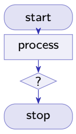
# Algorithm Design

AS Computer Science ▪ Topic 9

A-Level Computer Science






## What we will cover

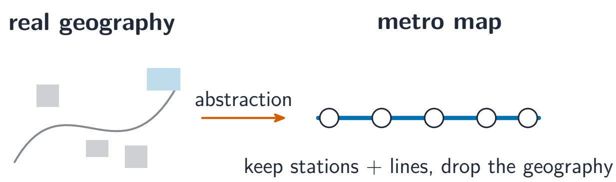


# 1 Computational thinking



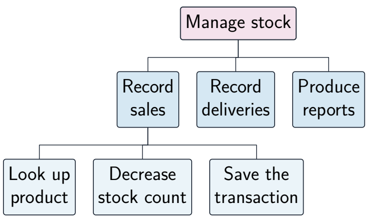
## Abstraction

**Abstraction** 抽象 keeps the essential features and drops the irrelevant detail. A metro map keeps the stations and lines, not the geography. A **class** in object-oriented programming does the same: only the attributes and methods the system needs.






## Decomposition




**Decomposition** 分解 breaks a big problem into smaller sub-problems, each easy to solve. Find the main parts, split each into steps, keep splitting until each is easy, then solve them and combine. Each module becomes a **procedure** 过程 or function.



## Algorithms and the identifier table

An **algorithm** 算法 is a sequence of steps that is unambiguous, deterministic, finite and effective. It says *what* to do – independent of the language.

| Name | Type | Description |
|---|---|---|
| `ItemCost` | `REAL` | cost |
| `InStock` | `BOOLEAN` | in stock? |
| `SaleDate` | `DATE` | sold on |

An **identifier table** 标识符表 names every variable first.

## Identifier table

**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

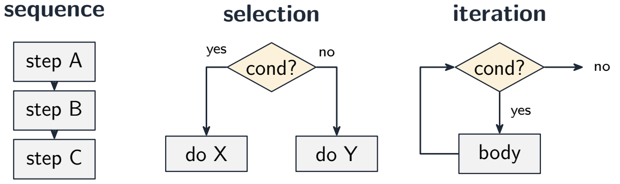
> An identifier table names every piece of data before you write code

# 2 Pseudocode



## The three basic constructs

- **sequence** 顺序 – one step after another
- **selection** 选择 – `IF`/`ELSE`; for more options, `CASE OF` … `ENDCASE`
- **iteration** 迭代 – `FOR`, `WHILE`, `REPEAT`






## Pseudocode operations

**Selection**

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

- assignment: `x` ← `5`
- arithmetic `+ - * /`, plus `DIV` (integer ÷) and `MOD` (remainder)
- `WHILE` tests before; `REPEAT` tests after
- `&` joins strings (**concatenation** 拼接)



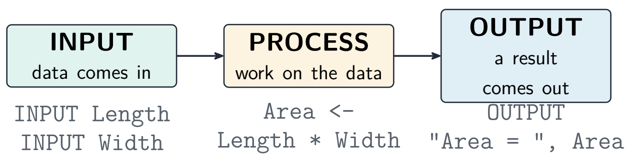
## Input → Process → Output

Every program follows this shape: `INPUT` the values, process them, then `OUTPUT` the result.

> Listing the inputs and outputs first makes the algorithm cleaner – and naming all three parts is what the mark scheme asks for.



## Input → Process → Output, in detail




> Every program follows the Input, Process, Output shape

## Worked example: trace the algorithm

```
x ← 0
FOR i ← 1 TO 5
  IF i MOD 2 = 0 THEN
    x ← x + i
  ENDIF
NEXT i
OUTPUT x
```

What is the output?

| i | i MOD 2 | x |
|---|---|---|
| – | – | 0 |
| 1 | 1 | 0 |
| 2 | **0** | **2** |
| 3 | 1 | 2 |
| 4 | **0** | **6** |
| 5 | 1 | 6 |

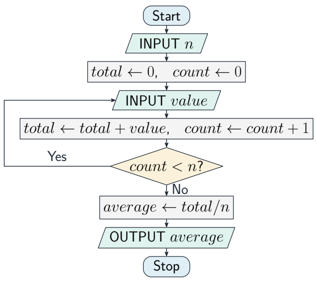
output: **6**

> Fill the **trace table** 跟踪表 row by row – never run the whole loop in your head. Only even `i` (2 and 4) change `x`.

# 3 Notations and refinement



## Three notations



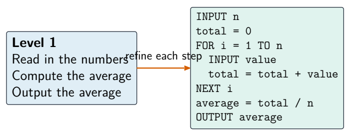

One algorithm, three ways:

- **structured English** 结构化英语 – high-level
- **flowchart** 流程图 – standard shapes
- **pseudocode** 伪代码 – closest to code

Each `IF` = a diamond; each loop = a back-arrow.



## Stepwise refinement

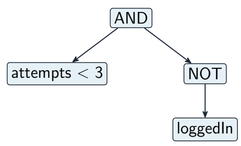



**Stepwise refinement** 逐步求精 starts with a high-level outline and expands each step until it can be coded. "Compute the average" → a loop that sums, then divides by $n$.

# 4 Logic



## Logic statements




NOT binds first, then AND

A **Boolean** 布尔 condition controls branching. **Precedence** 优先级: `NOT` > `AND` > `OR`.

- write `a = 1 OR a = 2` (not `a = 1 OR 2`)
- **De Morgan** 德摩根定律: `NOT (A AND B) = (NOT A) OR (NOT B)`

## Logic precedence, De Morgan, and the initialising trap

| | |
|---|---|
| **precedence 优先级** | `NOT`, then `AND`, then `OR` – and use brackets whenever it is not obvious |
| **De Morgan's law 德摩根定律** | `NOT (A AND B)` is the same as `(NOT A) OR (NOT B)` |
| **and the other half** | `NOT (A OR B)` is `(NOT A) AND (NOT B)` |
| **why it is handy** | for simplifying a condition – a negated compound becomes two simple negations |
| **the marked design decision** | initialising `Max`: it must start lower than any possible input |
| **the safe way** | set `Max` to the first element of the list rather than to zero – which fails on all-negative data |

> Initialising to zero is the classic bug: a list of negative temperatures returns a maximum of zero, and the program never errors.



## 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.

# 5 Recap



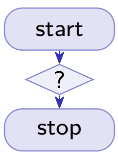
## Topic 9 – key takeaways

1. **Thinking**
   abstraction keeps essentials; decomposition splits
2. **Constructs**
   sequence, selection, iteration
3. **Notations**
   structured English, flowchart, pseudocode
4. **Logic**
   NOT > AND > OR; De Morgan



## Check yourself

1. Which technique drops irrelevant detail?
2. Which loop always runs at least once?
3. Flowchart shape for a decision?
4. Simplify `NOT (A AND B)`.




> **Answers** **1.** abstraction **2.** `REPEAT...UNTIL` **3.** a diamond **4.** `(NOT A) OR (NOT B)`