Algorithm Design and Problem-solving

A-Level Computer Science

Computational thinking

Computational thinking 计算思维 is the set of mental tools for **analysing a problem and designing a solution a computer can run**. Two key ones are abstraction and decomposition.

Computational thinking breaks a big problem into smaller, easier parts — like solving a jigsaw

Abstraction

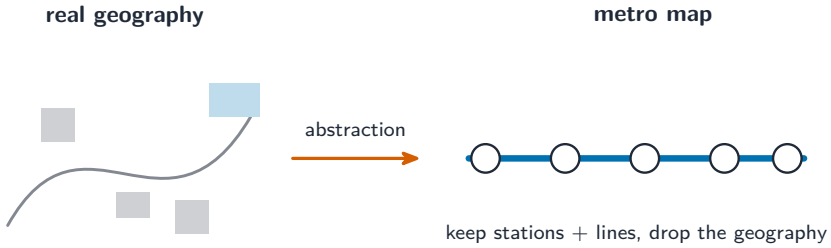
Abstraction 抽象 means **keeping the essential features** of a problem and **ignoring the irrelevant detail**, giving a simpler model.

Examples:

- a **train-network map** keeps the stations and lines but drops the geography.
- a **class** in object-oriented programming keeps only the attributes and methods the system needs.
- a **function** hides a piece of work behind a name.

A full model of any real problem would be too big to reason about, so abstraction is essential.

The examiner asks for the **purpose** of abstraction and for its **benefits**. Purpose: to produce a simpler model of a problem that contains only the details needed to solve it. Benefits: the problem is easier to understand and to program; the program is smaller and faster to write and test; the same model can be reused for similar problems. When you are asked to **produce an abstract model** of a system, list only the data and actions the task needs. For a school timetable that means the classes, rooms, teachers and periods; it does not mean the colour of the rooms or the age of the teachers.



Abstraction keeps the essentials (stations and lines) and drops irrelevant detail (the geography)

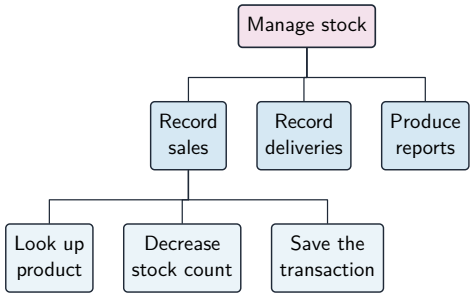
Decomposition

Decomposition 分解 means **breaking a large problem into smaller sub-problems**, each easier to solve and tackled one at a time.

1. find the main parts of the task.
2. break each into smaller sub-tasks.
3. continue until each is small enough to design directly.
4. solve the small tasks and combine them.

For stock control: "manage stock" → "record sales", "record deliveries", "produce reports" → ("record sales") "look up product", "decrease stock count", "save the transaction". Decomposition makes big problems manageable, lets a team divide the work, and gives modular code — each module becomes a **procedure** 过程 or function.

"Explain why decomposition is used" is a three-mark question with a fixed shape. Give three separate benefits: each **sub-problem** 子问题 is small enough to design, code and test on its own; different programmers can work on different **modules** 模块 at the same time; a module that already exists (or a library routine) can be reused, and a fault is easier to find because it lies inside one module. A structure chart (topic 12) is the diagram of a decomposition: the program at the top, its modules beneath, and the data passed between them.



Decomposing a program into modules and sub-modules

Algorithms

An **algorithm** 算法 is a **solution expressed as a sequence of defined steps**. Each step is **unambiguous** 无歧义 (one meaning), **deterministic** 确定性 (same input → same output), **finite** (the steps end), and **effective** (each can be done). An algorithm says *what to do*, independent of the programming language used to implement it.

Identifier table

When you start an algorithm, list every piece of data in an **identifier table** 标识符表 — its **identifier** 标识符 (the **variable** 变量 name), **data type** 数据类型, and description. The exam's table has exactly these three columns:

Identifier	Data type	Description
`Category`	`STRING`	the product category
`SaleDate`	`DATE`	when the item was sold
`ItemCost`	`REAL`	cost of the item
`InStock`	`BOOLEAN`	`TRUE` if in stock
`Sales`	`ARRAY[1:30] OF REAL`	the last 30 daily sales totals

Use **descriptive** names (`ItemCost`, not `x`): an identifier starts with a letter, contains no spaces, and is written the same way every time it appears. Common types are `INTEGER`, `REAL`, `STRING`, `CHAR`, `BOOLEAN`, `DATE`, plus arrays. The table forces you to name every piece of data before writing code, and a "complete the identifier table" question gives one mark for each correct data type or description, so write the type exactly as the pseudocode guide does.

list every piece of data first - name, type, description

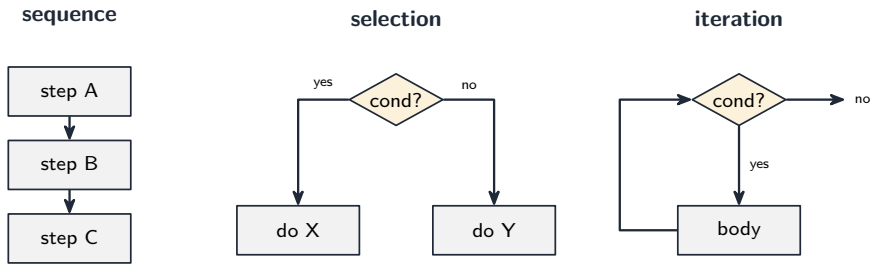
name	data type	description
`ItemCost`	`REAL`	cost of the item
`Category`	`STRING`	the product category
`SaleDate`	`DATE`	when the item was sold
`InStock`	`BOOLEAN`	TRUE if in stock

use **descriptive** names (`ItemCost`, not `x`) - the table forces you to plan the data

An identifier table names every piece of data before you write code

Pseudocode — the three basic constructs

Pseudocode 伪代码 is a structured, language-neutral way to describe algorithms.



The three building blocks of any algorithm: sequence, selection and iteration

1. Sequence

Steps run one after another (**sequence** 顺序):

```
INPUT Name
INPUT Age
OUTPUT "Hello", Name
```

2. Selection

A choice of which steps run, based on a condition (**selection** 选择):

```
IF Age >= 18 THEN
    OUTPUT "Adult"
ELSE
    OUTPUT "Minor"
ENDIF
```

For more options, use `CASE OF ... ENDCASE`.

3. Iteration

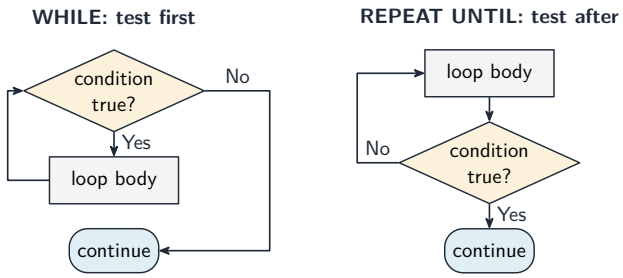
Repeating a block (**iteration** 迭代, a **loop** 循环):

```
FOR i ← 1 TO 10
    OUTPUT i
NEXT i
```

A **WHILE** loop tests the condition **before** each pass (may run zero times); a **REPEAT...UNTIL** loop tests **after** each pass (always runs at least once).

```
WHILE Total < 100 DO
    INPUT Value
    Total ← Total + Value
ENDWHILE

REPEAT
    INPUT Mark
UNTIL Mark >= 0 AND Mark <= 100
```



A WHILE loop tests before the body runs; a REPEAT ... UNTIL loop tests after it, so its body always runs at least once

Choosing the loop is itself a mark: `FOR` when you know how many times (a **count-controlled loop** 计数循环); `WHILE` when the loop might not run at all (a **pre-condition loop** 前测循环); `REPEAT ... UNTIL` when it must run at least once, as in validating an input (a **post-condition loop** 后测循环). A "describe the iteration construct" answer names the construct, says where the condition is tested, and gives the consequence (zero times or at least once).

Common operations

- **assignment** 赋值: `x ← 5` (an arrow; `=` is for comparison).
- input/output: `INPUT variable`, `OUTPUT expression`.
- comparisons `=`, `<>`, `<`, `>`, `<=`, `>=`; logic `AND`, `OR`, `NOT`.
- arithmetic `+ - * /`, plus `DIV` (integer division) and `MOD` (remainder).
- strings: `LENGTH`, `LEFT`, `RIGHT`, `MID`, and `&` for **concatenation** 拼接 (joining).

The pseudocode the exam expects

Every pseudocode answer is marked against Cambridge's published pseudocode guide. Write these forms exactly:

Construct	Pseudocode
declare a variable	`DECLARE Total : INTEGER`
declare an array	`DECLARE Marks : ARRAY[1:30] OF REAL`
a constant	`CONSTANT MaxTries = 3`
assignment	`Total ← Total + Value`
input and output	`INPUT Name` and `OUTPUT "Hello ", Name`
several options	`CASE OF Choice ... 1 : OUTPUT "Add" ... OTHERWISE OUTPUT "Error" ... ENDCASE`
count-controlled loop	`FOR i ← 1 TO 10 STEP 2 ... NEXT i`
pre-condition loop	`WHILE Total < 100 DO ... ENDWHILE`
post-condition loop	`REPEAT ... UNTIL Mark >= 0`
integer division and remainder	`DIV` and `MOD`: `17 DIV 5 = 3`, `17 MOD 5 = 2`
string functions	`LENGTH(S)`, `LEFT(S, 3)`, `RIGHT(S, 2)`, `MID(S, 2, 4)`, `UCASE(S)`, `LCASE(S)`
conversions	`INT(3.7) = 3`, `NUM_TO_STR(12)`, `STR_TO_NUM("4.5")`, `ASC('A') = 65`, `CHR(66) = 'B'`

Construct	Pseudocode
a random number	`RAND(100)` gives a real number from 0 up to (but not including) 100; `INT(RAND(100)) + 1` gives an integer from 1 to 100

Two habits earn marks on every question: **declare** every variable you use, with the type from your identifier table, and **initialise** 初始化 every **counter** 计数器 and total (`Count ← 0`, `Total ← 0`) before the loop that changes it.

Input → Process → Output

Every program follows this shape:

```
INPUT Length
INPUT Width
Area ← Length * Width
OUTPUT "Area = ", Area
```

Listing the inputs and outputs first makes the algorithm cleaner.

Worked example. Write pseudocode that inputs 100 integers and outputs how many of them, and the total of those, that lie between 10 and 20 inclusive.

Identifier table: `Count : INTEGER` (loop counter), `Value : INTEGER` (the integer just input), `InRange : INTEGER` (how many were in range), `Total : INTEGER` (their sum).

```
DECLARE Count, Value, InRange, Total : INTEGER
InRange ← 0
Total ← 0
FOR Count ← 1 TO 100
    INPUT Value
    IF Value >= 10 AND Value <= 20 THEN
        InRange ← InRange + 1
        Total ← Total + Value
    ENDIF
NEXT Count
OUTPUT InRange, Total
```

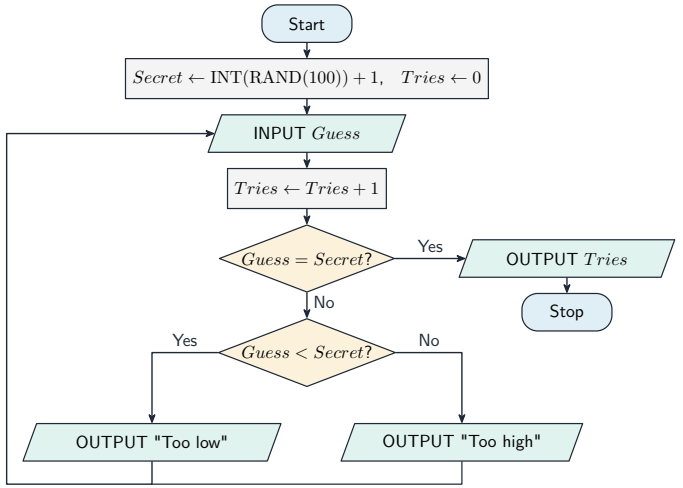
If the question then asks you to "identify two constructs and state how each is used", answer in the same shape: iteration, the `FOR` loop, repeats the input 100 times; selection, the `IF` statement, adds a value only when it is in range.

Worked example. A program picks a secret integer from 1 to 100. The user guesses until they are right; after each wrong guess the program says "Too low" or "Too high", and at the end it outputs how many guesses were made.

Identifier table: `Secret : INTEGER` (the number to guess), `Guess : INTEGER` (the user's input), `Tries : INTEGER` (how many guesses so far).

```
DECLARE Secret, Guess, Tries : INTEGER
Secret ← INT(RAND(100)) + 1
Tries ← 0
REPEAT
    INPUT Guess
    Tries ← Tries + 1
    IF Guess < Secret THEN
        OUTPUT "Too low"
    ELSE
        IF Guess > Secret THEN
            OUTPUT "Too high"
        ENDIF
    ENDIF
UNTIL Guess = Secret
OUTPUT "You took ", Tries, " guesses"
```

A `REPEAT ... UNTIL` loop is the right choice because the user must guess at least once. The marks are for: the random number in the right range, a loop that ends on a correct guess, the counter that starts at zero and increases inside the loop, the two messages under the right conditions, and the final output.

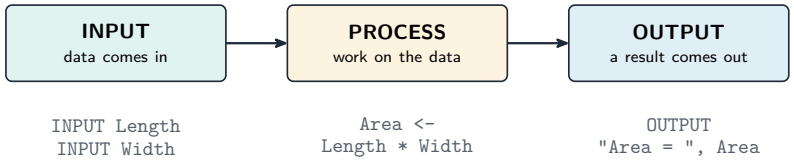


The same guessing game as a flowchart: the two decision diamonds are the two IF statements, and the return arrow is the REPEAT ... UNTIL loop

Worked example. Output two different random integers, each between −10 and 10 inclusive.

There are 21 possible values, so `INT(RAND(21))` gives 0 to 20 and subtracting 10 shifts it to the range −10 to 10. The second number must be generated again until it differs from the first:

```
DECLARE First, Second : INTEGER
First ← INT(RAND(21)) - 10
REPEAT
    Second ← INT(RAND(21)) - 10
UNTIL Second <> First
OUTPUT First, Second
```



Every program follows the Input, Process, Output shape

Three notations

The same algorithm can be written three ways.

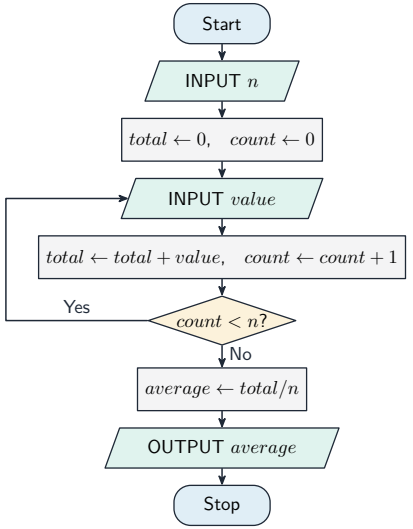
- **structured English** 结构化英语 — natural language with indentation and fixed keywords; good for a high-level description.
- **flowchart** 流程图 — a diagram with standard shapes:

Shape	Meaning
Rounded rectangle	Start / Stop
Parallelogram	Input / Output
Rectangle	Process
Diamond	Decision
Arrow	Flow of control

- **pseudocode** — the keyword notation above; closest to code.

You should be able to convert between any pair: each `IF` is a decision diamond, each loop is a back-arrow, and a sequence is stacked rectangles.

Converting a flowchart into pseudocode: start at the terminator and follow the arrows in order; a parallelogram becomes `INPUT` or `OUTPUT`; a rectangle becomes an assignment; a diamond with two exits that rejoin further down becomes `IF ... THEN ... ELSE ... ENDIF`; a diamond whose one exit arrows back upwards is a loop, and it is a `WHILE` loop if the diamond comes before the repeated boxes and a `REPEAT ... UNTIL` loop if it comes after them. Converting the other way, draw exactly one box per statement and one diamond per condition, and label every exit of a diamond `Yes` or `No`.



A flowchart for averaging a list of numbers, using the standard shapes

Stepwise refinement

Stepwise refinement 逐步求精 starts with a high-level outline and **expands each step** until it is small enough to code. For an average of n numbers:

Level 1:

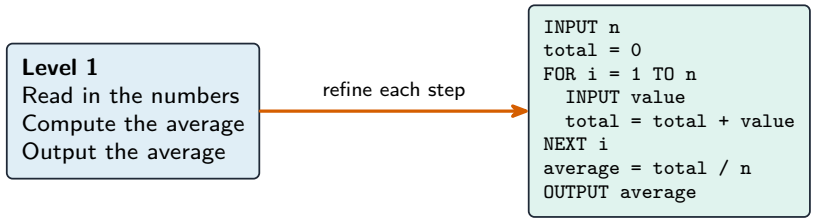
```
Read in the numbers
Compute the average
Output the average
```

Level 2:

```
INPUT n
total ← 0
FOR i ← 1 TO n
    INPUT value
    total ← total + value
NEXT i
average ← total / n
OUTPUT average
```

Each refinement keeps the previous structure and adds detail.

A six-mark "apply stepwise refinement" question gives you a high-level outline and wants each step expanded into the concrete statements a programmer could code. Keep the steps in the same order, name the data each step reads or produces, and stop when every line is a single input, assignment, output, loop or condition. For example, "validate the password" becomes: input the password; check its length is at least 8; check it contains at least one digit; output "accepted" if both checks pass, otherwise output "rejected".



Stepwise refinement: expand each high-level step into detailed pseudocode

Logic statements

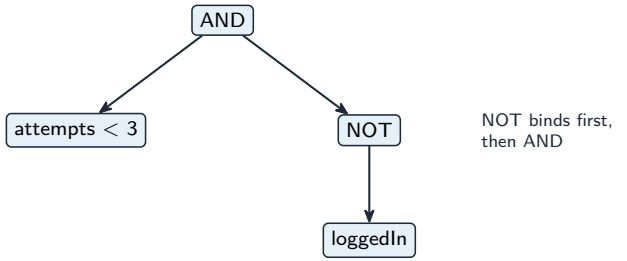
A **logic statement** 逻辑语句 is a **Boolean** 布尔 condition that controls branching, built from comparisons (`x > 10`), connectives (`AND`, `OR`, `NOT`) and brackets. Use it as the condition of `IF`, `WHILE` or `REPEAT...UNTIL`:

```
WHILE attempts < 3 AND NOT loggedIn DO
    INPUT password
    IF password = correctPassword THEN
        loggedIn ← TRUE
    ELSE
        attempts ← attempts + 1
    ENDIF
ENDWHILE
```

Precedence 优先级 (highest to lowest): `NOT`, then `AND`, then `OR`. Use brackets when unsure. Common mistakes:

- `a = 1 OR 2` is wrong — write `a = 1 OR a = 2`.
- `NOT a > 5` means `NOT (a > 5)`, i.e. `a <= 5`.
- `NOT (A AND B)` is the same as `(NOT A) OR (NOT B)` (**De Morgan's law** 德摩根定律) — handy for simplifying conditions.

Turning a sentence into a logic statement is a skill the papers test directly. "A ticket is free for anyone under 5 or over 65" becomes `Age < 5 OR Age > 65`. "A mark is valid if it is a whole number from 0 to 100" becomes `Mark >= 0 AND Mark <= 100`. "The loop stops when the file is finished or ten records have been read" becomes `UNTIL EOF(File) OR Count = 10`. Write each comparison in full: `Age > 65` and `Age < 5`, never `Age > 65 OR < 5`.



Precedence: NOT binds to loggedIn first, then AND combines the two sides

Worked example. Write an identifier table and pseudocode to read 10 numbers and output the largest. The **identifier table** names each variable with its data type and purpose: `Count : INTEGER` (loop counter), `Num : REAL` (the number just read), `Max : REAL` (largest so far).

```
Max ← -999999
FOR Count ← 1 TO 10
    INPUT Num
    IF Num > Max THEN
        Max ← Num
    ENDIF
NEXT Count
OUTPUT Max
```

The design decision carrying the marks is **initialising `Max`**: it must start **lower than any possible input** - or, safer still, be set to the **first** number read. Initialise it to `0` and the algorithm wrongly returns 0 for a list of negative numbers, a bug your trace only exposes if the test data include a negative.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
abstraction	keeping the essential details of a problem and leaving out the details that are not needed
decomposition	breaking a problem down into smaller sub-problems, each of which can be solved separately
algorithm	a solution to a problem expressed as a sequence of defined steps
identifier table	a table listing each identifier used in an algorithm with its data type and a description of its purpose
pseudocode	a structured, language-independent way of writing the steps of an algorithm
flowchart	a diagram that shows the steps and decisions of an algorithm using standard symbols joined by arrows
sequence	statements executed one after another in the order written
selection	choosing which statements to execute according to a condition
iteration	repeating a group of statements while, or until, a condition holds

Term	Definition
stepwise refinement	breaking each step of an outline into smaller steps, repeatedly, until each step can be coded directly
logic statement	a condition built from comparisons and the operators AND, OR and NOT that evaluates to TRUE or FALSE

Exam tips

- Define an **algorithm** as an unambiguous, finite, deterministic sequence of steps, independent of language.
- Use the three constructs correctly — **sequence, selection, iteration** — and keep an identifier table with data types.
- Break a problem down by **decomposition and abstraction**, then stepwise refinement.
- Write pseudocode that would actually run: declare variables and follow the exam's pseudocode style.

Common mistakes

- Using `=` to assign a value. Assignment is `←`; `=` is a comparison.
- Forgetting `ENDIF`, `ENDWHILE`, `ENDCASE` or `NEXT`. Every construct closes, and the closing word is where the mark for the construct is checked.
- Not initialising a total or counter before the loop, so the algorithm adds to a value that never existed.
- Using a `FOR` loop when the number of repetitions is unknown. Reading until a sentinel value or a correct guess needs `WHILE` or `REPEAT ... UNTIL`.
- Writing `Age > 65 OR < 5`. Each side of `OR` and `AND` must be a complete comparison.
- Answering "explain why decomposition is used" with one benefit written three ways. Three marks need three different benefits.