

Program development and design

Read online at pastopic.com/cambridge/computer-science-0478/notes/program-development-and-design

Last checked 30 September 2026

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

By the end of this topic you should be able to:

1. **Name the four stages of the program development life cycle**, in order: **analysis, design, coding, testing**, and say what is done at each one:
 - analysis: abstraction, decomposition of the problem, and identifying the problem and its requirements;
 - design: decomposition, structure diagrams, flowcharts and pseudocode;
 - coding: writing the program code, with iterative testing as it is written;
 - testing: running the program code with test data.
2. **Explain that every computer system is made of sub-systems**, and that each sub-system can be made of smaller sub-systems.
3. **Decompose a problem into its component parts**: inputs, processes, outputs and storage.
4. **Design and construct a solution with each of the three methods**: a structure diagram, a flowchart (using the standard symbols) and pseudocode.
5. **Explain the purpose of a given algorithm**: state what it is for, and describe the processes it carries out.

How to choose test data, fill in trace tables and find errors have their own topic, **Testing, trace tables and errors**; searching, sorting, totalling and counting are in **Standard methods of solution**. This note uses them only as far as the questions on design need them. Making a program easy to maintain (meaningful identifiers, comments) belongs to topic 8, Programming; it appears here only because one tagged design question asks about it.

Understand it

The program development life cycle

Programs are not written by sitting down and typing. A new program goes through four stages, one after the other.

1. **Analysis: what is the problem?** The programmer finds out exactly what the program must do. This means:

- **identifying the problem:** what is wrong or missing now, and who will use the program;
 - **identifying the requirements:** the list of things the finished program must do, which later decides whether it has succeeded (for example, "a member's time must be shown to the nearest second");
 - **abstraction:** keeping only the details that matter to this problem and leaving out the rest;
 - **decomposition:** breaking the problem into smaller parts.
2. **Design: how will it be solved?** The programmer plans the solution before writing any code, using structure diagrams, flowcharts and pseudocode, and decomposes the problem further into sub-systems.
 3. **Coding: build it.** The design is turned into program code in a programming language. Each part is run and tested as soon as it is written, and fixed before moving on. This is **iterative testing**.
 4. **Testing: does it work?** The finished program is run with **test data** chosen in advance, to check that it gives the right results and meets every requirement.

Abstraction

Abstraction means removing the details that are not relevant to the problem, so that you can concentrate on the ones that are. A route-planning app for a delivery van needs the roads, their lengths and the speed limits; it does not need the colour of the houses or the names of the shops. Leaving those out makes the problem simpler to solve.

Systems, sub-systems and decomposition

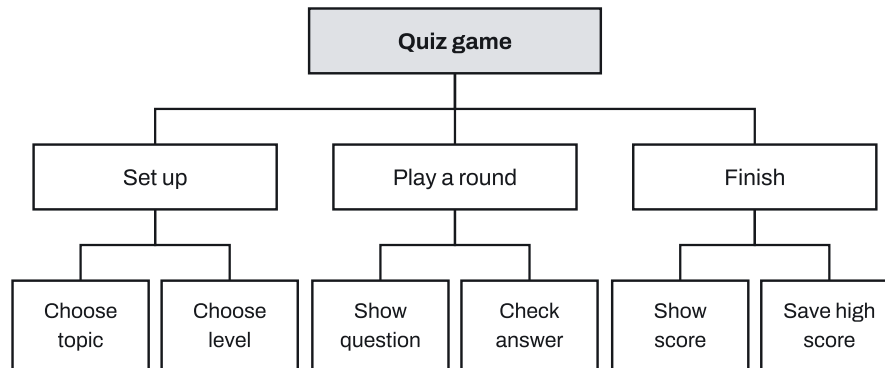
Every computer system is made of **sub-systems**, and each sub-system is made of smaller sub-systems, until each part is small enough to design and code on its own. A games console has a controller, a display system and a sound system; the display system itself has a part that draws the background and a part that draws the players.

Decomposition means breaking a problem into those smaller parts. The standard way to decompose a problem at IGCSE is into four **component parts**:

Component part	What it is	Example: a machine that makes a cup of tea
Inputs	the data that goes into the system	the drink chosen and the sugar level pressed on the buttons
Processes	what is done to the data	work out how much water to heat and for how long
Outputs	what comes out of the system	the cup of tea, and the message "Ready"
Storage	data that must be kept for later use	the recipe for each drink, the number of cups sold today

Structure diagrams

A **structure diagram** shows the decomposition as a hierarchy. The whole system is the single box at the top. Each level below splits the box above it into its sub-systems, joined by lines. It shows **what** the parts are, not the order in which they happen.

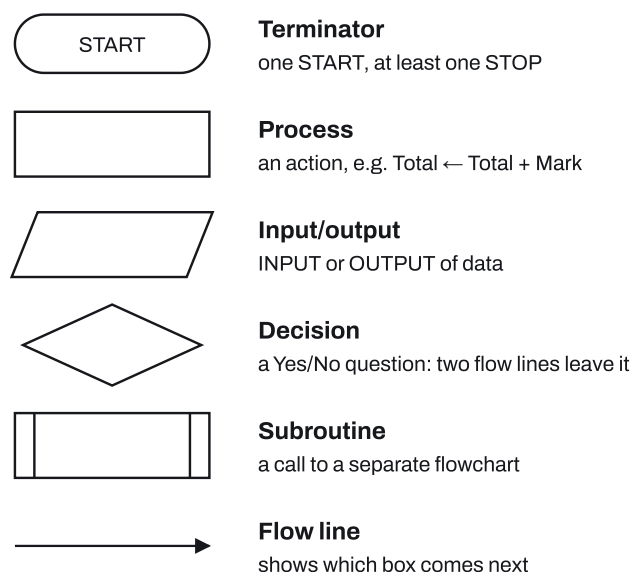


Rules for drawing one:

- one box at the top, with the name of the whole system;
- each box splits only into the boxes joined **below** it, so the lines always go down from a parent to its children;
- short names, one job per box, taken from the question.

Flowcharts

A **flowchart** shows the steps of an algorithm in the order they happen, using six standard symbols.



The rules:

- exactly one **START** and at least one **STOP** terminator;

- every box is joined by flow lines with arrows, so you can follow the path from START to STOP;
- a **decision** box holds only a question with a Yes/No answer, such as `IS Mark >= 50 ?`, and has exactly two lines leaving it, labelled **Yes** and **No**;
- calculations and assignments such as `Count ← Count + 1` go in **process** boxes; INPUT and OUTPUT go in **input/output** boxes;
- a **loop** is drawn by a flow line going back up to an earlier box, with a decision box deciding when to leave it.

Pseudocode

Pseudocode writes the steps as short statements that look like program code but belong to no programming language. Cambridge uses a fixed style: keywords in capitals, `←` for "becomes", and indentation to show what is inside an IF or a loop. The statements you will use most:

Pseudocode	Flowchart box
INPUT Mark, OUTPUT "Well done"	input/output
Total ← Total + Mark	process
IF ... THEN ... ELSE ... ENDIF	decision
WHILE ... DO ... ENDWHILE, REPEAT ... UNTIL ..., FOR ... TO ... NEXT	decision with a flow line back (a loop)
CALL ShowMenu	subroutine

For example, this counts how many marks of 50 or more are entered, stopping when `-1` is entered. For the marks 62, 48, 50, 71 and then `-1`, it outputs 3.

```

Passed ← 0
INPUT Mark
WHILE Mark <> -1 DO
    IF Mark >= 50
        THEN
            Passed ← Passed + 1
        ENDIF
    INPUT Mark
ENDWHILE
OUTPUT Passed

```

The three design methods show the same solution in different ways: a structure diagram shows the parts, a flowchart shows the order as a picture, and pseudocode shows the order as statements that are easy to turn into code.

Explaining the purpose of an algorithm

You will often be given an algorithm and asked what it does. There are two different questions:

- **State the purpose:** one sentence saying what the algorithm is **for**, in terms of its inputs and its output, for example "it finds and outputs the number of marks of 50 or more in a list of marks ending with -1 ".
- **Describe the processes:** the separate things it does, in order, for example "inputs marks until -1 is entered; checks each mark against 50; counts the marks of 50 or more; outputs the count".

To work either out, dry-run the algorithm with a few simple values and look at what comes out.

Making an algorithm easy to understand

Someone else may have to read, fix or update your program later (maintain it). It is easier to maintain if it uses:

- **meaningful identifiers:** `Total`, `Names[ ]`, `Counter` rather than `T`, `A`, `C`;
- **comments** (`// ...`) explaining what each part does;
- **indentation and white space**, so you can see what is inside each loop or `IF`;
- **subroutines** (procedures and functions) for jobs that are done more than once.

Key facts and formulas

There are no formulas in this topic and nothing is given in the exam. These are the facts to learn.

Fact	Status
Life cycle stages, in order: analysis → design → coding → testing	Learn it
Analysis: abstraction, decomposition, identifying the problem and the requirements	Learn it
Design: decomposition, structure diagrams, flowcharts, pseudocode	Learn it
Coding: writing program code, with iterative testing; testing: running the code with test data	Learn it
Abstraction: removing details that are not relevant to the problem	Learn it
Component parts: inputs, processes, outputs, storage	Learn it
Every system is made of sub-systems, which are made of further sub-systems	Learn it
Design methods: structure diagram, flowchart, pseudocode	Learn it
Flowchart symbols: terminator, process, input/output, decision, subroutine, flow line	Learn it
Purpose = what the algorithm is for; processes = the steps it carries out	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (41 questions, 2021–2025), the parts on design came in these types. 16 of the 41 questions mix more than one type, so the counts add up to more than 41.

Question type	Questions
Stages of the program development life cycle	10
Design methods and flowchart symbols	9
Dry-run a given algorithm	9
Complete or draw a flowchart	8
State the purpose or describe the processes of an algorithm	8
Explain how to change an algorithm, or correct its errors	8
Decomposition and structure diagrams	5
Write pseudocode from a design	4
Make an algorithm easier to understand	1

1. Stages of the program development life cycle (10 questions)

1. Learn the four stages **in order**, so "the first stage" is always **analysis**.
2. Learn what is done at each stage, from the syllabus lists in "Understand it".
3. For "**identify**", give the name only. For "**describe**", give the stage, what happens, and one more detail or example.

Variations you will meet:

- **Match descriptions to stages** (one line each, extra boxes not used). Look for key words: "requirements" or "identify the problem" → analysis; "structure diagrams, flowcharts, pseudocode" or "plan" → design; "write", "implement", "programming language" → coding; "test data", "detect errors", "works as expected" → testing. "Evaluation" and "reviewing the solution" are **not** stages in this syllabus.
- **Pick the stages out of a list of words** mixed with other terms (decomposition, pseudocode, variable). Only analysis, design, coding and testing are stages.
- **Tasks in the analysis stage**: abstraction; decomposition; identifying the problem; identifying the requirements.
- **Describe abstraction**: removing (hiding) details that are not relevant so the problem is simpler; keeping only the key elements. In a multiple-choice question, reject "testing", "writing code" and "defining sub-systems" (that is decomposition).
- **Identify and describe another stage** (6 marks for two): stage (1), what happens (1), a further detail (1). For example: design (1), the solution is planned (1) using structure diagrams, flowcharts and pseudocode (1).

2. Design methods and flowchart symbols (9 questions)

1. The three **design methods** are structure diagrams, flowcharts and pseudocode. Nothing else counts: not "iteration", not "inputs and outputs", not "test data".

2. To **describe** a method, add what it shows: a structure diagram shows the problem broken down into sub-systems in a hierarchy; a flowchart shows the steps in order using symbols; pseudocode shows the steps as plain-language statements, not in any real language.
3. For **symbols**, match the statement to the box: INPUT/OUTPUT/PRINT → input/output (parallelogram); ← assignments and calculations → process (rectangle); IF and loop conditions → decision (diamond); START/STOP → terminator (rounded box); a separate flowchart → subroutine (rectangle with double sides); arrows → flow lines.

Variations: draw the symbol for a statement (draw the shape and write the statement in it); name the kind of statement, for example IF `Number = 20` is selection, PRINT `Number` is output and `Number ← Number + 1` is counting.

3. Dry-run a given algorithm (9 questions)

1. Make one column for every variable, and one for the output.
2. Follow the flowchart one box at a time, with the first input value. Write a new value in a column **only when it changes**, one row further down each time a value changes.
3. At each decision, work out Yes or No with the current values and follow that line.
4. Write each output exactly, including any message, in the output column.
5. Stop at STOP; any input values left over are never used.

The full method, with test data and errors, is in the note on testing, trace tables and errors. In a design question, the dry run is usually the way into the next part: the pattern in the outputs tells you the algorithm's purpose.

4. Complete or draw a flowchart (8 questions)

1. **Read the whole description** and underline the variable names you are given. Use them exactly; don't invent your own.
2. List the jobs in order: **initialise** (usually given) → **input** → **check** (decisions) → **update** (totals and counters) → **loop test** → **output** after the loop.
3. Choose the right box for each job, and put only a question in each decision box.
4. Write each condition precisely, with the operator the description needs: "between 16 and 20 inclusive" is `Value >= 16 AND Value <= 20`; "rejects values of 0 or less and of 100 or more" is `Value <= 0 OR Value >= 100`.
5. Label every decision's two lines **Yes** and **No**, draw arrows on every flow line, and make sure the loop goes back to the **input** box.
6. Put the final OUTPUT (and any average) **after** the loop, not inside it.

Variations you will meet:

- **A count-controlled loop** ("50 numbers"): `Count ← 1` at the start, `Count ← Count + 1` in the loop, and leave the loop with `IS Count > 50 ?` (or start at 0 and test `= 50`).

- **A rogue value** ("until -1 is entered"): test the input with a decision straight after the input box; Yes goes to the outputs.
- **Rejecting bad input**: a decision after the input; the "bad" branch goes back to the input box (with an error message if asked) without counting the value.
- **A prompt**: an OUTPUT "Enter a number between 1 and 100" box before the input box.
- **Two outcomes** (accepted/rejected, pass/fail): a decision whose two branches each hold an output, or each update a different counter or array.

5. State the purpose or describe the processes (8 questions)

1. Dry-run the algorithm with two or three simple inputs.
2. Look at what the outputs have in common with the inputs: are they counted, totalled, sorted, compared, rejected?
3. **Purpose**: one sentence naming the result, for example "to find and output the largest of 20 numbers", "to convert a number of minutes into hours and minutes and output them". Be specific: "it processes numbers" earns nothing.
4. **Processes**: list the steps in order, one mark each, using the algorithm's own variable names: what is input, what is checked (and against what), what is counted or totalled, what is stored and where, when it stops, what is output.

6. Explain how to change an algorithm, or correct its errors (8 questions)

For "explain how you would alter ...", every mark point says **where** and **what**:

1. **New variable?** Say so and where it is initialised (at the start, before the loop).
2. **New step inside the loop?** Say exactly where (for example "in the Yes branch, just after the counter is increased") and what it does.
3. **New step after the loop?** Calculations such as an average, and the final output, go after the loop.
4. You may write pseudocode or draw boxes, but explain each one in words.

Common changes:

- an **average**: initialise a total, add each value in the right branch, divide by the right count **after** the loop, output it;
- **store every value**: input into an array with the counter as the index, then a second loop after the first to output the array;
- **any class size** or **any number of items**: input the number first, and use it as the loop's end condition;
- **accept lower-case letters**: after each input, a process box that converts it with UCASE, or change every decision to test both cases with OR;
- **stop an endless loop or a divide-by-zero**: a new decision that rejects the problem input and goes back to the input.

"Identify another problem with the inputs": look for a missing prompt, a possible division by 0, a letter typed where a number is expected, nothing typed at all, or an option letter the algorithm never checks (so it falls through to the last branch).

For **"identify the errors and suggest corrections"**, write the line (or box), what is wrong and the corrected statement. Look for: Yes/No labels swapped, AND where OR is needed (or the reverse), INPUT where OUTPUT is needed, the wrong variable name, an arrow pointing the wrong way, a statement in the wrong kind of box, and a total or counter reset inside a loop.

7. Decomposition and structure diagrams (5 questions)

1. **Component parts**: inputs, processes, outputs, storage. Give the names, or match each to an example from the scenario: the data typed in is an input, a calculation is a process, the result shown is an output, and a detail the problem doesn't need matches none of them.
2. **Structure diagram**: the system's name in one box at the top; under it one box for each sub-system the question names (often an input part and an output part); under each of those, the smaller parts the bullet points list.
3. Use **only** the parts the question gives, and join every box by a line to the one above it.

8. Write pseudocode from a design (4 questions)

1. Turn each flowchart box into one statement: input/output box → INPUT/OUTPUT; process box → an assignment with ←; decision → IF ... THEN ... ENDIF; a flow line going back up → a loop.
2. Choose the loop: FOR ... NEXT for a fixed number of times; WHILE ... DO ... ENDWHILE when the test comes first; REPEAT ... UNTIL when the body runs at least once.
3. Declare the variables if asked (DECLARE Count : INTEGER), use the given names and keep Cambridge's keywords and indentation.

9. Make an algorithm easier to understand (1 question)

Suggest **meaningful identifiers** that say what each item holds (Names for the array, Temp for a swap store, Counter, Length), then other ways: comments, indentation and white space, procedures and functions.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.

Example 1

A running club asks a student to write a program that records members' times in a weekly 5 km race and prints a ranking.

(a) For each task, name the stage of the program development life cycle it belongs to. [4]

- Interviewing the club secretary to find out what the ranking must show.
- Drawing a flowchart for putting the times in order.
- Writing the program in Python and running each part as soon as it is finished.
- Entering the times 00:00 and 99:59 to see whether they are rejected.

(b) List **two** details about each race that the ranking program can leave out, and say why leaving them out helps. [2]

(c) The student's plan for the analysis stage has three tasks: (i) deciding what the ranking must show; (ii) splitting the problem into inputs, processes, outputs and storage; (iii) drawing a structure diagram of the program. State which task does not belong in analysis, and name the stage it belongs to. [2]

(a) Interviewing: analysis **B1**. Flowchart: design **B1**. Writing and running each part: coding (this is iterative testing) **B1**. Entering chosen times after it is finished: testing **B1**.

(b) Any two details the ranking doesn't need, such as the weather on race day, the route of the course or each runner's shoe brand **B1**. Leaving them out (abstraction) means less data to input and store, so the program only has to deal with names and times and is simpler to design **B1**.

(c) Task (iii), drawing the structure diagram **B1**, belongs in the design stage **B1**. Tasks (i) and (ii) are identifying the requirements and decomposition, both part of analysis.

Example 2

A computer controls a greenhouse. Every minute it reads a soil-moisture sensor and a temperature sensor. If the soil is too dry it switches on a water pump; if the air is too hot it opens a roof window with a motor. The gardener types the target moisture and temperature on a keypad, and these are kept by the system.

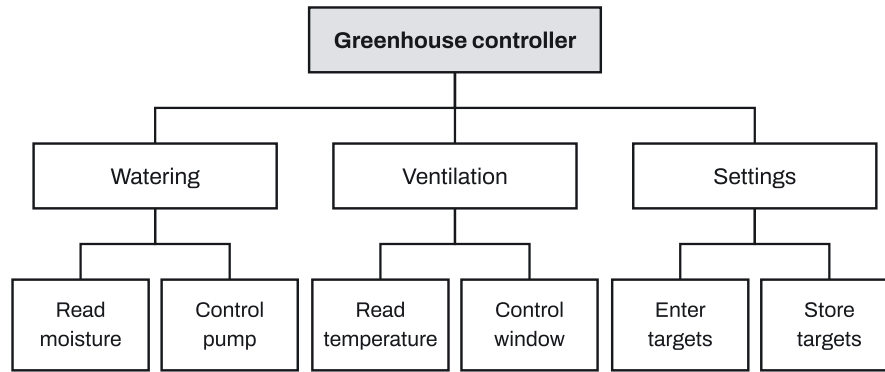
(a) The problem is decomposed into its component parts. Name **four** component parts and give an example of each from this system. [4]

(b) The system is made of three sub-systems: watering, ventilation and settings. Draw a structure diagram with two sub-systems under each of these. [3]

(a)

- Inputs: the sensor readings, and the targets typed on the keypad. **B1**
- Processes: comparing each reading with its target. **B1**
- Outputs: the signals that switch the pump on or off and open or close the window. **B1**
- Storage: the target values, kept until the gardener changes them. **B1**

(b)



B1 for a hierarchy with one box at the top and the three sub-systems below it. **B1** for two suitable sub-systems under watering and under ventilation. **B1** for two suitable sub-systems under settings.

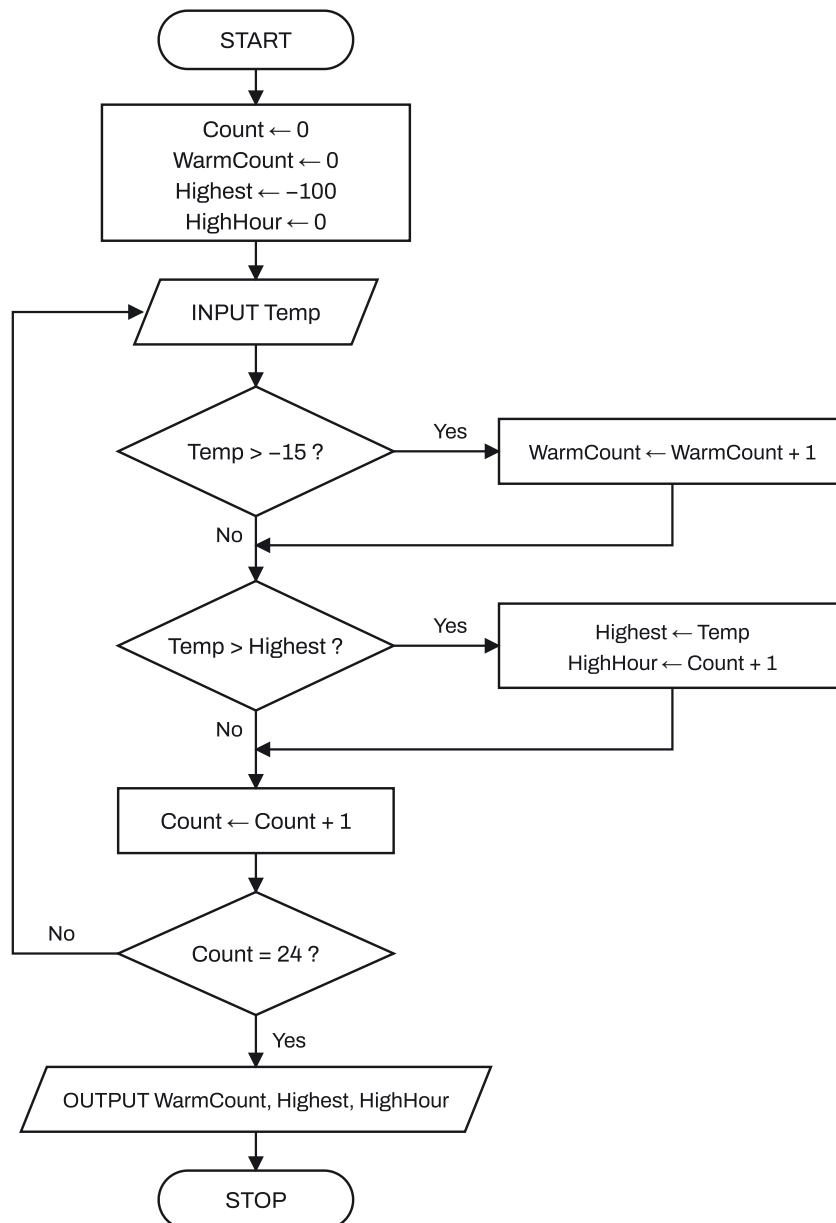
Example 3

A freezer takes its temperature once an hour, 24 times, and each reading is input into `Temp`. The algorithm must count in `WarmCount` how many readings are above -15°C , find the highest reading in `Highest`, and output both after the last reading. `Count`, `WarmCount` and `Highest` are set to 0, 0 and -100 at the start.

(a) Draw the flowchart. [6]

(b) Explain how you would change the flowchart so that it also outputs the hour (1 to 24) of the highest reading. [3]

(a) The flowchart is below; the `HighHour` parts are the answer to (b).



- START, the initial values, then INPUT Temp in an input/output box. **B1**
- A decision Temp > -15 ? whose Yes branch increases WarmCount. **B1**
- A decision Temp > Highest ? whose Yes branch sets Highest ← Temp. **B1**
- Count ← Count + 1, then a decision Count = 24 ? whose No line loops back to the input box. **B1**
- After the loop, OUTPUT WarmCount, Highest, then STOP. **B1**
- Correct symbols, arrows on every flow line, and Yes/No on every decision. **B1**

(b)

- Add a new variable HighHour, set to 0 in the first process box. **B1**
- In the Yes branch of Temp > Highest ?, next to Highest ← Temp, add HighHour ← Count + 1 (the count has not been increased yet, so the first reading is hour 1). **B1**
- Add HighHour to the output box after the loop. **B1**

A quick dry run with 4 readings instead of 24 (test Count = 4): $-18, -14, -20, -12$ gives WarmCount = 2, Highest = -12 and HighHour = 4.

Example 4

```
01 OUTPUT "Enter a year"
02 INPUT Year
03 Leap ← FALSE
04 IF Year MOD 4 = 0
05     THEN
06         Leap ← TRUE
07 ENDIF
08 IF Year MOD 100 = 0
09     THEN
10         Leap ← FALSE
11 ENDIF
12 IF Year MOD 400 = 0
13     THEN
14         Leap ← TRUE
15 ENDIF
16 IF Leap
17     THEN
18         OUTPUT Year, " has 366 days"
19     ELSE
20         OUTPUT Year, " has 365 days"
21 ENDIF
```

- (a) State the purpose of this algorithm. **[1]**
- (b) Describe the processes in this algorithm. **[3]**
- (c) Explain how you would change the algorithm so that it keeps asking for years until the user enters 0. **[3]**
- (d) Name the flowchart symbol you would use for line 02 and for line 04. **[2]**

Dry-running it first: 1900 → "1900 has 365 days", 2000 → "2000 has 366 days", 2024 → "2024 has 366 days", 2023 → "2023 has 365 days".

(a) To decide whether a year is a leap year, and output how many days it has. **B1**

(b) Any three:

- it asks for and inputs a year **B1**;
- it checks whether the year divides exactly by 4, by 100 and by 400, using MOD **B1**;
- it sets Leap to TRUE for multiples of 4, back to FALSE for multiples of 100, and to TRUE again for multiples of 400 **B1**;

- it outputs the year with 366 days if `Leap` is `TRUE`, otherwise with 365 days **B1**.

(c)

- Put lines 03 to 21 inside a `WHILE Year <> 0 DO ... ENDWHILE` loop, straight after the first input. **B1**
- At the end of the loop body (after line 21), add a new prompt and `INPUT Year`, so the next year is read before the condition is tested again. **B1**
- The loop ends when 0 is entered, and 0 itself is never checked. **B1**

(d) Line 02 (`INPUT Year`): input/output (parallelogram) **B1**. Line 04 (`IF Year MOD 4 = 0`): decision (diamond) **B1**.

Common mistakes

- **Giving the wrong kind of answer for "design methods"**. Examiners saw iteration, selection and sequence, and inputs/processes/outputs, given as ways to present a design. The three methods are structure diagrams, flowcharts and pseudocode.
- **Forgetting storage**. When asked for the component parts after decomposition, many candidates could not give all of them. There are four: inputs, processes, outputs **and storage**.
- **Adding extra boxes to a structure diagram**. Use only the sub-systems the question describes, arranged as a hierarchy; boxes for things the question never mentions don't earn marks.
- **Inventing your own variable names**. Completed flowcharts lost marks when candidates used their own names, or ignored the arrays they were given. Use the names in the question, spelt the same way every time, with no spaces.
- **Putting the wrong thing in a box**. An `OUTPUT` in a decision box, a question in a process box, or a decision with no Yes/No labels loses marks. Each symbol has one job.
- **A vague purpose**. "It works with numbers" earns nothing, and guessing a familiar purpose ("finds prime numbers") when the algorithm counts factors is wrong. Dry-run it and describe what it actually outputs.
- **Explaining a change without saying where**. Changes to a flowchart were often correct in idea but lost marks because the answer didn't say where each new box goes. Say "before the loop", "in the Yes branch", "after the loop".
- **Describing when asked to explain**. An "explain" question wants how something is done and why; a list of what is done gets fewer marks.
- **Answering in the wrong form**. If the question asks for pseudocode, a flowchart earns nothing, and the reverse. Flowcharts drawn instead of pseudocode also tended to lack detail.
- **Writing trace tables in pencil and going over them in ink**. Both answers show when scanned, and the examiner can't tell which is meant.

How to get the marks

- **Identify / state / name:** a word or short phrase each (for example "design"). **Describe:** say what it is or what happens. **Explain:** say how or why, with a reason.
- **Two-part answers** ("identify and describe"): one mark for the name, one for what happens, one for a further detail. Write each on its own line so each point is clear.
- **Match the context.** In "decompose this system" or "draw a structure diagram for this system", the marks are for names taken from the scenario, not general words.
- **Flowcharts** are marked point by point: input, each check, each update, the loop test, the outputs, then one mark for correct symbols, arrows and Yes/No labels throughout. A flowchart that is right except for its arrows still earns most marks, so draw every part you can.
- **Precision:** write conditions with operators (`Speed >= 30 AND Speed <= 50`), not words such as "weight is between". "Inclusive" means `>=` and `<=`.
- **"Explain how you would alter":** one mark per change, each with where and what. You may add pseudocode or boxes, but explain them.
- **Purpose** questions are usually one mark: one clear sentence. **Processes** are one mark per separate step, so give as many steps as there are marks.

Quick check

1. Name the stage of the program development life cycle for each task: (i) listing everything a new booking app must do before any planning starts; (ii) writing pseudocode for the app's log-in routine; (iii) fixing an error the translator reports while the program is being written.
2. Name the flowchart symbol used for each statement: `Total ← Total + Price`, `IS Age >= 18 ?`, `OUTPUT "Done"`, `CALL ShowMenu`.
3. A museum ticket machine lets a visitor choose the number of adult and child tickets, works out the price, takes payment, prints the tickets and keeps a record of the day's sales. Identify one input, one process, one output and one item of storage.
4. State the purpose of this algorithm.

```

Smallest ← 1000
FOR Count ← 1 TO 10
    INPUT Number
    IF Number < Smallest
        THEN
            Smallest ← Number
    ENDIF
NEXT Count
OUTPUT Smallest

```

5. This algorithm should output the total of five prices, but it doesn't. Identify the error and correct it.

```
FOR Count ← 1 TO 5
  Total ← 0
  INPUT Price
  Total ← Total + Price
NEXT Count
OUTPUT Total
```

Answers

1. (i) Analysis (identifying the requirements); (ii) design; (iii) coding (iterative testing as it is written).
2. Process; decision; input/output; subroutine.
3. Input: the numbers of adult and child tickets (or the payment). Process: working out the total price from the numbers of tickets. Output: the printed tickets (or the price shown). Storage: the record of the day's sales (or the ticket prices).
4. To input 10 numbers and find and output the smallest one. For example, for 12, 7, 30, 7, 19, 4, 88, 51, 6 and 40 it outputs 4. (It assumes every number is below 1000.)
5. `Total ← 0` is inside the loop, so the total is reset to 0 for every price and only the last price is output: for 3, 5, 2, 8 and 4 it outputs 4 instead of 22. Move `Total ← 0` to before the `FOR` line.

Past questions to try

- [0478/22/M/J/25 Q2](#): matching the component parts of a decomposed problem to examples.
- [0478/21/O/N/25 Q7](#): dry-run a flowchart, state its purpose and rewrite it in pseudocode.
- [0478/23/O/N/24 Q9](#): find and correct four errors in a flowchart.
- [0478/22/O/N/24 Q5](#): another life-cycle stage, and the tasks of the analysis stage.