

Algorithm Design

IGCSE Computer Science ■ Topic 7

IGCSE Computer Science



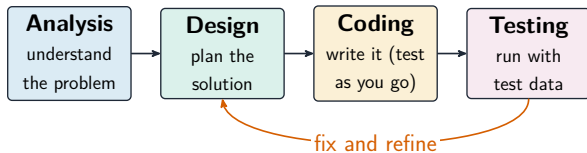
What we will cover

- 1 The development life cycle
- 2 Design tools
- 3 Algorithms & IPO
- 4 Validation
- 5 Trace tables
- 6 Standard algorithms

1

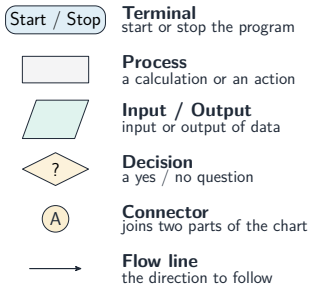
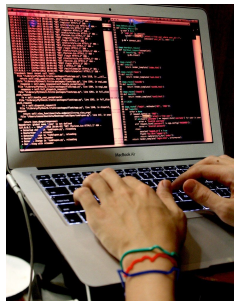
Designing

The development life cycle



- **abstraction** 抽象 – keep only the important details and ignore the rest;
- **decomposition** 分解 – break a big problem into smaller, easier parts.

The program development life cycle



Software is written by programmers, who follow the development life cycle

A program flowchart sets out the steps and decisions of a program during the design stage

Decomposing a problem, and testing what you build

Sub-systems

Break the problem into **sub-systems** 子系统, each with its own inputs, **processing** 处理 and outputs – then solve them one at a time.

Test data

Choose **test data** 测试数据 deliberately: normal, boundary, abnormal – and know the expected result *before* running it.

Iterative testing

Iterative testing 迭代测试: test each module as it is written, fix, and test again, rather than waiting for the whole program.

Then the whole

Final testing runs the assembled program against the original requirements.

A test with no expected result is not a test. Write the expected value first, or you will accept whatever appears.

Three design tools

structure diagram

the parts of a system
and how they fit

flowchart 流程图

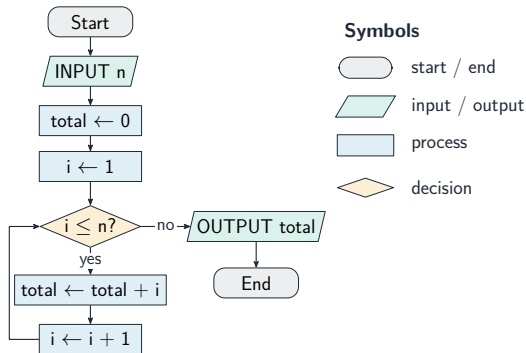
boxes and arrows for
the steps in order

pseudocode 伪代码

steps in simple,
code-like English

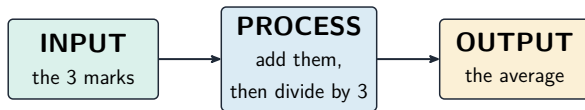
None is a real language – they **plan** the solution before any code is written.

A flowchart for the sum algorithm



A flowchart for the sum algorithm, using the standard symbols (start/end, input/output, process, decision)

Algorithms: input, process, output



An **algorithm** 算法 is an ordered set of steps that solves a problem. Every one splits into:

input → **process** → **output**

“Average of three marks”:
input the marks, **process** add and divide by 3, **output** the average. Name all three in an exam answer.

2

Checking

Validation checks

range 范围 – between
a low and high value

length 长度 – allowed
number of characters

type 类型 – the right
kind (number, not letters)

presence 存在性 –
something was entered

format 格式 – the right
pattern (dd/mm/yyyy)

Validation checks data is *sensible*; **verification** (visual or double-entry) checks it was *copied right*.

The validation checks, by name

Presence check

A **presence check** 存在性检查: the field must not be left empty.

Length check

A **length check** 长度检查: the entry has the right number of characters – a password of at least eight, a code of exactly six.

Range and type

A range check keeps a number between limits; a type check rejects letters in a numeric field; a format check enforces a pattern.

Check digit

A **check digit** 校验码 is calculated from the other digits and appended, to help detect errors when the number is entered.

Validation checks that the data is *sensible*; verification checks it was *copied correctly*. They catch different mistakes.

Worked example – a trace table

Trace with input $n = 5$. What does it output?

```
total ← 0
FOR i ← 1 TO n
  total ← total + i
NEXT i
OUTPUT total
```

i	total	OUT
1	1	
2	3	
3	6	
4	10	
5	15	15

It sums 1 to n : output 15. Fill *one row per pass* – never run the loop in your head.

Trace tables

count	total	output
1	5	
2	12	
3	20	20

A trace table records each variable's value as the program runs

3

Standard algorithms

Try it: search without sorting first

A list contains 8, 3, 5, 1, at positions 1 to 4.

Use a **linear search** 线性查找 from the start to find 5. Record each comparison. When can you stop?

Start again and search for 9. How many comparisons are needed? What tells you that the value is absent?

Would sorting the list be necessary before either search? Explain.

A search stops on a match or at the end

Position	Value	Search for 5	Search for 9
1	8	no match	no match
2	3	no match	no match
3	5	match: stop	no match
4	1	not checked	no match: end

Finding 5 takes three comparisons; ruling out 9 takes four. The list need not be sorted: unchecked positions may still contain the target.

Start the found flag at FALSE and set it TRUE on a match. Never read beyond the last valid position.

Try it: one bubble-sort pass is not a full sort

Sort 8, 3, 5, 1 into ascending order using **bubble sort** 冒泡排序.

In each pass, compare positions 1 and 2, then 2 and 3, then 3 and 4. Swap a pair only if its left value is larger.

Write the list after each comparison in the first pass. Is the result sorted?

Continue with complete passes. For a version that stops on a pass with no swaps, which pass proves that the algorithm can stop?

Trace the passes, including the final check

First pass: $8, 3, 5, 1 \rightarrow 3, 8, 5, 1 \rightarrow 3, 5, 8, 1 \rightarrow 3, 5, 1, 8$.

Pass	List after the complete pass	Any swaps?
1	3, 5, 1, 8	yes
2	3, 1, 5, 8	yes
3	1, 3, 5, 8	yes
4	1, 3, 5, 8	no: stop

Reset the swap flag to FALSE at the start of **each** pass; set it TRUE whenever a swap happens.

One pass puts the largest value at the end, but may leave smaller values out of order. A no-swap pass confirms ascending order in this version.

Linear search and bubble sort, in detail

search for 5: check each item in turn (left to right)



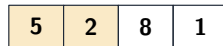
4	9	2	7	5	1	8	3
---	---	---	---	---	---	---	---

$\neq 5$ $\neq 5$ $\neq 5$ $\neq 5$ $= 5$ **found**

linear search – check each in turn

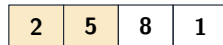
compare the pair 5 and 2: $5 > 2$, so swap them

before



5	2	8	1
---	---	---	---

after



2	5	8	1
---	---	---	---

then repeat for each pair, until no swaps are needed

bubble sort – swap neighbours each pass

Linear search checks each item in turn from the start; bubble sort makes repeated passes, swapping neighbours that are the wrong way round.

Try it: total and count only accepted values

A program examines four attempted inputs: -20 , 4 , 50 , 6 . It accepts a value only if $-20 < \text{Value} < 50$.

Start `Total` and `Count` at zero. After each attempt, record the accepted total and accepted count.

Find the average of the accepted values. Should you divide by four or by the final count?

Explain the difference between **totalling** 求和 and **counting** 计数, and what the program should do if no value is accepted.

Keep both updates inside the acceptance test

Processing four attempted inputs

```
Total ← 0
Count ← 0
FOR I ← 1 TO 4
  INPUT Value
  IF Value > -20 AND Value < 50
    THEN
      Total ← Total + Value
      Count ← Count + 1
  ENDIF
NEXT I
```

Input	Total	Count
-20	0	0
4	4	1
50	4	1
6	10	2

The strict limits reject both -20 and 50. Rejected attempts change neither the accepted total nor the accepted count.

Use the count of the values actually totalled

Two accepted values, 4 and 6, give $\text{Total} = 10$ and $\text{Count} = 2$.

$$\text{Average} = \frac{\text{Total}}{\text{Count}} = \frac{10}{2} = 5$$

Dividing by four would count the rejected attempts, even though their values were not added.

If $\text{Count} = 0$, output a message such as “No accepted values”. Do not divide by zero.

A counter adds 1 per accepted value; a total adds the value itself. Use the same acceptance rule when adding values and counting them.

Transfer: collect 50 accepted values

Replace the fixed-attempt loop

```
Total ← 0
Count ← 0
WHILE Count < 50
  INPUT Value
  IF Value > -20 AND Value < 50
    THEN
      Total ← Total + Value
      Count ← Count + 1
    ENDIF
  ENDWHILE
```

A task asking for 50 **accepted** values may need more than 50 attempts.

A rejected value leaves Count unchanged, so the program asks for another input.

When the loop ends, Count is 50. Output Total and calculate the average as $\text{Total} / \text{Count}$.

Try it: can zero be the starting maximum?

A non-empty list contains -8 , -3 , -12 , -5 . Find its **maximum** 最大值, **minimum** 最小值 and **average** 平均值.

A student starts Maximum at zero, then replaces it only when the next value is larger. What will the algorithm output? Why is that wrong?

Choose starting values that work without assuming all numbers are positive. Trace the largest and smallest values seen so far after each item.

Start with a value that is actually in the list

Set Maximum and Minimum to the first value, -8 . For each remaining value, update the maximum if it is larger, and the minimum if it is smaller.

Value just considered	Maximum so far	Minimum so far
-8 (initialisation)	-8	-8
-3	-3	-8
-12	-3	-12
-5	-3	-12

Starting the maximum at zero would incorrectly keep zero: no list value is greater than it. The result must be a value from this non-empty list.

The average uses all four values

The maximum is -3 and the minimum is -12 . The average is a separate calculation: total all values, then divide by their count.

$$\text{Total} = \sum \text{Value} = (-8) + (-3) + (-12) + (-5) = -28$$

$$\text{Average} = \frac{\text{Total}}{\text{Count}} = \frac{-28}{4} = -7$$

Check that -7 lies between the minimum and maximum. This is a useful check, but does not by itself prove that the average is correct.

Initialising from the first item works for negative, positive or mixed values. It requires at least one item; handle an empty list separately.

Exam tips

- Learn the four life-cycle stages: analysis → design → coding → testing.
Abstraction keeps only the important details; **decomposition** breaks a problem into smaller parts.
- **Validation** checks data is *sensible* (range, length, type, presence, format checks); **verification** checks it was *copied correctly* (a visual check or double entry).
- Learn the four test-data types: **normal** (accepted), **abnormal** (rejected), **extreme** (the largest/smallest still allowed), **boundary** (the values either side of a limit).
- To work out what an algorithm does, fill in a **trace table** – write down every variable's value at each step.
- Know the standard algorithms: **linear search** (check each item in turn) and **bubble sort** (swap side-by-side pairs until no swaps are needed).

4

Recap

Topic 7 – key takeaways

1 Design

abstraction + decomposition; 3
design tools

2 IPO

every algorithm: input, process,
output

3 Check

validation = sensible; verification
= copied right

4 Trace

one row per pass to see what
code does

Check yourself

- 1 What are the three parts of every algorithm?
- 2 Range check or type check for “age 0–120”?
- 3 Validation or verification: double entry?
- 4 When does a bubble sort stop?



Answers 1. input, process, output 2. range 3. verification 4. when a pass makes no swaps