

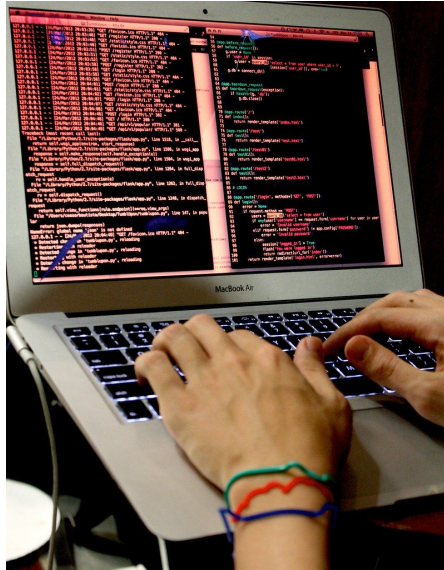
Topic 07

IGCSE Computer Science

The program development life cycle

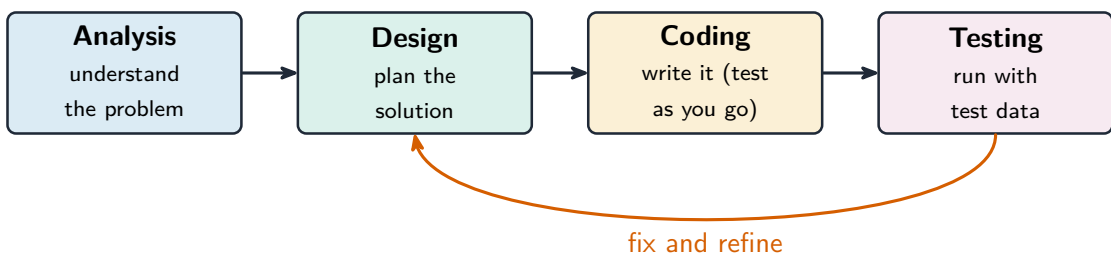
The **program development life cycle** is the set of stages used to make a program. There are four stages.

is the set of stages used

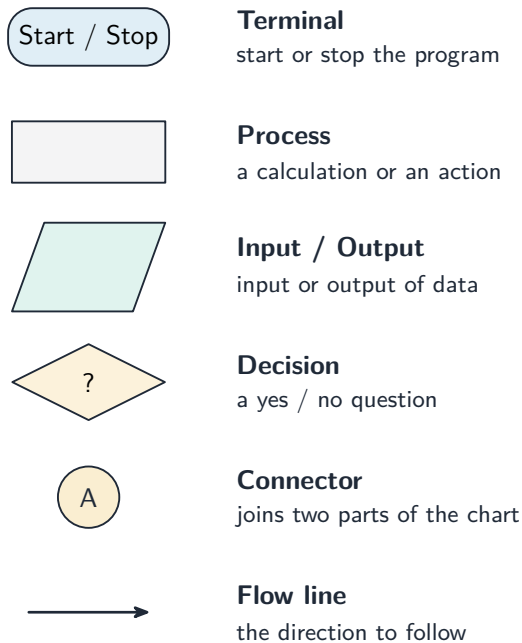


Software is written by programmers, who follow the development life cycle

Stage	What you do
analysis	study the problem and work out what is needed
design	plan how the program will work
coding	write the program code and test it as you go
testing	run the finished program with test data to find errors



The four stages of program development; testing feeds back to fix and refine the design



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

■ Analysis

In analysis you understand the problem. Two key skills help:

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

■ Design

In design you plan the solution, often using decomposition. You can show the parts as **sub-systems** in a **structure diagram** (a chart that splits a system into smaller boxes).

■ Coding and testing

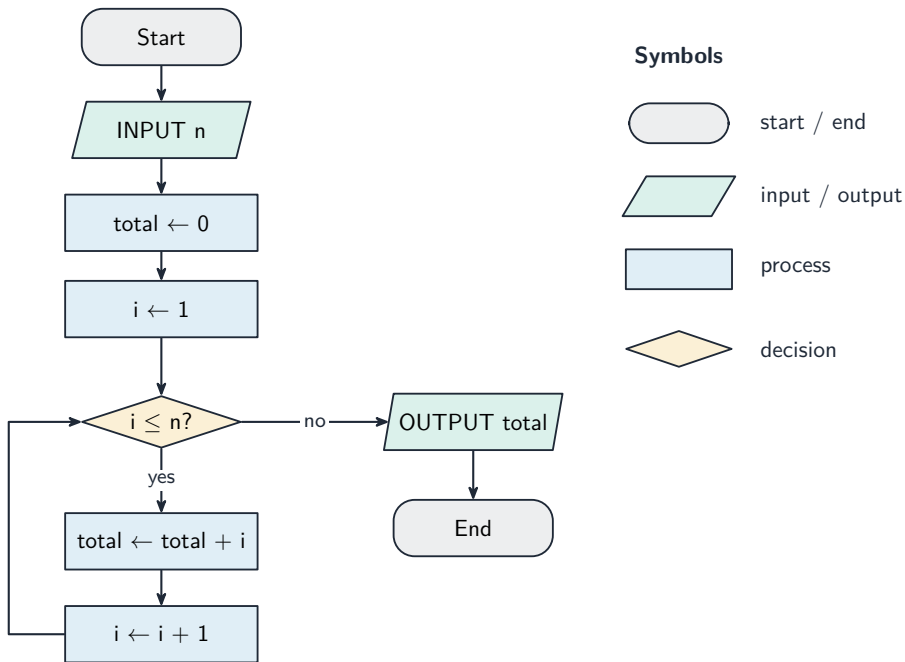
In **coding** you write the program code. You use **iterative testing** – test small parts again and again as you build them. In **testing** you run the whole program with **test data** to check it works.

Design tools

You can plan a solution in three main ways.

- a **structure diagram** – shows the parts of a system and how they fit together;
- a **flowchart** – a diagram using boxes and arrows to show the steps in order;

- **pseudocode** – steps written in simple, code-like English (not a real language).



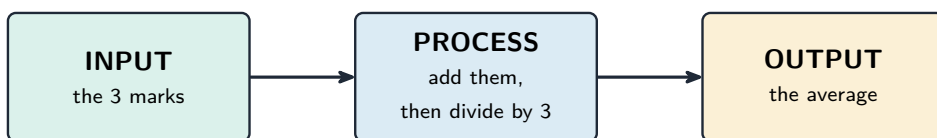
A flowchart for the sum algorithm, using the standard symbols (start/end, input/output, process, decision)

Algorithms

An **algorithm** is a set of steps, in the right order, that solves a problem. Every algorithm can be split into three parts:

- **input** – the data that goes in;
- **processing** – the work done on the data;
- **output** – the result that comes out.

This is called decomposition into inputs, processes and outputs. For example, for “find the average of three marks”: the inputs are the three marks; the processing is adding them and dividing by 3; the output is the average.



Every algorithm decomposes into input, processing and output – here, finding the average of three marks

Validation and verification

When data is entered, you check it to reduce mistakes.

Validation checks that the data is **sensible** and follows the rules. It cannot check that the data is true, only that it is allowed.

Validation check	What it checks
range check	the value is between a lowest and highest allowed value
length check	the number of characters is allowed (e.g. a password $\geq$ 8)
type check	the data is the right type (e.g. a number, not letters)
presence check	something has actually been entered (not left blank)
format check	the data is in the right pattern (e.g. a date as dd/mm/yyyy)
check digit	an extra calculated digit helps detect errors when a number is entered

Verification checks that data was **copied or entered correctly** (no mistakes while typing it in). Two methods:

- **visual check** – a person compares the typed data with the original;
- **double entry** – the data is entered twice and the two copies are compared.

Trace tables

A **trace table** records the value of each variable as an algorithm runs, step by step. It helps you:

count	total	output
1	5	
2	12	
3	20	20

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

- check that an algorithm works correctly;
- work out **what an algorithm does** by following it with given data.

Example: trace this algorithm with the input 5.

```
INPUT N
Total ← 0
FOR I ← 1 TO N
    Total ← Total + I
NEXT I
OUTPUT Total
```

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

The trace shows the algorithm adds up 1 to n. With input 5 the output is 15.

Worked example. Trace this algorithm and give the output.

```
X ← 20
Count ← 0
WHILE X > 1
    X ← DIV(X, 2)
    Count ← Count + 1
ENDWHILE
OUTPUT Count
```

DIV gives only the whole-number part of a division. Take one row per pass: x becomes 10 (count 1), then 5 (count 2), then 2 (count 3), then 1 (count 4). Now $x > 1$ is false, so the loop stops and the output is **4**. Two habits protect these marks: test the condition **before** each pass rather than after, and write a new row for **every** pass - trying to hold the values in your head is what makes traces go wrong.

Test data

Test data is data you use to test a program. There are four types you must know.

Type	Meaning	Example (age 0–120 allowed)
normal	sensible data that should be accepted	25
abnormal	wrong data that should be rejected	-4 or “cat”
extreme	the largest and smallest values still allowed	0 and 120
boundary	the values on each side of a limit (one allowed, one not)	120 and 121

Standard methods of solution

You must know these common algorithms.

■ Linear search

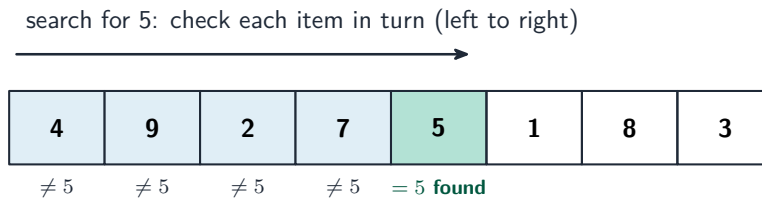
A **linear search** compares each item with the target, starting at the beginning. Stop when the value is found or the list ends. The list does not need to be sorted. The following example uses positions 1 to 4.

```

DECLARE List : ARRAY[1:4] OF INTEGER
DECLARE I, SearchValue : INTEGER
DECLARE Found : BOOLEAN
List[1] ← 8
List[2] ← 3
List[3] ← 5
List[4] ← 1
INPUT SearchValue
Found ← FALSE
I ← 1
WHILE I <= 4 AND NOT Found
    IF List[I] = SearchValue
        THEN
            Found ← TRUE
        ENDIF
    I ← I + 1
ENDWHILE
OUTPUT Found

```

Worked comparison. Searching for 5 compares 8, 3 and then 5: three comparisons, with Found becoming TRUE. The fourth item is not checked. Searching for 9 compares all four values; Found stays FALSE. Stop before trying to read position 5, which is outside this array.



Another example: linear search checks each item in turn from the start until it finds the value

■ Bubble sort

A **bubble sort** puts a list in order. Compare neighbouring values and swap them if the left value is greater. A single pass is not enough in general. This complete version repeats passes until a whole pass makes no swaps.

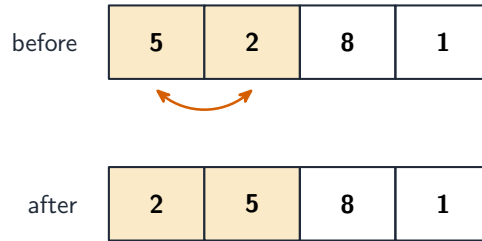
```

DECLARE List : ARRAY[1:4] OF INTEGER
DECLARE I, Temp : INTEGER
DECLARE Swapped : BOOLEAN
List[1] ← 8
List[2] ← 3
List[3] ← 5
List[4] ← 1
REPEAT
    Swapped ← FALSE
    FOR I ← 1 TO 3
        IF List[I] > List[I + 1]
            THEN
                Temp ← List[I]
                List[I] ← List[I + 1]
                List[I + 1] ← Temp
                Swapped ← TRUE
            ENDIF
    NEXT I
UNTIL NOT Swapped
OUTPUT List[1], ", ", List[2], ", ", List[3], ", ", List[4]

```

Worked passes. Start with 8, 3, 5, 1. In the first pass the list becomes 3, 8, 5, 1, then 3, 5, 8, 1, then 3, 5, 1, 8. This puts the largest value last but leaves the rest unsorted. The second pass ends at 3, 1, 5, 8; the third at 1, 3, 5, 8. The fourth makes no swaps and confirms that this version can stop. Reset **Swapped** to **FALSE** at the start of every pass; set it to **TRUE** on any swap.

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



then repeat for each pair, until no swaps are needed

One comparison in another example: bubble sort swaps neighbours that are out of order, then continues through the pass

■ Totalling and counting

Totalling adds each required value to a running total: $\text{Total} \leftarrow \text{Total} + \text{Value}$.

Counting adds 1 for each required item: $\text{Count} \leftarrow \text{Count} + 1$. Initialise both to zero before processing the values.

Worked example. Four attempted inputs are -20 , 4 , 50 and 6 . Accept a value only if it is greater than -20 and less than 50 . This example processes four attempts, not four accepted values.

```
DECLARE I, Count : INTEGER
DECLARE Value, Total : REAL
Total  $\leftarrow$  0
Count  $\leftarrow$  0
FOR I  $\leftarrow$  1 TO 4
    INPUT Value
    IF Value > -20 AND Value < 50
        THEN
            Total  $\leftarrow$  Total + Value
            Count  $\leftarrow$  Count + 1
        ENDIF
NEXT I
IF Count > 0
    THEN
        OUTPUT Total, ", ", Count, ", ", Total / Count
    ELSE
        OUTPUT "No accepted values"
    ENDIF
```

Attempted value	Total after attempt	Count after attempt
−20	0	0
4	4	1
50	4	1
6	10	2

The strict limits reject −20 and 50. Both updates belong inside the acceptance test, so rejected attempts change neither total nor count. The average of the accepted values is:

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

Do not divide by four: only two values were added. If no value is accepted, the count is zero and an average cannot be calculated; output a message instead of dividing by zero.

Transfer to a fixed number of accepted values. If a task asks for 50 accepted values, rather than four attempts, replace the fixed **FOR** loop with **WHILE Count < 50** and close it with **ENDWHILE**. Keep the same input and acceptance test inside it. Rejected inputs leave **Count** unchanged, so another input is requested. On completion **Count** is 50, although the number of attempts may be greater. Output the total and calculate the average as **Total / Count**.

■ Maximum, minimum and average

The **maximum** is the largest value and the **minimum** is the smallest. The **average** is the total divided by the number of values. For a non-empty list, initialise both from its first item. For each remaining item, replace the maximum if the new value is larger, and replace the minimum if it is smaller.

Worked example. The list is −8, −3, −12, −5. Starting the maximum at zero would fail: no list item is greater than zero, so the algorithm would output a number that is not in the list. Starting from −8 works without assuming that the data is positive.

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

```

DECLARE List : ARRAY[1:4] OF INTEGER
DECLARE I, Maximum, Minimum, Total : INTEGER
List[1] ← -8
List[2] ← -3
List[3] ← -12
List[4] ← -5
Maximum ← List[1]
Minimum ← List[1]
Total ← List[1]
FOR I ← 2 TO 4
    IF List[I] > Maximum
        THEN
            Maximum ← List[I]
    ENDIF
    IF List[I] < Minimum
        THEN
            Minimum ← List[I]
    ENDIF
    Total ← Total + List[I]
NEXT I
OUTPUT Maximum, ", ", Minimum, ", ", Total / 4

```

The maximum is -3 and the minimum is -12 . The average uses all four values:

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

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

The average lies between the minimum and maximum; this is a useful check, not proof that the calculation is correct. Initialising from the first item requires a non-empty list. Handle an empty list separately before trying to read that item.

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).