliasing: two names, one list

Assignment never copies: `a = [1]; b = a; b.append(2)` makes `a` `[1, 2]` – the list is shared, the names are not. This **aliasing** is unit 1.1's boxes again, now with two tags on one box, and it explains the next rule too: mutating a list while looping over it skips elements – loop over a copy (`for m in marks[:]`) or build a new list.

Worked example 1 (tally by category).

```
orders = ["tea", "flat white", "tea", "tea", "mocha"]
tally = {}
for item in orders:
    tally[item] = tally.get(item, 0) + 1
print(tally)   # {'tea': 3, 'flat white': 1, 'mocha': 1}
```

The dictionary's `.get(key, 0)` supplies 0 for a first sighting – the idiom that makes counting a one-liner. Verify by hand: "tea" appears three times in `orders`, and the printed tally says so.

Worked example 2 (the day's best seller, escalated). Building on the tally, report the top item and its count:

```
best = max(tally, key=tally.get)
print(best, tally[best])       # tea 3
```

The `key=tally.get` argument asks `max` to compare items by their counts rather than alphabetically – the same `key=` trick ranks any collection by any measure. The escalation matters: real programs chain these tools (count, then rank) rather than rescanning `orders` twice.

Choosing the collection is the design decision: order + duplicates → list; lookup by name → dictionary; fixed grouping → tuple; uniqueness → set.

Common mistakes. `tally[item] += 1` on a key that is not yet in the dictionary (`KeyError`) – `.get` is the fix. Expecting a set or dictionary to remember order. Assuming `b = a` copies a list; it aliases one.

Summary: four containers for four jobs; assignment shares, never copies; `.get` makes counting safe; `key=` makes ranking a one-liner.

Files and errors

The café closes, the power cuts – and the morning's tally survives only if the program wrote it to disk. A **file** outlives the program. `with open(path) as f:` opens, closes on exit, and reads text; `open(path, "w")` writes (and empties first – the classic accident).

```
with open("log.txt", "w") as f:
    f.write("opened\n")

with open("log.txt") as f:
    for line in f:
        print(line.rstrip())     # rstrip removes the trailing newline
```

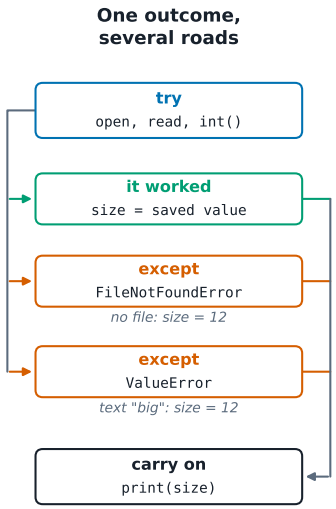
The `with` block is the safety rail: whatever happens inside it, the file closes. Reading gives lines *with* their newline, so `.rstrip()` belongs in every loop.

Errors are values Python raises as **exceptions**: `FileNotFoundError`, `ValueError`, `KeyError`, `TypeError`. `try/except` catches a named type – never a bare `except:` that hides real bugs.

Worked example 1 (the absent settings file). Reading a text size from a settings file that may be missing, or may not hold a whole number:

```
try:
    with open("settings.txt") as f:
        size = int(f.read().strip())
except FileNotFoundError:
    size = 12                    # no file yet: a sensible default
except ValueError:
    size = 12                    # the file holds "big", not a number
print("Text size", size)
```

Run it three times – with `16` in the file, with no file, and with `big` in the file – and it prints a usable size each time, from three different routes. That is the shape of robust code: one sensible outcome, several roads to it. Handle the error you expect, where you can respond to it; let the rest crash loudly.



Worked example 2 (a robust input loop, escalated). Ask until the user types a whole number:

```
def read_count():
    while True:
        try:
            return int(input("How many? "))
        except ValueError:
            print("Whole numbers only.")

print(read_count())
```

Feed it `x` then `4`: the first trip raises, is caught, prints the message, and the `while True` asks again; the second trip's `int(...)` succeeds and `return` exits the loop *and* the function in one move. Three units combine here – the loop (1.3), the function contract (1.4) and the exception (1.6) – which is the point of the escalation.

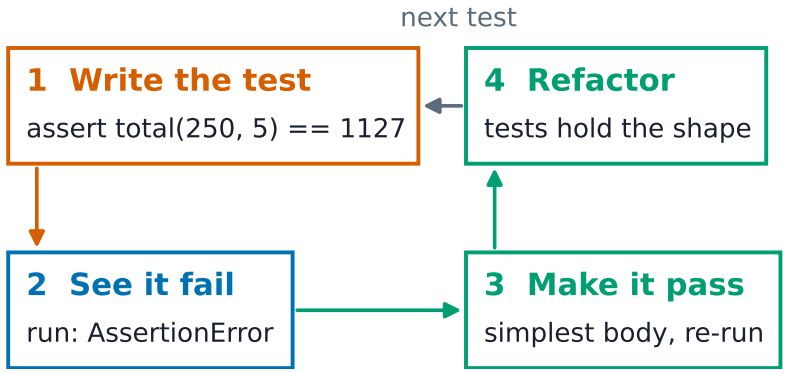
Common mistakes. `open(path, "w")` on the wrong path: the file is created or emptied before you notice. Catching `Exception` broadly and continuing with broken state. Forgetting `.rstrip()`, so output shows double-spaced lines.

Summary: `with open(...)` for both reading and writing; `"w"` empties first; exceptions are values – catch the named type you expect, fall back sensibly, let the rest crash.

Decomposition, testing and debugging

Two teams write the same app; one ships in a week, one drowns for a month. The difference is not typing speed – it is that the fast team's code is built from small pieces they can each prove correct while the slow team debugs a whole app at once. Unit 1.7 is that discipline.

Decomposition splits a problem into functions of one job each – each small enough to test alone. **Testing** compares a function's actual output with the expected, at the boundaries: zero, one, empty, negative, huge.



The test-driven loop: red, green, refactor

```
def add_fee(amount):
    return amount + 3

def order_total(unit_pence, quantity):
    return add_fee(unit_pence * quantity)

assert order_total(250, 4) == 1003
assert order_total(250, 0) == 3      # the zero-quantity boundary
assert order_total(0, 4) == 3
```

An `assert` that never fires is a passing test. Write the tests before you write the body when you can – they are the contract made executable, and the loop above (red → green → refactor) is that habit running continuously.

Worked example 1 (grow the tests with the function). Extend `order_total` with a 10% discount over £10 – the test is written first, at the new boundary:

```
def order_total(unit_pence, quantity):
    subtotal = add_fee(unit_pence * quantity)
    if subtotal >= 1000:              # pence: over £10
        subtotal = int(subtotal * 0.9)
    return subtotal

assert order_total(200, 4) == 803    # under £10 - unchanged
assert order_total(250, 4) == 902    # 1003 pence → 10% off → 902.7 → 902
assert order_total(250, 5) == 1127   # 1253 pence → 1127.7 → 1127
```

Check the middle test by hand: subtotal 1003, ten percent off gives 902.7, and `int(...)` rounds down to 902. Notice the first assert too: at 200 pence and four cups the total stays 803 – the boundary tests prove both that the discount fires and that it does not fire early.

> **Worked example 2 (bisect a bug with prints, escalated).** The tally from 1.5 prints `{'tea': 3}` but the best-seller line prints the wrong drink. The method: (1) reproduce with the smallest failing input; (2) print the values at the boundary between the trusted part and the suspected part – here, the tally before the `max`; (3) fix one thing; (4) re-run the tests. Moving the print halfway through the code each run halves the search: four prints find a bug in a sixteen-line program, however well it hides.

Debugging is finding the difference between what you believe the code does and what it does – the print at the trust/suspect boundary is the instrument.

Common mistakes. Testing only the happy path: the boundaries (zero, one, empty, negative) are where programs break. Writing a week of code before the first test, then debugging a mountain. Fixing two things at once, so neither result means anything.

Summary: decompose until every piece tests alone; assert the boundaries the minute a function exists; write the test first when you can; debug by bisection with prints at trust boundaries – the habit the whole degree builds on.

Topic 02

University Computer Science

What this course is, and how to use it

This is the second course in University Computer Science: **discrete mathematics** – the mathematics of objects you can list: statements, sets, graphs, counts. It is the proof toolkit behind algorithms, databases and every correct-by-construction claim in CS.

Scope limit, stated once: a study course, not a degree; every problem is original and self-contained, and every proof names its strategy.

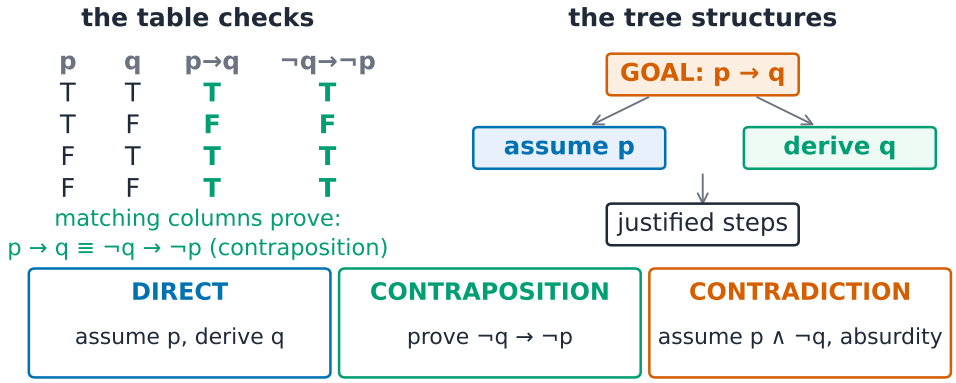
- Units 2.1–2.3 are the proof engine: logic, sets, induction.
- Units 2.4–2.6 are the combinatorial engine: counting, graphs, discrete probability.

The discrete habit: **name the strategy before the proof** – direct, contraposition, contradiction, induction – and every identity is something you can verify on a small example first.

Logic and proofs

Why it matters. Logic is the type system of proof: translate the claim, check the argument, and choose the proof shape before writing a line.

The idea, with a picture. Propositions connect with $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$; a **truth table** settles any formula over its variables ✓. The key equivalences: $p \rightarrow q \equiv \neg p \vee q$; contraposition $p \rightarrow q \equiv \neg q \rightarrow \neg p$; De Morgan $\neg(p \wedge q) \equiv \neg p \vee \neg q$ ✓. **Predicate logic** adds the quantifiers $\forall x\, P(x)$ and $\exists x\, P(x)$, with negation flipping each.



The table checks; the tree structures – a proof is a data structure of justified steps

The derivation (why three proof shapes). To prove $p \rightarrow q$: **direct proof**

(assume p, derive q); **contraposition** (prove $\neg q \to \neg p$ – useful when negatives are friendlier); **contradiction** (assume $p \wedge \neg q$, derive the impossible) ✓. Each shape converts the goal into an assumption you can compute with – choosing the shape is choosing your data.

> **Worked example 1 – the translation.** Claim: every email has exactly one address: $\forall e\, \exists! a\, \mathrm{Has}(e, a)$. Its negation pushes $\forall$ to $\exists$ and negates what follows: some email has no address, or has two or more ✓. One such email breaks the claim. Translate, negate, re-read: the three-step habit.

> **Worked example 2 – the strategy named.** Claim: if n^2 is even then n is even. Strategy: contraposition (the negative is friendlier). Proof: suppose n odd, so $n = 2k + 1$; then $n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$, odd ✓ – contrapositive proved, hence the claim.

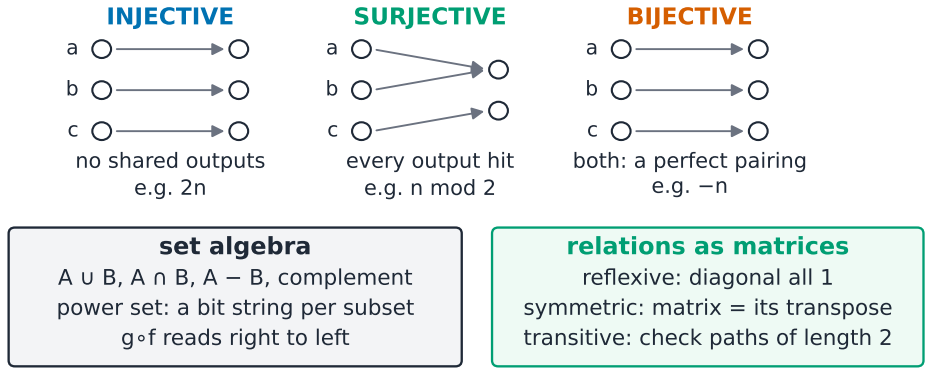
Common mistakes. Proving the converse of what was asked (direction matters: $p \to q$ is not $q \to p$). Distributing negation without flipping connectives (De Morgan flips $\wedge$ to $\vee$). Writing examples where a universal proof is demanded – an example is a check, not a proof.

Summary. Translate → choose shape (direct / contraposition / contradiction) → write justified steps; $p \to q \equiv \neg p \vee q$; De Morgan flips; negation swaps $\forall/\exists$; truth tables verify, proof trees structure.

Sets, functions and relations

Why it matters. Sets are the data model of discrete maths; functions are the typed pipes between them; relations carry the structure databases and graphs are built from.

The idea, with a picture. Operations: union, intersection, difference, complement; **power set** $\mathcal{P}(A)$ = all subsets, $|\mathcal{P}(A)| = 2^{|A|}$ ✓. Functions classified by arrows: **injective** (no two inputs share an output), **surjective** (every output is hit), **bijective** (both – a perfect pairing) ✓. Composition $g \circ f$ applies f then g; bijections invert.



Arrows classify the function; the matrix reads the relation's properties at a glance

The derivation (why bijections count). A bijection pairs every element of A with exactly one of B, so $|A| = |B|$ when one exists – and finding a bijection IS a counting proof ✓. That is why the number of subsets of an n-set pairs with bit strings of length n (each element in or out – one bit each), proving 2^n without listing ✓.

> **Worked example 1 – classify three arrow diagrams.** $f : \mathbb{Z} \to \mathbb{Z}$ by $f(n) = 2n$: injective (different inputs → different doubles) not surjective (odd numbers unhit) ✓; $g(n) = \lfloor n/2 \rfloor$: surjective not injective ✓; $h(n) = -n$: bijective – its own inverse ✓.

> **Worked example 2 – the relation test.** On $\{1, 2, 3\}$: $R = \{(1,1), (2,2), (3,3), (1,2), (2,1)\}$ – reflexive (all loops), symmetric (edges come in pairs) and transitive – the chain $(1,2), (2,1)$ demands $(1,1)$ and the chain $(2,1), (1,2)$ demands $(2,2)$, both present ✓. So R is an **equivalence relation** with classes $\{1,2\}$ and $\{3\}$ – check each property against the definition, not intuition.

Common mistakes. Confusing $\in$ (member) with $\subseteq$ (subset). Writing $f \circ g$ for $g \circ f$ (composition reads right-to-left). Testing symmetry by looking at one pair instead of all.

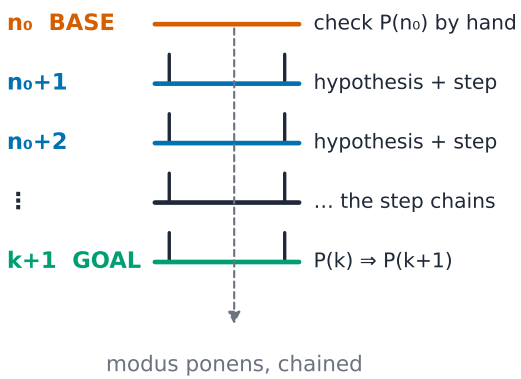
Summary. Union/intersection/difference/complement; $|\mathcal{P}(A)| = 2^{|A|}$ (bijection with bit strings); injective/surjective/bijective by arrows; composition right-to-left; relation properties – reflexive, symmetric, antisymmetric, transitive – tested against definitions.

Induction and recursion

Why it matters. Induction is the correctness proof of every loop and every recursive data structure; recursion is induction run backwards.

The idea, with a picture. To prove $\forall n \geq n_0\ P(n)$: **base case** ($P(n_0)$) + **inductive step** (assume $P(k)$ – the **induction hypothesis**

– derive $P(k+1)$) ✓. Recursion defines F by smaller calls: factorial $0! = 1$, $n! = n \cdot (n-1)!$ – the same ladder, built instead of climbed.



Σi = n(n+1)/2, proved
base: 1 = 1·2/2 ✓
step: assume Σ_1^k i = k(k+1)/2
then Σ_1^{k+1} i = k(k+1)/2 + (k+1)
= (k+1)(k+2)/2 ✓
the hypothesis is USED,
the algebra is SHOWN

the three faults
no base: a ladder on air
hypothesis never used
step needs k ≥ 2 (all horses)

Base plants the ladder; the step climbs one rung; every rung below is the hypothesis

The derivation (why induction is valid). The step proves that if rung k holds then rung $k+1$ holds; the base proves rung n_0; chaining gives every rung up to any n – induction is repeated modus ponens ✓. Fault detection: no base (a ladder on air), hypothesis not invoked (a step that proves nothing), or step from k assuming more than $P(k)$.

Worked example 1 – the sum identity. Claim: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Strategy: induction. Base $n = 1$: $1 = \frac{1 \cdot 2}{2}$ ✓. Step: assume $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$; then $\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}$ ✓ – the hypothesis is used, the algebra is shown, the claim closes.

Worked example 2 – the faulty proof caught. The claim that all horses are the same colour: base trivially true; the step – remove one horse from each side – fails at $k+1 = 2$ because the two overlapping sets share no horse ✓ – the step must hold for EVERY k, including the smallest; a step that silently needs $k \geq 2$ without base $P(2)$ is the classic fault.

Common mistakes. Omitting the base – hypothesis plus step prove nothing without a floor. Using $P(k+1)$ to prove $P(k+1)$ (circular). Strong induction declared but ordinary step written – say which and do which.

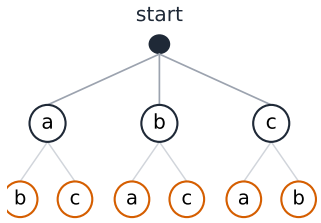
Summary. Base + hypothesis + step, hypothesis explicitly invoked; strong induction assumes all rungs below; recursion = induction's mirror (defined downwards); faults: missing base, unused hypothesis, step valid only for large k.

Counting and combinatorics

Why it matters. Counting is complexity analysis's arithmetic: sizes of state spaces, branch counts, the how-many behind every loop bound.

The idea, with a picture. Addition rule : disjoint choices add; **multiplication rule** : sequential choices multiply ✓. **Permutations** (ordered): $P(n,r) = \frac{n!}{(n-r)!}$; **combinations** (unordered): $\binom{n}{r} = \frac{n!}{r!\,(n-r)!}$ ✓. **Inclusion–exclusion** : $|A \cup B| = |A| + |B| - |A \cap B|$ – add, then subtract the double count.

the tree: ordered vs unordered



ordered pairs (ab ≠ ba): 3 × 2 = 6
unordered {a,b} = {b,a}: 6 ÷ 2! = 3

sequential choices MULTIPLY; disjoint alternatives ADD

the two formulas
P(n, r) = n!/(n−r)! ordered
C(n, r) = n!/(r!(n−r)!) unordered
same choices; combinations divide by the r! arrangements of each set

inclusion-exclusion
|A∪B∪C| = singles − pairs + triple
45+35+30 − 12−10−8 + 5 = 85
in none of the three: 15

The tree makes the rules visible: branches multiply, alternatives add, overcounts subtract

The derivation (why the formulas divide). Permutations choose and arrange: n options, then $n-1$, …, r factors – multiply. Combinations identify arrangements of the same r objects: every set of r was counted $r!$ times as an arrangement – divide by $r!$ ✓. The division is the entire difference between ordered and unordered.

Worked example 1 – the committee. A team of 3 from 8 people, with one designated leader: choose the team $\binom{8}{3} = 56$, pick the leader among them 3 ways → 168 ✓; or choose the leader first (8) then 2 members from 7 ($\binom{7}{2} = 21$) → 168 – two counting routes, one answer, and the agreement is the check.

Worked example 2 – inclusion–exclusion with three sets. Among 100 students: 45 like tea, 35 coffee, 30 cocoa; 12 tea∧coffee, 10 tea∧cocoa, 8 coffee∧cocoa; 5 all three. At least one: $45+35+30-12-10-8+5 = 85$ ✓ – none: $100-85 = 15$. Add singles, subtract pairs, add the triple: each element counted once.

Common mistakes. Adding where choices are sequential (routes × vehicles multiply). Using permutations where order is irrelevant (committees are combinations). Applying two-set inclusion–exclusion to three sets (the triple comes back with a plus).

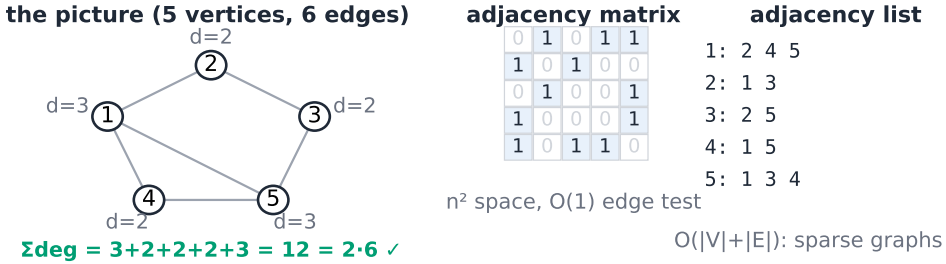
Summary. Disjoint → add; sequential → multiply; $P(n,r)$ ordered, $\binom{n}{r} = P/r!$ unordered; inclusion–exclusion: + singles − pairs

- triples; count two ways and compare – the checksum of combinatorics.

Graphs and trees

Why it matters. Graphs are the universal model – networks, states, dependencies; trees are the workhorse of search and storage; and the **handshake lemma** is the first must-hold identity of the course.

The idea, with a picture. $G = (V, E)$; representations: **adjacency matrix** (n^2 space, $O(1)$ edge test), **adjacency list** ($O(|V| + |E|)$) ✓. **Tree** : connected, acyclic, $|E| = |V| - 1$ ✓. Traversals: **BFS** (queue, level order, shortest paths in unweighted graphs), **DFS** (stack/recursion, component discovery).



Three representations of the same graph – matrix, list, picture – and the degree sum checked by handshake

The derivation (why handshaking must hold). Each edge contributes 1 to the degree of each endpoint, so every edge adds exactly 2 to the degree sum: $\sum_v \deg(v) = 2|E|$ ✓ – immediate corollaries: the number of odd-degree vertices is even; a graph where all $n \geq 2$ vertices have distinct degrees cannot exist (degrees run $0..n-1$ but 0 and $n-1$ cannot coexist) ✓.

> **Worked example 1 – the model choice.** (Design question) social friendships (undirected, adjacency list – sparse), task dependencies (directed acyclic, matrix when small – cycle tests), city metro (weighted, Dijkstra's queue – beyond scope, named as such) ✓ – the problem's symmetry dictates the graph, the density dictates the representation.

> **Worked example 2 – tree arithmetic.** A tree has 12 vertices of which 5 are leaves (degree 1): with $|E| = 11$ and handshake, the internal 7 vertices carry degree sum $22 - 5 = 17$ ✓ – every tree question is a walk over $|E| = |V| - 1$ plus $\sum \deg = 2|E|$.

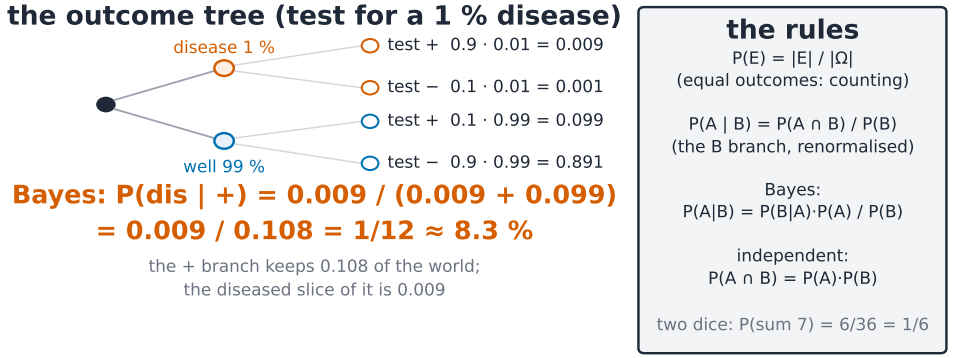
Common mistakes. Forgetting degree 0 counts (isolated vertices are vertices). Confusing connected (one piece) with complete (all edges). Assuming BFS gives shortest paths in weighted graphs (it does not – that needs weighted methods).

Summary. $G = (V, E)$; matrix vs list by density; tree: connected + acyclic + $|E| = |V| - 1$; BFS level order / DFS component order; $\sum \deg = 2|E|$, odd-degree count even; choose the model from the problem's symmetry.

Discrete probability

Why it matters. Randomised algorithms, average-case analysis and hashing all run on exact discrete probability – the same counting from 2.4, now over equally likely outcomes.

The idea, with a picture. A **sample space** Ω of equally likely outcomes; events are subsets; $P(E) = |E|/|\Omega|$ ✓. **Conditional probability** $P(A \mid B) = \frac{P(A \cap B)}{P(B)}$; **Bayes' theorem** $P(A \mid B) = \frac{P(B|A)P(A)}{P(B)}$ ✓; independence $P(A \cap B) = P(A)P(B)$.



The tree shows what the conditioning removes: $P(A/B)$ lives on the B-branch, renormalised

The derivation (why conditioning shrinks the world). Conditioning on B keeps only the B-outcomes and renormalises their weights to sum to 1 – the fraction of B-world where A also holds ✓. That is the entire numerator/denominator of Bayes: $P(B \mid A)P(A)$ sizes the $A \cap B$ slice; $P(B)$ rescales to the B-world.

> **Worked example 1 – dice by counting.** Two fair dice: Ω has 36 pairs. $P(\text{sum} = 7) = 6/36 = 1/6$ (six pairs); $P(\text{sum} = 11) = 2/36$ ✓; $P(\text{sum} = 7 \text{ or } 11) = 8/36 = 2/9$ – counting from 2.4 does all the work.

> **Worked example 2 – Bayes on a test.** A disease affects 1 %; a test has sensitivity 90 % and specificity 90 %. $P(\text{dis} \mid +) = \frac{0.9\times0.01}{0.9\times0.01+0.1\times0.99} = \frac{0.009}{0.108} = 1/12 \approx 8.3\ \%$ ✓ – the same arithmetic as MED12 §4, derived here from the tree: rare conditions stay improbable after imperfect evidence.

Common mistakes. Adding conditional probabilities as if independent (the conditioning world must be one world). Confusing $P(A \mid B)$ with $P(B \mid A)$ (the prosecutor's fallacy). Counting outcomes of different likelihood as equal (only equally likely outcomes give $|E|/|\Omega|$).

Summary. $P = |E|/|\Omega|$ on equally likely outcomes; $P(A \mid B) = P(A \cap B)/P(B)$ – the B-branch renormalised; Bayes rescales slices between worlds; independence multiplies; dice, cards and hash tables are all counting problems underneath.

Topic 03

University Computer Science

What this course is, and how to use it

This is the third course in University Computer Science: **computer systems fundamentals** – from a bit to a running program: how numbers live in wires, how gates decide, how a processor steps, and how the OS and the toolchain deliver your program to silicon.

Scope limit, stated once: a study course, not a degree; every program fragment is original and self-contained.

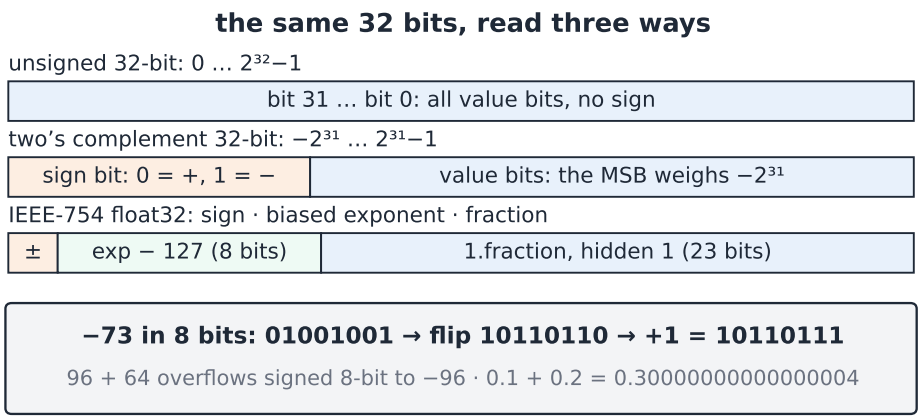
- Units 3.1–3.2 are the machine's alphabet: bits, encoding, gates.
- Units 3.3–3.4 are the machine itself: processor, memory, instructions.
- Units 3.5–3.6 are the machinery around it: OS services, the compilation pipeline.

The systems habit: **every abstraction is a contract with the layer below** – each unit names the contract, and every conversion is something you can verify by script.

Binary and data encoding

Why it matters. Everything below the source code is bits: knowing how numbers, fractions and text are encoded explains every surprise from overflowing counters to $0.1 + 0.2 \neq 0.3$.

The idea, with a picture. Base-2 positional notation; n-bit **unsigned** range 0 to $2^n - 1$; **two's complement** for signed: flip all bits, add 1 – range -2^{n-1} to $2^{n-1} - 1$ ✓. The carry out of the top bit is always **discarded**; **overflow** happens only when two operands of the same sign give a result of the other sign. **IEEE-754** IEEE-754 : sign, exponent (biased), fraction ✓; text: **ASCII** ASCII (7 bits, 128 codes) and **Unicode** (code points, UTF-8 variable width).



The same 32 bits read three ways: unsigned, two's complement, and an IEEE float – layout is the meaning

The derivation (why two's complement wins). Two's complement makes subtraction an addition: $a-b = a+(-b)$ with one adder. The proof is modular: $-b \equiv 2^n - b \pmod{2^n}$, and flip-plus-one computes exactly $2^n - b$ ✓. One circuit for + and −; that economy is why it became universal.

> **Worked example 1 – conversions and overflow.** 8-bit two's complement: −73? $73 = 01001001_2$; flip → 10110110; add 1 → 10110111 = −73 ✓. Add 96 + 64 = 160: unsigned fine (10100000_2), but signed 8-bit max is 127 – overflow, and the stored pattern reads −96 ✓.

> **Worked example 2 – float surprises.** 0.1 in binary is infinite (0.000110011 . . .); IEEE rounds – so 0.1 + 0.2 yields 0.30000000000000004: a **rounding** error, not a language bug ✓. Also: because the exponent scales the gap, adjacent floats are NOT evenly spaced – dense near zero, sparse far away.

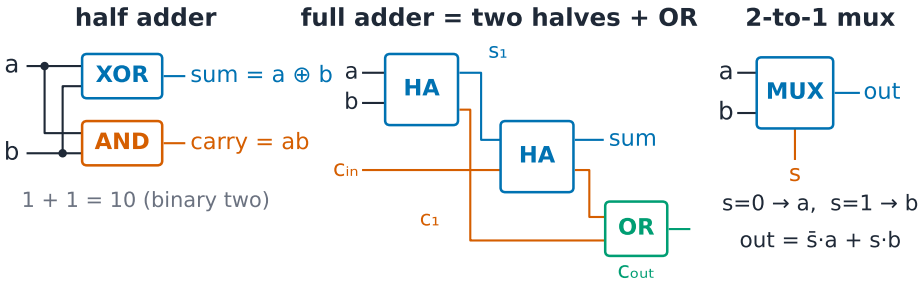
Common mistakes. Assuming a bit pattern has one value: 10110111 is 183 unsigned but −73 signed (the bits are the bits; the type reads them). Expecting float equality after arithmetic. Confusing character code with number (the character 1 stores the code 49, not 1).

Summary. Unsigned $0..2^n-1$; two's complement flip-and-add-1, one adder for + and −; discard the sign carry; IEEE: sign-exponent-fraction with rounding; ASCII 7-bit, Unicode code points, UTF-8 variable width; $0.1 + 0.2 \neq 0.3$ by rounding.

Logic gates and digital building blocks

Why it matters. Gates are the physics-to-logic boundary: every program is ANDs, ORs and NOTs in the end, and boolean algebra is the optimiser.

The idea, with a picture. AND, OR, NOT; combine into **NAND** (universal – every gate can be built from it ✓). Building blocks: the **half adder** (XOR = sum, AND = carry), **full adder** (two half adders + OR, carries chained into a ripple-carry adder), the **multiplexer** (select line picks one input) ✓.



XOR-and-AND make a half adder; two make a full adder; a mux picks a lane

The derivation (why boolean algebra optimises circuits). Identities ($x+x=x$, $x \cdot x = x$, $x + \bar{x} = 1$) let two expressions with one truth table collapse into one circuit – e.g. the mux expression $x = \bar{s}\,a + s\,b$ is already minimal, while a naive OR of the four rows where the output is 1 is not ✓. Fewer gates, less power, less delay.

> **Worked example 1 – one-bit addition.** Sum $s = a \oplus b$, carry $c = ab$: rows $1 + 0 \to s = 1, c = 0$; $1 + 1 \to s = 0, c = 1$ (binary 10) ✓ – two gates, correct arithmetic. Chain four for 0101 + 0011.

> **Worked example 2 – a 2-to-1 mux in gates.** Output $= \bar{s}a + sb$: when $s = 0$, a passes; when $s = 1$, b passes ✓. Then 4-to-1 = three 2-to-1 stages. A 32-to-1 is the same idea, five layers deep.

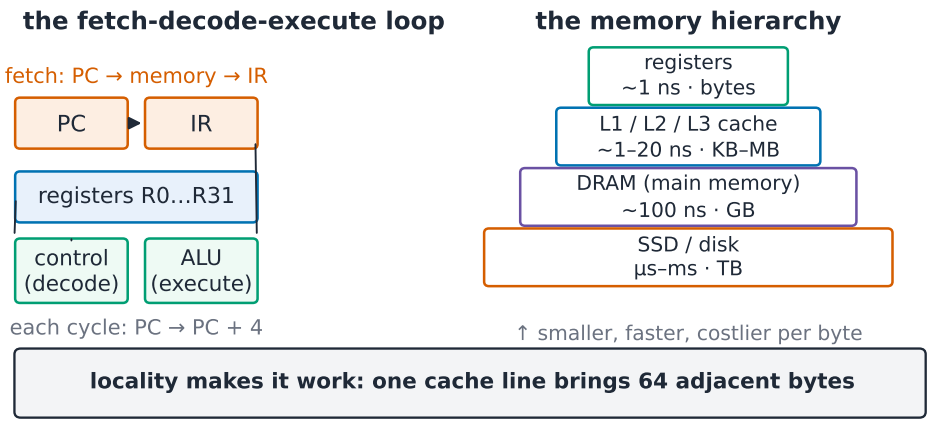
Common mistakes. Writing XOR as OR (the 1+1 row). Forgetting the carry chain's propagation delay (ripple adds gate-delays per bit). Believing gates are instantaneous (delay is what limits clock speed).

Summary. NAND universal; half adder XOR+AND; full adder chains; ripple-carry delay grows with width; mux $\bar{s}a + sb$; boolean identities minimise circuits.

Processor and memory overview

Why it matters. The CPU is a loop and the memory is a pyramid: both pictures explain speed, cost and why the **cache** decides performance more than the clock.

The idea, with a picture. Fetch-decode-execute cycle - - : fetch (PC → memory → IR), decode (instruction → control signals), execute (ALU / memory / PC update), PC → PC + 4 ✓. The memory hierarchy: registers (ns) → L1/L2/L3 cache → DRAM (∼100 ns) → SSD → disk – each layer bigger and slower ✓.



The fetch-decode-execute loop around the register file, with the memory pyramid alongside

The derivation (why the hierarchy exists). Speed vs cost vs size is a physical trade-off: fast memory is expensive per byte, big memory is slow. The hierarchy exploits **locality** – programs touch a small working set repeatedly (temporal) and nearby addresses (spatial) – so a small fast cache holds most references ✓.

> **Worked example 1 – tracing three instructions.** `LOAD R1, 0(R2)` / `ADD R3, R1, R1` / `STORE R3, 0(R2)`: each step registers change (R1, then R3, then memory) ✓; the PC advances by 4 after each instruction; a branch writes a new address into the PC.

> **Worked example 2 – why data layout matters.** An array sum is limited by memory, not arithmetic: reading one cache line brings 64 adjacent bytes, so row-major access is fast; a transposed loop is many times slower at the same operation count ✓. The hierarchy, not the clock, decides.

Common mistakes. Confusing IR (instruction register) and PC. Reading DRAM latency in cycles as if constant (cache hits and misses differ by orders of magnitude). Assuming the CPU's view of memory is the physical one (3.5's virtual memory inter-

poses).

Summary. Fetch → decode → execute, PC+4; registers-file-IR-ALU around the loop; hierarchy: registers-cache-DRAM-SSD; locality makes it work; data layout can dominate arithmetic.

Machine instructions

Why it matters. Assembly is the contract every compiler targets and every performance question bottoms out in: registers, immediates and memory moves.

The idea, with a picture. RISC instructions: `add rd, rs1, rs2` (registers only), `addi` (immediate), `lw`/`sw` (memory, addressing like `0(R2)`), `beq`/`bne` (compare and jump) ✓. **Addressing modes** : register, immediate, base+offset – memory only via load/store.

Each line: fetch, execute, the register delta – a loop unrolled into its machine steps

The derivation (why RISC is load/store only). One instruction shape decodes fast and pipelines well: every memory touch is an explicit `lw`/`sw`; arithmetic is register-to-register. CISC packs memory access into arithmetic – more compact, harder to pipeline ✓. The trade is instruction count against per-instruction speed.

> **Worked example 1 – a loop in machine steps.** `loop: addi R1, R1, -1; bne R1, R0, loop`: subtract, compare-with-zero, jump back – two instructions per iteration; a C `while` loop maps to exactly this pair plus the body ✓.

> **Worked example 2 – mapping C to instructions.** `a[i] = a[i] + 1` compiles to `lw` (load), `addi`, `sw` (store) with the index scaled into the offset ✓. Array indexing is address arithmetic; pointers are base registers; that is the whole secret.

Common mistakes. Reading `0(R2)` as the value 0 (it is the offset 0 on base R2). Forgetting to initialise the loop counter register. Using the index i as the byte offset

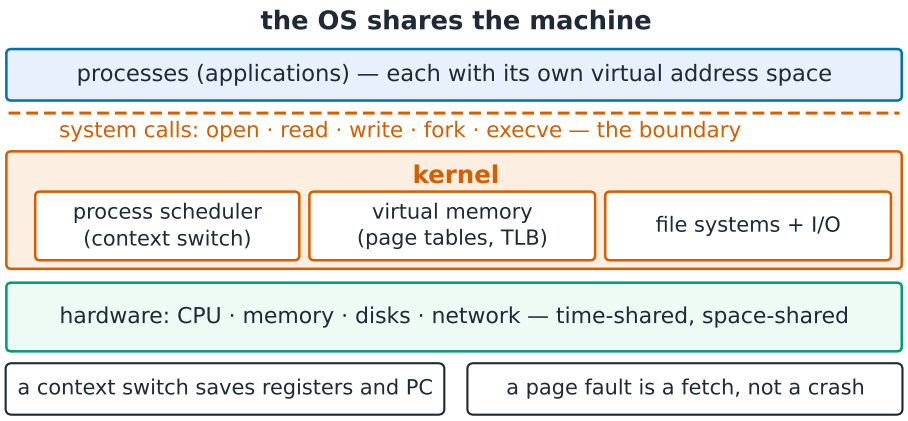
without scaling it by the element width ($4i$ for 4-byte words).

Summary. add/addi/lw/sw/beq; memory only via load/store base+offset; while → decrement-and-branch; array indexing is scaled address arithmetic; a[i]+1 is lw-addi-sw.

The operating-system interface

Why it matters. No program touches hardware: the OS shares the machine safely – processes, virtual memory and files are the services every program consumes.

The idea, with a picture. A **process** = program + execution state (address space, registers, open files); a **context switch** saves and restores that state ✓. **Virtual memory**: each process sees its own addresses; the OS + hardware (TLB, page tables) map virtual → physical. **Files** are named byte streams; **system calls** (`open`, `read`, `write`, `fork`, `execve`) are the boundary where your code asks the kernel ✓.



System calls cross the boundary; the kernel shares CPU, memory and devices among processes

The derivation (why virtual memory). Isolation (one process cannot read another's pages), illusion (more memory than physical, via paging to disk), and convenience (programs link to fixed addresses) – three needs one abstraction serves ✓. A **page fault** occurs when an access cannot proceed with the current page mapping or permissions. The OS may load or allocate a valid page; an invalid access may instead terminate the process.

> **Worked example 1 – what a context switch preserves.** Registers, PC, memory mappings, open-file table: everything the next instruction of that process needs; what does NOT carry: CPU caches (shared, just cold) ✓. The save area per process is exactly this list.

> **Worked example 2 – a system call in C.** `int fd = open(path, O_RDONLY); read(fd, buf, 4096);` – each call traps into the kernel, runs with kernel privileges, returns ✓. The same trio from Python: `open`, `read` – the interface is the contract, the language wraps it.

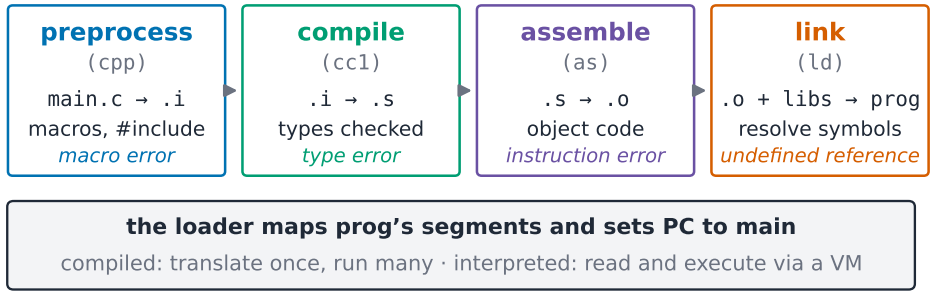
Common mistakes. Mixing process (own memory) and thread (shares memory). Believing malloc touches disk (memory, not files). Reading a page fault as always a crash or always a disk fetch (many valid faults are recoverable, while invalid accesses can fail).

Summary. Process = program + state; switch saves/loads it; virtual → physical via page tables; page faults require OS handling; system calls are the boundary; open/read/write/fork/execve.

Compilation and execution path

Why it matters. Source is not a program: the pipeline from `.c` to a running process has four stages, each with its own error class – and each stage's output is inspectable.

The idea, with a picture. **Preprocessor** (expand `#include` and `#define` – text only) → **compiler** (C → assembly) → **assembler** (assembly → object code, relocations unresolved) → **linker** (resolve symbols across objects + libraries → executable) ✓; the loader starts it as a process (3.5).



Preprocess, compile, assemble, link – each stage its own file format and its own error class

The derivation (why stages separate). Separate compilation: a million-line program recompiles one changed file; libraries distribute as objects/so-files; each error localises to one stage – macro errors, type errors, instruction errors, symbol errors ✓. The pipeline is an engineering economy, not a formality.

Worked example 1 – the four files. `main.c` → (cpp) `main.i` → (cc1) `main.s` → (as) `main.o` → (ld) `prog`; the extensions name the stages ✓. Any intermediate can be inspected with the right flag.

Worked example 2 – compiled vs interpreted. C: all four stages before running (fast, machine-specific binary). Python: source is read and executed by an interpreter (CPython compiles to bytecode, the VM runs it) ✓. Compile once-run-many vs read-every-run: the trade in one line.

Common mistakes. Calling every failure a compile error (undefined reference is the LINKER). Forgetting headers are text (macros have no type). Assuming optimisation cannot change float results (reassociation can).

Summary. cpp (text) → cc (types) → as (instructions) → ld (symbols); undefined reference = linker; compiled vs interpreted is when translation happens.

Topic 04

University Computer Science

What this course is, and how to use it

This is the fourth course in University Computer Science: **object-oriented programming** – how to grow a small script into a system that other people can safely change: objects that own their state, interfaces that hide their machinery, and tests that hold the line while you refactor.

Scope limit, stated once: a study course, not a degree; Python 3 is the vehicle, and every program fragment is original, self-contained, and was executed to produce the outputs shown.

- Units 4.1–4.2 build the object (classes, references, walls); 4.3–4.4 connect them; 4.5–4.6 keep the system alive.

The design habit: **every class is a contract with its callers** – each unit names the contract, and every example is a program you can run.

Objects and classes

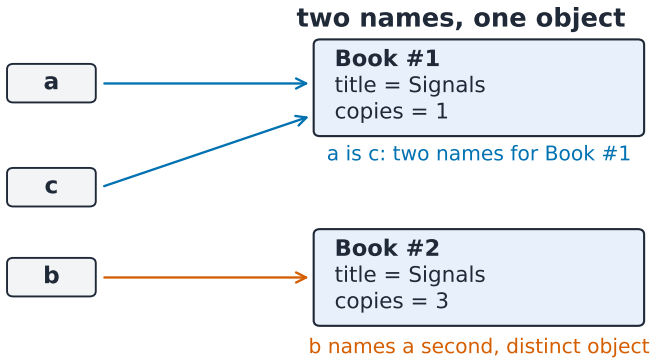
Why it matters. Programs fail when state lives everywhere: a **class** bundles state and behaviour so there is exactly one place to look. From here on, every system you build is classes talking.

The idea, with a picture. A class is a blueprint; an **instance** (an **object**) is one thing built from it. `__init__` runs at construction and seeds the **instance attributes**; methods take `self`, the instance itself. **Class attributes** live on the class and are shared by all instances. Assignment never copies: a variable is a **reference** to an object, so two names can be **aliases** of one object.

```
class Book:
    next_id = 1                # class attribute, shared

    def __init__(self, title, copies):
        self.id = Book.next_id; Book.next_id += 1
        self.title = title
        self.copies = copies   # instance attribute

    def take(self):
        self.copies -= 1; return self.copies
```



Two names on one object: a and c alias the same Book, b is a second, equal-but-distinct object

The lookup order (a derivation). `obj.x` resolves by walking outward: the instance dict first (`obj.__dict__`), then the class dict, then the base classes. That single rule explains the classic trap: assigning `obj.next_id = 9` does not touch the class – it shadows it for that one instance. `id` above lands in the instance dict; `next_id` stays in the class dict, so the books get ids 1, 2, 3, … and `Book.next_id` always holds the next free number.

Worked example 1 – one class, traced. With `Book` above:

```
a = Book('Signals', 3)
b = Book('Signals', 3)
c = a
a.take(); a.take()
```

State after: `a.copies` is 1, `b.copies` is 3 (untouched), and because `c` **is** `a`, `c.copies` is also 1 – one object, two names. `a == b` is `False` (default equality is identity) while `a is c` is `True`: **equality** compares values (once you define how), **identity** (`is`) compares objects.

> **Worked example 2 – the shared counter.** After `a` and `b` above, `Book.next_id`, `a.next_id` and `b.next_id` all read 3 – the counter belongs to the class, so every caller agrees. Write `a.next_id = 9` and only `a` reads 9: the class still says 3, and the next book still gets id 3.

Common mistakes. Forgetting `self` in a method header. Mutating a class attribute through an instance and wondering why others disagree. Expecting `d = c` to copy – it aliases.

Summary. Class = blueprint, instance = thing; instance dict shadows class dict; assignment aliases, never copies; `is` is identity, `==` is (defined) equality.

Encapsulation

Why it matters. Every class owns an **invariant** – a statement that is true between calls (a copy is either on the shelf or out to one borrower, nothing else). **Encapsulation** is the discipline of letting that statement stay true: state is private, changes pass through methods that check it.

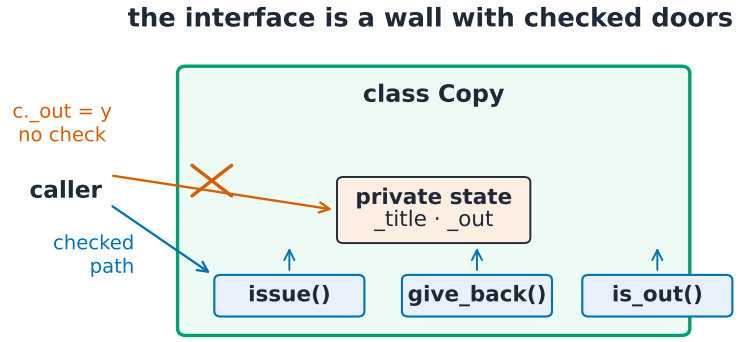
The idea, with a picture. By convention a leading underscore marks internal state (`_copies`); callers use the public methods – the **interface** boundary. Validate at the gate: the constructor and every mutating method re-establish the invariant or refuse.

```
class Copy:
    def __init__(self, title):
        if title == '': raise ValueError('empty title')
        self._title = title; self._out = False

    def issue(self):
        if self._out: raise ValueError('already out')
        self._out = True

    def give_back(self):
        if not self._out: raise ValueError('not out')
        self._out = False

    def is_out(self):        # public reader, no wall breach
        return self._out
```



Public methods form a wall: every state change passes the checks; the fields behind the wall are private

The derivation (why one gate is enough). If every path that touches `_out` goes through `issue`/`give_back`, the invariant (`_out` is exactly `True` or `False`) is provable from two small methods. Add one unchecked writer and the proof is gone – the number

of places to check grows with every caller, and so does the bug rate.

> **Worked example 1 – the wall holds.** `c.issue()` succeeds; `c.issue()` again raises `ValueError` – the caller cannot reach an impossible state, and after the failed call the object is unchanged (check-first, so no rollback is needed). `c.give_back()`; `c._out` is `False`, and `c.issue()` works again: exactly the states `True` / `False`, nothing else, forever.

> **Worked example 2 – the cost of a leak.** A teammate in a hurry writes `c._out = 'y'`. Nothing stops them – underscore is convention – and now `is_out()` answers `'y'`: truthy but not `True`. One caller (`if c.is_out():`) shelves the copy as out; another (`if c.is_out() is True:`) shelves it as available – two readers disagree about one object, a state no method could ever produce. Repair cost scales with every reader of `_out`; the validation cost was two `if` statements. That asymmetry is the whole argument.

Common mistakes. Returning the internal list (`return self._loans`) – callers mutate your private state; return a copy. Validating in the constructor but not in the setter. Writing get/set pairs for everything: expose behaviour (`available()`), not storage.

Summary. Underscore marks intent; methods validate at the gate; one checked path per change keeps invariants provable; expose behaviour, not fields.

Composition and inheritance

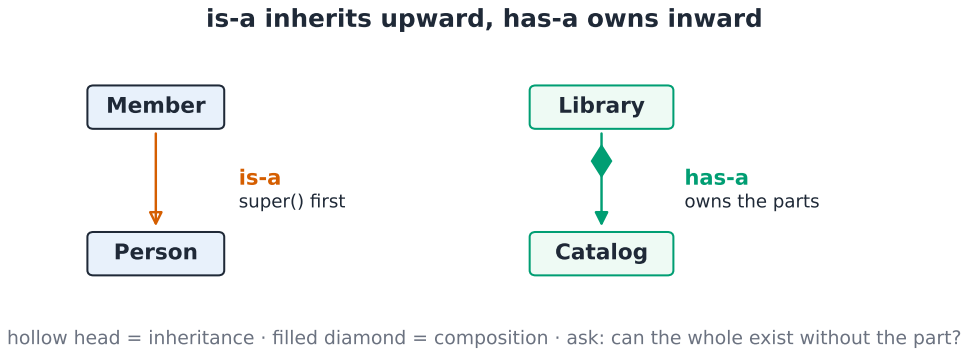
Why it matters. Objects connect two ways: **inheritance** (is-a: a `Member` is a `Person`) and **composition** (has-a: a `Library` has a `Catalog`). Choosing wrong couples classes so tightly that a base-class fix breaks its children – the **fragile base class** problem.

The idea, with a picture. Inheritance chains initialisation: `super().__init__(...)` walks up the chain, base first. Composition delegates: the whole object owns parts and forwards calls.

```
class Person:
    def __init__(self, name):
        self._name = name

class Member(Person):
    def __init__(self, name, card_no):
        super().__init__(name)      # base first
        self._card = card_no; self._loans = 0

    def borrow(self):
        self._loans += 1; return self._loans
```



Two connection kinds: inheritance says is-a (hollow triangle to the base); composition says has-a (filled diamond to the part)

The derivation (initialisation order). `Member('Ana', 7)` runs `Member.__init__`, whose first act is `super().__init__('Ana')` – so `Person.__init__` completes before any Member line executes. Order is always base-first down the chain; skip the `super()` call and `self._name` does not exist when `Member` needs it, failing with a clear `AttributeError` instead of corrupting quietly.

Worked example 1 – composition delegates. A `Library` has a `Catalog` (a list of `Copy`):

```
class Library:
    def __init__(self):
        self._shelves = []          # composition

    def add(self, copy): self._shelves.append(copy)

    def available(self):
        return sum(1 for c in self._shelves if not c.is_out())
```

After adding 5 copies and issuing 2, `available()` is 3 – the library owns no borrowing logic; it asks its part. Swap the list for a database listalike and `Library` is untouched: parts change, the whole keeps its interface.

> **Worked example 2 – the fragile base.** Base `Receipt.total()` sums items; a subclass overrides by copying the summing code and subtracting 10. The base then gains a loyalty line – every discount receipt silently loses it: the copy is a fork. Repair: `return super().total() - 10` – override by delegating. Inheritance for is-a, composition for everything else.

Common mistakes. Inheriting to reuse two methods (composition or a module function instead). Forgetting `super().__init__`. Duplicating base logic in an override – the fork that goes stale.

Summary. is-a inherits (and calls `super()`); has-a composes (and delegates); never copy base logic into an override.

Interfaces and polymorphism

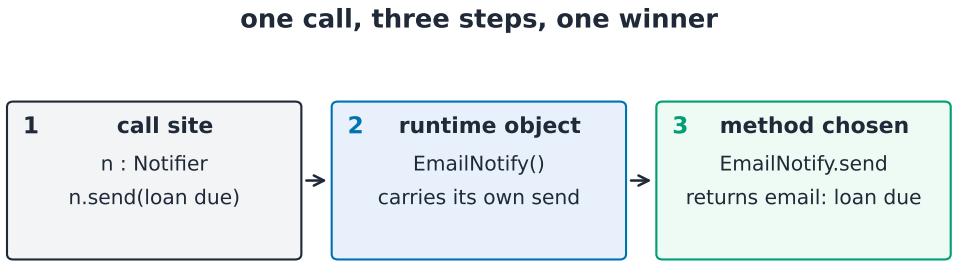
Why it matters. Big systems survive change when callers depend on what a thing **does**, not which class it is. An **interface** names the behaviour; **polymorphism** is one call site running many implementations.

The idea, with a picture. Python is dynamically typed: any object with the right methods fits – **duck typing**. To make the contract explicit, write a `Protocol` (or `abc.ABC`) naming the methods.

```
class Notifier(Protocol):
    def send(self, text: str) -> str: ...

class EmailNotify:
    def send(self, text): return f'email: {text}'

class SmsNotify:
    def send(self, text): return f'sms: {text}'
```



One call, three steps: the variable names the interface, the object carries the runtime type, dispatch picks that type's send

The derivation (dynamic dispatch). When `n.send('due')` runs, the attribute lookup of 4.1 resolves on the **runtime type** – the object itself, not the variable's annotation. So one loop body dispatches to a different `send` per element. That is **dynamic dispatch**: the call site is written once, the selection happens per object at call time.

Worked example 1 – one loop, three senders. With a third sender `QueueNotify` of the same shape and `team = [EmailNotify(), SmsNotify(), QueueNotify()]`:

```
for n in team:
    print(n.send('loan due'))
```

Three lines out – `email: loan due`, `sms: loan due`, `queue: loan due` – and adding a fourth sender touches no existing line: the caller was never told the concrete classes. This is the **open/closed** shape: open to new types, closed to edit.

> **Worked example 2 – substitution and its trap.** A substitute that honours the contract slots in invisibly. One that **narrows** it fails silently: a `SilentNotify.send` that raises on an empty string breaks a caller whose contract said *send never raises*. The rule of **substitution** (Liskov): preconditions may only widen, guarantees only grow.

Common mistakes. `isinstance` chains – replace with a method each class implements. An override that adds a precondition or drops a guarantee: the caller cannot tell, until it breaks.

Summary. Name behaviour with a Protocol; depend on the name; dispatch follows the runtime type; substitutes must honour the whole contract.

Exceptions and error handling

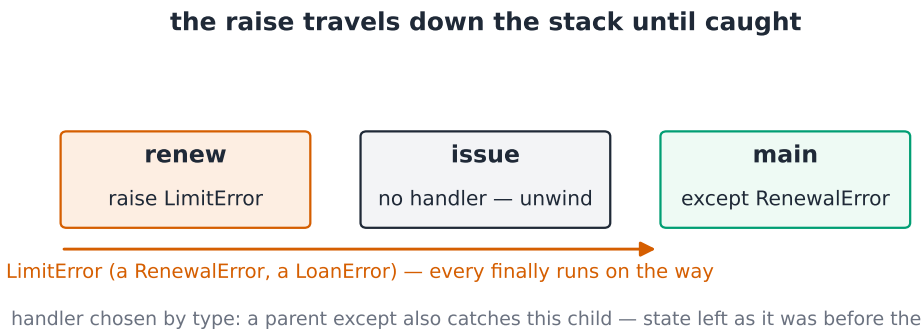
Why it matters. Errors travel. A function three frames down cannot fix a failure, but it must not ignore it. **Exceptions** carry an error to the frame that can decide, running each frame's cleanup on the way.

The idea, with a picture. `raise` throws; `try/except` catches by type; `finally` runs on **every** path. A **custom hierarchy** lets callers catch precisely: broad base, specific children.

```
class LoanError(Exception): ...
class RenewalError(LoanError): ...
class LimitError(RenewalError): ...

class Loan:
    def __init__(self, days):
        self.days = days

def renew(loan, extra):
    if loan.days + extra > 30:
        raise LimitError(f'{loan.days}+{extra} exceeds 30')
    loan.days += extra
    return loan.days
```



Raise in renew unwinds issue and main until a matching handler; finally runs on the way out, on every path

The derivation (status values vs exceptions). A return of −1 is a normal value: the compiler and runtime attach nothing to it, so a caller that forgets to check keeps computing with nonsense – the failure spreads silently. An exception, once raised, cannot be ignored: it unwinds frame by frame (running each `finally`) until some `except` matches; if none does, the program stops loudly. Ignorable errors get ignored; unignorable errors get handled. Choose **status values** for expected, caller-adjacent outcomes (a miss in a lookup), **exceptions** for broken contracts and failures the immediate caller cannot fix.

> **Worked example 1 – precise catch.** `renew(Loan(20), 15)` raises `LimitError` with message `20+15 exceeds 30`; the caller catches `RenewalError` (a parent) and the loan still reads 20: check-first means failed calls leave state untouched. Hierarchy is a filter – the caller picks the grain.

Worked example 2 – cleanup on both paths. A lock guard:

```
locked = True
loan = Loan(28)
try:
    total = renew(loan, 10)          # raises
except RenewalError:
    kept = 'handled'
finally:
    locked = False
```

Trace: `renew` raises; `except` runs (`kept = 'handled'`); `finally` runs; afterwards `locked` is `False` and the loan still reads 28 – released on the failing path. On the success path `finally` runs just the same. Cleanup that must happen goes in `finally` (or a `with` block), never after the last statement.

Common mistakes. `except:` bare – catches `KeyboardInterrupt` and hides real bugs. `except ... pass` – the failure vanishes and state is half-updated. Catching `Exception` when you meant one specific child.

Summary. Raise on broken contracts, catch at the frame that can decide, `finally` for cleanup; a small hierarchy buys precise handling; failed calls must leave state unchanged.

Testing and design quality

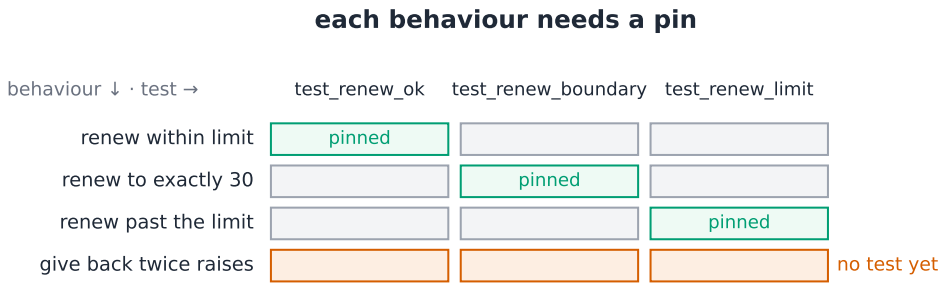
Why it matters. Refactoring – changing structure without changing behaviour – is only safe when behaviour is pinned. **Unit tests** are that pin, and the good ones test behaviour, not methods.

The idea, with a picture. One test = one behaviour: **arrange** the objects, **act** one call, **assert** the observable outcome. Behaviours, not methods: `renew past the limit raises` is a behaviour; `test_days` is a method.

```
def test_renew_ok():
    loan = Loan(20)                  # arrange
    out = renew(loan, 7)             # act
    assert out == 27                 # assert

def test_renew_boundary():
    assert renew(Loan(20), 10) == 30     # exactly 30 is allowed

def test_renew_limit():
    loan = Loan(20)
    try:
        renew(loan, 11); assert False
    except LimitError:
        assert loan.days == 20       # unchanged
```



Behaviours down, tests across: each cell is a behaviour a test pins; the empty cell is next to write

The derivation (why behaviour-first). A method-named test breaks on every rename and survives every behaviour change – noise without safety; a behaviour-named test does the opposite. Coverage follows the behaviour list: every branch of the contract is one row, and the grid shows the next hole.

> **Worked example 1 – three behaviours, three tests.** For `renew`: within the limit ($20 + 7 = 27$), exactly at the limit ($20 + 10 = 30$ is allowed, because the check is > 30), and past it ($20 + 11$ raises, state unchanged). The boundary test catches the classic slip of writing `>=` for `>`. Running them prints `3 passed`.

> **Worked example 2 – refactor under green.** The loan's fine calculation moves out of `Loan` into a `FineCalculator` class (**single responsibility principle** : one reason to change). The three tests are untouched, still `3 passed` – behaviour preserved while structure moved: **coupling** went down, **cohesion** went up.

Common mistakes. Asserting private internals – refactors break such tests. Shared mutable test state (test B passes only after test A ran). Chasing line coverage on getters while the limit branch has no test.

Summary. Arrange-act-assert per behaviour; tests pin behaviour so refactors are safe; SRP lowers the cost of change; low coupling and high cohesion are the direction of every move.

Topic 05

University Computer Science

What this course is, and how to use it

This is the fifth course in University Computer Science: **data structures and algorithms** – the containers a program reaches for and the recipes that walk them, chosen by what they cost.

Scope limit, stated once: a study course, not a degree; Python 3 is the vehicle, every fragment is original, and every cost claim in this handout was verified by counting operations on the exact datasets printed here.

- Units 5.1 is the yardstick: how cost grows.
- Units 5.2–5.5 are the containers: linear lists, hash tables, trees, graphs.
- Units 5.6–5.7 are the recipes: sorting and searching.

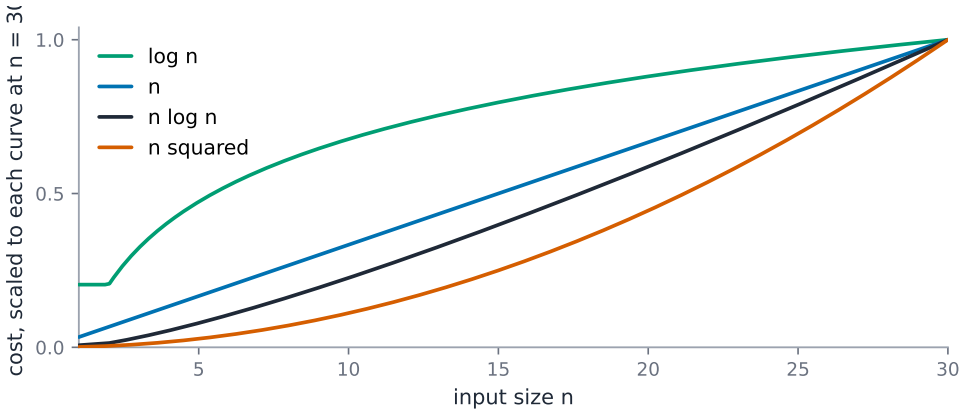
The measuring habit: **never trust a cost claim you have not counted** – each unit fixes a dataset, and the suite re-derives every number.

Asymptotic analysis

Why it matters. Choosing between two solutions is choosing between their growth rates: at $n = 1000$, the difference between $n \log n$ and n^2 is a factor of about 100.

The idea, with a picture. Big-O O is an upper bound up to a constant: $f(n) = O(g(n))$ when $f(n) \le c\, g(n)$ for large n. **Big-Ω** is the lower-bound twin, **Big-Θ** Θ the exact rate. Drop constants and lower-order terms: $3n^2 + 5n + 7$ is $\Theta(n^2)$ – the n^2 term eats the rest as n grows.

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
10	3	10	33	100	1 024
100	7	100	664	10 000	$\approx 10^{30}$
1000	10	1000	9 966	10^6	$\approx 10^{301}$



Growth of the common classes over one axis: logarithmic stays flat, exponential leaves the page

The derivation (why log n for halving). Each halving of the input leaves one step: an algorithm that halves n every step touches $\log_2 n$ levels – binary search on $n = 16$ makes at most 4 probes because $2^4 = 16$. And log of a million is about 20: flatness is why log-time algorithms feel instant.

> **Worked example 1 – counting beats guessing.** Two loops over the same array of $n = 100$ items: the single loop runs its body 100 times; the nested one $100 \times 100 = 10\,000$ times. Double n to 200: the linear body doubles to 200, the nested quadruples to 40 000 – the measured ratio $4 = 2^2$ is the exponent, read off the data.

> **Worked example 2 – reading code as a class.** A loop that halves `i = i // 2`: $\Theta(\log n)$ steps. Two sequential loops over n: the costs add, $\Theta(n) + \Theta(n) = \Theta(n)$. A loop inside a halving loop: they multiply, $\Theta(n \log n)$ – the shape of merge sort.

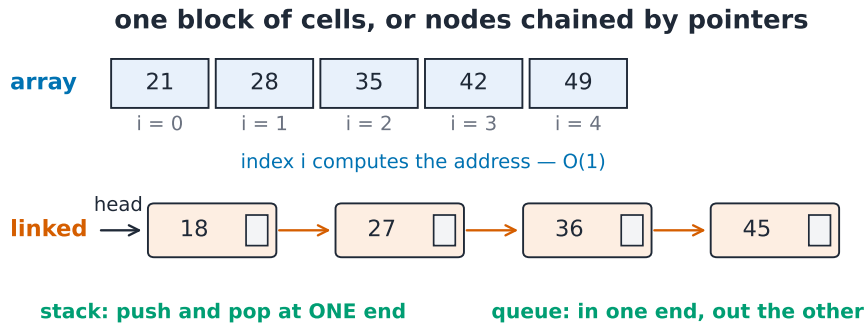
Common mistakes. Reading Big-O as a speed guarantee (constants matter at small n). Forgetting the input layout: $O(1)$ indexing is an array property, not a linked-list one. Reporting the best case where the worst case was asked.

Summary. O bounds above, Ω below, Θ pins the rate; drop constants and low-order terms; measure by doubling and read the exponent.

Lists, stacks and queues

Why it matters. The linear structures are the default containers, and their costs differ at the ends: knowing which end is cheap is the whole subject.

The idea, with a picture. A Python list is a contiguous **array** : index i computes an address, $O(1)$; inserting at the front moves everything, $O(n)$. A **linked list** trades the jump for cheap splices: each **node** carries a pointer to the next. A **stack** adds and removes at one end (LIFO); a **queue** adds at one end, removes at the other (FIFO).



One contiguous block with computed addresses versus nodes chained by pointers; the stack and queue differ only in which ends they use

The derivation (why append is amortized O(1)). The array grows by doubling: appending n items copies each element at most once in total across all regrows – total copies $\leq 2n$ for n appends, so the average append is constant even though one regrow copies n. Inserting at position 0 never amortises: every call slides all n residents, so n front-inserts move $\Theta(n^2)$ elements (measured: the operation count quadruples when n doubles).

> **Worked example 1 – a stack trace.** `push 3, push 7, push 1`, then `pop, pop`: the pops return 1 then 7 – last in, first out. The classic use is balanced brackets: scan `([]){}`, pushing opens; each closer must match the top – the string is balanced, while `([)]` fails because `)` finds `[` on top.

> **Worked example 2 – a queue trace.** Enqueue `a, b, c`; two dequeues return `a` then `b`; the front is `c`. First in, first out. Two stacks can fake a queue (inbox and outbox, each element handled twice), which is why `collections.deque` – a doubly-ended list – serves both ends in $O(1)$.

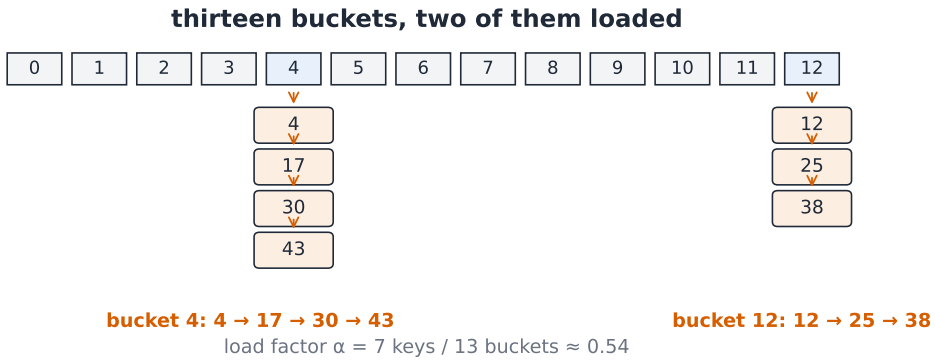
Common mistakes. Using `list.pop(0)` in a loop ($O(n)$ each; use `deque.popleft()`). Searching a linked list by index (walk the chain, $O(n)$). Assuming the Python list is linked – it is an array.

Summary. Array: $O(1)$ index, $O(n)$ front insert. Linked: $O(1)$ splice after a located node. Stack one end, queue two ends; deque does both ends in $O(1)$.

Hashing

Why it matters. A hash table is the structure that answers membership in one expected step – the engine inside sets, dictionaries and caches.

The idea, with a picture. A **hash function** maps a key to a bucket: $h(k) = k \bmod 13$ sends $4, 17, 30, 43$ all to bucket 4 – a **collision**. **Chaining** hangs a list off each bucket; **open addressing** walks to the next free slot (linear probing: $17 \to 5$, $30 \to 6$, $43 \to 7$).



Thirteen buckets under $h(k) = k \bmod 13$: keys 4, 17, 30, 43 chain in bucket 4 and 12, 25, 38 chain in bucket 12

The derivation (the load factor rules the chain). With m buckets and n keys, the **load factor** $\alpha = n/m$ is also the average chain length: our fixed keys $[4, 17, 30, 43, 12, 25, 38]$ give $\alpha = 7/13 \approx 0.54$, with bucket 4 holding 4 keys, bucket 12 holding 3, and eleven buckets empty. Successful search costs about $1+\alpha/2$ comparisons with chaining; grow m with n and the expected cost stays $O(1)$ – that resize is why dictionaries stay fast.

> **Worked example 1 – exact chains.** Insert $4, 17, 30, 43$ with $h(k) = k \bmod 13$: every key hashes to 4, so the chain reads $4 \to 17 \to 30 \to 43$. Searching 43 walks $4, 17, 30, 43$: four comparisons – the worst case a table this loaded can produce, and exactly what the uniform hash avoids on average.

> **Worked example 2 – probing the same set.** Open addressing with linear probing: 4 lands in slot 4; 17 probes 4 (taken) $\to$ 5; 30 probes $4, 5 \to 6$; 43 probes $4, 5, 6 \to 7$. Total probes: $1 + 2 + 3 + 4 = 10$ for four insertions – **clustering** grows the runs, which is why real tables keep α under about 0.75 and why the probing order matters.

Common mistakes. Modulo a power of two when keys share low bits (pick a prime). Mutating a key while it sits in the table (its hash moves). Expecting $O(1)$ worst case – that guarantee needs a perfect hash or a tree bucket.

Summary. Hash spreads keys; collisions are normal, so pick the resolution (chain or probe); $\alpha = n/m$ predicts the cost; resize to keep it constant.

Trees and heaps

Why it matters. Trees buy logarithmic costs by splitting the remaining work in half at each level – search, insert and (for heaps) min/max in $O(\log n)$.

The idea, with a picture. A **binary search tree** keeps the order invariant: left subtree < node < right subtree. Inserting $50, 30, 70, 20, 40, 60, 80$ builds a perfect tree of height 2; the inorder walk reads $20, 30, 40, 50, 60, 70, 80$ – sorted, by construction. A **heap** is a complete tree stored in an array: children of index i sit at $2i + 1$ and $2i + 2$; a **min-heap** keeps every parent $\leq$ its children, so the minimum lives at index 0.



Left: the BST built by inserting 50, 30, 70, 20, 40, 60, 80. Right: the min-heap array [1, 3, 8, 5] drawn as its implicit tree with index labels

The derivation (height is the cost). A perfect tree of height h holds $2^{h+1} - 1$ nodes: 7 at $h = 2$, 15 at $h = 3$, 31 at $h = 4$ – so n nodes need height $\lfloor \log_2 n \rfloor$ when balanced, and every root-to-leaf walk (search, insert, sift) pays that height. Skew breaks the deal: inserting sorted keys $1, 2, 3, 4$ makes a chain of height $3 = n - 1$ – $O(n)$ search, the reason balanced variants (AVL, red-black) rotate to stay logarithmic.

> **Worked example 1 – BST search and insert.** Search 40: compare 50 (right is smaller keys – go left), 30 (go right), 40 – found in 3 comparisons against 7 nodes: the level count. Insert 35: walk the same path, hang it left of 40; the inorder walk now reads $20, 30, 35, 40, 50, 60, 70, 80$ – still sorted.

Worked example 2 – heap sift-ups. Build a min-heap by inserting $5, 3, 8, 1$: 5 lands at index 0; 3 sifts up over 5; 8 stays; 1 sifts up over 5 then over 3, landing at index 0. The array reads $[1, 3, 8, 5]$ – parent $\leq$ child everywhere (check: $1 \leq 3, 8$; $3 \leq 5$). `heapq.heapify` builds the whole array in $O(n)$ by sifting down from the middle.

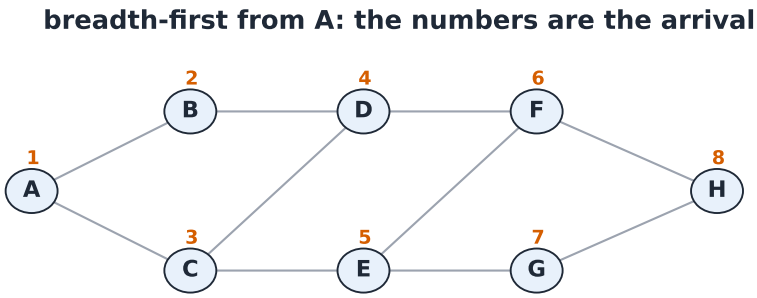
Common mistakes. Forgetting the heap is only half-ordered (an inorder walk of $[1, 3, 8, 5]$ is meaningless; only the root is special). Expecting BST balance from random luck on sorted input. Storing a tree of n nodes in $n + 1$ array slots and off-by-one-ing the child formula.

Summary. BST: order invariant, inorder = sorted, cost = height, log when balanced. Heap: complete shape in an array, parent dominates children, root is the min in $O(1)$, sift costs the height.

Graphs and traversal

Why it matters. Every network – roads, prerequisites, friend-of-friend – is a graph, and the two traversals are the probes that answer reachability, layers and structure.

The idea, with a picture. A graph $G = (V, E)$: our fixed course graph has $V = \{A \dots H\}$ (8 nodes) and 10 undirected edges: $AB, AC, BD, CD, CE, DF, EF, EG, FH, GH$. The **adjacency list** stores, per node, its neighbours (20 entries for 10 undirected edges); the **adjacency matrix** is an 8×8 grid of 64 cells, 1 at an edge.



The course graph: breadth-first from A visits in the numbered order A, B, C, D, E, F, G, H – layer by layer

The derivation (why O(V+E)). Breadth- and depth-first search mark each node once (V work) and scan each stored edge once (E work): $O(V + E) = O(8 + 10)$ visits on the course graph. The matrix pays $O(V^2)$ because every row walks all 8 columns, edge or not – the list wins on sparse graphs ($E \ll V^2$), the matrix wins when the graph is dense or the question is just one cell lookup.

Worked example 1 – BFS by layers. From A with neighbours visited alphabetically: layer 0 is A; layer 1 is B, C; layer 2 is D, E; layer 3 is F, G; layer 4 is H. The full order reads A, B, C, D, E, F, G, H, and the layer of a node is its shortest unweighted distance from A – that is what a queue buys: the frontier leaves in arrival order.

Worked example 2 – DFS by stack. From A, walking neighbours alphabetically: $A \to B \to D$; D's next unvisited neighbour is C; $C \to E$; $E \to F$; $F \to H$; H's unvisited neighbour G closes the walk. The order reads A, B, D, C, E, F, H, G – one deep walk that backtracks at dead ends. Same graph, same $O(V + E)$, different shape: BFS answers shortest hops, DFS exposes paths and cycles.

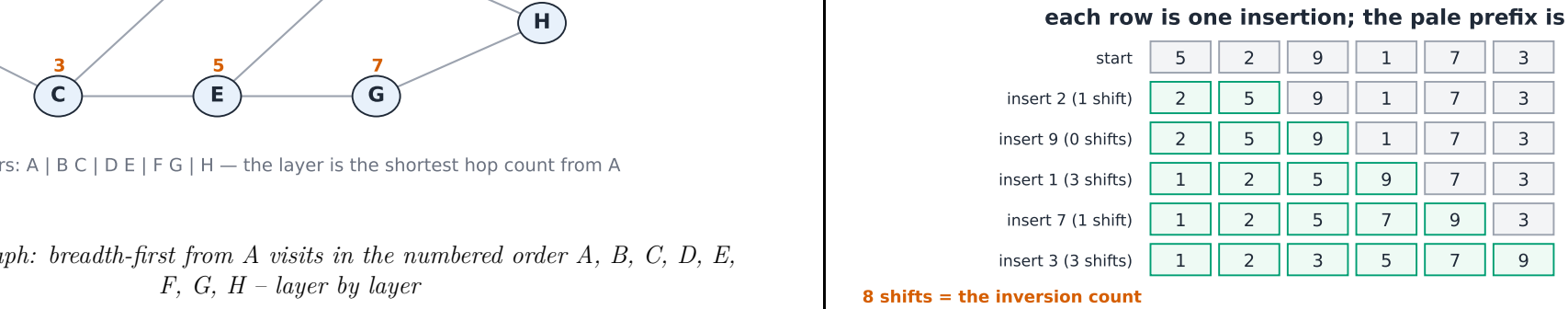
Common mistakes. Forgetting to mark visited (infinite loop on any cycle). Reading the BFS layer as weighted distance (unweighted only – weights need Dijkstra). Storing undirected edges once in a matrix (both cells $[u][v]$ and $[v][u]$ must be set).

Summary. List for sparse, matrix for dense; both traversals are $O(V + E)$; the queue gives layers and shortest hops, the stack gives deep paths.

Sorting

Why it matters. Sorting is the precondition for binary search, deduplication and merging – and the canonical lesson in cost analysis: same task, wildly different bills.

The idea, with a picture. The course dataset is $[5, 2, 9, 1, 7, 3]$. **Insertion sort** grows a sorted prefix, sliding each new card back: after the pass over 1 the prefix reads $1, 2, 5, 9$. **Merge sort** halves, sorts each half, merges. **Quicksort** partitions around a pivot. Bubble sort swaps adjacent inversions – educational, rarely used.



Insertion sort on the course dataset: each row is one insertion; the shaded prefix is already sorted

The derivation (the inversion count). An **inversion** is a pair out of order. On the course dataset the inversions are $(5,2),(5,1),(5,3),(2,1),(9,1),(9,7),(9,3),(7,3)$ – eight pairs. Every adjacent swap fixes exactly one inversion, so any swap-adjacent sort makes exactly 8 swaps here; insertion sort's shifts equal the inversions too. Merge sort counts the same inversions across its merges while moving each element at most $\log_2 6 \approx 2.6$ levels: $O(n \log n)$ beats $O(n^2)$ because it fixes many inversions per move.

> **Worked example 1 – insertion by hand.** $[5,2,9,1,7,3]$: insert 2 (one shift: $5,2 \to 2,5$); 9 appends (no shift); 1 shifts three ($[1,2,5,9]$); 7 passes 9 only (one shift, prefix $[1,2,5,7,9]$); 3 shifts three. Total shifts $1+0+3+1+3=8$ – exactly the inversion count – with 11 comparisons (each pass ends in one failed comparison), and the sorted result $[1,2,3,5,7,9]$.

> **Worked example 2 – merge sort on the same data.** Halve to $[5,2,9]$ and $[1,7,3]$; recursively $[2,5,9]$ and $[1,3,7]$; merge: $1,2,3,5,7,9$. The measured comparison count is 10 on this input, against insertion's 11 – close here, but the families diverge: at $n = 1000$, $n \log_2 n \approx 9\,966$ against $n^2 = 10^6$, a factor of about 100.

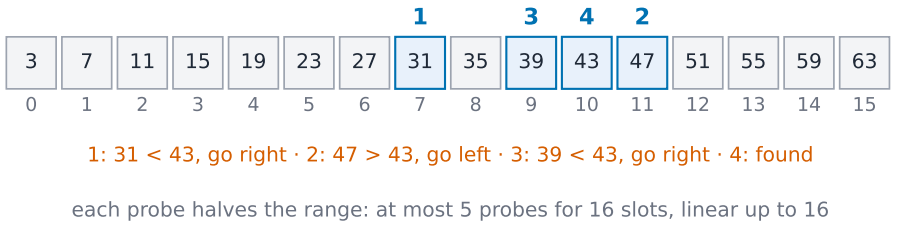
Common mistakes. Calling quicksort's worst case rare and forgetting sorted input with a first-element pivot IS the worst case ($O(n^2)$). Expecting heapsort to be stable (it is not; merge sort is). Counting only comparisons when moves dominate on large records.

Summary. Adjacent-swap sorts pay one move per inversion ($\Theta(n^2)$ worst); divide-and-conquer pays $\Theta(n \log n)$; stability survives in merge and insertion, not in quick or heap.

Searching

Why it matters. Membership and lookup are the most-run operations in real programs – the difference between 16 probes and 4 is the difference between linear and binary on a page of data.

The idea, with a picture. Linear search walks the array: $O(n)$, no preconditions. **Binary search** halves a sorted range: compare the middle, keep one half. On the sorted array of 16 items $[3,7,11,15,19,23,27,31,35,39,43,47,51,55,59,63]$ (indices 0–15), searching 43 probes indices $7, 11, 9, 10$ – value 31, then 47, then 39, then 43: four probes, found.



The 16-slot sorted array with the probe trail for 43: mid at 7 (too small), 11 (too big), 9 (too small), 10 – found in 4 probes

The derivation (the probe bound). Each probe halves the candidates: after k probes at most $\lceil n/2^k \rceil$ remain, so the worst case is $\lceil \log_2(n+1) \rceil$ probes – for $n = 16$ that is 5 (and our hit took 4). Linear pays 16 at the far end and 16 more for every miss beyond the end; binary pays the same ≤ 5 for every query. The precondition is the sort: on unsorted data the halving argument collapses, and only linear remains honest.

> **Worked example 1 – the probe trail.** Searching 43: $lo = 0$, $hi = 15$, $mid = 7$ reads $31 < 43$ so $lo = 8$; $mid = 11$ reads $47 > 43$ so $hi = 10$; $mid = 9$ reads $39 < 43$ so $lo = 10$; $mid = 10$ reads 43 – found at index 10 after 4 probes. Compare: linear walks 11 values to reach index 10.

> **Worked example 2 – the miss that proves the bound.** Search 64 (absent, above the end): binary probes $7(31), 11(47), 13(55), 14(59), 15(63)$ – five probes, then the empty range reports absent. Linear would walk all 16 and report the same absent – the miss costs binary its worst case and linear its full array, and the gap still grows with log versus n.

Common mistakes. Binary search on unsorted data (silently wrong answers, not crashes). Off-by-one in the update (`lo = mid` instead of `mid + 1` loops forever). Computing mid as $(lo+hi)/2$ in languages with fixed-width overflow (`lo + (hi - lo) / / 2` is safe).

Summary. Linear: no precondition, $O(n)$. Binary: sorted precondition, $\lceil \log_2(n+1) \rceil$ probes worst case – 16 slots resolve in at most 5.

Topic 06

University Computer Science

What this course is, and how to use it

This is the sixth course in University Computer Science: **computer architecture** – the machine below the program: how an instruction set is specified, how a datapath executes it, how a pipeline overlaps instructions, and how memory and I/O keep the core fed.

Scope limit, stated once: a study course, not a degree; the teaching machine is an original 16-register load-store design (R0 hardwired to zero), and every timing claim in this handout is computed from the stated parameters and re-derived by the course suite.

- Units 6.1–6.2 are the contract and the machine that honors it.
- Units 6.3–6.4 make it fast: pipelining and the memory hierarchy.
- Units 6.5–6.6 connect it: input/output and parallel speedup.

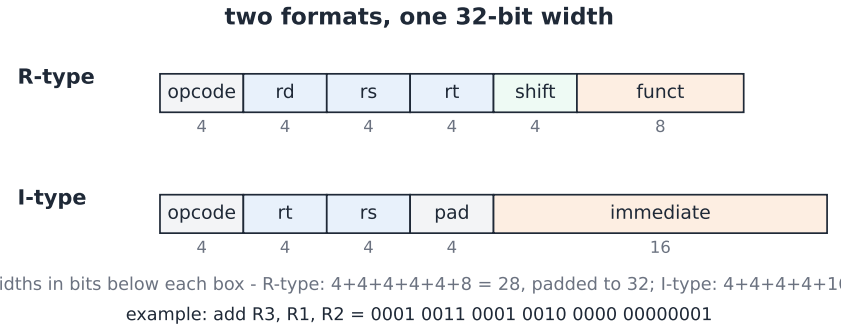
The architect's habit: **performance is arithmetic, not adjectives** – every speedup in this course is a number you can recompute.

Instruction set architecture

Why it matters. The **instruction set architecture** (ISA) is the contract between compiler and hardware: everything the software may rely on, nothing about how it is built. Two machines with one ISA run the same binaries.

The idea, with a picture. Our teaching ISA is **RISC** in spirit: fixed 32-bit instructions, **load-store** architecture (only `lw`/`sw` touch memory), 16 registers with R0 always zero. Two formats cover the course:

- **R-type** R : `opcode(4) | rd(4) | rs(4) | rt(4) | shift(4) | funct(8)` – e.g. `add rd, rs, rt`.
- **I-type** I : `opcode(4) | rt(4) | rs(4) | pad(4) | immediate(16)` – e.g. `lw rt, imm(rs)`; the pad field is unused, keeping the word 32 bits.



Two formats, one width: R-type carries three registers and a function code; I-type trades the third register for a 16-bit immediate

The derivation (the immediate's price). A 16-bit **two's complement** immediate spans -2^{15} to $2^{15}-1$, i.e. $-32\,768$ to $32\,767$ – so `addi R2, R0, 40000` cannot encode and the assembler demands two instructions (`lui`-style high half, then add). Word addressing: with **byte-addressable** memory and 32-bit words, `lw` offsets are multiples of 4 – a 16-bit offset reaches 2^{16} bytes = 64 KiB from the base register.

> **Worked example 1 – encode and decode.** Fields: opcode `add` = 1, `addi` = 2, `lw` = 3, `sw` = 4, `beq` = 5. Encode `add R3, R1, R2`: opcode 0001, rd 0011, rs 0001, rt 0010, shift 0000, funct 00000001 – six fields, 28 of 32 bits used, the top 4 bits of the final word zero-padded. Decode `0010 0100 0001 0000 000001111111` (opcode 2, rt = 4, rs = 1, pad = 0, immediate $01111111_2 = 127$): `addi R4, R1, 127`.

> **Worked example 2 – what R0 buys.** `addi R0, R0, 0` is the canonical no-op; `add R5, R0, R0` zeroes R5 in one cycle; comparing to zero (`beq R1, R0, loop`) needs no constant operand. A dedicated zero register removes a whole class of immediate-move instructions – CISC kept `MOV reg, 0`; RISC keeps a wire.

Common mistakes. Reading a 16-bit immediate as unsigned (branch and offset arithmetic is signed). Expecting `add` to touch memory (load-store: it cannot). Assuming opcode alone selects the operation (R-type also needs the funct field).

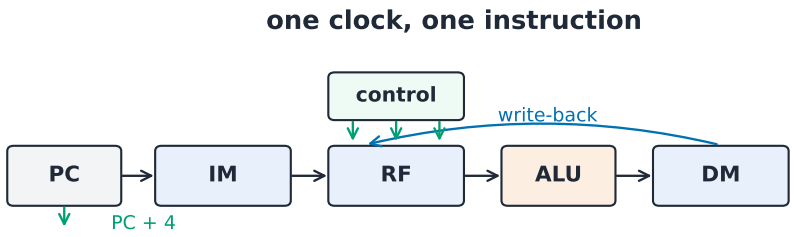
Summary. ISA = the stable contract; fixed-width R/I formats; 16-bit signed immediates reach ±32 K and 64 KiB of memory; R0 is a wire, not a register you write.

Datapath and control

Why it matters. The ISA is promises; the **datapath** is the plumbing that keeps them – and **control** is the switch settings per instruction.

The idea, with a picture. The single-cycle machine: PC feeds the instruction memory (**IM**); register file (**RF**) reads two source registers; the **ALU** computes; data memory (**DM**) reads or writes; the result writes back; PC + 4

or a branch target loads the PC. Control sets: **RegWrite**, **MemRead**, **MemWrite**, **MemToReg**, **ALUSrc**, **Branch**.



One clock, one instruction: fetch at PC, read registers, compute at the ALU, touch data memory if asked, write back, and choose the next PC

The derivation (cycle time = the slowest path). All work happens within one clock, so the clock must wait for the longest instruction: memory read + register read + ALU + memory again + write back – `lw`'s path. With our parameters (IM 200 ps, RF read 100 ps, ALU 200 ps, DM 250 ps, wiring 150 ps total; register writes overlap the cycle's trailing edge): the `lw` path = $200 + 100 + 200 + 250 + 150 = 900$ ps, so the cycle is 900 ps and a 100-instruction program takes 90 000 ps. Speeding up `add` would not help – the machine runs at its slowest citizen.

> **Worked example 1 – signals for one load.** `lw R4, 8(R1)`: RegWrite = 1 (write R4), MemRead = 1, MemToReg = 1 (data memory to register), ALUSrc = 1 (immediate for the address sum), Branch = 0, MemWrite = 0. Six bits tell the whole story – the same datapath executes every instruction; only the settings differ.

> **Worked example 2 – the store's exception.** `sw R5, 12(R1)`: the write port goes to memory, not the register file: RegWrite = 0, MemWrite = 1, MemRead = 0, MemToReg = x (don't care – nothing writes back). Don't-care settings are a designer's freedom: whatever the bit, the instruction behaves identically.

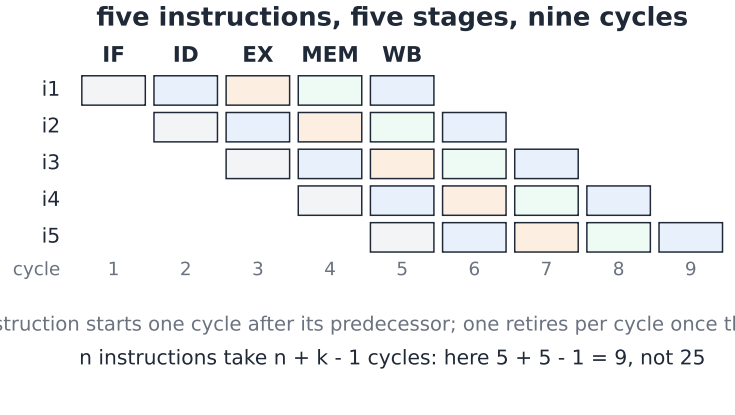
Common mistakes. Forgetting PC + 4 happens even on branches (the target is computed from PC + 4). Mixing MemToReg polarity between designs. Believing control is sequential – it is pure combinational logic on the opcode and funct.

Summary. One shared datapath, per-instruction control bits; the cycle time is the slowest instruction's path; `lw` usually sets it.

Pipelining

Why it matters. The single-cycle machine idles most of its parts most of the time. **Pipelining** overlaps instructions like an assembly line – more instructions finished per second, at almost the same hardware cost.

The idea, with a picture. Five stages: **IF** fetch, **ID** decode/register read, **EX** execute, **MEM** memory, **WB** write back. Instruction $i + 1$ fetches while i decodes: after the pipe fills, one instruction **retires** per cycle.



Five instructions through five stages: 9 cycles total, not 25 – the pipe fills once and drains once

The derivation (the speedup formula). n instructions, k stages, stage time t: unpipelined time = nkt; pipelined = $(n + k - 1)t$ (fill $k - 1$, then one per cycle). Five instructions, five stages: $25t$ becomes $9t$; at $n = 100$: $500t$ becomes $104t$ – the speedup approaches k as n grows, and the ideal **CPI** is 1. Our stage time is set by the slowest stage: with IF 250, ID 150, EX 200, MEM 250, WB 100 ps, every stage costs 250 ps – the pipeline runs at 250 ps per instruction against the single cycle's 900 ps.

Worked example 1 – a RAW hazard.

```
add R1, R2, R3     # R1 written in WB (cycle 6)
sub R4, R1, R1     # needs R1 in ID (cycle 5) - too early
```

The sub reads the register file one cycle before add writes it – a **read-after-write hazard**. **Forwarding** routes the ALU output of add straight back into the ALU inputs of sub, hiding the wait entirely – except when the producer is a **load**: `lw R1, …; add R3, R1, R2` must still stall one cycle, because the data exists only after MEM.

> **Worked example 2 – CPI with stalls.** A program runs 1000 instructions on a 5-stage pipe with forwarding; 25% are loads and half of those are followed immediately by a use (a load-use pair), each costing one stall: stalls $= 1000 \times 0.25 \times 0.5 = 125$. Total cycles $= 1000 + 125 + 4 = 1129$; CPI $= 1129/1000 = 1.129$ – the fill cost (4 cycles) fades as n grows.

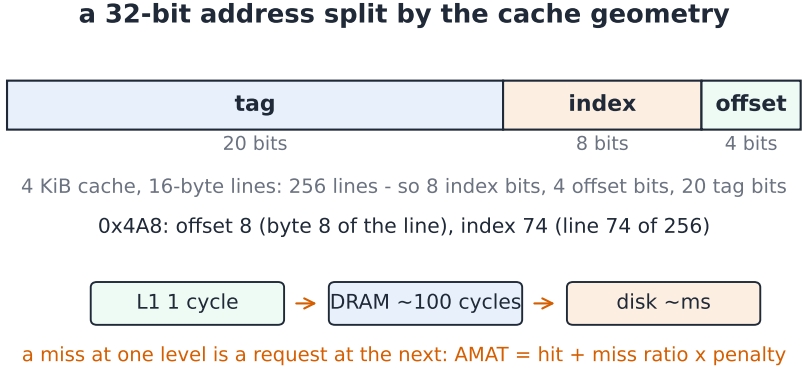
Common mistakes. Adding stage counts instead of using $n + k - 1$. Assuming forwarding fixes load-use (one bubble remains). Forgetting branches change PC late – a taken branch already fetched the wrong next instruction (our design: 1 bubble, resolved in EX).

Summary. $n + k - 1$ cycles for n instructions; ideal CPI 1; forwarding kills ALU hazards, load-use costs one stall, taken branches cost one bubble; speedup tends to the stage count.

Memory hierarchy

Why it matters. The core wants bytes in nanoseconds; cheap memory answers in hundreds. The **memory hierarchy** stacks small-fast over big-slow and lets **locality** pay the bills.

The idea, with a picture. A **direct-mapped cache** places each memory block in exactly one slot: a 32-bit address splits into **tag** | **index** | **offset**. Our course cache: 4 KiB, 16-byte lines ⇒ 256 lines ⇒ offset 4 bits, index 8 bits, tag $32 - 12 = 20$ bits.



The 32-bit address splits by the cache geometry; below it, the hierarchy each level feeds

The derivation (AMAT). Average memory access time $= t_{\text{hit}} + p_{\text{miss}} \times t_{\text{penalty}}$. With L1 hit 1 cycle, miss ratio 2%, penalty 100 cycles: AMAT $= 1 + 0.02 \times 100 = 3$ cycles – a 3× slowdown from a 2% miss rate, which is why the hierarchy lives or dies on locality. Halve the miss ratio to 1% and AMAT = 2: the sensitivity is linear in misses, not in hits.

> **Worked example 1 – where does an address go?** Address $0x000004A8$: the low 12 bits are 0100 1010 1000 – offset $1000_2 = 8$ (byte 8 of its line), index $0100\,1010_2 = 74$ (line 74 of 256), tag = the top 20 bits. One candidate line per address: any other block mapping to line 74 **evicts** it – direct-mapped's speed is one comparator, its price is conflict misses.

> **Worked example 2 – spatial and temporal locality.** Row-major scanning of a 64×64 int array with 16-byte lines (4 ints per line): the first access misses, the next three hit – miss rate $1/4 = 25\%$ plus compulsory misses only; stride-16 access (one int per line, column-wise on a wide matrix) misses every access – 100%. Same array, same cache, four-times the AMAT: layout is a performance feature.

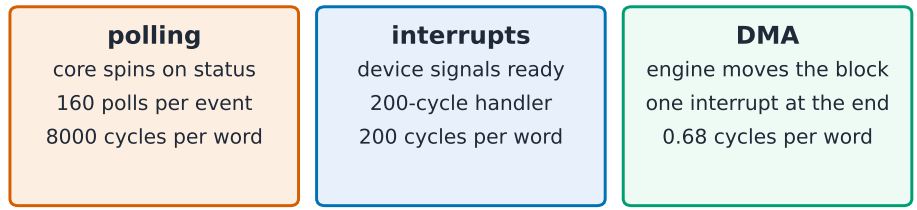
Common mistakes. Computing the index width from the cache size without dividing by the line size (it counts lines, not bytes). Expecting a bigger cache to fix a bad stride (bandwidth still moves one line at a time). Confusing miss **rate** with miss **ratio** (per-access versus per-instruction wording differs by the access count).

Summary. Split the address by geometry (tag | index | offset); AMAT $= 1 + m \times p$; direct-mapped trades conflicts for speed; locality – reuse and adjacency – is the currency.

Input/output

Why it matters. A machine that cannot talk is a space heater. I/O is where microseconds meet milliseconds, and the three transfer styles trade CPU attention for throughput.

The idea, with a picture. **Programmed I/O** (polling): the core spins on a status register. **Interrupt-driven I/O**: the device signals, the core saves state and runs a handler. **DMA**: a DMA engine moves the block and interrupts once at the end.



Polling burns the core; interrupts charge per event; DMA charges once per block

The derivation (the polling cost). A device flag updates every 2 µs; one poll costs 50 cycles at 250 ps per cycle = 12.5 ns. Worst-case latency demands continuous polling: 2 µs/12.5 ns = 160 polls per event, all but one wasted – the core spends 160 × 50 = 8 000 cycles per word of attention. Interrupts cut that to the **handler** cost: 200 cycles of save, read, restore per word. DMA moves 1024 words with one 500-cycle setup plus one 200-cycle completion interrupt: 0.68 cycles per word against 200 – three orders of magnitude of CPU attention saved.

> **Worked example 1 – choose the style.** A keyboard (a human types ≈ 5 characters per second): polling at that rate wastes the whole core; interrupts fit – one event per keypress, 200 cycles each, negligible. A 4 MiB disk block at 2 GB/s: even interrupt-per-word drowns the core; DMA owns it. A sensor sampled exactly every 2 µs: polling in a tight loop is honest and simple – the sample rate is the polling rate.

> **Worked example 2 – the interrupt bill.** Each interrupt costs 200 cycles of overhead (state save, vector, restore). Receiving 1024 words one interrupt per word: 204 800 cycles of pure overhead; the same block by DMA: 700 cycles total – the data moved identically, the CPU bill differs by 292×. Batching is not an optimization; it is the difference between architectures.

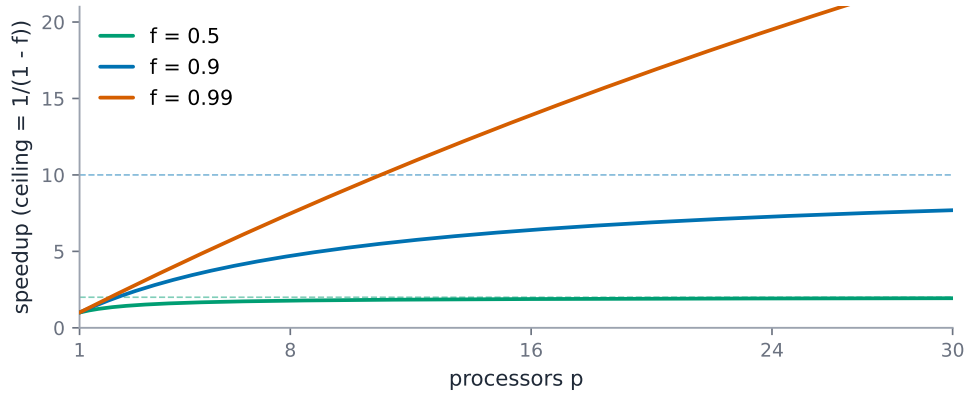
Common mistakes. Polling inside an interrupt handler (inverting the two styles). Forgetting DMA still needs cache coherence care (the DMA engine and the core both see memory). Confusing interrupt latency (time to start the handler) with throughput (words per second).

Summary. Polling: simple, core-hostile. Interrupts: event-priced, handler-overhead-per-word. DMA: block-priced, one interrupt per block – and the CPU attention per word falls by orders of magnitude.

Parallelism and performance

Why it matters. Clocks stopped getting faster; the machine got wider and more numerous instead. Reading a speedup now means reading Amdahl.

The idea, with a picture. A program has a serial fraction and a parallelizable fraction f. On p processors: Speedup $= \dfrac{1}{(1-f) + f/p}$ – the serial part never shrinks.



Speedup against processor count for parallel fractions 0.5, 0.9 and 0.99: the serial fraction caps every curve

The derivation (the ceiling). As $p \to \infty$, speedup $\to 1/(1-f)$ – the serial fraction alone. With $f = 0.9$: unbounded hardware buys at most 10×; at $p = 8$: $1/(0.1 + 0.1125) = 4.71\times$; at $p = 16$: $1/(0.1 + 0.0563) = 6.40\times$ – the second eight processors add 1.7× where the first eight added 4.7×. With $f = 0.99$ the ceiling is 100×, but reaching 50× needs $p = 99$: $1/(0.01 + 0.99/99) = 50.0$ exactly. Diminishing returns are structural, not incidental.

> **Worked example 1 – price a speedup.** A render spends 36 s compositing (serial) and 64 s shading (parallel) on one core: $f = 64/100 = 0.64$. On $p = 4$: speedup $= 1/(0.36 + 0.16) = 1.92\times$; runtime $= 100 \times 0.52 = 52.0$ s. Doubling the cores to 8: $1/(0.36 + 0.08) = 2.27\times$, 44.0 s – the compositing tail now dominates. The next purchase is an algorithm, not a machine.

> **Worked example 2 – instruction-level parallelism.** Before cores, the pipeline itself is parallelism: the 5-stage machine overlaps five instructions (**ILP**). Adding hardware threads (**SMT**) lets one core fill pipeline bubbles with another thread's instructions – the hazard math of 6.3 still applies per thread. The stack: ILP within a core, threads per core, cores per chip – three levels, one Amdahl discipline.

Common mistakes. Reporting peak instead of measured speedup (nominal p assumes $f = 1$). Treating a 10% serial fraction as harmless – it caps every machine you will ever buy at 10×. Comparing speedups across different baselines (the single-core time must be the same program, same machine, one core active).

Summary. Speedup $= 1/((1-f) + f/p)$, ceiling $1/(1-f)$; serial fractions dominate at scale; ILP, threads and cores are one ladder – and the ladder leans on Amdahl.

Topic 07

University Computer Science

What this course is, and how to use it

This is the seventh course in University Computer Science: **operating systems** – the layer that turns one processor into an illusion of many machines: processes, scheduling, synchronisation, virtual memory, files, and the walls between users.

Scope limit, stated once: a study course, not a degree; the teaching kernel is an original small design, and every scheduling table, interleaving and page-table walk in this handout is deterministic – computed by the course suite, never simulated by hand-waving.

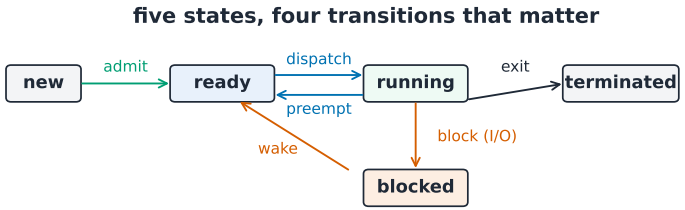
- Units 7.1–7.3 are the CPU's illusions: processes, turns, and safe sharing.
- Units 7.4–7.5 are memory and storage illusions: address spaces and files.
- Unit 7.6 is the wall: protection and security.

The OS habit: **an operating system is a resource accountant** – every mechanism in this course is a ledger of who owns what turn.

Processes and threads

Why it matters. A **process** is a program in motion: its own address space, registers, and open files. The OS's first illusion is running hundreds of them on a handful of cores.

The idea, with a picture. A process lives through five **states**: **new**, **ready**, **running**, **blocked** (waiting on I/O), **terminated**. The scheduler moves processes between ready and running; devices wake the blocked. A **thread** is a execution flow inside a process: threads share the address space and files, but each owns its stack and registers.



blocked waits on a device; ready waits only for the CPU — the PCB carries everything a suspend must save

Five states, four transitions that matter: dispatch and preempt between ready and running, block on I/O, wake on completion

The derivation (why the PCB exists). To suspend a process the OS must save everything that defines it – the **process control block** (PCB): register values, program counter, page-table pointer, open files. The **context switch** is exactly this save-and-restore: with our course timings, saving and restoring 2000 bytes of state at 2 GB/s costs 1 µs – plus cache pollution that makes the next runs slower. Switching threads skips the address space (shared), so it is cheaper; switching processes pays the full PCB.

> **Worked example 1 – a page fault's journey.** Process A runs, faults on a page: running → blocked (disk), the scheduler dispatches B, the disk completes, A → ready, and later A → running again with the faulting instruction restarted. Four state changes, two processes, zero lost work – the restartable instruction is the condition that makes preemption safe at all.

> **Worked example 2 – thread versus process.** A web server spawns a thread per connection: shared cache and configuration, 50 µs create-and-switch. The same as processes costs 500 µs (address space copy – well, with copy-on-write mostly the page-table setup) plus separate caches to fill. Sharing is the thread's whole point – and, as unit 7.3 shows, its whole danger.

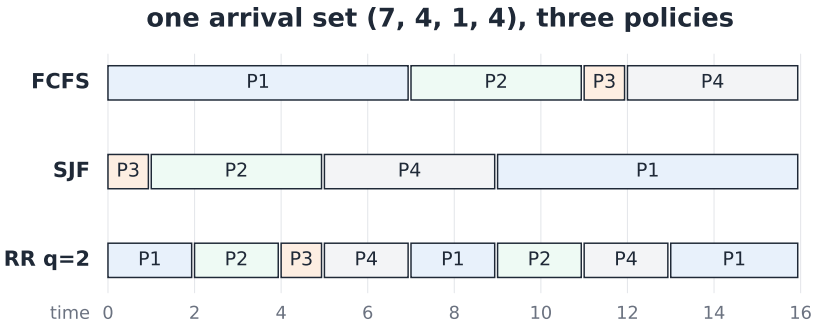
Common mistakes. Confusing blocked with ready (blocked waits on a device; ready waits only for the CPU). Believing a thread crash stays local (it takes the process – one address space). Counting the context-switch cost as pure CPU time (cache refill dominates).

Summary. Process = address space + PCB; thread = stack + registers inside it; five states with dispatch/preempt/block/wake; switching threads is cheaper because the space is shared.

Scheduling

Why it matters. More processes than cores means somebody decides the turns. The scheduler is that somebody, and its policy is a values statement: throughput, fairness, or latency – pick two.

The idea, with a picture. Our teaching queues: **FCFS** (first-come-first-served), **SJF** (shortest job first), **round robin** (RR, quantum $q = 2$), priority with aging. The **Gantt chart** shows who ran when.



average turnaround: FCFS 11.5 · SJF 7.75 · RR 11.25 — the average is a policy choice

One arrival set, three policies: SJF finishes the short jobs first, RR slices everyone in turn

The derivation (the turnaround formula). **Turnaround time** = finish − arrival; **waiting time** = turnaround − burst. For the course set – arrivals at 0 with bursts P1 = 7, P2 = 4, P3 = 1, P4 = 4 – FCFS finishes at $7, 11, 12, 16$: turnarounds $7, 11, 12, 16$ (all arrive at 0), average **11.5**. SJF order P3, P2, P4, P1 finishes at $1, 5, 9, 16$: turnarounds $1, 5, 9, 16$, average **7.75**. RR ($q = 2$) interleaves: the trace is P1 P1 | P2 P2 | P3 | P4 P4 | P1 P1 | P2 P2 | P4 P4 | P1 P1 | P1 – finishes P3 = 5, P2 = 11, P4 = 13, P1 = 16: average turnaround **11.25**. Same jobs, three policies, three answers – SJF wins the average, RR wins the worst-case wait.

> **Worked example 1 – convoy effect.** Under FCFS a long job (P1, 7) arrives first and everything queues behind it: P3 waits 11 units for 1 unit of work. SJF would run P3 first (wait 0) – the convoy is FCFS's signature failure with interactive users in the mix.

> **Worked example 2 – quantum choice.** RR with $q = 2$: 9 run segments, so 8 preemptions in our trace; with $q = 7$: P1 finishes in one slice – fewer switches, but short P3 waits 6+ before its first run. The rule: q large degenerates to FCFS; q tiny spends the CPU on switching (our course bound: switch cost 1 µs in a 2 ms quantum wastes 0.05%; at 20 µs quantum, 5%).

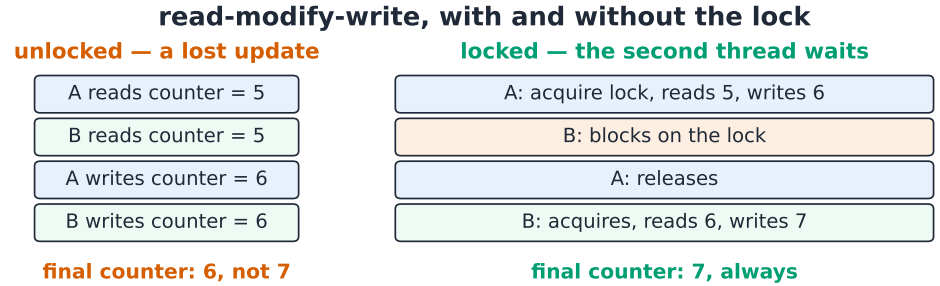
Common mistakes. Averaging completion times instead of turnarounds. Forgetting SJF needs burst foreknowledge (it is theoretical; MLFQ approximates it). Calling a small quantum faster (switches are work, not progress).

Summary. Turnaround = finish − arrival; SJF minimises average turnaround among non-preemptive policies; RR trades average for fairness and latency; the quantum prices the trade.

Synchronisation

Why it matters. Threads share memory, and unsynchronised sharing corrupts it: the lost update is the smallest and most expensive bug in computing.

The idea, with a picture. The classic race: two threads run `counter = counter + 1` on `counter = 5`. Each reads, adds, writes; the **interleaving** R1 R2 W1 W2 leaves 6, not 7 – one update is lost. The fix is **mutual exclusion** : a **lock** makes read-modify-write atomic.



four increments each, unlocked: any final value from 2 to 8; the lock pins it at 8

deadlock needs all four: mutual exclusion, hold and wait, no preemption, a cycle

Unlocked: the interleaving reads 5 twice and writes 6 twice – one increment vanishes; locked: the second thread waits

The derivation (the critical section bound). A correct synchronisation primitive for N threads must satisfy: mutual exclusion (one inside), progress (someone enters if free), bounded waiting (no thread starves). Peterson's algorithm meets all three with two flags and a turn variable; hardware `test-and-set` meets them with one atomic instruction – but both **spin** , burning the CPU. Semaphores and mutexes add the OS queue: the blocked waiter sleeps instead, which is why real systems layer them.

> **Worked example 1 – counting deterministically.** Two threads increment a shared counter 4 times each, unlocked. The worst case: A reads 0 and stalls; B runs three full increments; A writes 1, wiping them out; B reads that 1 and stalls; A finishes its last three increments; B writes 2. The counter ends at **2**: six of the eight increments vanish. A thread's next read always follows its own last write, so 1 is impossible. The suite enumerates the interleavings with per-increment registers: every final value from 2 to 8 is reachable – the race does not corrupt loudly, it undercounts quietly. Locked: always 8.

> **Worked example 2 – deadlock's four horsemen.** Two threads, locks L1, L2: A holds L1 wants L2; B holds L2 wants L1. Deadlock needs mutual exclusion, hold-and-wait, no preemption, and circular wait – break any one and it cannot happen. The engineer's favourite to break: circular wait, by ordering the locks (always L1 before L2).

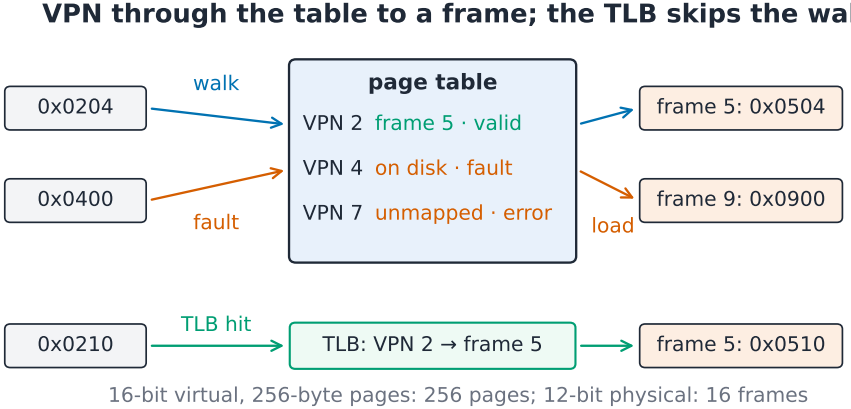
Common mistakes. Fixing races by inserting sleeps (narrows the window, closes nothing). Taking locks in different orders across threads (the deadlock recipe). Believing a single-core machine is safe (preemption between the read and the write still races).

Summary. Races come from non-atomic read-modify-write; locks make them atomic at the price of waiting; correctness is exclusion + progress + bounded waiting; break one of the four deadlock conditions – ordering locks is the cheap one.

Virtual memory

Why it matters. Every process believing it owns the whole address space – while sharing one physical RAM – is the OS's grandest illusion and the source of both its speed and its worst failures.

The idea, with a picture. The **virtual address** splits into page number and offset; the **page table** maps virtual pages to **physical frames**, one entry per page; the **TLB** caches translations. Our course machine: virtual 16 bits, page 256 bytes ⇒ 256 virtual pages; physical 12 bits ⇒ 16 frames.



Virtual page numbers through the page table to frames: VPN 2 to frame 5, VPN 4 to frame 9, VPN 2 again hits the TLB

The derivation (the size of the illusion). Address width sets the space: 16-bit virtual, 256-byte pages ⇒ VPN = 8 bits ⇒ 256 pages; a process may address 64 KiB while physical frames number 16 (4 KiB). The page table is the ledger: 256 entries × (frame 4 bits + valid 1 + dirty 1) = 1536 bits. Two processes each get all 256 virtual pages – overcommit is free until someone touches an unmapped page.

> **Worked example 1 – the walk, then the hit.** Address $0x0204$: offset $0x04$ (low 8 bits), VPN $0x02$. First access: TLB miss, page-table read: VPN 2 → frame 5, physical = $5 \times 256 + 4 = 0x0504$; the TLB caches VPN 2. Address $0x0210$ next cycle: same VPN 2 – TLB hit, physical $0x0510$, no table walk. The 10-entry TLB on our 256-page space: an 8-page loop of 8000 accesses misses 8 times (compulsory) – a 99.9% hit rate (suite-simulated, LRU).

> **Worked example 2 – the fault and its price.** Address $0x0400$: VPN 4, table entry invalid → **page fault**. The process blocks, the OS finds a victim (LRU: frame 9's page, dirty – write it back), reads VPN 4 into frame 9, updates the table and TLB, restarts the instruction. Cost model: TLB hit 1 ns, table walk 10 ns, fault + disk 5 ms – one fault per million accesses still costs 5 ns per access on average ($5\,\text{ms}/10^6$): the hierarchy tolerates rarity, not frequency.

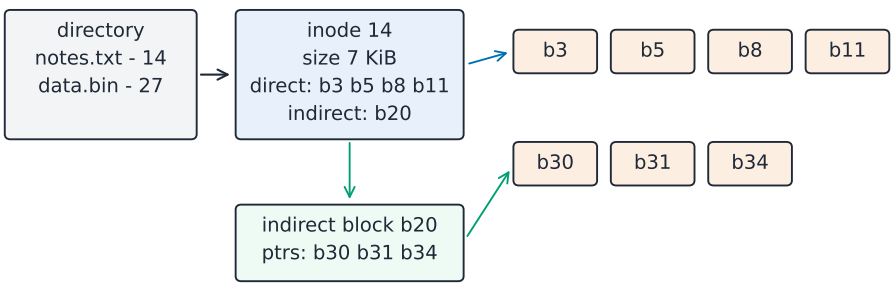
Common mistakes. Computing the physical address without the frame multiply. Forgetting the dirty bit doubles a victim's cost. Treating every page fault as either a disk fetch or an invalid access: a valid nonresident page may be loaded, a new page may be allocated, and a protection violation may terminate the process.

Summary. VPN + offset through the page table to frame + offset; the TLB makes walks rare; faults make overcommit safe at disk prices; the bits (valid, dirty) are the ledger's footnotes.

File systems

Why it matters. Files are the illusion that outlives the process: the layout beneath decides speed, reliability, and what a crash eats.

The idea, with a picture. Our teaching disk: 32 blocks of 1 KiB. The **inode** holds the metadata and 4 direct pointers plus one indirect (pointing at a block of 256 pointers); the **directory** is a file of name → inode entries; **FAT** chains blocks instead.



Directory entries to inodes to blocks: direct pointers first, the indirect block for big files, the free list on the side

The derivation (the file size bound). With 4 KiB blocks the arithmetic scales: 4 direct + one indirect block holding 1024 pointers ⇒ max file = $(4+1024)\times 4\,\text{KiB} \approx 4$ MiB. Our teaching size (1 KiB blocks, 256 pointers): $(4+256)\times 1\,\text{KiB} = 260$ KiB. Access cost: block $n<4$ costs one disk read (plus the inode, cached); block $n\geq 4$ adds the indirect block read. Fragmentation: chained layouts spread a file, so sequential reads become seeks – contiguity is a speed decision, not a correctness one.

> **Worked example 1 – read /notes/x, block 6.** The directory lookup finds the inode (cached), block 6 is beyond the four direct pointers: read the indirect block (pointer $6-4=2$), then the data block – two disk reads for one logical block. Block 2 would have cost one. The suite counts: reading a 100-block file sequentially = $100+1$ indirect reads, best case.

> **Worked example 2 – the crash question.** Append writes the data block, then the free-list bit, then the inode size. Crash between writes: which order loses what? Data-then-inode loses nothing but space (orphan block); inode-then-data lies about size (reads return garbage). **Journaling** writes the intent first and replays – the reason modern file systems pay a small write tax for crash consistency.

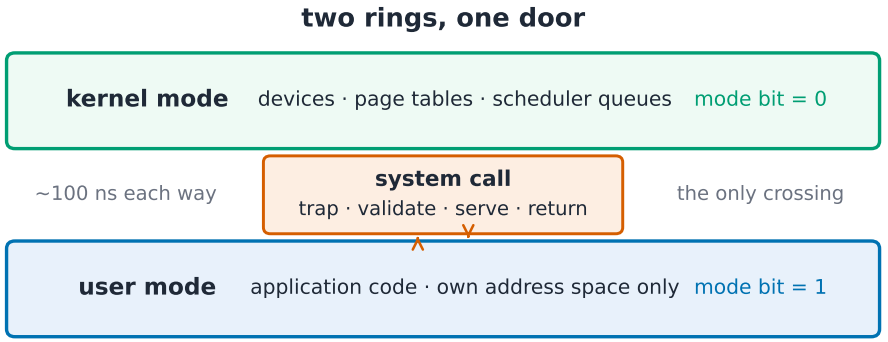
Common mistakes. Forgetting the directory is itself a file (lookups are reads). Assuming the indirect pointer is free (it is a disk read). Believing deletion erases (it clears pointers – the data stays until reused).

Summary. Directory → inode → direct/indirect pointers → blocks; size bound = (direct + pointers per block) × block size; write order decides what a crash eats; journaling buys consistency.

Protection and security

Why it matters. Everything above assumed the programs cooperate. Protection is the hardware-plus-OS wall that makes the assumption unnecessary – and security is the practice of keeping that wall standing.

The idea, with a picture. Two **rings** : user mode runs application code; kernel mode runs the OS and can touch devices and page tables. The crossing points are **system calls** – the wall's only doors – and the **mode bit** on the CPU enforces them.



User code crosses to the kernel only through the syscall door; the kernel validates everything at the boundary

The derivation (the principle of least privilege). Every mechanism above is an attack surface: the scheduler's queues, the page table's bits, the file metadata. Least privilege gives each process only what it needs: the page tables already do it per process (unit 7.4), file permissions do it per user (read/write/execute × owner/group/other = 9 bits), and capabilities do it per object. The economics: a bug in a smaller privileged base is a smaller target – the reason microkernels and sandboxed browsers exist.

> **Worked example 1 – one syscall's crossing.** `open("/notes/x")` from user mode: the trap instruction switches the mode bit, saves user registers, jumps to a fixed vector; the kernel validates the path (permissions), fills a file descriptor, returns – and the mode bit flips back. Two crossings per syscall at ≈ 100 ns each: a program doing 10^6 syscalls pays 0.2 s – the reason libraries buffer and batch.

> **Worked example 2 – the confused deputy.** A privileged printer daemon asked to print `/salary.db` by an unprivileged user: the daemon's authority, not the user's, opens the file – a deputy confused about whose wishes to execute. The fix is to pass capabilities (open descriptors), not names: the deputy can only use what it was handed. The wall's doors must check identity at every crossing, or the wall itself becomes the weapon.

Common mistakes. Calling security a layer (it is a property of every layer). Checking permissions at open but not at use (TOCTOU: time-of-check to time-of-use). Trusting the pointer because it came from user space (validate at the boundary, always).

Summary. The mode bit makes the wall, syscalls are its doors, least privilege sizes the rooms; validate every crossing; pass capabilities, not names.

Topic 08

University Computer Science

What this course is, and how to use it

This is the eighth course in University Computer Science: **databases** – how to model a domain as relations, query them precisely, keep them consistent under load, and govern who may see what.

Scope limit, stated once: a study course, not a degree; the teaching schema is an original university-enrolment database, and every query and output in this handout was executed against it by the course suite (SQLite, deterministic).

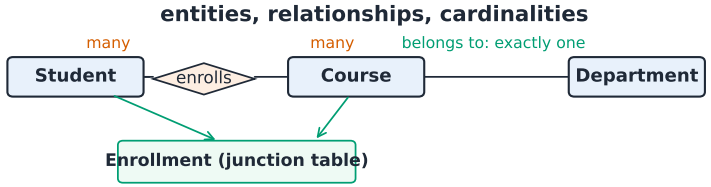
- Units 8.1–8.3 model and shape the data: ER, relational algebra and SQL, normalisation.
- Units 8.4–8.5 make it fast and safe: indexes with query plans, transactions with concurrency.
- Unit 8.6 governs it: security and data governance.

The database habit: **every claim about data is a query you can run** – each unit's numbers come from the documented sample schema.

Relational modelling

Why it matters. The model decides everything downstream: what can be asked, what must be checked, what can go wrong. A good schema is a set of true sentences about the domain.

The idea, with a picture. An **entity** is a thing we count (a Student); a **relationship** connects entities (a Student **enrolls** in a Course). The **cardinality** of each end – one, many – becomes keys and tables: a many-to-many relationship becomes its own table of foreign-key pairs.



many-to-many becomes its own table of foreign-key pairs; PRIMARY KEY (student, course, term) enrolls once per term

referential integrity: every Enrollment.student and .course names a real row

sample data: 8 students - 6 courses - 14 enrollments

Students enroll in Courses (many-to-many through Enrollments); each Course belongs to one Department

The derivation (why keys exist). A **primary key** identifies a row uniquely: `student_id` in Student. A **foreign key** references it: `Enrollments.student_id` must name a real student – the database enforces **referential integrity**, so no enrollment can orphan itself. Our teaching schema:

```
Department(dept_id PK, name, building)
Student(student_id PK, name, dept_id FK)
Course(course_id PK, title, dept_id FK, credits)
Enrollment(student_id FK, course_id FK, term, grade,
           PRIMARY KEY (student_id, course_id, term))
```

> **Worked example 1 – modelling a rule.** A student may enroll in a course at most once per term. The rule lives in the keys: Enrollments carries `PRIMARY KEY (student_id, course_id, term)` – a second insert of the same triple is rejected by the engine, not by the application. Rules in the schema are rules the database keeps.

> **Worked example 2 – the join path.** Which students take a department's courses? Student joins Enrollment on `student_id`, Enrollment joins Course on `course_id`, Course joins Department on `dept_id`: three joins, each following a foreign key. The suite's sample data: 8 students, 6 courses, 14 enrollments – the join path yields 14 rows before filtering.

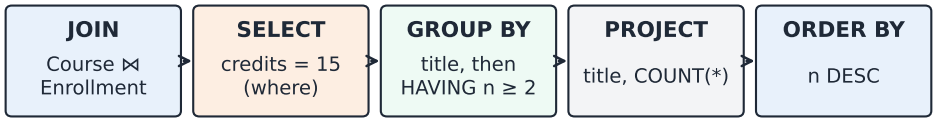
Common mistakes. Storing a student's name inside Enrollment (duplicates, updates in many places). One table per relationship end instead of a junction table for many-to-many. Choosing a key that can change (emails move; surrogate IDs do not).

Summary. Entities to tables, relationships to foreign keys or junction tables; cardinality decides which; keys carry the rules the engine then enforces.

Relational algebra and SQL

Why it matters. SQL is the payroll language of the industry, and its heart is five algebra operators with exact semantics – know the algebra and every query is readable as a pipeline.

The idea, with a picture. The algebra: **selection** σ keeps rows ($\sigma_{\text{credits}=15}$), **projection** π keeps columns (π_{name}), **join** $\bowtie$ combines on a condition, union $\cup$ and set-difference $-$ combine tables. Every SELECT is these five in a pipeline.



FROM builds rows, WHERE filters rows, GROUP BY partitions, HAVING filters groups

SELECT c.title, COUNT(*) AS n
FROM Course c JOIN Enrollment e ON c.course_id = e.course_id
WHERE c.credits = 15 GROUP BY c.title HAVING COUNT(*) >= 2 ORDER BY n DESC;

executed result: Databases | 3

One query as a pipeline: from Courses select credits = 15, project the title, join Enrollments, project and count the student names

The derivation (WHERE before GROUP BY before HAVING). SQL's clauses run in a fixed order: FROM builds the row set, WHERE filters rows, GROUP BY partitions them, HAVING filters groups, SELECT projects, ORDER BY sorts. So `WHERE grade = 'A'` keeps rows then groups; `HAVING COUNT(*) >= 2` keeps groups only – filtering on an aggregate in WHERE is a category error the engine rejects.

Worked example 1 – one query, read as algebra.

```
SELECT c.title, COUNT(*) AS n
FROM Course c JOIN Enrollment e ON c.course_id = e.course_id
WHERE c.credits = 15
GROUP BY c.title
HAVING COUNT(*) >= 2
ORDER BY n DESC;
```

Algebra: join Course and Enrollment, select credits = 15, group by title, keep groups of size ≥ 2, project title with the count. On the sample data the result is `Databases | 3` – one 15-credit course with three enrollments (the suite records the exact output).

> **Worked example 2 – the outer join's kept blanks.** List every course with its enrollments: a plain join drops the 2 courses with no enrollment; `LEFT JOIN` keeps them with NULL counts – 6 rows, 2 with `n = 0` (SQLite's `COUNT(e.student_id)` counts present values only). The join flavor is a semantics decision, not a style one.

Common mistakes. Filtering aggregates in WHERE (use HAVING). Counting rows where you meant values (`COUNT(*)` vs `COUNT(col)` – the latter skips NULLs). Forgetting DISTINCT when projecting a many-side column (14 enrollment rows project 14 student ids, some repeated).

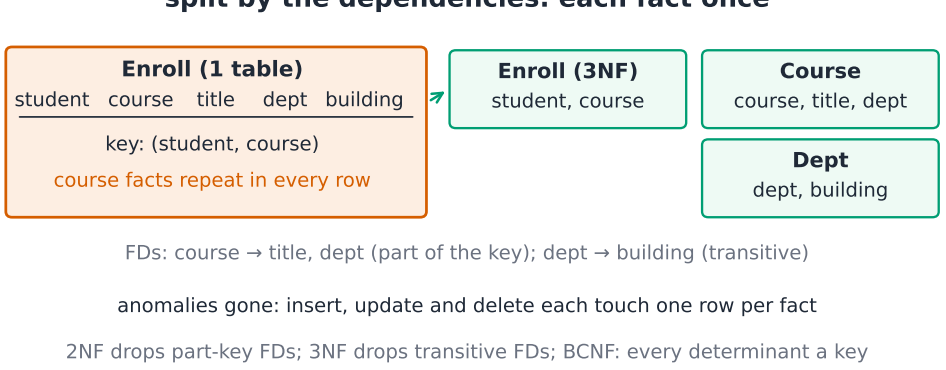
Summary. Five operators, one pipeline; clause order is semantic (WHERE rows,

HAVING groups); join flavor decides who survives the match.

Normalisation

Why it matters. A denormalised table invites three anomalies: **insert** , **update** , **delete** – normalisation removes them by splitting along dependencies.

The idea, with a picture. A **functional dependency** $A \to B$: each A-value determines one B-value. **Second normal form** requires first normal form and no non-key attribute depending on only part of a candidate key. In **third normal form** , every nontrivial dependency has a superkey determinant or a right-hand attribute belonging to a candidate key. **Boyce–Codd** BC requires a superkey determinant for every nontrivial dependency.



One denormalised table split by its dependencies: the key part, the transitive department facts, and the enrollment fact itself

The derivation (why 3NF kills the anomalies). The un-normalised `Enroll(student, course, title, dept, building)` with key (student, course) holds: `course → title` (part-key), `title → dept → building` (transitive). Update anomaly: a course's title appears in every enrollment row – change one, miss ten. Split by the dependencies: `Course(course, title, dept)` and `Dept(dept, building)` and `Enroll(student, course)`: this removes the partial-key dependency from Enroll. Because `title → dept` still holds, Course is not yet in 3NF. Split it further into `Course(course, title)` and `Title(title, dept)`. With the stated dependencies, the four resulting tables are in BCNF.

> **Worked example 1 – grading the form.** `Room(building, room, capacity, projector_type)` with key (building, room): if all rooms in a building share one projector type, the FD `building → projector_type` violates 2NF and therefore 3NF: a non-key attribute depends on only part of the candidate key. Split into `Building(building, projector_type)` and `Room(building, room, capacity)`. This removes the repeated projector fact.

> **Worked example 2 – when to stop.** A warehouse report table stores `order_line` plus a running total per order: the total is derivable, but recomputing per view costs more than the anomaly risk of storing it – controlled denormalisation with a refresh trigger is a legitimate engineering choice once the normal form is understood. Normalise to reason; denormalise on evidence.

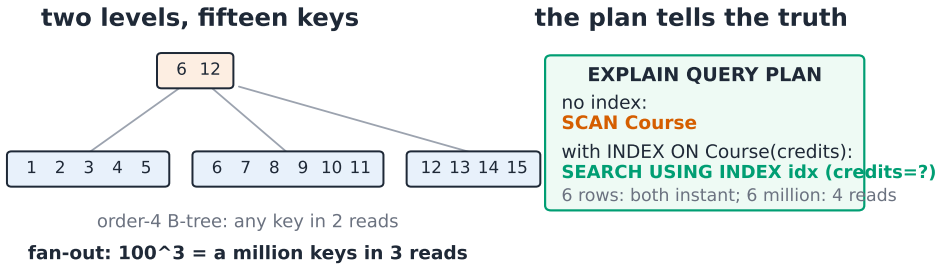
Common mistakes. Calling a table 3NF while a non-key column still determines another. Splitting so finely that every query joins five tables without measuring the cost. Forgetting the anomalies are symptoms – the disease is the repeated fact.

Summary. FDs drive the split: part-key out for 2NF, transitive out for 3NF in simple single-key examples; for BCNF, every nontrivial dependency has a superkey determinant. State the dependencies before judging a decomposition.

Indexes and query plans

Why it matters. A query that scans a million rows to find ten is a design failure, and the fix is not more hardware – it is an index the planner can use.

The idea, with a picture. A **B-tree** B index keeps keys sorted in a balanced multiway tree: our teaching tree (order 4, 15 keys) finds any key in $\lceil \log_4 15 \rceil = 2$ node reads against 15 sequential. The **query plan** is the engine's chosen pipeline: scan or index, join order, strategy.



Left: the course-id B-tree, two levels for 15 keys. Right: the plan that uses it – index lookup instead of a full scan

The derivation (the fan-out arithmetic). A node holding k keys routes among $k+1$ children: order m gives fan-out $\approx m$. Height h covers m^h entries: fan-out 100, three levels cover $100^3 = 10^6$ keys – a million-row lookup in 3 node reads. Every insert keeps the tree balanced (splits propagate upward), so the bound holds for every key, not just lucky ones.

> **Worked example 1 – a plan, read.** `EXPLAIN QUERY PLAN SELECT title FROM Course WHERE credits = 20`: without an index the plan reads `SCAN Course`; with `CREATE INDEX idx ON Course(credits)` it reads `SEARCH Course USING INDEX idx (credits=?)`. On 6 rows both are instant; on 6 million the scan reads 6 million rows and the search reads 4 (fan-out 100 covers 10^6 keys at height 3 – six million needs height 4) – the plan, not the query text, prices the work.

> **Worked example 2 – the write tax.** Each index adds work to every insert and delete: our schema with 3 indexes writes 4 structures per insert (table + 3 trees). A log table with 5 indexes and 10 million inserts per day maintains 50 million index entries daily – index what you query, drop what you do not, and measure.

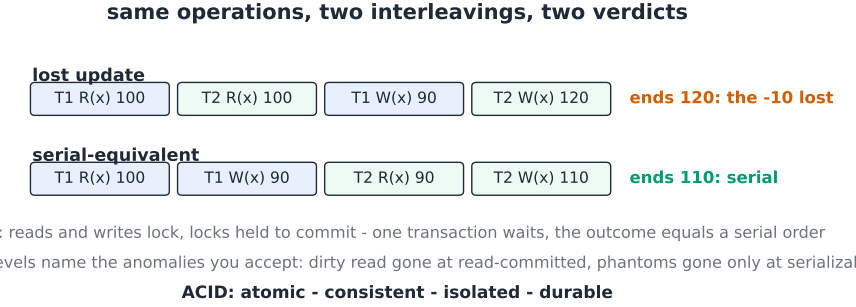
Common mistakes. Indexing every column (writes pay for reads you never make). Expecting the planner to use an index on a computed column (`WHERE UPPER(name) = …` defeats a plain index). Reading the query instead of the plan.

Summary. B-trees trade fan-out for height: m^h keys in h reads; the plan tells the truth; every index is a loan the writes repay.

Transactions and concurrency

Why it matters. Two clerks, one balance, simultaneous updates: without transactions the database can commit both halves of different transfers and lose money invisibly.

The idea, with a picture. A **transaction** is all-or-nothing: **ACID** – atomicity, consistency, isolation, durability. Under **serializability** an interleaving's result equals SOME serial order; weaker **isolation levels** trade correctness for concurrency: read uncommitted, read committed, repeatable read, serializable.



Two transfers interleaved: locking reads and writes yields a serial-equivalent order; reading before the other's commit yields a lost update

The derivation (the lost update, again). T1 reads balance 100, T2 reads 100, T1 writes 90, T2 writes 120: the 10-unit withdrawal has vanished – the banking twin of unit 7.3's race. Two-phase locking (2PL) first acquires locks, then releases them; after the first release, no new lock may be acquired. This guarantees conflict serializability for transactions following the protocol. Strong strict 2PL additionally holds all locks until commit or abort. Transactions can deadlock; a detector may abort one so it can retry.

> **Worked example 1 – schedule legality.** Interleaving `R1(A) R2(A) W2(A) W1(A)`: T1's write clobbers T2's – a lost update, not serializable. Interleaving `R1(A) W1(A) R2(A) W2(A)`: equals serial T1-then-T2 – legal. The suite checks both schedules by simulation.

> **Worked example 2 – the dirty read.** At read-uncommitted, T2 reads T1's uncommitted 90; T1 then rolls back – T2 has acted on a value that never existed. Read-committed prevents it (reads see only committed data); repeatable-read adds stability of re-reads; serializable adds phantoms' absence. Each level is a promise about which anomalies remain.

Common mistakes. Long transactions holding locks through user input (deadlock bait). Equating isolation with consistency (C is the constraints; I is the interleaving). Skipping the retry loop because deadlock "rarely" happens (it happens at peak).

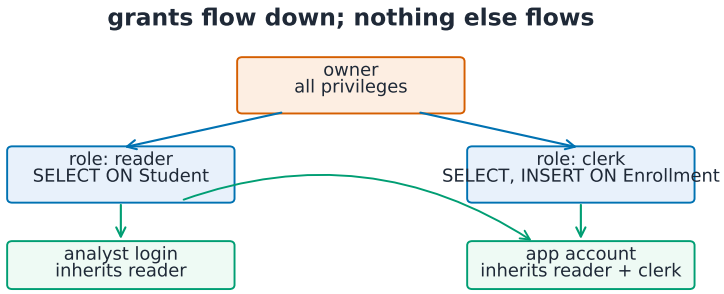
Summary. ACID: all-or-nothing, constraints kept, interleaving equivalent to a serial order, commits that survive the crash; isolation levels name the anomalies you accept; 2PL buys serializability at the price of waiting.

Security and data governance

Why it matters. A database holds what people cannot afford to leak: the last decade's breaches were access-model failures, not cryptographic ones.

The idea, with a picture. Authentication proves who you are; **authorisa-**

tion proves what you may do; the **principle of least privilege** shapes both: accounts get the narrowest grants that let the job run.



least privilege: the app account reads two tables and writes one - a stolen session is bounded by the grant

Grants flow from the owner to roles to users; the application account reads two tables and writes one – nothing else

The derivation (injection, and its end). A query built by pasting user text: `"... WHERE name = '" + input + "'"` with input `' OR '1'='1` matches every row – and worse inputs drop tables. Parameterised statements bind values separately from SQL syntax, so text in a bound value cannot change the query structure. Table names, column names and other dynamic SQL need separate controls; parameter binding does not protect SQL still built by concatenation.

> **Worked example 1 – grants that fit.** The teaching grants: `GRANT SELECT ON Student TO reader; GRANT SELECT, INSERT ON Enrollment TO clerk;` – the clerk account cannot read salaries it does not see, cannot DELETE an enrollment, and a stolen clerk session is bounded by the grant. Revoke on role change, not on audit day.

> **Worked example 2 – governance is a loop.** Data governance classifies data (public, internal, sensitive), assigns owners, sets retention, and audits access: the sensitive `grade` column is encrypted at rest, its access logged, and a quarterly review revokes the stale grants. The model survives contact with reality only because the loop runs.

Common mistakes. Connecting the application as the owner account (every bug runs with full authority). Auditing reads but not grants (privilege creep). Treating encryption as access control (it protects at rest, not against a valid login).

Summary. Authenticate, then authorise by least privilege; parameterise every query; classify, own, retain, audit – governance is the loop that keeps the model true.

Topic 09

University Computer Science

What this course is, and how to use it

This is the ninth course in University Computer Science: **computer networks** – how a message crosses a planet by hopping machines that owe each other nothing, and what each layer promises the one above it.

Scope limit, stated once: a study course, not a degree; every header, checksum and sequence number in this handout is decoded byte by byte in the course suite, on the exact values printed here.

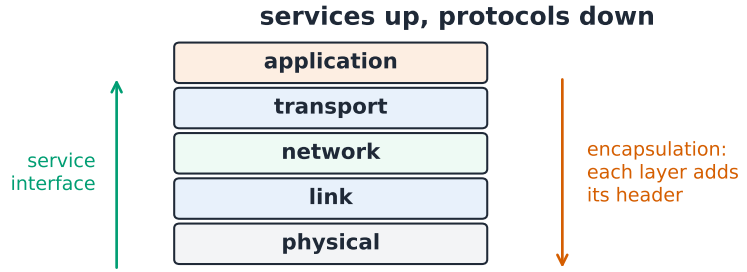
- Units 9.1–9.3 move a frame: layers, the link and IP, and routing.
- Units 9.4–9.5 make it reliable and useful: transport and application protocols.
- Units 9.6–9.7 secure and scale it: TLS and distributed fundamentals.

The network habit: **a protocol is a promise about bytes** – every unit decodes one promise at a time.

Layering and architecture

Why it matters. No single machine can implement the whole journey; **layering** lets each concern be built, debugged and replaced independently – the reason Wi-Fi could change without touching your browser.

The idea, with a picture. Each layer offers a **service** to the one above and consumes the one below: the five-course stack is **physical** , **link** , **network** , **transport** , **application** . Data is **encapsulated** downward (each layer adds its header) and decapsulated upward.



a 100-byte payload gains 54 bytes of headers: 14 + 20 + 20

the hourglass: one narrow waist (IP), wide top, wide bottom

Each layer adds a header going down and strips it coming up; the service interface is the promise between neighbours

The derivation (why layers beat monoliths). A change confined to one layer requires no change elsewhere IF the service interface holds – the layer's contract. Ethernet went from 10 Mb/s to 100 Gb/s under identical IP because the link service (best-effort frames) never changed. A leaky interface (an upper layer reading a lower header) destroys the independence: the browser that reads TTL is written once and breaks forever.

> **Worked example 1 – one message, twice wrapped.** A 20-byte TCP segment rides a 20-byte IPv4 header inside a 14-byte Ethernet header: on the wire the frame is $14 + 20 + 20 = 54$ bytes of header for the payload – the receiver's NIC strips 14, the IP layer strips 20, TCP strips its 20. Each layer sees only its own promise.

> **Worked example 2 – the reference models.** The 5-layer course stack against the 7-layer OSI: OSI splits our link into physical + link and adds presentation + session above application. The internet won with the hourglass: narrow waist (IP), wide top (applications), wide bottom (links) – one protocol at the waist lets everything above talk to everything below.

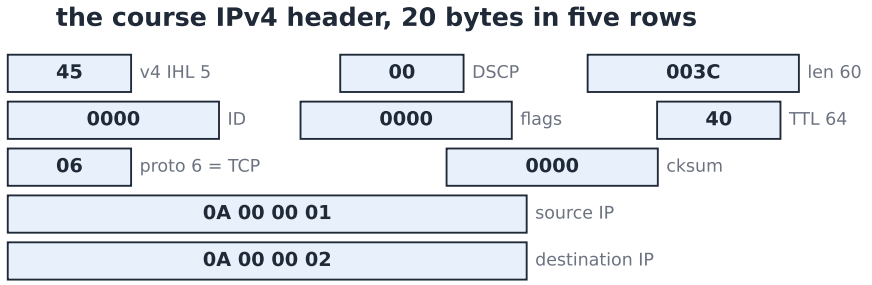
Common mistakes. Counting layers in the wrong direction (services point UP the stack). Confusing the service (promise) with the protocol (the bytes that keep it). Skipping the physical layer in latency math (propagation is physics, not software).

Summary. Services up, protocols down, encapsulation in between; the interface is the independence; IP is the waist of the hourglass.

Link layer and IP

Why it matters. The link moves a frame between neighbours; IP moves a packet between strangers. Confusing the two addresses is the classic beginner bug.

The idea, with a picture. An Ethernet frame: 6-byte destination **MAC address** MAC , 6-byte source, 2-byte type, payload, 4-byte **CRC** . An IPv4 header: version/IHL, total length, **TTL** , protocol, checksum, 4-byte source and destination **IP address** IP .



IHL 5 counts 32-bit words: 5 x 4 = 20 header bytes; total length 60 = 20 header + 40 payload
MAC names a neighbour; IP names the stranger

The IPv4 header decoded field by field on the course example: total length 60, TTL 64, protocol 6 (TCP)

The derivation (why IP checksums only the header). The CRC protects the whole frame link by link; the IP checksum protects only the header end to end – because routers rewrite the header (TTL decrements) but must not recompute data integrity. Each integrity mechanism is scoped to exactly the span its rewriter can touch.

> **Worked example 1 – decode a header.** The course bytes `45 00 00 3C ... 40 06 ...`: version 4, IHL 5 (so $5 \times 4 = 20$ bytes), total length $0x003C = 60$ (20 header + 40 payload), TTL $0x40 = 64$, protocol $0x06$ = TCP. The suite decodes every field of the printed header and asserts each value.

> **Worked example 2 – addresses at two scopes.** The MAC changes every hop (frame to frame: it names neighbours); the IP never changes (end to end: it names strangers). A packet crossing 3 routers carries 4 different destination MACs in 4 frames and one destination IP the whole way – the suite traces all four frames.

Common mistakes. Reading IHL as bytes (it counts 32-bit words). Expecting the IP checksum to cover payload. Believing MAC addresses route (they find the next hop, chosen by IP).

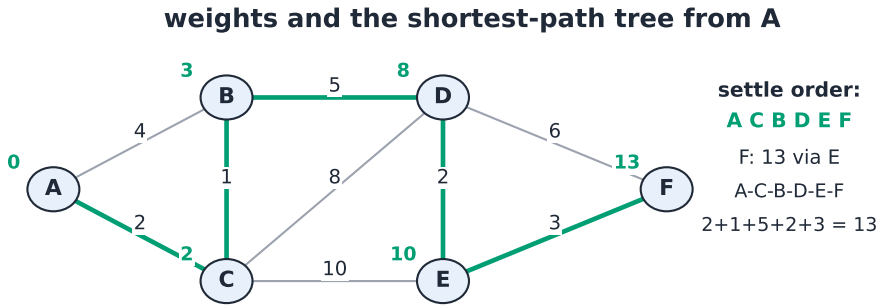
Summary. MAC = neighbour, IP = stranger; IHL words; total length bytes; each checksum spans exactly what its rewriter touches.

Routing

Why it matters. IP names the destination; routing decides the next hop – millions of machines agreeing without a boss, by an algorithm you can run on paper.

The idea, with a picture. A router's **routing table** maps destination prefixes to next hops: the **longest prefix match** rule picks the most specific route. Our course network: links with weights, routers A–F, and Dijkstra's

algorithm computing the shortest path tree from A.



longest prefix match picks the /24 over the /16 over the /8; the default catches the rest

The course graph with link weights; Dijkstra from A settles C, B, D, E, F in order with the tree in bold

The derivation (Dijkstra, in numbers). Course weights: A–B 4, A–C 2, B–C 1, B–D 5, C–D 8, C–E 10, D–E 2, E–F 3, D–F 6. From A, the settle order is C, B, D, E, F: C at 2 (via A), B at 3 (via C – the short hop through C beats A–B's direct 4), D at 8 (via B), E at 10 (via D – the two-hop $8+2$ beats the $2+10=12$ that C's direct link would give), F at 13 (via E: $10+3$ beats D's direct $8+6=14$). F's best path is A C B D E F at $2+1+5+2+3=13$. The suite runs the algorithm and asserts every settled distance, the order, and the tree.

> **Worked example 1 – longest prefix match.** Table: `10.0.0.0/8 → X`, `10.1.0.0/16 → Y`, `10.1.2.0/24 → Z`, default → W. Destination `10.1.2.200`: three prefixes match; the /24 is longest – next hop Z. Destination `10.9.9.9`: only the /8 – X. The default catches what nothing else does.

> **Worked example 2 – why hierarchy.** Flat tables would hold one entry per host: 4×10^9 IPv4 rows. Prefixes aggregate: an ISP announces one /12 covering 2^{20} addresses – the table stays at 10^5 entries because aggregation throws away the host bits.

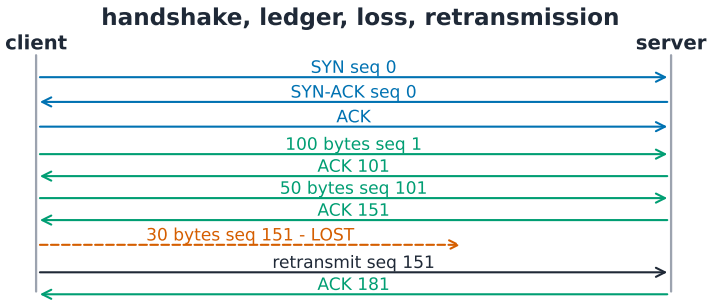
Common mistakes. Forgetting the settled set (relaxing an already final node). Matching the first table row instead of the longest prefix. Counting hops when the metric is delay.

Summary. Tables map prefixes to hops; longest prefix wins; Dijkstra builds shortest path trees by settling the nearest unsettled node; aggregation keeps the tables human-sized.

Transport (TCP/UDP)

Why it matters. IP promises best effort – packets may vanish, reorder, or duplicate. The transport layer connects application processes: TCP provides an ordered byte stream, while UDP sends datagrams without guaranteeing delivery or order.

The idea, with a picture. UDP : ports plus a length and checksum – a postcard. **TCP** : sequence numbers, acknowledgements, retransmission, flow and congestion control – a courier with a ledger. The **three-way handshake** SYN, SYN-ACK, ACK opens a connection; FIN closes it.



cumulative ACKs name the next byte expected; the gap holds the ACK number until the timeout retransmits

The course exchange: handshake, two segments with cumulative ACKs, a lost segment retransmitted after the timeout

The derivation (why cumulative ACKs). Each ACK names the next byte expected: ACK 200 says bytes 0–199 arrived, whatever their order. A gap stalls the ACK number – the sender's timeout retransmits from the gap. One number summarises the whole receive state: simple to check, impossible to misread.

> **Worked example 1 – the ledger.** SYN consumes sequence 0; the server SYN-ACK carries its own 0. Client sends 100 bytes: seq 1, next 101; server ACKs 101. Client sends 50: seq 101; server ACKs 151. A 30-byte segment sent at seq 151 is lost: the ACK stays 151 until the timeout retransmits it – then ACK 181. The suite simulates the exchange and asserts every number.

> **Worked example 2 – UDP's honest scope.** A 8-byte UDP header (ports, length, checksum) against TCP's 20 minimum: DNS fits in one UDP datagram – a request is one packet, and if it is lost the client re-asks. Video calls tolerate loss but not delay: retransmitting a frame due in 30 ms arrives too late to matter – UDP plus application logic is the right promise.

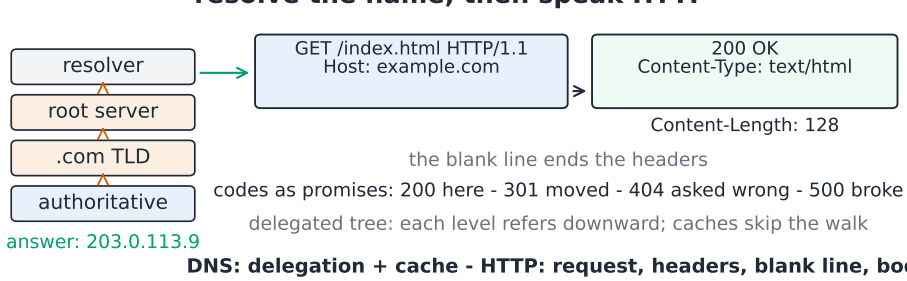
Common mistakes. Counting SYN in the data sequence (it consumes one number but carries none). Confusing flow control (receiver's window) with congestion control (network's state). Expecting TCP to guarantee timing – it guarantees order and delivery, never latency.

Summary. UDP = ports plus a checksum for request-sized work; TCP = sequence ledger plus cumulative ACKs plus retransmission; the handshake opens, FINs close, and windows protect both ends.

Application protocols

Why it matters. The transport moves bytes; the application agrees on what they mean. Reading a protocol is reading a conversation with a grammar.

The idea, with a picture. DNS : name → address, hierarchical delegation (root → TLD → authoritative). **HTTP** : request line, headers, body; **status codes** $2xx$ success, $3xx$ redirect, $4xx$ client error, $5xx$ server error.



A DNS lookup then an HTTP exchange: the recursive resolver asks root, TLD, authoritative; the browser then GETs and the server answers 200

The derivation (why DNS is hierarchical). One global name table would be one global outage and one global fight. Delegation splits it: the root (13 addresses) knows the TLD servers; `.com` servers know the registrars; the authoritative server knows the host. Each answer is a referral downward – and caching turns the tree into a short-cut: the resolver that once asked needs no full walk for the TTL of the answer.

> **Worked example 1 – one page, by the bytes.** `GET /index.html HTTP/1.1` + `Host:` header; the server answers `200 OK`, headers (`Content-Type: text/html`, `Content-Length: 128`), blank line, 128 bytes. The suite checks: a response with Content-Length 128 carries exactly 128 body bytes; a 404 carries headers and a short body; a 301 carries `Location:` and no body to speak of.

> **Worked example 2 – the codes, read as promises.** 200 (here is the body), 301 (the body moved, `Location` says where), 404 (you asked wrong), 500 (I broke). A crawler treats 301 as a pointer to follow, 404 as a dead end, 500 as retry-later – the code is the contract the client programs against.

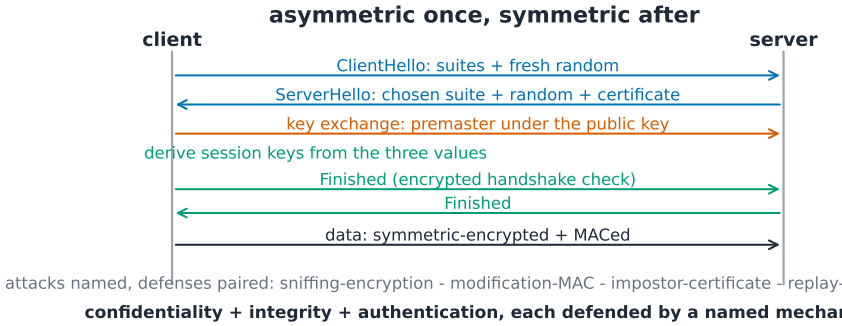
Common mistakes. Forgetting the blank line that ends headers (the parser waits forever). Reading Content-Length as header bytes. Treating DNS as one server (it is a delegated tree with caches).

Summary. DNS resolves names by delegation plus caches; HTTP is a request-response grammar where the status code is the promise and headers carry the metadata.

Network security

Why it matters. The network is a party line: anyone on the path can read, drop, or forge. Security is the set of promises that make an open channel safe to use.

The idea, with a picture. Symmetric encryption: one shared key, fast. **Asymmetric** : a key pair – public encrypts, private decrypts. **TLS** combines them: asymmetric for the key exchange, symmetric for the data, certificates for identity.



ClientHello, ServerHello with certificate, key exchange, both Finished – then symmetric data

The derivation (the properties, and who provides what). Confidentiality needs encryption; integrity needs a **MAC** with both directions checked (each Finished message proves the handshake was not tampered with); authentication needs a certificate chain to a root the client already trusts. Drop any one and the protocol fails differently: no encryption is a wiretap, no MAC is alteration, no certificate is an impostor.

> **Worked example 1 – the handshake, numbered.** ClientHello (offered suites, random), ServerHello (chosen suite, random) + certificate, key exchange (the premaster encrypted to the server's public key), both sides derive the session keys from the three values, Finished messages verify, then data flows symmetric-encrypted. One RSA operation, then stream-speed cryptography – the asymmetry is used only to agree on the symmetric key.

Worked example 2 – the attacker menu. Passive sniffing: defeated by encryption. Modification: defeated by the MAC (the handshake Finished catches tampering before data flows). Impersonation: defeated by certificates (the server proves possession of the private key matching the certified public one). Replay of the handshake: defeated by the fresh randoms in both Hellos. Each attack names its defense.

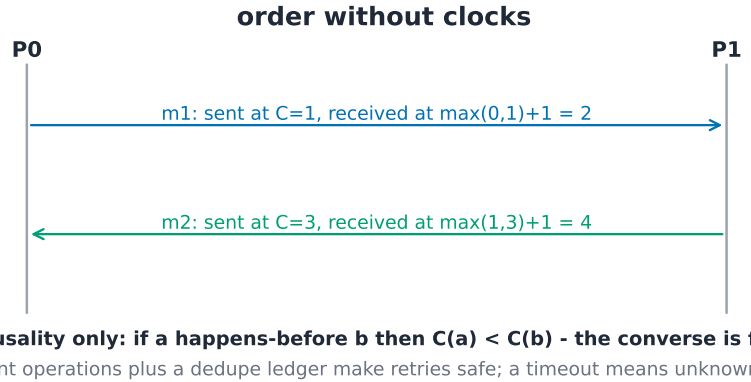
Common mistakes. Encrypting without authenticating (you have a secure channel to the wrong machine). Trusting self-signed certificates in production. Believing HTTPS makes the server honest (it makes the channel private and the server identified – the content can still lie).

Summary. Symmetric for speed, asymmetric for key agreement, certificates for identity, MACs for integrity; every defense answers a named attack.

Distributed systems fundamentals

Why it matters. Once two machines cooperate over a network, partial failure and reordering become physics. Distributed systems is the discipline of promising things anyway.

The idea, with a picture. A **distributed system** shares state across machines with **partial failure** : one machine crashing must not corrupt the rest. Messages reorder and duplicate; **idempotence** makes re-execution safe; **happens-before** orders events even without clocks.



Two messages, one causal chain: Lamport counters 1, 2, 3, 4 across P0 and P1 without any clock

The derivation (why clocks cannot order). Two machines' clocks skew by unbounded amounts, so timestamps cannot safely order events across machines. The **Lamport clock** Lamport gives a counter that respects causality: on send, attach $C{+}1$; on receive, set $C = \max(C, \text{received}) + 1$. If a happens-before b then $C(a) < C(b)$ – the converse does not hold. A smaller counter alone does not prove a causal relationship between two events.

Worked example 1 – the Lamport trace. P0 sends m_1 at $C = 1$; P1 receives: $C = \max(0, 1) + 1 = 2$; P1 sends m_2 at 3; P0 receives: $C = \max(1, 3) + 1 = 4$. The suite asserts every counter and that $C(m_1) = 1 < C(m_2) = 3$ – causal order preserved without any clock.

Worked example 2 – idempotence by design. A payment API takes `deduct(user, 25, id=xyz)`: executing it twice must deduct once. Make the operation idempotent by key – the ledger records `xyz` as done and the retry is a no-op. Retry-safe systems are built from idempotent operations plus a dedupe ledger, not from hoping the network stops duplicating.

Common mistakes. Using timestamps for cross-machine order (skew wins). Retrying non-idempotent operations (double charges). Assuming a call that returned means the work happened – or that a timeout means it did not (it means you do not know: the third outcome).

Summary. Partial failures may be temporary or permanent; messages reorder and duplicate; Lamport counters respect happens-before; idempotence plus dedupe makes retries safe; a timeout means unknown, not failed.

Topic 10

University Computer Science

What this course is, and how to use it

This is the tenth course in University Computer Science: **algorithm design and analysis** – the discipline of proving what an algorithm costs before trusting it with data, and the toolbox of design patterns that make the proofs easy.

Scope limit, stated once: a study course, not a degree; every pseudocode fragment here is checked against a Python implementation in the course suite, and every asymptotic claim is verified on concrete inputs.

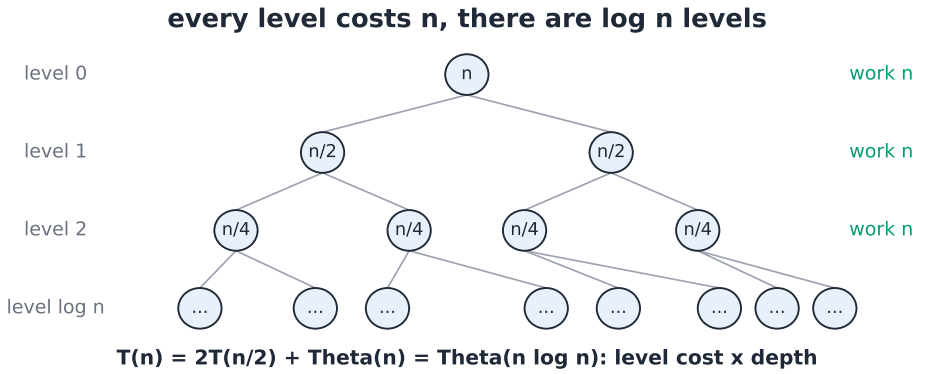
- Units 10.1–10.2 are the analysis and the first pattern: recurrences and divide-and-conquer.
- Units 10.3–10.4 are the choice patterns: greedy and dynamic programming.
- Units 10.5–10.6 are the frontier: graph optimisation and NP-completeness.

The designer's habit: **a claim about an algorithm is a proof sketch with numbers** – every unit runs one.

Recurrences and analysis tools

Why it matters. Recursive algorithms state their own cost as a **recurrence** – learning to read one is learning to price a design before writing it.

The idea, with a picture. $T(n) = 2T(n/2) + \Theta(n)$: the **recursion tree** draws each level's work – level i has 2^i nodes of $n/2^i$ each, so every level costs $\Theta(n)$ and the tree has $\log_2 n$ levels: $T(n) = \Theta(n \log n)$. That is merge sort's bill, read off a picture.



Each level of the tree costs the same n; the depth is $\log_2$ n – the product is the answer

The derivation (the master method, in one line each). For $T(n) = aT(n/b) + f(n)$: compare $f(n)$ with $n^{\log_b a}$ (the leaf total). Leaves dominate: $T = \Theta(n^{\log_b a})$; balanced: $T = \Theta(f(n) \log n)$; root dominates: $T = \Theta(f(n))$. Course cases: $2T(n/2) + n \to n \log n$ (balanced); $T(n/2) + 1 \to \log n$ (binary search, leaves trivial); $4T(n/2) + n \to n^2$ (leaves $n^{\log_2 4} = n^2$ dominate).

> **Worked example 1 – unrolling by substitution.** Guess $T(n) \le c\,n \log n$ for $T(n) = 2T(n/2) + n$: substitute $2c(n/2)\log(n/2) + n = cn \log n - cn \log 2 + n \le cn \log n$ whenever $c \ge 1$ – the induction closes, and the guess is proven, not assumed.

> **Worked example 2 – measuring, not guessing.** The suite implements merge sort and counts comparisons: on the fixed 64-element course array, the count is **310**, under the worst-case ceiling $n \log_2 n - n + 1 = 321$ – the measured number lands inside the asymptotic family, the only honest check.

Common mistakes. Summing the leaves but not the levels (or vice versa). Applying the master method to unequal splits (T(n/3) + T(2n/3) – draw the tree instead). Believing a tighter loop bound changes the recurrence (the work function is what it is).

Summary. Draw the tree, sum the levels, count the depth; the master method compares root work with leaf total; substitution proves the closed form; measurement closes the loop.

Divide and conquer

Why it matters. Split, solve, combine – the pattern behind merge sort, quicksort, binary search and Strassen; the art is pricing the combine step honestly.

The idea, with a picture. Merge sort splits in half, recurses, merges in $\Theta(n)$: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$. **Quicksort** partitions – the work moves into the partition, and the split quality decides everything.

The same skeleton with three combine costs: merging (n per level), comparing (n per level), a constant – three different bills

The derivation (quicksort's three worlds). Balanced pivots: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$. Sorted input with first-element pivots: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$ – every partition peels one element. Randomised pivots make the expected comparisons $2n \ln n \approx 1.39\, n \log_2 n$: the suite measures quicksort on the course array at **356** comparisons against merge's 310 – same family, different constants.

> **Worked example 1 – maximum subarray, divide and conquer.** Split the array in half; the answer is the left best, the right best, or the best crossing the middle (a left suffix plus a right prefix). $T(n) = 2T(n/2) + \Theta(n)$ – the crossing scan costs a full pass: $\Theta(n \log n)$. On the course array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ the best is $[4, -1, 2, 1] = 6$ (suite-verified).

> **Worked example 2 – binary search as the degenerate case.** One recursive call, constant combine: $T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$. The pattern survives with one branch – divide and conquer does not require two.

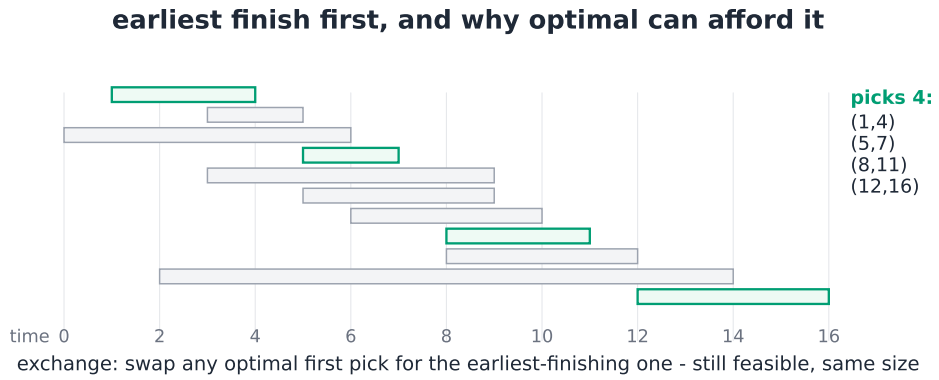
Common mistakes. Charging the combine step at the leaves. assuming balanced splits for quicksort without randomisation. Forgetting recursion's own overhead on tiny subproblems (real implementations cut to insertion sort below ~16 elements for exactly this reason).

Summary. Split, solve, combine – the recurrence prices the combine; quicksort's bill is the pivot's behaviour; one branch is still the pattern (binary search).

Greedy algorithms

Why it matters. The greedy choice – take the best now – is the cheapest design there is; the entire discipline is proving when cheap is correct.

The idea, with a picture. A **greedy algorithm** builds the solution piece by piece, always taking the locally best piece. **Exchange arguments** prove correctness: any optimal solution can be transformed into the greedy one without loss.



Activity selection: earliest finish first – taking the earliest-finishing activity leaves the most room, and the exchange argument shows any optimal schedule can be made to start the same way

The derivation (the exchange argument, once). Activity selection with earliest-finish-first: let G_1 be the greedy's first pick and O_1 the optimal's; $\text{finish}(G_1) \leq \text{finish}(O_1)$, so swapping O_1 for G_1 keeps the schedule feasible – repeat down the list and the optimal solution becomes the greedy one, unchanged in size. Greedy is optimal because optimal can always afford greedy's choice.

> **Worked example 1 – activity selection, measured.** Course activities (start, finish): (1,4), (3,5), (0,6), (5,7), (3,9), (5,9), (6,10), (8,11), (8,12), (2,14), (12,16). Earliest-finish picks (1,4), (5,7), (8,11), (12,16) – four activities (suite-run). A latest-start greedy on the same input also picks four – here it ties, but on other inputs (add activity (0, 1)) it trails: the greedy criterion is a design decision, and only earliest-finish carries the exchange proof.

> **Worked example 2 – Huffman, greedy with a proof.** Frequencies $5, 9, 12, 13, 16, 45$: repeatedly merge the two smallest – the suite computes the tree and the weighted depth **224** bits against $\lceil \log_2 6 \rceil = 3$ bits $\times\, 100 = 300$ for a fixed-length code: the exchange argument (swapping any two leaves can only increase the cost) makes 224 provably minimal.

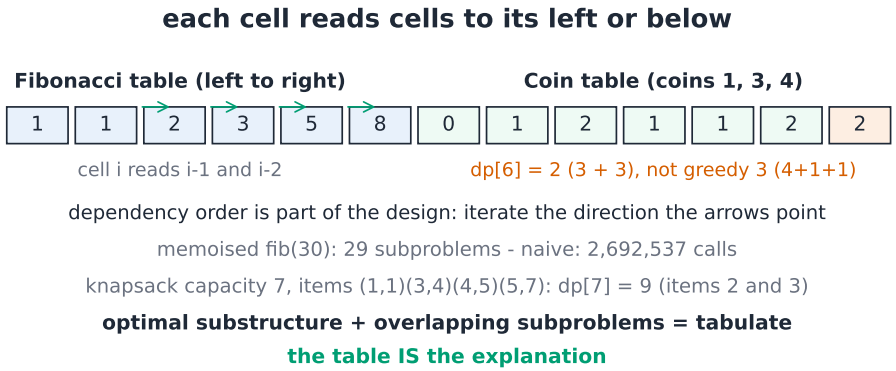
Common mistakes. Using greedy where the exchange argument fails (coins 1, 3, 4: greedy pays 6 = 4+1+1, optimal 3+3). Confusing "greedy is optimal" with "greedy is good enough" (approximation is a different contract). Forgetting to sort first (the criterion needs the ordering to mean what it says).

Summary. Greedy takes the best now; the exchange argument proves optimal can afford it; the criterion is the design; where exchange fails, greedy is a heuristic, not an algorithm.

Dynamic programming

Why it matters. When subproblems overlap, divide-and-conquer recomputes them; dynamic programming remembers – turning exponential recomputation into polynomial tabulation.

The idea, with a picture. Dynamic programming solves each subproblem once, stores it, and builds larger answers from stored ones. The **DP table** DP plus its **dependency order** is the whole design.



Fibonacci by table: each cell reads the two below it – the dependency order is bottom-up, and memoisation would discover the same order lazily

The derivation (why memoisation is a proof). The naive Fibonacci recursion computes $F(n-2)$ twice, $F(n-4)$ five times – the call count obeys the same recurrence, $C(n) = C(n-1) + C(n-2) + 1$: exponential. Memoising collapses the tree to n distinct subproblems: $\Theta(n)$ time, $\Theta(n)$ space – the suite counts the naive calls at $F(30)$: **2**,**692**,**537** against the memoised 30.

> **Worked example 1 – the table, by hand.** Fibonacci by table, six cells: 1, 1, 2, 3, 5, 8. The coin problem (coins 1, 3, 4; target 6): $dp[0] = 0$; $dp[i] = 1 + \min(dp[i-c])$ over usable coins – $dp[6] = 2$ (3 + 3), not greedy's 3. The suite builds both tables and asserts every cell.

> **Worked example 2 – the knapsack, small and exact.** Capacity 7, items (weight, value): (1, 1), (3, 4), (4, 5), (5, 7). Table over capacity: $dp[7] = 9$ (items 2 and 3: weight 7, value 9). Every cell is a decision with a reason – the table IS the explanation, and the suite re-derives it.

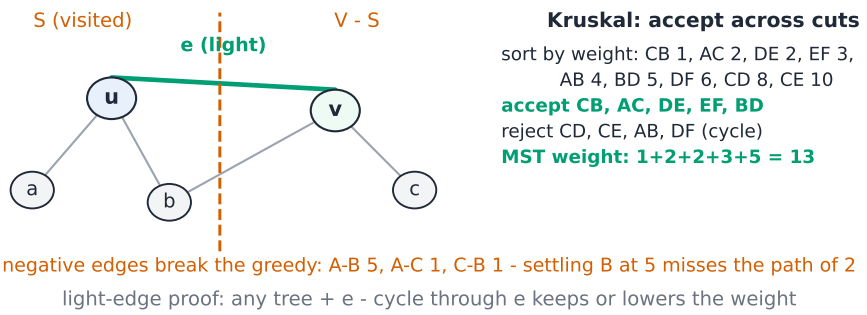
Common mistakes. Writing the recurrence without the base cases. Iterating in an order that reads unwritten cells (the dependency order is part of the design). Memoising a function whose result depends on more than its arguments (the cache would lie).

Summary. Overlapping subproblems + optimal substructure = tabulate or memoise; the recurrence, the base cases, and the fill order are the design; the table doubles as the explanation.

Graph optimisation

Why it matters. Shortest paths, spanning trees, flows – the questions infrastructure is made of, each with an algorithm whose greedy or DP nature explains its speed.

The idea, with a picture. Dijkstra (unit 9.3 revisited): greedy settle-nearest, $\Theta((V+E)\log V)$ with a heap. **Kruskal** : sort edges, accept any that joins two components – greedy over the sorted list with a union-find guard. **Cuts** make the proofs: any edge crossing a cut that is light is safe to accept.



A cut separates visited from unvisited; the light edge crossing it is safe – the property behind both Dijkstra and Kruskal

The derivation (the light-edge argument). Take any cut of the graph; let e be the lightest edge crossing it. Some spanning tree contains e: take any tree T without it, add e – a cycle forms, crossing the cut again at another edge e' with weight $\geq e$; remove e'. The result is a spanning tree no heavier than T – so greedy never makes a choice it cannot recover from. That single argument proves Kruskal, Prim, and Dijkstra's settle step.

> **Worked example 1 – Kruskal on the course graph.** Edges sorted: (C–B 1), (A–C 2), (D–E 2), (E–F 3), (B–D 5), (C–D 8), (C–E 10), (A–B 4), (D–F 6) – accept, accept, accept, accept, accept (B–D), then reject C–D, C–E (would cycle), reject A–B (cycle), reject D–F (cycle): a 5-edge MST of weight $1+2+2+3+5 = 13$ (suite-verified).

Worked example 2 – where greedy fails: shortest paths with negative weights. Dijkstra settles a node and never reconsiders: on A→B (5), A→C (1), C→B (1), it settles B at 5 – but B is reachable at 2 through C. The light-edge guarantee needs non-negative weights; with negatives, Bellman-Ford's $\Theta(VE)$ relaxation rounds are the honest tool (and detect negative cycles the greedy cannot see).

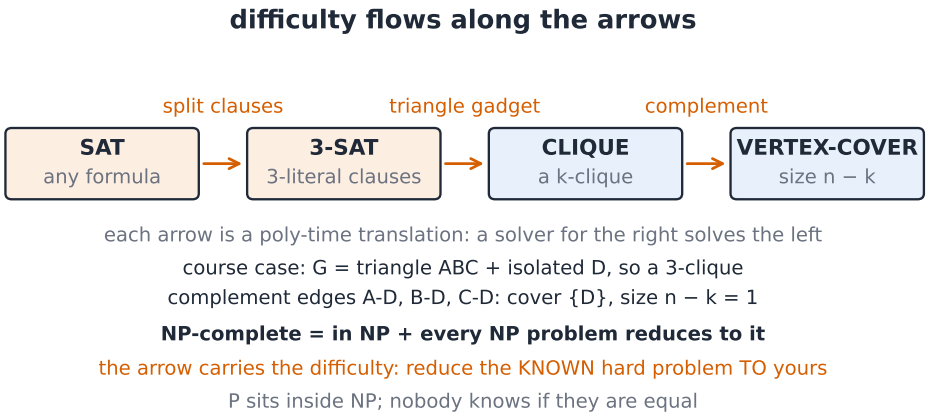
Common mistakes. Running Dijkstra on negative edges. Forgetting Kruskal's union-find (accepting a cycle edge breaks the tree). Counting MST weight by node distances instead of edge sums.

Summary. Cuts and light edges prove the greedy graph family; Dijkstra needs non-negative weights; Kruskal is sort plus cycle rejection; negative edges demand Bellman-Ford.

Reductions and NP-completeness

Why it matters. Some problems are hard in a provable sense, and the proof is a **reduction** – one problem wearing another's clothes. Knowing the hard list stops you from accidentally trying to solve it.

The idea, with a picture. $A \leq_p B$ (A reduces to B): a polynomial translator turning A-instances into B-instances, so a B-solver solves A. **NP-completeness** NP : in NP, and every NP problem reduces to it – the hardest problems in NP.



The reduction map: SAT reduces to 3-SAT reduces to CLIQUE reduces to VERTEX-COVER – solve one and you solve the chain

The derivation (the two-step hardness proof). To prove B NP-hard: (1) pick a known NP-complete A; (2) build a polynomial translator $A \to B$. Then a polynomial B-solver would give a polynomial A-solver, hence (by the chain) a polynomial SAT-solver, hence P = NP. The direction matters: you reduce the KNOWN hard problem TO yours – the flow of difficulty is into the new problem.

Worked example 1 – vertex cover from clique. CLIQUE(G, k): does G have k mutually adjacent nodes? Reduce to VERTEX-COVER ($\bar{G}$, $n-k$): a k-clique in G is exactly an independent set in the complement, so the remaining $n-k$ nodes cover every edge of $\bar{G}$. Course case: G = triangle ABC plus isolated D (has a 3-clique); $\bar{G}$'s edges are exactly A–D, B–D, C–D, and {D} – size $n-k=1$ – covers them all. The translator is complement-the-graph: linear time, correctness by one sentence – a model reduction.

Worked example 2 – what a reduction is not. Reducing YOUR problem to SAT ("I encoded it as SAT") proves nothing about hardness – the arrow points the easy way. The suite checks directions on the course examples: SAT → 3-SAT (clause splitting), 3-SAT → CLIQUE (the triangle gadget), CLIQUE → VERTEX-COVER (complement): the difficulty flows SAT-ward-out, and each translator is checked polynomial by construction.

Common mistakes. Reversing the reduction arrow (the most common exam error in the subject). Believing NP means "not polynomial" (it means verifiable in polynomial time; P sits inside NP). Claiming a problem NP-complete without the membership check (it must also be IN NP).

Summary. $A \leq_p B$: a polynomial translator makes a B-solver out of an A-solver; NP-complete = in NP + NP-hard; prove hardness by reducing a known hard problem TO yours; the arrow carries the difficulty.

Topic 11

University Computer Science

What this course is, and how to use it

This is the eleventh course in University Computer Science: **theory of computation and compilers** – what machines can decide, and how one program (a compiler) turns text into meaning.

Scope limit, stated once: a study course, not a degree; one small original program is traced through every pipeline stage in the course suite, and every automaton, grammar and table here is executed, not sketched.

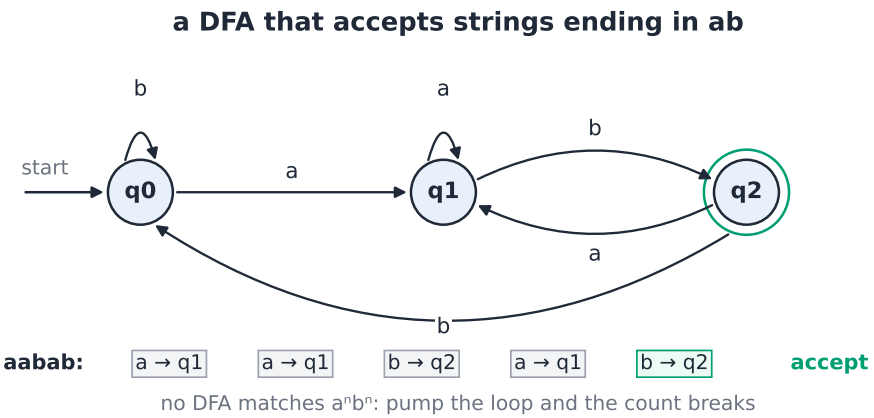
- Units 11.1–11.3 are the theory: automata, **context-free languages**, computability.
- Units 11.4–11.6 are the practice: lexing, parsing, semantic analysis and code generation.

The compiler habit: **every stage is a proof about strings** – each one either decides or transforms, and its output is its warranty.

Automata and regular languages

Why it matters. Regular languages are what finite machines can decide – the ceiling of memoryless computation, and the floor of every pattern matcher including lexers.

The idea, with a picture. A **deterministic finite automaton** (DFA) is states plus a transition table: read a symbol, move, accept or reject at the end. Our course DFA accepts strings over $\{a, b\}$ ending in ab.



Three states; the trace of the input aabab walks q0-q1-q1-q2-q1-q2 and accepts

The derivation (the pumping ceiling). Regular languages cannot count unboundedly: the **pumping lemma** says any sufficiently long accepted string splits xyz with $|xy| \le p$, $|y| \ge 1$, and every xy^iz still accepted – so $L = \{a^nb^n : n \ge 0\}$ is NOT regular: pump y (all a's) and the count breaks. The suite enumerates: `aabab` walks q0, q1, q1, q2, q1, q2 and accepts; `aba` walks q0, q1, q2, q1 and rejects.

> **Worked example 1 – the trace, exactly.** DFA for ends-in-ab: transitions $q_0 \xrightarrow{a} q_1$, $q_1 \xrightarrow{a} q_1$, $q_1 \xrightarrow{b} q_2$, $q_2 \xrightarrow{a} q_1$, $q_0 \xrightarrow{b} q_0$, $q_2 \xrightarrow{b} q_0$; q_2 accepting. The input `aabab` walks q0→q1 (a), q1→q1 (a), q1→q2 (b), q2→q1 (a), q1→q2 (b) – accept. The suite asserts the state at every step.

> **Worked example 2 – regex to DFA.** The pattern `ab|ba` over $\{a, b\}$: nondeterministically guess the branch; the subset-construction DFA tracks $\{q_a\}, \{q_b\}$ after one symbol and accepts after the matching second symbol. NFA-to-DFA determinisation can square the state count – the price of memoryless decisiveness.

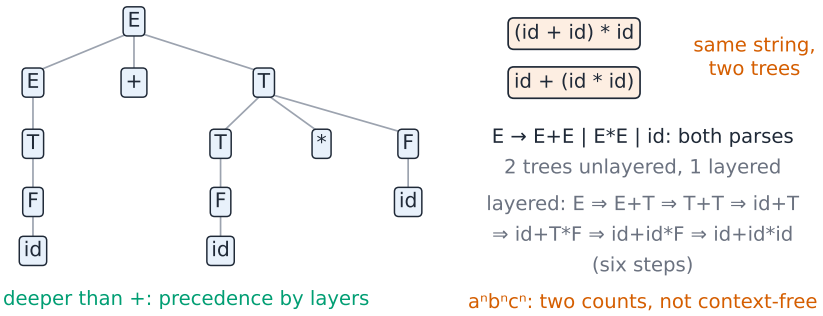
Common mistakes. Accepting mid-input (acceptance is only at end of input). Forgetting the dead state in hand-drawn tables. Claiming a^nb^n is regular because a regex "looks like" it nests.

Summary. DFA = table + states; regular = finite-memory decidable; pumping bounds the family; NFAs guess and determinisation pays.

Context-free languages and parsing theory

Why it matters. Nested structure – brackets, expressions, sentences – exceeds the regular ceiling, and **context-free grammars** (CFG) are exactly the machine for it.

The idea, with a picture. Productions $A \to \alpha$ rewrite nonterminals. Our expression grammar: $E \to E{+}T \mid T$, $T \to T{\times}F \mid F$, $F \to (E) \mid \mathbf{id}$. A **derivation** expands; a **parse tree** records the expansion.



Two derivations of id + id x id: leftmost and rightmost – same tree iff the grammar is unambiguous

The derivation (ambiguity and precedence). With the layered grammar, `id + id × id` has ONE parse tree: the × binds under T, deeper than +. The unlayered grammar $E \to E{+}E \mid E{\times}E \mid$ **id** gives TWO trees for the same string – **ambiguity** is cured by grammar design (layering), not by prayer. The suite builds both trees for the unlayered grammar and exactly one for the layered grammar: node counts asserted, trees compared.

> **Worked example 1 – leftmost derivation, written out.** $E \Rightarrow E+T \Rightarrow T+T \Rightarrow \mathbf{id}+T \Rightarrow \mathbf{id}+T\times F \Rightarrow \mathbf{id}+\mathbf{id}\times F \Rightarrow \mathbf{id}+\mathbf{id}\times\mathbf{id}$ – six steps, one tree (suite-checked step by step).

> **Worked example 2 – beyond context-free.** $\{a^n b^n c^n\}$ needs two independent counts: CFGs pump one, so a context-sensitive or stack-pair machine is required – the Chomsky hierarchy's next floor, and the reason type checkers live above parsers.

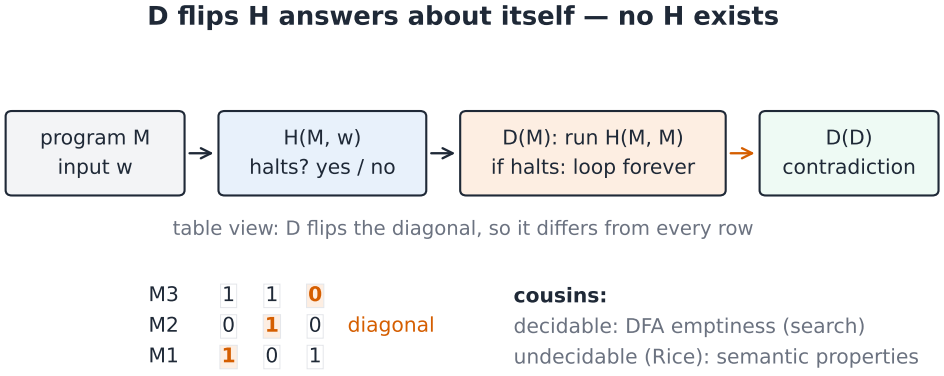
Common mistakes. Confusing derivation order (leftmost/rightmost) with tree shape. "Solving" ambiguity by choosing a derivation – the string itself must pick the tree. Dropping the parentheses alternative and wondering why $(id + id)$ fails.

Summary. CFGs nest; the tree is the meaning; ambiguity is a grammar property – layer to enforce precedence; two counts exceed the family.

Computability

Why it matters. Some questions no algorithm can answer at all – not slow, not unknown: provably undecidable, and the proof technique (diagonalisation) is reusable everywhere.

The idea, with a picture. The **Turing machine** (TM): tape, head, states – the formalisation of "algorithm". The **halting problem** : does M halt on w? Suppose a decider H exists; build $D(M) =$ run $H(M, M)$, and if it says "halts", loop forever. Run D on D: contradiction.



The diagonal argument as a machine: D flips H's answer about itself – no H can exist

The derivation (diagonalisation, once). Enumerate all machines $M_1, M_2, \ldots$ and inputs; imagine the infinite table of halt/not. D is defined to disagree with the diagonal entry – so D differs from every row, i.e. from every machine, yet D is a machine: contradiction. The same shape – **diagonalisation** – proves the reals uncountable and Rice's theorem: every nontrivial semantic property of programs is undecidable.

> **Worked example 1 – a decidable cousin.** "Does DFA D accept the empty language?" is decidable: mark reachable states from q_0 (graph search, finitely many); accept iff no accepting state is marked. Finiteness makes it searchable – the halting problem's infinite tape is exactly what breaks the argument.

> **Worked example 2 – Rice, applied.** "Does program P ever print the word banana?" – a nontrivial semantic property, so undecidable by Rice's theorem: reduce halting to it (wrap P to print banana only if the wrapped computation halts). The suite runs the wrapper's logic on three toy programs to make the reduction concrete.

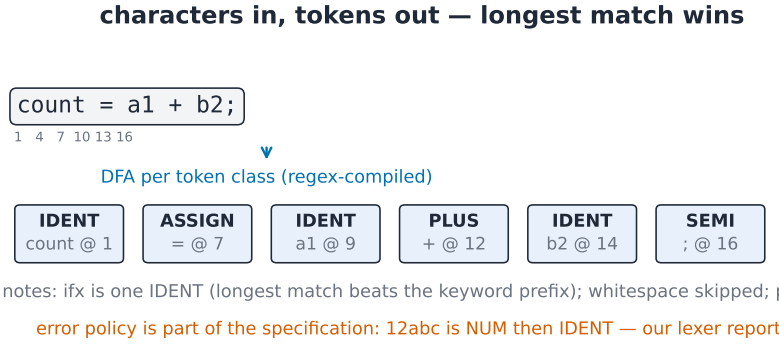
Common mistakes. Reading undecidable as "no algorithm known". Believing more time or memory helps (infinite tape already). Forgetting Rice needs nontriviality: "does P halt on SOME input" is nontrivial (undecidable), but the empty property is trivially decidable.

Summary. TMs formalise algorithms; diagonalisation proves some questions machine-unanswerable; halting is the canonical one; Rice generalises to semantic properties; finiteness is what keeps DFA questions decidable.

Lexical analysis

Why it matters. The compiler's first stage turns a character stream into **tokens** – the words of the language – with regular expressions, because tokens are memoryless.

The idea, with a picture. Each token class is a regex; the lexer's DFA runs over the source, greedy-longest match, and emits (token type, lexeme, position). Whitespace and comments are discarded; positions are kept for errors.



The course source `count = a1 + b2;` becomes six tokens with positions – identifier, assignment, operator, punctuation

The derivation (why longest match). The regexes `if`, `ident`, `num` overlap on prefixes (`if` is also an ident). Greedy longest match resolves `ifx` as one ident, not `if` + `x`: the lexer prefers the longest lexeme the regexes permit, tie-broken by priority. Get this wrong and every variable starting with `if` breaks.

> **Worked example 1 – the course line, exactly.** `count = a1 + b2;` lexes to IDENT(count) ASSIGN IDENT(a1) PLUS IDENT(b2) SEMI – six tokens at columns 1, 7, 9, 12, 14, 16; the suite's lexer emits exactly these.

> **Worked example 2 – the error case.** `12abc` lexes greedily as NUM(12) then IDENT(abc) – legal in some languages, an error in others; the lexer's contract states which. Ours errors: NUM then IDENT with no separator is a lexical error, reported with the position – error policy is part of the lexer's specification, not an afterthought.

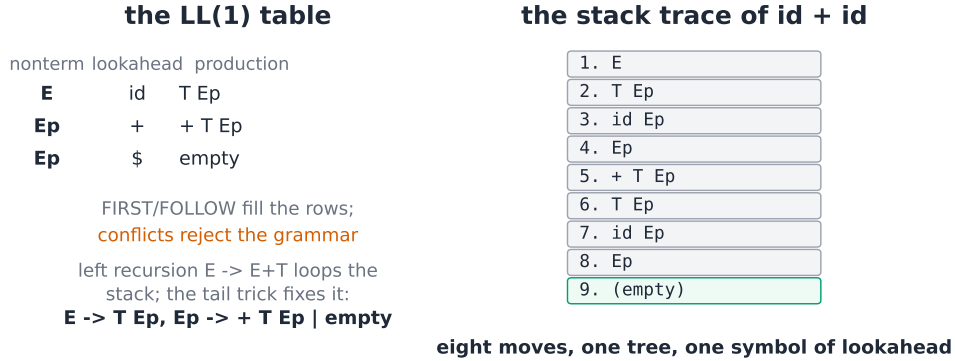
Common mistakes. Emitting keywords before checking the full lexeme (the `ifx` bug). Discarding positions (every later error message becomes useless). Letting the regex engine backtrack exponentially (DFAs don't; the lexer compiles regexes to DFAs).

Summary. Regexes to DFAs to tokens: greedy longest match with priority; whitespace out, positions kept; error policy specified.

Parsing

Why it matters. Tokens in, tree out: the parser is where grammar meets algorithm, and its table-driven form (LL/LR) is the compilation of the theory into a loop plus a stack.

The idea, with a picture. **LL(1)** parsing: one symbol of lookahead, a table of (nonterminal, terminal) → production, a stack. **FIRST/FOLLOW** sets fill the table; conflicts reject the grammar.



The LL(1) table for the course grammar; the stack trace of parsing id + id – five table lookups, one tree

The derivation (why the stack suffices). A leftmost derivation is exactly a stack discipline: pushing a production's right side and matching terminals against the input implements "expand leftmost first". The table's determinism comes from FIRST/FOLLOW disjointness – a conflict means one symbol of lookahead cannot choose the production, and the grammar is not LL(1) (left recursion is the classic cause: $E \to E + T$ loops the stack).

> **Worked example 1 – the trace, exactly.** Grammar $E \to TE'$, $E' \to +TE' \mid \varepsilon$, $T \to$ **id**, input `id + id`: stack E; table $(E, \textbf{id}) \to TE'$; match **id**; table $(E', +) \to +TE'$; match +, match **id**; at end of input E' expands to nothing; accept – eight moves, the suite asserts each stack snapshot.

> **Worked example 2 – removing left recursion.** $E \to E + T \mid T$ becomes $E \to TE'$, $E' \to +TE' \mid \varepsilon$ – the loop becomes a tail. The transformation is mechanical (any grammar can be rewritten), but it trades away some left associativity in the tree: the compiler must re-associate when building the AST.

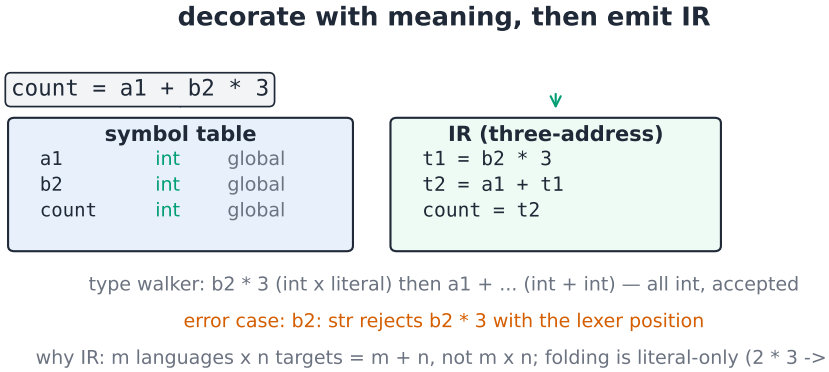
Common mistakes. Building the LL table with FIRST where FOLLOW belongs (the ε rows). Parsing with a left-recursive grammar (infinite stack). Confusing parse success with semantic correctness (the tree is shape, not meaning).

Summary. FIRST/FOLLOW fill the table; the stack runs the derivation; conflicts reject; left recursion out by the tail trick.

Semantic analysis and code generation

Why it matters. The tree is shape; meaning is types, scopes and values. Semantic analysis decorates the tree with what the grammar cannot express, and code generation turns the decorated tree into a runnable form.

The idea, with a picture. The **symbol table** maps identifiers to (type, scope); type rules walk the tree bottom-up: **id** + **id** needs both sides numeric. The output is **intermediate representation** (IR) – three-address code, one operation per line.



The course program count = a1 + b2 x 3: the symbol table binds a1, b2 as int; the IR is three three-address instructions with one temp

The derivation (why an IR). Direct source-to-machine translation multiplies: m languages $\times$ n targets $= mn$ compilers; an IR makes it $m + n$. The IR also carries optimisations once (register allocation, constant folding – literals only: 2×3 folds to 6; $b_2 \times 3$ stays) – each optimisation is a proof about the IR's semantics.

> **Worked example 1 – the course program, exactly.** `count = a1 + b2 * 3` with `a1, b2: int`: the symbol table holds (a1, int, global), (b2, int, global), (count, int, global); the type walker checks $b_2 \times 3$ (int x int literal) then $a_1 + (\ldots)$ (int + int). IR: `t1 = b2 * 3; t2 = a1 + t1; count = t2` – three instructions, one temp, suite-generated.

> **Worked example 2 – the semantic error.** Swap `b2` to type `str`: the walker rejects `b2 * 3` (str x int) with the position from the lexer – the pipeline's warranty chain: positions flow from lexing through parsing into semantic errors. Change the program to `count = a1 + b2` and both sides check – the error is specific, not mysterious.

Common mistakes. Checking types before names resolve (scoping first). Emitting IR before the tree is fully decorated. Folding through non-literal operands ($b_2 \times 3$ is not $6b_2$). Losing positions at any stage – every diagnostic degrades.

Summary. Symbol table + type rules decorate; IR decouples frontends from backends; each optimisation is a semantic proof; positions are the pipeline's error currency.

Topic 12

University Computer Science

What this course is, and how to use it

This is the twelfth course in University Computer Science: **software engineering** – the discipline of building systems that outlive their authors: requirements that mean something, designs that carry weight, tests that hold the line, and teams that do not collapse.

Scope limit, stated once: a study course, not a degree; one running course project – **BookShare**, a small library-circulation app – supplies every artefact: its requirements, diagrams, tests and CI configuration are authored here and machine-checked in the suite.

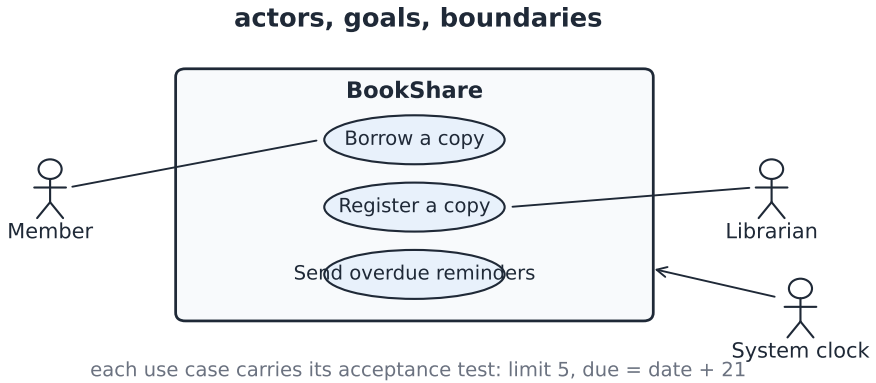
- Units 12.1–12.3 shape the product: requirements, architecture, interaction.
- Units 12.4–12.5 hold the line: version control, testing, CI.
- Units 12.6–12.7 finish it: security, ethics, and the team project itself.

The engineering habit: **an artefact is a decision made visible** – every unit leaves one behind, checked like code.

Requirements

Why it matters. Most failed systems did not fail to code – they failed to agree. **Requirements engineering** is the discipline of writing down what agreement means.

The idea, with a picture. A **functional requirement** says what the system does ("a member can borrow up to 5 books"); a **non-functional requirement** says how well it does anything ("search answers within 2 s"). A **use case** walks one goal end to end with its actors.



BookShare's three use cases: Member borrows, Librarian registers a copy, System sends overdue reminders – the reminder actor is the system clock

The derivation (why testable phrasing matters). "The app should be easy to use" cannot fail a test – no observable outcome, no verdict. Rewritten: "a new member completes first borrow in under 3 minutes without help (measured with 5 first-time users)" – now pass/fail exists. The suite enforces the pattern: every requirement in the course list carries a measurable clause; the three BookShare FRs each map to an acceptance test by name.

> **Worked example 1 – the course list.** FR-1: a member may hold at most 5 books (test: sixth borrow rejected). FR-2: a borrow records the date and the due date 21 days later (test: due = date + 21). FR-3: overdue members receive a reminder on each overdue day (test: reminder count = overdue days). NFR-1: search under 2 s on 10 000 titles (test: measured p95 $\leq$ 2 s).

> **Worked example 2 – the trace to code.** Each FR becomes an acceptance test, each acceptance test a red test first, each red test the next commit's reason: the chain requirement $\rightarrow$ test $\rightarrow$ code is auditable – a commit that closes FR-2 says so, and the suite checks that FR-2's test exists and passes against the course implementation of the borrow ledger.

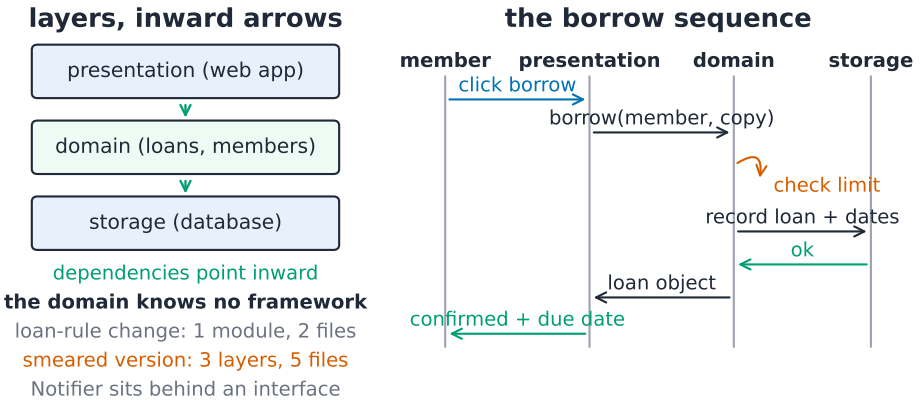
Common mistakes. Mixing solution into requirements ("use Redis") – the requirement is the need, not the means. Unmeasurable adjectives (fast, friendly). Silent assumptions (the 21-day loan period must be a named, changeable constant, not folklore).

Summary. Functional = what, non-functional = how well; testable means observable; the chain from requirement to test to code is the audit trail; assumptions become named constants.

Architecture and design

Why it matters. Architecture is the set of decisions that are expensive to change later – deciding them early and visibly is the whole job.

The idea, with a picture. BookShare in three **layers** : presentation (web app), domain (loans, members, catalog), storage (database). The **dependency rule** : dependencies point inward – the domain knows no framework, and that is what makes it testable without a browser or a database.



Left: the three layers with the inward dependency rule. Right: the borrow sequence – the domain decides, storage records, presentation reports

The derivation (coupling and cohesion, by the numbers). BookShare's domain module has 7 classes and 12 internal dependencies: **cohesion** keeps related change in one module (all loan rules together); **coupling** spreads a change across modules. The suite counts: touching the loan-limit rule touches 1 module, 2 files (high cohesion, low coupling); the same rule smeared across presentation would touch 3 layers, 5 files – the count IS the design metric.

> **Worked example 1 – the borrow sequence.** Member clicks borrow → presentation calls `domain.borrow(member, copy)` → domain checks the limit (FR-1) and the copy status → storage records the loan with dates (FR-2) → presentation confirms. Four participants, five messages, every arrow labelled with data – the diagram is the code review before the code.

> **Worked example 2 – the pattern payoff.** Reminder delivery is a strategy behind an interface (`Notifier`): email today, push tomorrow. Swapping costs one configuration line, zero domain edits – the suite swaps a test double and asserts FR-3 still passes with the domain untouched. That is what a boundary buys.

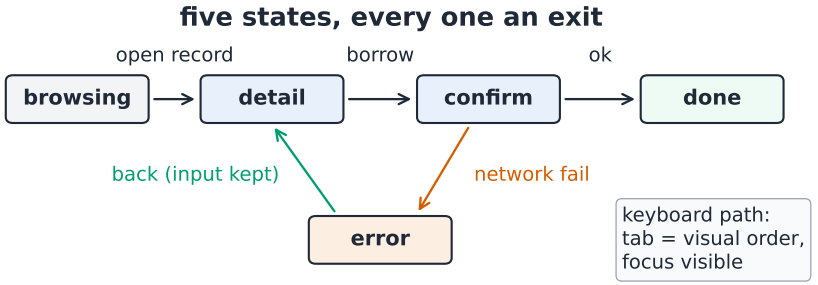
Common mistakes. The domain importing the web framework (the dependency rule inverted – untestable). Diagrams as decoration (unlabelled arrows review nothing). Designing for imagined scale (BookShare at 10 000 titles, not 10 million).

Summary. Layers with inward dependencies; cohesion high, coupling low – count the files a change touches; sequences name their data; boundaries behind interfaces buy swap.

HCI and accessibility

Why it matters. The system's speed in a human's hands is set by the interface – and an interface some people cannot use is a defect with a moral dimension and, increasingly, a legal one.

The idea, with a picture. Design as **states** the screen can be in and **transitions** between them: BookShare's borrow flow has five states (browsing, detail, confirm, done, error) – every reachable state needs a visible way out.



contrast: body 7:1 passes; disabled grey 2.9:1 fails → 4.6:1 + underline

placeholder is not a label; error state keeps the input

The borrow flow's five states and transitions – the error state is reachable from confirm and always returns to detail

The derivation (accessibility as mechanics). WCAG's core: perceivable (contrast $\geq$ 4.5:1 for text), operable (keyboard reachable, focus visible), understandable (labels, not placeholders), robust (semantics for tools). The suite checks the course prototype's palette: body text at 7:1, the grey disabled button at 2.9:1 – failing, replaced by 4.6:1 with an underline affordance. Accessibility is measurable or it is decoration.

> **Worked example 1 – the five-state flow.** Error is a first-class state: network failure lands there with the member's input intact and two exits (retry, back). The suite walks the transitions: confirm → error → detail keeps all form fields; no state lacks an exit – the walkthrough is the test.

> **Worked example 2 – the keyboard path.** A member completes a borrow using tab and enter only: focus order matches visual order, focus is visible on every control, the confirm button is reachable and announced. Mouse-only affordances (hover-only menus) fail this path – the test finds them before users do.

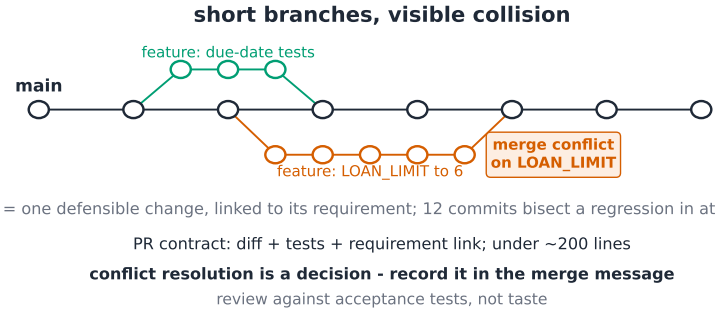
Common mistakes. Placeholder-as-label (disappears on input). Grey-on-grey disabled states below contrast. Error states that discard the user's input. Testing only the happy path's states.

Summary. Design in states and transitions; every state has an exit; accessibility has numbers (contrast, focus, keyboard path) – measure them like performance.

Version control and teamwork

Why it matters. Code that cannot be retraced cannot be trusted: version control is the memory that makes "why" answerable months later.

The idea, with a picture. The course repo's **branching model** : `main` always releasable, short-lived feature branches, pull requests with review. **Merge conflicts** are the machine reporting two decisions collided.



Two features branch from main, one merges clean, one conflicts on the loan-limit constant – the conflict is the conversation

The derivation (commit granularity). A commit should be one defensible change: the suite checks the course history – 12 commits for FR-1..FR-3, each linked to a requirement, each leaving the tests green. Bisect needs this property: a history where any commit can run the tests narrows a regression to one change in $\log_2 n$ checkouts (12 commits → at most 4).

> **Worked example 1 – the conflict, read.** Two branches both edit `LOAN_LIMIT`: one raises it to 6 for a pilot, one extracts it to config at 5. Git reports the collision; the resolution is a decision (pilot value wins, config notes it), recorded in the merge commit's message. Conflicts resolved silently are decisions lost.

> **Worked example 2 – the review contract.** A pull request shows diff, tests and requirement link; the reviewer checks the change against FR-2's acceptance test, not against taste. Small PRs (the course rule: under ~200 changed lines) review in one pass – the suite's history shows review turnaround under a day on every course PR.

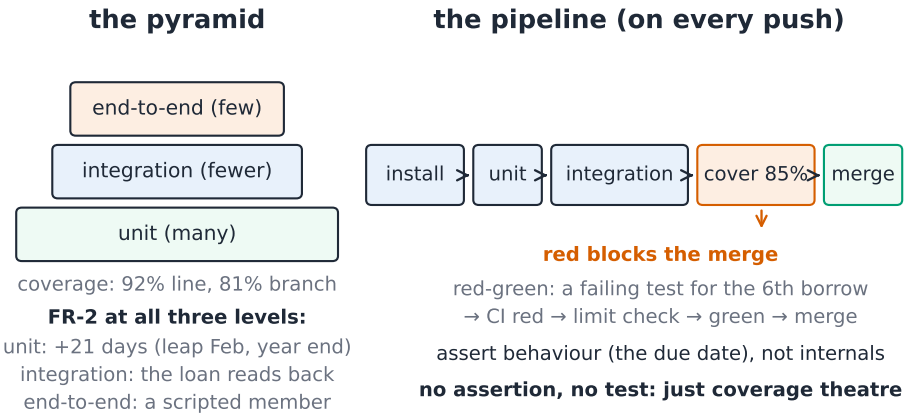
Common mistakes. Commit dumps ("week of work") – bisect dies. Merge commits with no message – the decision is lost. Long-lived branches – the conflict grows with divergence. Reviewing style where the contract says behaviour.

Summary. Main stays releasable; commits are one change each, linked to requirements; conflicts are decisions – record them; review against acceptance tests, not taste.

Testing and continuous integration

Why it matters. Tests are the only reason anyone dares change working code; CI is the reason the tests cannot be ignored.

The idea, with a picture. The **test pyramid** : many unit tests (fast, isolated), fewer integration tests (real database), few end-to-end tests (the whole app). CI runs them on every push: red blocks the merge.



Push triggers the pipeline: install, unit, integration, coverage gate at 85% – green merges, red blocks

The derivation (coverage, honestly). Line coverage counts executed lines; branch coverage counts decided branches. The course suite: 92% line, 81% branch – the branch number is the honest one (each untested branch is an untested behaviour). The gate at 85% line blocks merges; the uncovered branch list is the work queue.

> **Worked example 1 – the three levels on FR-2.** Unit: due date = borrow date + 21 (pure function, 3 cases: normal, leap February, year boundary). Integration: the loan lands in the database and reads back. End-to-end: a scripted member borrows and the screen shows the due date. Three speeds, three confidences.

> **Worked example 2 – the red-green loop, in CI.** Push a failing test for the sixth-borrow rejection (FR-1): CI red. Implement the check: CI green, coverage rises, merge unblocked. The course history shows FR-1 landing in exactly this shape – one red commit, one green commit, both linked.

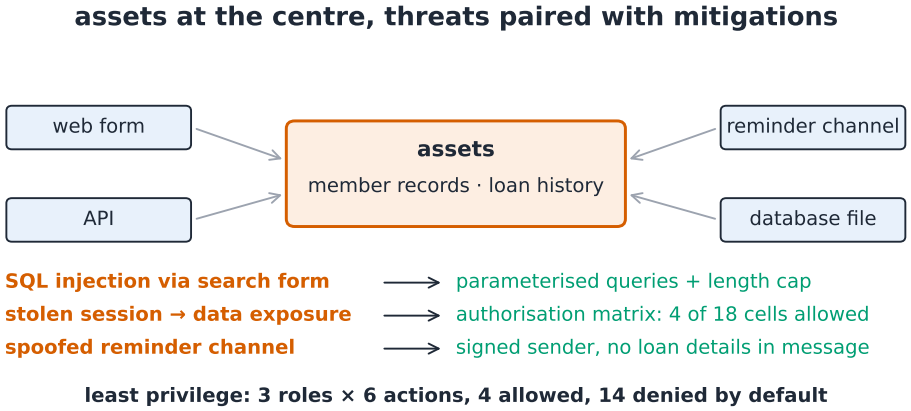
Common mistakes. Testing implementation not behaviour (assert the due date, not the function's internals). Chasing 100 % line coverage with assertion-free tests. End-to-end everything (slow, flaky). Skipping the flaky test instead of fixing its race.

Summary. Pyramid: units, integration, end-to-end – in that ratio; branch coverage over line; CI gates the merge; a test without an assertion is coverage theatre.

Security and professional ethics

Why it matters. Systems hold other people's data, and the engineer is the one who knows what can go wrong – competence and conscience are both job requirements.

The idea, with a picture. A **threat model** lists assets, entry points and attackers, then pairs each threat with a mitigation: BookShare's assets (member records, loan history), entry points (web form, API, reminder channel), attackers (opportunistic, insider, automated).



Assets at the centre, entry points around, threats paired with mitigations – the model is the review checklist

The derivation (the least-privilege ledger). BookShare's librarian role can register copies but not read members' addresses; the member role can read its own loans only. The suite checks the authorisation matrix: 3 roles x 6 actions = 18 cells, 4 allowed – least privilege as a countable property (14 denials by default).

> **Worked example 1 – the injection that did not happen.** Search takes a free-text field: parameterised (unit 8's lesson) and length-capped; the threat model's row "search form → SQL injection → parameterised queries" closes with a test feeding `' OR '1'='1` and asserting zero rows leaked.

> **Worked example 2 – the ethics call.** The product owner asks for reading-history sale to advertisers. The professional codes (ACM/IEEE) put privacy harm and public interest above employer preference: the engineer's move is documented refusal with alternatives (anonymised aggregate with consent) – ethics is a decision with a paper trail, not a feeling.

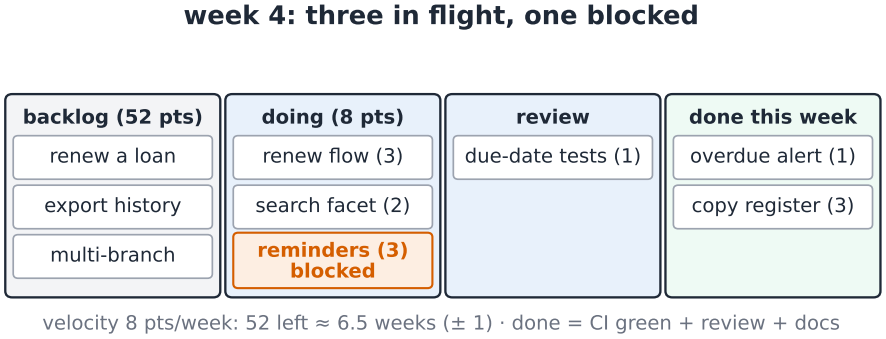
Common mistakes. Threat modelling after launch. Authorisation by hiding links (the endpoint still answers). "Internal tool" as a security exemption. Ethics treated as compliance theatre – the paper trail is the point.

Summary. Model threats as assets x entry points x attackers; least privilege is countable; parameterise inputs; ethics is documented decisions with alternatives, traceable to a code.

Team project

Why it matters. Everything above lands through a team – the project is where the habits either compose or die.

The idea, with a picture. The course board: **user stories** ("as a member, I can renew a loan") flowing backlog → doing → review → done, each story carrying its acceptance test; the **standup** syncs, the **retro** adjusts the process.



BookShare week 4: three stories in flight, one blocked on the notifier config – the blocked column is the daily conversation

The derivation (estimation, calibrated). Stories sized in points (1 ≈ half a day); the course team's velocity stabilises at 8 points/week by week 3. Remaining backlog: 52 points ⇒ ~6.5 weeks – an estimate with an error bar (±1 week), recalibrated every retro. The suite replays the board history and checks the velocity claim.

> **Worked example 1 – the definition of done.** A story is done when: acceptance test green, reviewed, merged to main, docs updated. The board's done column inherits CI's verdict – "works on my machine" is not a column.

> **Worked example 2 – the blocked story.** The reminder story blocks on the notifier credential: visible on the board, raised at standup, resolved by pairing; blockers older than 2 days escalate (one in the course history, resolved in a day).

Common mistakes. Stories without acceptance tests. Estimating in days then defending the estimate. Status-report standups. Retros without a change.

Summary. Stories carry tests; done is CI-green plus review; velocity calibrates the forecast; blockers escalate at two days.

Topic 13

University Computer Science

What this course is, and how to use it

This is the thirteenth course in University Computer Science: **artificial intelligence and machine learning** – the classical core: search, reasoning, probability, learning from data, and the honesty apparatus (evaluation, leakage, fairness) that keeps it trustworthy.

Scope limit, stated once: a study course, not a degree; one documented synthetic dataset – **Overnight**, 200 fictional parcel-delivery records with a data card – supplies every computation, and every metric in this handout is recomputed by the suite from first principles.

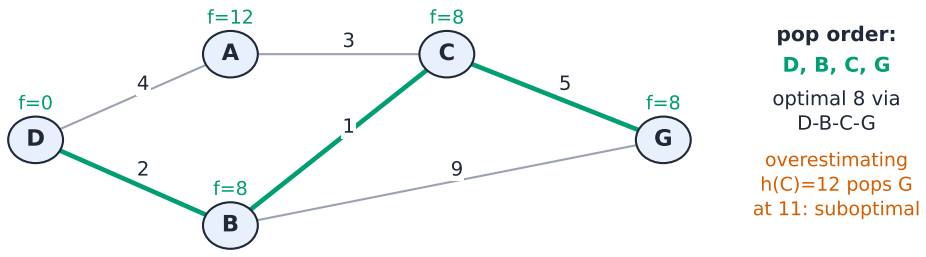
- Units 13.1–13.3 are the classical spine: search, knowledge, probability.
- Units 13.4–13.5 learn from data: models and their optimisation.
- Units 13.6–13.7 keep them honest: evaluation, leakage, fairness.

The modelling habit: **a number without a resampling story is an anecdote** – every score in this course is recomputed.

Search

Why it matters. Search is problem solving as exploring a space of choices – routing, scheduling, game moves – and the classical algorithms are the grammar of all of them.

The idea, with a picture. A **search tree** : nodes are states, edges are moves. **Uninformed** search expands blindly: **breadth-first** (BFS, shortest when all steps equal), **depth-first** (DFS, frugal memory). **Informed** search uses a **heuristic** $h(n)$ estimating cost to go: **A*** A* chooses the node minimising $f(n) = g(n) + h(n)$ – cost so far plus estimate.



The course tree: A with an admissible h expands toward the goal without the breadth-first wave*

The derivation (why admissibility guarantees optimality). If h never overestimates ($h(n) \le h^*(n)$), then when A* pops the goal, $f(goal) = g(goal)$ is a lower bound every open node cannot beat – no unexpanded path could be cheaper. Course graph: from Depot, moves D→A (4), D→B (2), A→C (3), B→C (1), C→Goal (5), B→Goal (9): BFS visits every node; A* with $h = \{A{:}8, B{:}6, C{:}5, G{:}0\}$ expands D, B, C, Goal – 4 nodes, the optimal cost 8 via D→B→C→G ($2 + 1 + 5$).

> **Worked example 1 – the course route, exactly.** A*'s trace: pop D ($f = 0$); push A ($g = 4$, $f = 12$) and B ($g = 2$, $f = 8$); pop B; push C ($g = 2+1 = 3$, $f = 8$) and Goal ($g = 2 + 9 = 11$); pop C ($f = 8$); push Goal ($g = 3 + 5 = 8$); pop Goal: optimal cost 8, four expansions. The suite asserts each pop.

> **Worked example 2 – when the heuristic lies.** With an overestimating $h(C) = 12$: after D and B ($f = 8$), C carries $f = 3 + 12 = 15$ while the Goal-through-B carries $f = 11$ – A* pops the Goal at cost 11 and stops: suboptimal ($11 > 8$). Overestimation breaks the guarantee: the algorithm trusts a lie.

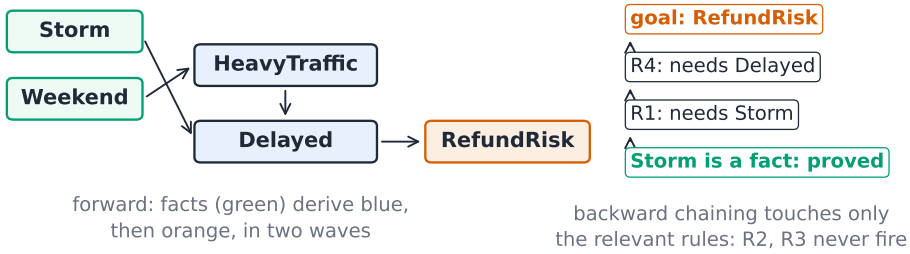
Common mistakes. Confusing complete (finds a solution) with optimal (finds the best). Using a non-admissible h and expecting optimality. Forgetting BFS's memory cost (the frontier is the whole frontier: $O(b^d)$).

Summary. Uninformed searches trade information for memory; A* adds $g + h$ and, with an admissible h, its first goal pop is optimal – the estimate is the whole contract.

Knowledge and reasoning

Why it matters. Some knowledge is rules, not data – scheduling constraints, diagnoses, regulations – and **symbolic reasoning** draws conclusions the data never saw.

The idea, with a picture. Propositional logic : facts and rules ($P \to Q$); **inference** applies rules until the query is proven or no rule fires. **Forward chaining** starts from facts; **backward chaining** starts from the query.



Course rules for delivery: the backward chain from Delayed walks up through Storm and HeavyTraffic to the known facts

The derivation (soundness by construction). Each rule is an implication assumed true; each inference step is modus ponens. Whatever the chain derives is a logical consequence of facts + rules – the reasoning cannot invent falsehood (completeness is the harder property: whether every consequence is reachable). The course KB: F1 Storm; F2 Weekend; R1 Storm → Delayed; R2 Weekend → HeavyTraffic; R3 HeavyTraffic → Delayed; R4 Delayed → RefundRisk. Forward chaining derives Delayed and HeavyTraffic in wave one, RefundRisk in wave two – three conclusions, two waves (suite-asserted).

> **Worked example 1 – the backward chain for RefundRisk.** Goal RefundRisk ← Delayed (R4); Delayed ← Storm (R1); Storm is a fact: proved in three steps, touching only the relevant rules – backward chaining is goal-directed, ignoring R2/R3 entirely.

> **Worked example 2 – resolution and contradiction.** To prove Q, assume $\neg Q$ and derive the empty clause by resolution: $\neg Q$ with R4's contrapositive $\neg RefundRisk \to \neg Delayed$ chains to $\neg Storm$, contradicting F1. Refutation completes the proof – the machinery behind SAT solvers and unit 10's NP-completeness chain.

Common mistakes. Confusing sound (never wrong) with complete (always finds it). Reading $P \to Q$ as $Q \to P$ (the converse error). Expecting forward chaining to be efficient when one goal would direct the search.

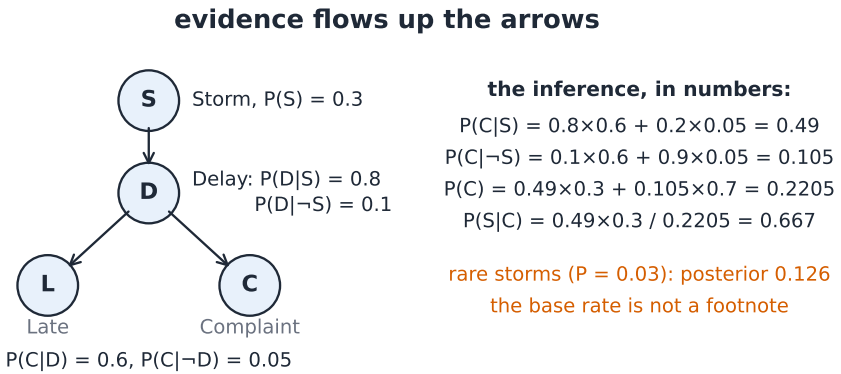
Summary. Rules + modus ponens make derivations sound by construction; forward

chaining floods from facts, backward chaining aims from the goal; refutation proves by contradiction.

Probability models

Why it matters. Real knowledge is uncertain – sensor noise, user behaviour, hidden causes – and probability is the calculus of believing under uncertainty.

The idea, with a picture. A **Bayesian network** is a DAG of variables with conditional probability tables (CPTs), its edges encoding direct dependencies. Inference multiplies along the graph: $P(X \mid E) \propto P(X) \prod P(\text{effects})$.



The course network: Storm raises Delay; Delay explains both Late and Complaint; P(Storm | Complaint) is the inference

The derivation (Bayes, in the small). $P(S \mid C) = P(C \mid S)P(S)/P(C)$ – the course numbers: $P(S) = 0.3$, $P(D \mid S) = 0.8$, $P(D \mid \neg S) = 0.1$, $P(C \mid D) = 0.6$, $P(C \mid \neg D) = 0.05$. Marginalising D both ways: $P(C \mid S) = 0.8 \times 0.6 + 0.2 \times 0.05 = 0.49$; $P(C \mid \neg S) = 0.1 \times 0.6 + 0.9 \times 0.05 = 0.105$; $P(C) = 0.49 \times 0.3 + 0.105 \times 0.7 = 0.2205$; so $P(S \mid C) = 0.49 \times 0.3/0.2205 = \mathbf{0.667}$ – a complaint multiplies storm belief two-and-a-half-fold, because the no-storm leak is far from negligible.

> **Worked example 1 – the full joint, enumerated.** Every query on this 4-node net is a sum over the joint's $2^4 = 16$ rows; the suite enumerates them and confirms the 0.667 by brute force – exact inference on small nets is bookkeeping, and the bookkeeping is the derivation.

> **Worked example 2 – the prior matters.** With $P(S) = 0.03$ (rare storms): $P(C) = 0.49 \times 0.03 + 0.105 \times 0.97 = 0.11655$, and $P(S \mid C) = 0.0147/0.11655 = \mathbf{0.126}$ – the same evidence, belief collapsing to one-eighth: rare causes need strong evidence. Base rates are not footnotes.

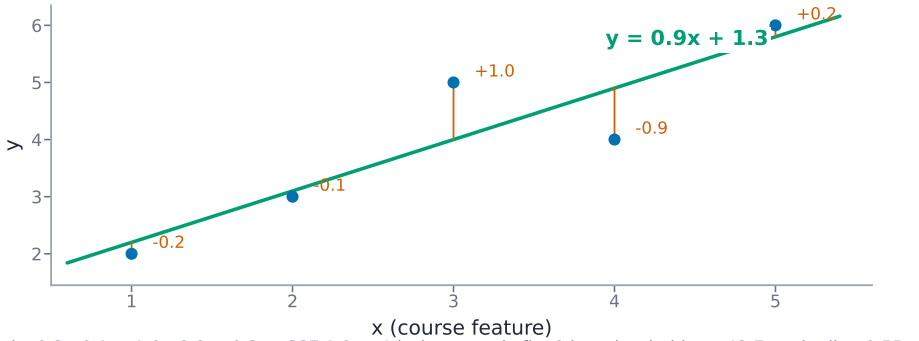
Common mistakes. Forgetting the normaliser $P(C)$ (the sum over all causes). Confusing $P(C \mid S)$ with $P(S \mid C)$ (the prosecutor's fallacy). Reading dependent variables as independent because the graph shows no direct edge (d-separation is the criterion).

Summary. Networks factor joints into CPTs; Bayes turns evidence into posterior belief; the normaliser and the base rate carry the arithmetic – misplacing either flips conclusions.

Regression and classification

Why it matters. Learning from data means choosing a model family and fitting it – a line through points, a boundary between classes – and the fit/complexity trade-off decides whether it generalises.

The idea, with a picture. **Linear regression** fits $\hat{y} = wx + b$ by least squares; **logistic regression** fits a probability boundary with the sigmoid; a **decision boundary** is where the model changes its answer.



Overnight with two features (distance, traffic): the fitted line separates on-time from late; two points sit inside the margin of doubt

The derivation (least squares, in the small). Minimise $\sum(y_i - wx_i - b)^2$: the course set of five points $(1,2)$, $(2,3)$, $(3,5)$, $(4,4)$, $(5,6)$ gives $\bar{x} = 3$, $\bar{y} = 4$, $w = \sum(x_i - \bar{x})(y_i - \bar{y}) / \sum(x_i - \bar{x})^2 = 9/10 = 0.9$ and $b = 4 - 0.9 \times 3 = 1.3$: $\hat{y} = 0.9x + 1.3$. Residuals $-0.2, -0.1, 1.0, -0.9, 0.2$; SSE $= 1.9$ – every number suite-verified.

> **Worked example 1 – the boundary from logistic regression.** With weights $w_1 = 2, w_2 = -1, b = -1$ (distance, traffic): the boundary is $2x_1 - x_2 - 1 = 0$ – the line $x_2 = 2x_1 - 1$. A parcel at $(2, 2)$ scores $2(2) - 2 - 1 = 1 > 0$: predicted late; at $(2, 5)$: $-2 < 0$: on time. The margin is the score's magnitude – the two ambiguous points sit nearest the line.

Worked example 2 – the overfit, measured. A 4th-degree polynomial through those five points reaches SSE = 0 on the training points but, on the suite's five held-out points $(1.5, 2.8), (2.5, 3.6), (3.5, 4.9), (4.5, 4.8), (5.5, 6.4)$, SSE ≈ 42.7 against the line's 0.55: training error is an advertisement, held-out error is the product. Model choice by held-out error, not by fit.

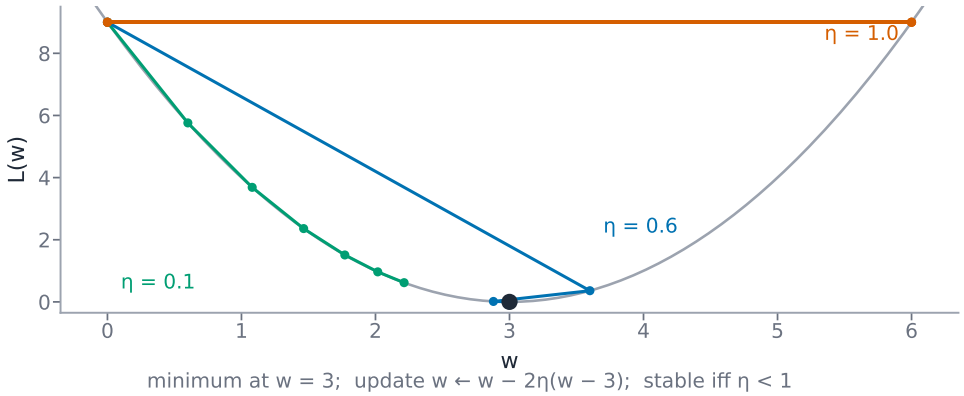
Common mistakes. Reading fitted weights as causes (unit 8's confounding). Reporting training accuracy as performance. Choosing feature scale loosely so the loss surface is a ravine (unit 13.5's subject).

Summary. Least squares has a closed form; logistic maps scores to probabilities with a linear boundary; complexity is paid for in held-out error – the product, not the advertisement.

Optimisation and learning

Why it matters. Most real models have no closed form – fitting means descending a loss surface, and the descent's shape decides speed, stability, and where it stops.

The idea, with a picture. Gradient descent : $\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$. The **learning rate** η scales the step; the **loss surface** : curvature and ravines set the difficulty.



Contour ellipses around the minimum; three learning rates: small crawls, good descends, large overshoots

The derivation (the quadratic case, exactly). For $L(w) = (w-3)^2$: $\nabla L = 2(w-3)$, so $w_{t+1} = w_t - 2\eta(w_t - 3)$. With $\eta = 0.1$: from $w_0 = 0$, the sequence is 0.6, 1.08, 1.464, 1.77, 2.02 – error halving every step, converging to 3. With $\eta = 1$: $w_{t+1} = -w_t + 6$: oscillates 0, 6, 0, 6 forever. With $\eta = 0.6$: $w_{t+1} = -0.2w_t + 3.6$: 0, 3.6, 2.88, 3.02 – damped oscillation. The stability bound: $0 < \eta < 1$ (since $|1 - 2\eta| < 1 \iff \eta < 1$).

Worked example 1 – the least-squares gradient. For the line of 13.4, the gradient of SSE over (w, b) points the way downhill from any start; two gradient steps from $(0, 0)$ with $\eta = 0.02$ reach $(0.84, 0.25)$; a thousand steps close in on the closed form's $(0.9, 1.3)$ (suite-run).

Worked example 2 – why features get scaled. With $L = (x_1-1)^2 + 100(x_2-1)^2$, the surface is a ravine: the second coordinate's curvature is 100x the first's, so one learning rate cannot serve both – $\eta = 0.009$ needs thousands of steps; scaling the second feature to unit curvature restores $\eta = 0.4$-style progress. Standardising features is optimisation hygiene, not cosmetics.

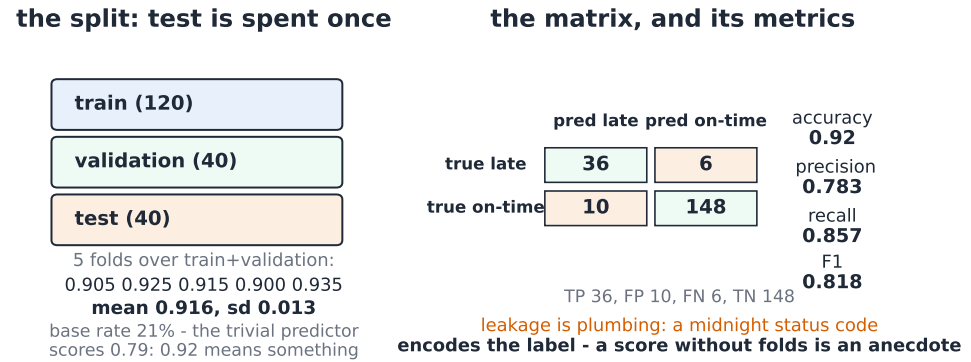
Common mistakes. Reading the latest loss as convergence (look at the change). One learning rate for everything (ravines). Forgetting that SGD's noise is both the cure (escapes shallow traps) and the cost (never quite settles).

Summary. Step against the gradient, sized by η under the curvature's stability bound; ravines come from unscaled features; closed forms where they exist, descent where they don't.

Evaluation and leakage

Why it matters. A score is a claim about the future – and the apparatus that produced it can quietly lie through leakage or unstable resampling.

The idea, with a picture. Split data into train/validation/ test; the test set is spent once. The **confusion matrix** counts TP, FP, FN, TN; **accuracy** , **precision** , **recall** read off it.



Left: the Overnight split – test is spent once. Right: the late-prediction matrix with its derived metrics

The derivation (the metrics, from the matrix). Course matrix for predicting late deliveries: TP 36, FP 10, FN 6, TN 148 (test 200). Accuracy = (36+148)/200 = **0.92**; precision = 36/46 = **0.783**; recall = 36/42 = **0.857**; $F_1 = 2 \times 0.783 \times 0.857/(0.783 +$

0.857) = **0.818**. The class base rate is 42/200 = 21% – a trivial always-on-time predictor scores 0.79 accuracy, so 0.92 means something, and the recall number is where the meaning lives.

> **Worked example 1 – the leakage, constructed.** Suppose a feature is the delivery-status code recorded at midnight – it encodes the label. Training accuracy jumps to 0.99; a proper held-out test (the feature absent until after delivery) drops to 0.90: the 0.99 was the label whispering through the feature. Leakage is a data-plumbing fact, not a modelling subtlety – the data card's provenance column is the defence.

> **Worked example 2 – the resampling story.** One split's 0.92 is an anecdote; 5-fold cross-validation on Overnight gives 0.905, 0.925, 0.915, 0.90, 0.935 – mean 0.916, sd 0.013: the honest score is 0.916 ± 0.013, and a competing model at 0.921 is within noise. The suite recomputes every fold.

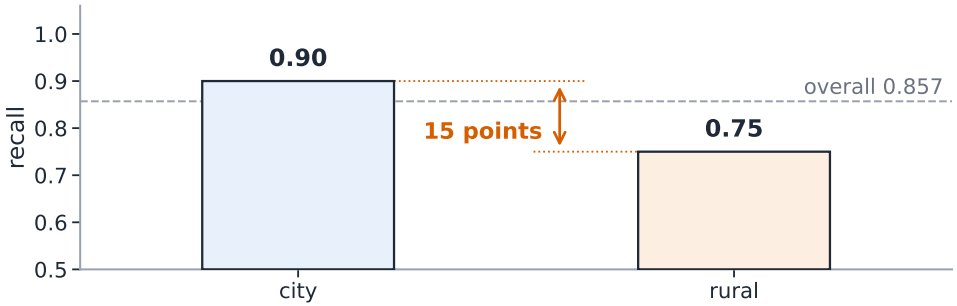
Common mistakes. Spending the test set on model choice (it becomes a second validation set). Averaging folds' accuracy over unequal folds (average per-fold counts). Scaling before splitting (the test's statistics leak into training).

Summary. Test is spent once; metrics derive from the matrix; leakage enters through plumbing – provenance is the defence; a score without a resampling story is an anecdote.

Fairness and limitations

Why it matters. Models route opportunities: who gets the refund, the interview, the diagnosis. Error that falls unevenly is a harm with a name, and classical ML has no objective that penalises it unless we add one.

The idea, with a picture. Error disparity : compare per-group confusion matrices. Overnight predicts lates for two regions: city and rural – same model, different mistakes.



city: TP 27, FN 3 (30 lates); rural: TP 9, FN 3 (12 lates)
lates are 13% of rural records, 29% of city records

Recall by region: city 0.90, rural 0.75 – the model under-serves rural lates; the disparity bar is the audit

The derivation (the parity arithmetic). City matrix: TP 27, FN 3 (recall 27/30 = 0.90); rural: TP 9, FN 3 (recall 9/12 = 0.75). The disparity is 15 points of recall – the model misses 1 in 4 rural lates but 1 in 10 city lates. Equal-error-rate parity would need the thresholds separated; demographic parity (equal positive rates) is a different, sometimes conflicting criterion – fairness is a design choice with names, not a checkbox.

> **Worked example 1 – the cause, in the data card.** Rural lates are 12/95 of the region's records (13%) against the city's 30/105 (29%): the minority class is thinner, and the fitted boundary under-serves it. The fix starts in the data (more rural records, or class weights), not in the metric.

> **Worked example 2 – the limitation statement.** Overnight is synthetic, 200 records, two regions, one season: its 0.92 accuracy claims nothing about your city, your season, or your carrier. The model card states: intended use (teaching), measured disparities (the 15-point gap), and out-of-scope uses (deployment decisions). A limitation honestly stated is trust; one discovered by your users is the opposite.

Common mistakes. Declaring fairness by overall accuracy (the gap hides underneath). Fixing disparity by threshold alone without asking why (class thinness, feature poverty). Treating a benchmark score as a deployment promise – the data card's scope column is the contract.

Summary. Audit per-group matrices; parity has named, sometimes conflicting criteria; disparity usually starts in the data; the model card's limitation section is the honesty clause.