A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

9.1 Computational Thinking Skills

Name: _____ Class: _____ Date: _____ **Total: 31 marks**

KEY WORDS decomposition 分解 • abstraction 抽象 • computational thinking 计算思维 • pattern recognition 模式 • loop 循环

Objective

Build the skills to answer exam questions on **computational thinking** 计算思维 — the two key skills of abstraction and decomposition.

You must be able to:

- explain and use **abstraction** 抽象
- explain and use **decomposition** 分解

1 Worked examples

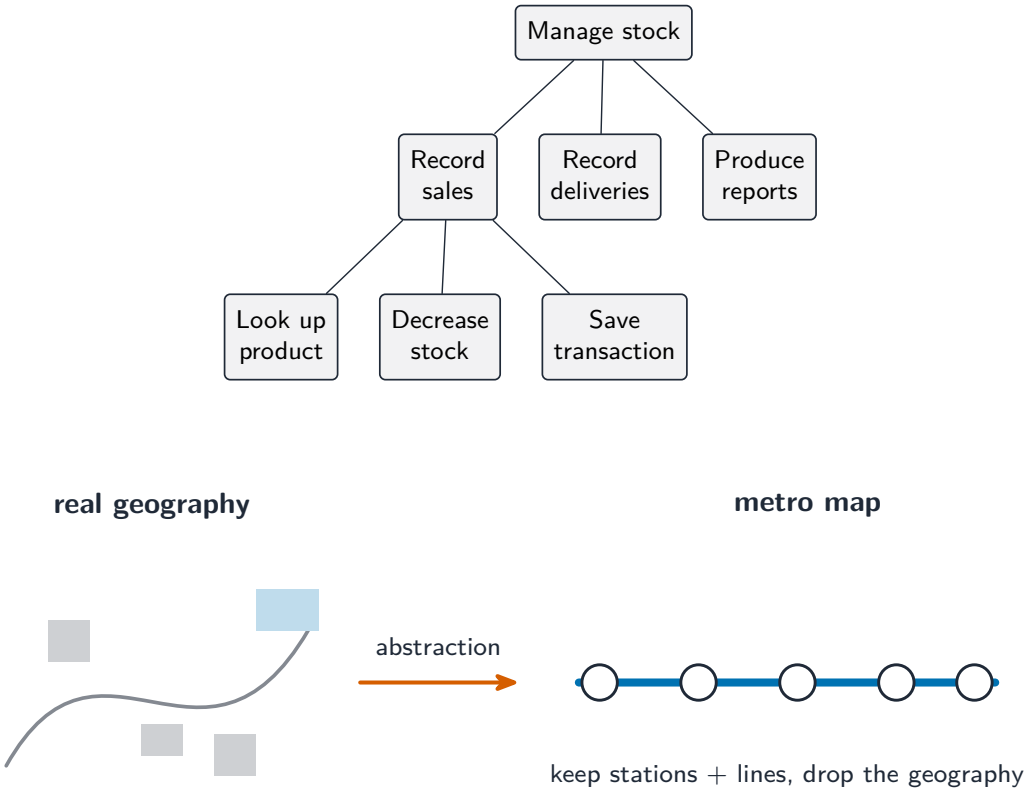
Study these first. Each one shows the method for a question type used later — follow the steps and you can do the Practice and Exam-style questions yourself.

■ Abstraction

Abstraction means keeping the **essential** features of a problem and **ignoring irrelevant detail**, to give a simpler model. Examples: a train-line map keeps the stations and lines but drops the real geography; a function hides a job behind a name; a class keeps only the attributes the system needs. Without abstraction, a full model of a real problem would be too big to reason about.

■ Decomposition

Decomposition means breaking a large problem into smaller **sub-problems**, each easier to solve and combined at the end:



Abstraction hides detail to manage complexity

It makes a big task manageable, lets a **team** divide the work, and produces **modular** code that is easier to test and maintain.

■ Pattern recognition

Pattern recognition 模式识别 means spotting where parts of a problem are the **same or similar**, so one solution can handle them all — for example, noticing that several jobs repeat lets you use a **loop** or reuse one **module** instead of writing the same code many times.

In the exam you may be asked to **identify which skill** a designer used: keeping only the needed detail → **abstraction**; splitting the problem into parts → **decomposition**; spotting repetition → **pattern recognition**.

2 Practice

Now apply the methods above.

2.1 State what **abstraction** means. [1]

2.2 State what **decomposition** means. [1]

2.3 Give one benefit of decomposing a problem. [1]

2.4 Give one everyday example of abstraction. [1]

3 Exam-style questions

3.1 Explain what is meant by **abstraction**, giving an example. [3]

3.2 A team must build a system to run a school library (lending and returning books, managing members, fines). Use **decomposition** to break this into **three** sub-tasks. [3]

3.3 Explain **two** benefits of decomposing a large program into modules. [2]

3.4 For each, state the computational thinking skill used: **(a)** a designer keeps only the data the system needs and ignores the rest; **(b)** a designer notices the same calculation is needed in five places and writes it once as a function. [2]

3.5 A shop's stock-control program is designed using **decomposition**. One module, `Sales()`, records a sale and updates the stock level.

(a) **Explain** what is meant by decomposition, and **give** two benefits of designing this program that way. [4]

(b) **Complete** the identifier table for `Sales()`. [4]

identifier	data type	description
`ItemCode`		
`QuantitySold`		
`UnitPrice`		
`InStock`		

(c) **State** the purpose of an identifier table at the **design** stage, and **explain** why it is written before any code. [3]

(d) The shop manager also wants a weekly report of items that fell below their reorder level. **Outline**, using **stepwise refinement**, the steps of a module `LowStockReport()`. Do **not** write pseudocode. [4]

(e) **Explain** why `Sales()` and `LowStockReport()` should be separate modules rather than one long block of code. [2]

Solutions

2.1 Keeping the **essential features** of a problem and **ignoring irrelevant detail** (a simpler model).

2.2 Breaking a large problem into smaller, easier **sub-problems**.

2.3 Any one: makes the problem manageable; lets a team share the work; gives modular, reusable, easier-to-test code.

2.4 Any one: a map; a function/method; a model/diagram that drops detail.

3.1 Abstraction keeps the **essential** features and removes irrelevant detail, giving a simpler model that is easier to work with; example, e.g. a metro map shows lines and stops but not real distances.

3.2 Any three sensible sub-tasks, e.g.: lend/return books; manage member records; calculate and record fines; search the catalogue.

3.3 Any two: smaller parts are easier to solve/test; a team can work in parallel; modules can be reused; maintenance is easier.

3.4 (a) **Abstraction**. (b) **Pattern recognition**.

3.5 (a) Decomposition means breaking a problem into smaller sub-problems, each solved by its own module. Benefits, any two: each module can be written and tested independently, so faults are easier to find; modules can be reused elsewhere, and several people can work on different modules at once.
(b) `ItemCode` — `STRING`, the unique code identifying the product. `QuantitySold` — `INTEGER`, the number of units in this sale. `UnitPrice` — `REAL` (or `CURRENCY`), the price of one unit. `InStock` — `INTEGER`, the number of units remaining after the sale.
(c) It lists every variable, its data type and what it is for, so the programmer knows exactly what to declare and nobody invents a second name for the same thing. Writing it first forces the data to be thought through before the logic, which is where most design faults are cheapest to fix.
(d) Any sensible four steps, in order: read the reorder level and the stock level for each item in turn; compare the two and select the items whose stock is below the level; accumulate or format those items into a report, including code, description and quantity remaining; output the report and record the date it was produced.
(e) Each does a **different job**, so keeping them separate means one can be changed or tested without disturbing the other; it also allows the low-stock check to be called from more than one place, such as at the end of a sale and again weekly.