Computational thinking

International Baccalaureate · IB Diploma · Computer Science · SL

Scope and prerequisites

This handout teaches Theme B1, Computational thinking, for the course first assessed in 2027. All of it is SL and HL content.

Prerequisites: none beyond basic arithmetic. Theme B2 then turns these ideas into Python code.

Thinking like a computer scientist

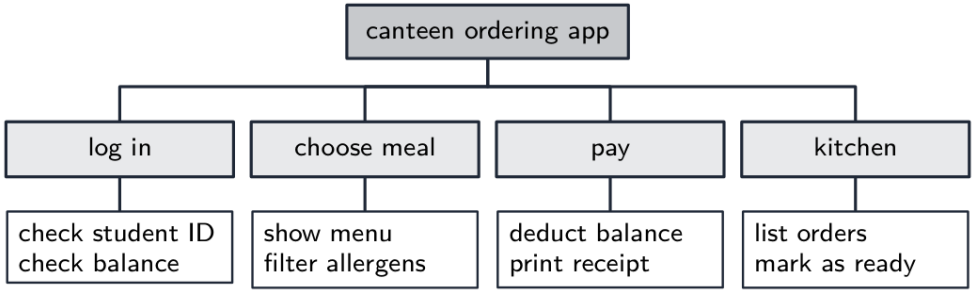
How do you plan a whole app?

Your school wants a canteen app: students order lunch on a phone, pay from a card balance, and the kitchen sees the orders. Nobody can write that in one go. Computational thinking is the set of habits that turns a big, vague problem into small, exact steps a computer can follow.

Four approaches

Approach	What you do	Canteen example
decomposition	split the problem into smaller sub-problems	log in, choose meal, pay, kitchen
pattern recognition	spot what sub-problems have in common	every screen shows a list and takes a choice
abstraction	keep the details that matter, drop the rest	a meal is a name, a price and allergens, not its photo or recipe
algorithmic design	write exact steps for each sub-problem	the steps to deduct a price from a balance

Decomposition



Each box is small enough to design, write and test on its own

Decomposition lets a team share the work, test each part separately, and reuse parts: the "check balance" step serves both logging in and paying.

Pattern recognition and abstraction

Pattern recognition finds the same structure in different places, so one solution can be reused. Calculating a total for a lunch order, a sports-kit order and a trip payment is the same pattern: add the prices, then deduct from a balance.

Abstraction builds a **model** that leaves out what does not affect the problem. A map of the school for fire drills shows exits and corridors, not wall colours. Too much abstraction loses something that matters: a canteen model with no allergens would be dangerous.

Algorithms

What makes an algorithm

An **algorithm** is a finite sequence of exact steps that solves a problem. A good algorithm is:

- **unambiguous**: each step has one meaning;
- **finite**: it stops after a limited number of steps;
- **correct**: it gives the right output for every valid input;
- defined by its **inputs** and **outputs**.

"Add some salt" is not an algorithm step; "add 2 g of salt" is.

Sequence, selection and iteration

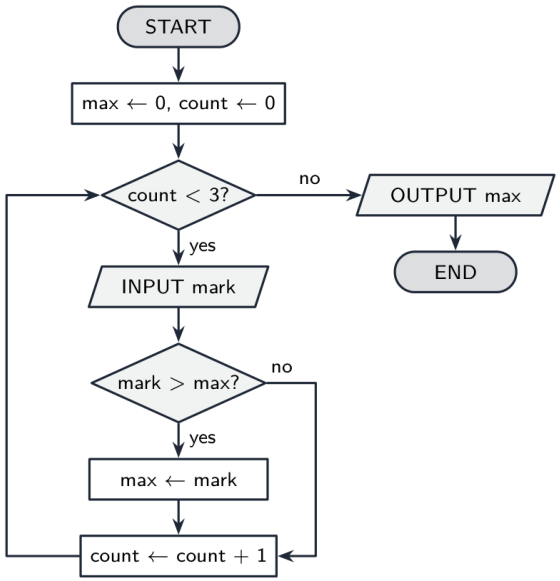
Every algorithm is built from three constructs:

Construct	Meaning	Flowchart shape
sequence	steps run one after another	arrows joining boxes
selection	a decision chooses the path	a diamond with yes and no
iteration	steps repeat	an arrow looping back

Flowcharts

A **flowchart** shows an algorithm as shapes joined by arrows.

Symbol	Use
rounded rectangle (terminator)	START and END
parallelogram	INPUT or OUTPUT
rectangle	a process, such as an assignment
diamond	a decision with yes and no exits



Find the largest of three marks: a loop with a decision inside it

`max ← 0` means "store 0 in max": the arrow is assignment.

Tracing a flowchart

A **trace table** records each variable after each step. For the inputs 62, 85, 71:

`count`	`count < 3?`	`mark`	`mark > max?`	`max`	Output
0				0	
0	yes	62	yes	62	
1	yes	85	yes	85	
2	yes	71	no	85	
3	no				85

The algorithm outputs **85**. Note the trap: if every mark could be negative, starting `max` at 0 gives a wrong answer; starting it at the first mark is safer.

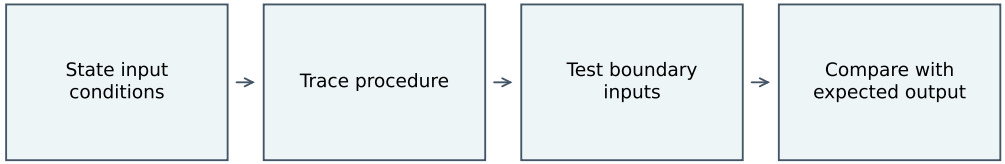
Pseudocode

The same algorithm written as **pseudocode**, structured English that is close to code but not tied to one language:

```
max ← 0
count ← 0
loop while count < 3
    input MARK
    if MARK > max then
        max ← MARK
    end if
    count ← count + 1
end loop
output max
```

Checking that an algorithm works

Test data



Correctness evidence: state the inputs, trace, test the boundaries, compare with the expected output

Choose tests from the problem's **specification**, not just the cases you expect. For a mark from 0 to 100:

Type	Purpose	Examples
normal data	typical valid data	45, 72
boundary data	the edges of the valid range, and just outside	0, 100, -1, 101
invalid data	data that must be rejected	-20, 150, "ten"

One passing test proves only that the algorithm works for that input. When a test fails, decide whether the **algorithm** is wrong, the **implementation** (the code) does not match the algorithm, or the **requirement** never covered that case.

■ Efficiency

Two correct algorithms can differ greatly in speed. Finding a name in a register of 1000 by checking each in turn takes up to 1000 steps; halving a sorted register each time (binary search, B2) takes about 10. Theme B2 measures this with Big O.

Practice

1. Define decomposition, and decompose "organise a school sports day" into four sub-problems.
2. Explain abstraction using a metro map as the example.
3. Give **one** example of pattern recognition in a school app.
4. Explain why "stir until it looks ready" is not a good algorithm step.
5. Name the flowchart symbol used for (a) a decision; (b) OUTPUT; (c) START.
6. Trace the flowchart above for the inputs 40, 40, 90 and state the output.
7. Trace the flowchart for the inputs -5, -8, -2. State the output and explain the error.
8. A ticket costs 12 for ages 0 to 17 and 20 for ages 18 and over; ages above 120 are rejected. Give two normal, two boundary and two invalid test values.
9. Draw a flowchart that inputs numbers until 0 is entered, then outputs how many numbers were entered, not counting the 0.

Answers.

1. Decomposition is splitting a problem into smaller sub-problems that can be solved separately. E.g. events and timetable, registering students, scoring and results, equipment and staff.
2. A metro map keeps what a traveller needs, the stations, lines and connections, and drops real distances, streets and buildings.
3. E.g. every form in the app (sign-up, booking, feedback) checks that required fields are filled, so one checking routine can be reused.
4. It is ambiguous: "looks ready" means different things to different people, so the step does not have one exact meaning.
5. (a) diamond; (b) parallelogram; (c) rounded rectangle (terminator).
6. `max` becomes 40 after the first input; the second 40 is not greater than 40, so `max` stays 40; 90 is greater, so `max` becomes 90. Output **90**.
7. No mark is greater than 0, so `max` stays 0. Output **0**, which is not one of the marks: `max` should start at the first input, not 0.
8. Normal: e.g. 10, 35. Boundary: 17 and 18 (the price change), or 0 and 120. Invalid: e.g. -1, 121.
9. START → `count ← 0` → INPUT `n` → decision `n = 0?`; no: `count ← count + 1`, INPUT `n`, back to the decision; yes: OUTPUT `count` → END.

Exam tips

- Name the approach the question asks for, then apply it to **its** scenario: a generic definition alone earns little.
- For abstraction, say both what is kept and what is removed, and why.
- Use the correct flowchart shapes; every decision needs labelled yes and no exits.
- In a trace table, add a new row each time a variable changes, and put outputs in their own column.
- Boundary test data sits at both edges of the range and just outside them.
- Check an algorithm with unusual inputs: all equal, all negative, empty, or the largest allowed.