

Algorithm design and problem-solving

AS

Read online at pastopic.com/cambridge/computer-science-9618/notes/algorithm-design-and-problem-solving

Last checked 5 October 2026

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

What you need to know

This topic is examined in Paper 2. By the end of it you should be able to:

1. **Abstraction:** explain why it is needed and what it gives you, describe its purpose, and build an abstract model of a system that keeps only the details that matter.
2. **Decomposition:** describe it and use it to split a problem into sub-problems, each of which becomes a program module (a procedure or a function).
3. **What an algorithm is:** a solution to a problem written as a sequence of clearly defined steps.
4. **Identifiers:** choose sensible names for the data a problem uses, and record them in an **identifier table**.
5. **Write pseudocode** that has input, processing and output, built from the three basic constructs: **sequence**, **selection** and **iteration** (repetition).
6. **Document an algorithm** in three ways: **structured English**, a **program flowchart** or **pseudocode**.
7. **Convert between them:** write pseudocode from structured English or from a flowchart, and draw a flowchart from structured English or from pseudocode.
8. **Stepwise refinement:** describe it and use it to break an algorithm down until each step is detailed enough to be programmed.
9. **Logic statements:** use conditions built with relational operators and AND, OR and NOT to define parts of an algorithm.

Structure charts and state-transition diagrams belong to the Software development topic, but Paper 2 often puts them in the same questions as this topic, so "How to do it" shows you how to answer them too.

Understand it

Algorithms

An **algorithm** is a sequence of defined steps that solves a problem. "Defined" means each step is exact: someone (or a computer) following it never has to guess. Almost every algorithm has three parts:

- **input:** the data it is given (typed in, read from a file, or passed as a parameter);
- **process:** what it does with that data (calculations, decisions, repeats);
- **output:** the result (displayed, written to a file, or returned).

Abstraction

Abstraction means building a model of a problem that keeps only the details you need for the task and leaves out the rest.

A school's records about a student might include name, address, date of birth, tutor group, medical notes, timetable, exam results and a photo. A module that emails parents about a cancelled trip needs only the student's name, the parent's email address and whether the student was booked on the trip. Everything else is filtered out.

Why it helps:

- the model is **simpler**, so the solution is quicker to design, write and test;
- less data is stored and processed, so the program needs less memory and runs faster;
- the people who will use the system can follow the model, because it shows only what matters to them.

Abstraction is not the same as hiding data from users for security, and it happens while the problem is being designed, not afterwards to a finished program.

Decomposition

Decomposition means breaking a problem down into smaller **sub-problems**, then breaking those down further until each one is small enough to solve on its own. Each sub-problem becomes a **module**: a **procedure** (it does a job) or a **function** (it does a job and returns one value).

For an online cinema booking system, the problem "let customers book seats" splits into sub-problems such as: show today's films, show the free seats for a chosen screening, take the customer's choice of seats, take payment, and email a confirmation. Each becomes a module.

Why it helps: each module is easier to understand, write and test than the whole program; different programmers can work on different modules at the same time; and a module (for example, "take payment") can be reused in other programs.

Identifiers and the identifier table

An **identifier** is the name you give a variable, constant, array or module. Good identifiers:

- describe what they hold (`TotalCost`, not `X` or `Data1`);
- start with a letter and contain only letters and digits, with no spaces (Cambridge pseudocode joins words with capitals: `DateSold`);
- are not keywords such as `STRING`, `INPUT` or `DATE`.

An **identifier table** lists each identifier with its data type and what it is for:

Identifier	Data type	Description
PlayerName	STRING	the name typed in by the player
Score	INTEGER	number of points scored so far
AverageTime	REAL	mean time per answer, in seconds
GameOver	BOOLEAN	TRUE once the player has no lives left
Initial	CHAR	first letter of the player's name
LastPlayed	DATE	the date of the player's last game

Choose the type from the values it will hold: whole numbers are INTEGER; numbers that can have a decimal part (money, averages, measurements) are REAL; TRUE/FALSE values are BOOLEAN; a single character is CHAR; text, and codes such as phone numbers or IDs that you never do arithmetic on, are STRING; dates are DATE.

Writing pseudocode

Pseudocode is a way of writing an algorithm that looks like a program but is not tied to any language. Cambridge uses a fixed style; use it exactly.

```
DECLARE Count : INTEGER
DECLARE Prices : ARRAY[1:20] OF REAL
CONSTANT MaxItems = 20
```

```
Count ← 0                // assignment uses ←
INPUT Price              // input
OUTPUT "Total: ", Total  // output (a comma joins items)
Name ← First & " " & Last // & joins strings
```

- **Arithmetic:** +, −, *, / (gives a REAL), DIV (whole-number division) and MOD (the remainder). 17 DIV 5 is 3 and 17 MOD 5 is 2.
- **Comparisons:** =, <> (not equal), <, >, <=, >=.
- **Arrays:** declare the bounds and type, then use the index in square brackets: `Prices[3] ← 4.99`. The bounds don't have to start at 1: if a value from 0 to 40 will be used as the index, declare `ARRAY[0:40]`.
- **Built-in functions** are defined in the insert that comes with Paper 2. The ones you'll use most are `LENGTH(S)`, `MID(S, Start, Count)`, `LEFT(S, Count)`, `RIGHT(S, Count)`, `TO_UPPER(S)`, `NUM_TO_STR(N)`, `STR_TO_NUM(S)`, `IS_NUM(S)`, `INT(X)` (the whole-number part) and `RAND(X)` (a random real number from 0 up to, but not including, X). Use only functions in the insert, never ones from Python or another language.

Text files use `OPENFILE "Data.txt" FOR READ (or WRITE, or APPEND), READFILE "Data.txt", Line, WRITEFILE "Data.txt", Line, EOF("Data.txt")` (TRUE at the end of the file) and `CLOSEFILE "Data.txt"`.

The three constructs

Every algorithm is built from three constructs.

Sequence: statements run one after another, in order.

Selection: a condition decides which statements run.

```
IF Mark >= 50 THEN
    OUTPUT "Pass"
ELSE
    OUTPUT "Fail"
ENDIF
```

```
CASE OF Choice
    'A' : CALL AddItem()
    'D' : CALL DeleteItem()
    OTHERWISE : OUTPUT "Invalid choice"
ENDCASE
```

Iteration (repetition): statements are repeated. There are three kinds of loop:

```
FOR Index ← 1 TO 10                // count-controlled: you know how many times
    Total ← Total + Prices[Index]
NEXT Index
```

```
WHILE Value <> -1                    // pre-condition: tested before each pass,
    INPUT Value                      // so it may run 0 times
ENDWHILE
```

```
REPEAT                             // post-condition: tested after each pass,
    INPUT Guess                     // so it always runs at least once
UNTIL Guess = Secret
```

```
FOR Points ← 0 TO 20 STEP 5 counts in fives: 0, 5, 10, 15, 20.
```

How to choose: use **FOR** when the number of repeats is known before the loop starts (every element of an array, every character of a string, 52 lines of a file). Use a **conditional loop** (WHILE or REPEAT) when the loop must stop as soon as something happens (a value is found, a rogue value is input, the end of a file is reached). A search that uses FOR keeps checking after the item has been found, which wastes time.

Modules: procedures and functions

```
PROCEDURE ShowLine(Symbol : CHAR, Length : INTEGER)
  DECLARE Count : INTEGER
  FOR Count ← 1 TO Length
    OUTPUT Symbol
  NEXT Count
ENDPROCEDURE
```

```
FUNCTION Square(Number : INTEGER) RETURNS INTEGER
  RETURN Number * Number
ENDFUNCTION
```

A procedure is used with `CALL ShowLine( ' ', 10 )`. A function returns a value, so it is used inside an expression: `Area ← Square(Side)`. A parameter declared `BYREF` lets the module change the caller's variable; the default, `BYVAL`, passes a copy.

Logic statements

A **logic statement** is a condition that is TRUE or FALSE, used in an IF, a CASE or a loop. Join conditions with:

- **AND**: TRUE only if both sides are TRUE;
- **OR**: TRUE if either side (or both) is TRUE;
- **NOT**: swaps TRUE and FALSE.

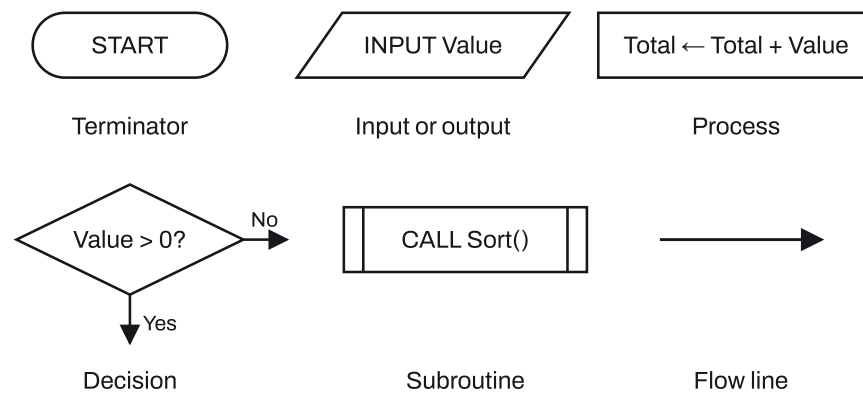
NOT is worked out first, then AND, then OR; brackets make your meaning clear. For example, with `Age ← 15` and `Member ← FALSE`, the condition `Age >= 13 AND NOT Member` is TRUE AND TRUE, which is TRUE.

Ranges are where marks are lost. "Between 10 and 20 inclusive" is `X >= 10 AND X <= 20`; "outside 10 to 20" is `X < 10 OR X > 20`. Each is the NOT of the other. In general, `NOT (A OR B)` is the same as `NOT A AND NOT B`, and `NOT (A AND B)` is the same as `NOT A OR NOT B`. Use these to simplify a condition: `(X > Y) OR (X = Y)` is simply `X >= Y`.

Structured English, flowcharts and pseudocode

The same algorithm can be written in three ways.

- **Structured English**: numbered steps in plain words, using a few fixed words such as "input", "if", "repeat" and "until". It is easy for anyone to follow but not exact enough to type in as code.
- **Program flowchart**: a diagram of boxes joined by arrows. It shows the order and the decisions clearly.
- **Pseudocode**: the most detailed of the three, and the closest to real code.



In a flowchart a loop is drawn as a decision box whose exit arrow goes **back up** to an earlier box; there is no "loop" symbol. Every decision box has exactly two exits, labelled Yes and No (or TRUE and FALSE), and every arrow leads somewhere.

Converting between the forms keeps the logic exactly the same. Each process box becomes an assignment, each input/output box an `INPUT` or `OUTPUT`, a decision that skips forward becomes an `IF`, and a decision that sends you back up becomes a loop (`WHILE` if the test comes before the repeated boxes, `REPEAT ... UNTIL` if it comes after). Going the other way, each line of pseudocode becomes one box.

Stepwise refinement

Stepwise refinement means starting with a short outline of an algorithm and repeatedly breaking each step into smaller, more detailed steps, until every step is simple enough to be written directly as a line of code. Decomposition splits the *problem* into modules; stepwise refinement adds detail to the *algorithm* inside a module.

For example, "calculate the average of the valid marks" refines to:

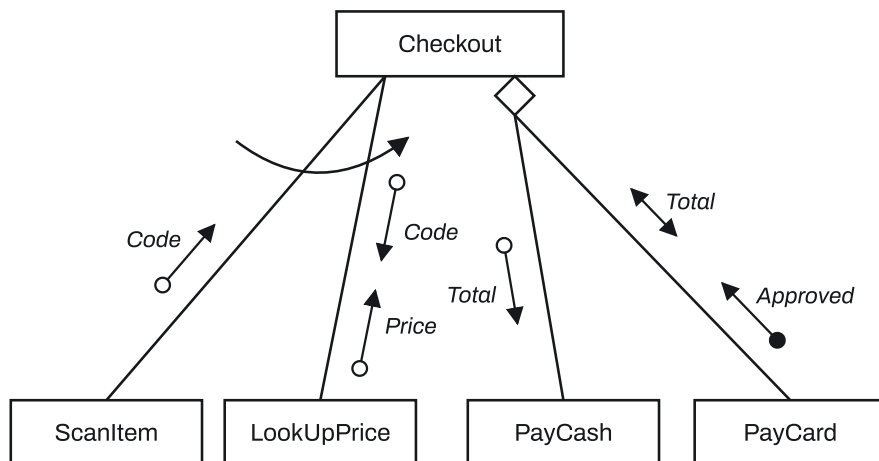
1. set Total and Count to 0;
2. repeat for each of the 30 marks:
 1. input the mark;
 2. if the mark is between 0 and 100 inclusive, add it to Total and add 1 to Count;
3. if Count is more than 0, output Total divided by Count; otherwise output "No valid marks".

Program design diagrams that appear with this topic

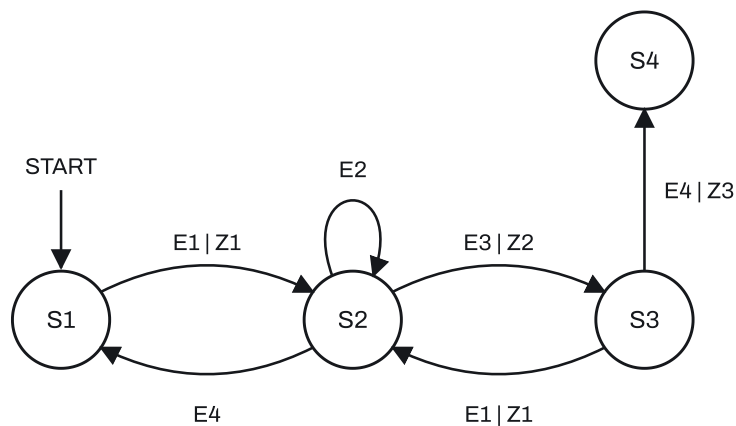
A **structure chart** shows how the modules of a program fit together. Each box is a module; a line from a box down to another means "calls". Modules are called from left to right. Small arrows beside each line show the data passed:

- an arrow with an **open circle** carries data: pointing down for a parameter, up for a returned value;
- an arrow with a **filled circle** carries a flag, a BOOLEAN value;
- a **two-headed arrow** shows a `BYREF` parameter (it goes down and its new value comes back up);

- a **diamond** where the lines leave a box means the module calls **either** one module **or** another (selection);
- a **curved arrow** across lines means the module calls those modules **repeatedly** (iteration).



A **state-transition diagram** shows how a system moves between **states**. Each state is a circle. An arrow from one state to another is a **transition**; its label is the input (event) that causes it, then a bar, then the output it produces, such as $E3 \mid Z2$. If there is no output the label is just the input. An arrow that leaves a state and comes back to the same state is a transition that doesn't change the state. An arrow from nowhere (often labelled START) points to the starting state.



Key facts and formulas

There is no formula list for 9618. The insert with Paper 2 defines the built-in functions, but the patterns below are not given: learn them.

Formula	Status
Random whole number from Low to High inclusive: $\text{INT}(\text{RAND}(\text{High} - \text{Low} + 1)) + \text{Low}$	Learn it

Formula	Status
Last digit of a whole number: $N \bmod 10$; remove the last digit: $N \div 10$	Learn it
Add a digit D to the end of a whole number: $N * 10 + D$	Learn it
One character of a string: $MID(S, Index, 1)$, for Index from 1 to $LENGTH(S)$	Learn it
X is even: $X \bmod 2 = 0$	Learn it
Inclusive range: $X \geq Low$ AND $X \leq High$; outside it: $X < Low$ OR $X > High$	Learn it
$NOT (A \text{ OR } B) = NOT A \text{ AND } NOT B$; $NOT (A \text{ AND } B) = NOT A \text{ OR } NOT B$	Learn it
FOR: number of repeats known; WHILE: tested first, may run 0 times; REPEAT ... UNTIL: tested last, runs at least once	Learn it
Function header: <code>FUNCTION Name (Param : TYPE) RETURNS TYPE ... ENDFUNCTION</code> ; a procedure has no RETURNS	Learn it
Data types: INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE	Learn it
Flowchart symbols: terminator, input/output (parallelogram), process (rectangle), decision (diamond), subroutine, flow line	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (115 questions, 2021–2025), the types came up this often. 42 of the 115 questions mix types, so the counts add up to more than 115; 4 tagged questions use the topic only inside a trace table or test plan and are not counted.

Question type	Questions
Write pseudocode for a module or algorithm from a description	56
Describe an algorithm in steps without pseudocode	16
Draw or complete a program flowchart	15
Write, simplify or evaluate logic conditions	10
Decomposition: describe it or name sub-modules	8
Abstraction: pick the essential data and operations	8
State-transition diagram: complete or read	8

Question type	Questions
Structure chart: draw, complete or read	8
Name the constructs or choose a loop type	7
Convert a flowchart or structured English to pseudocode	5
Stepwise refinement: describe it or apply it	5
Complete an identifier table (names and data types)	5
Explain or correct a given algorithm	5

1. Write pseudocode for a module or algorithm from a description (56 questions)

These are worth 5 to 9 marks (up to 16 on the last question), one mark for each feature the mark scheme lists.

- Header.** Read the bullet points: does the module return a value? If so, write `FUNCTION Name ( Param : TYPE ) RETURNS TYPE`; if not, `PROCEDURE Name ( Param : TYPE )`. Use the name and parameters given. Finish with `ENDFUNCTION` or `ENDPROCEDURE`.
- Declare every local variable** with its type, if the question asks (it usually does: "all variables used must be declared").
- Initialise** totals, counters and flags **before** the loop (`Total ← 0, Found ← FALSE`).
- Choose the loop:** `FOR` to visit every element or character; `WHILE` or `REPEAT` to stop early (rogue value, item found, array full, end of file).
- Do the key work inside the loop:** take a character with `MID`, a digit with `MOD 10`, an element with `Array [ Index ]`, a line with `READFILE`.
- After the loop,** calculate the result (an average, a check digit) and `RETURN` or `OUTPUT` it.

Patterns that come up again and again:

- Digit sums** (check digits, totals of digits). Either convert to a string and loop `FOR Index ← 1 TO LENGTH(NumString), adding STR_TO_NUM(MID(NumString, Index, 1))`; or keep it numeric: `WHILE Num > 0, add Num MOD 10, then Num ← Num DIV 10`. To put a check digit on the end: `Number * 10 + CheckDigit`. To check one: compare the last digit (`Number MOD 10`) with the check digit worked out from `Number DIV 10`.
- A check digit made from a plain digit sum misses some errors.** Any wrong number whose digits have the same sum (or a sum with the same last digit) gives the same check digit, so it is accepted. Changing two digits so that one goes up by as much as the other goes down, or swapping two digits, does this.
- Splitting a string with separators**, such as "318,072,945": if every item has the same length, step through with `MID(S, Start, 3)` and `Start ← Start + 4`; otherwise go one character at a time, building up the current item with `&` until a comma, then store it and reset it to `" "`.
- Random numbers:** a whole number from 1 to 6 is `INT(RAND(6)) + 1`; from -5 to 5 (11 values) it is `INT(RAND(11)) - 5`. To get a value different from an earlier one, generate again in a `REPEAT ... UNTIL New <> Old` loop.

- **Counting characters** (vowels, 1s for parity, a given letter): loop over the string with `MID`, compare, add 1 to a counter; for parity, test `Count MOD 2`.
- **Arrays**: total, average (divide by the number of elements actually used), largest value (start with the first element), search (conditional loop with a `Found` flag), fill or output in reverse (`FOR Index ← 100 TO 1 STEP -1`).
- **Text files**: `OPENFILE` in the right mode, a loop `WHILE NOT EOF (...)` with `READFILE`, `STR_TO_NUM` before any arithmetic, `CLOSEFILE` at the end.
- **Gap-fill or "complete the pseudocode"**: use exactly the identifiers already in the code, and check each gap against the loop bounds (a nested `FOR` with `STEP 3` and `STEP 5` that marks every possible total, for example).

2. Describe an algorithm in steps without pseudocode (16 questions)

1. Write **numbered steps in plain English**. Don't write pseudocode statements such as `FOR Index ← 1 TO 52`; write "repeat 52 times" instead.
2. Name the **objects** precisely: the file and its mode ("open Sales.txt in read mode"), the variable, the array, the condition ("if the number is greater than 500").
3. Include every small step the code would need: initialise the counter, read a line, **convert the string to a number**, test, output, add 1, repeat until the end of the file, close the file.
4. Imagine you are explaining it to someone who has just started programming: they must be able to write the code from your steps.

Variants:

- **Complete the given numbered steps** (fill in the file name and mode, which variable each line goes into, what to repeat and until when).
- **A lookup array**. If a `BOOLEAN` array holds `TRUE` at every index that is a valid score, a module to validate a score takes four steps: check the score is within the array's bounds; if not, return `FALSE`; otherwise use the score as the index into the array; return the value stored there.
- **Describe a change**: say what is added to the algorithm (a separator, a length field, a new test) and where.

3. Draw or complete a program flowchart (15 questions)

Marks are given for each **functional group**, so make sure each one appears:

1. **Initialise** the index or counter before the loop.
2. **Input** inside the loop.
3. **Test for the stopping value** (a rogue value such as `-999`) and leave the loop on Yes, without storing it.
4. **Test for the array being full** (or for the number of passes) and leave on Yes.
5. **Store and increment**: add 1 to the index and put the value in the array, inside the loop.
6. **Output** the right results, under the right conditions, after the loop.

Draw one statement per box, put a condition with a question mark in each diamond, label both exits, and make the loop by an arrow going back up into an earlier box. Example 2 below is a full answer.

Variant: write the text for labelled boxes in a given flowchart. Follow the arrows: what must have happened before each box, and what does the Yes/No branch after it need?

4. Write, simplify or evaluate logic conditions (10 questions)

1. **Writing:** translate each phrase. "Between Min and Max inclusive" is $\text{Weight} \geq \text{Min} \text{ AND } \text{Weight} \leq \text{Max}$. "A seat in an even-numbered row, in column A or column F" is $(\text{Row} \bmod 2 = 0) \text{ AND } (\text{Column} = 'A' \text{ OR } \text{Column} = 'F')$: the brackets matter.
2. **Simplifying:** look for two conditions that together mean one operator: $(\text{Total} = \text{Limit}) \text{ OR } (\text{Total} > \text{Limit})$ is $\text{Total} \geq \text{Limit}$. Push a NOT inside the brackets and flip each part: $\text{NOT } (\text{Mark} < 0 \text{ OR } \text{Mark} > 100)$ is $\text{Mark} \geq 0 \text{ AND } \text{Mark} \leq 100$.
3. **Evaluating:** replace each comparison by TRUE or FALSE first, then do NOT, then AND, then OR, working inside brackets first.
4. **Finding inputs** that make a condition TRUE: pick values for each part, checking each boundary (is = included?).

5. Decomposition: describe it or name sub-modules (8 questions)

1. **Describe it** (2 or 3 marks): breaking a **problem** down into smaller sub-problems, each of which can be solved separately, leading to the program being designed as modules (procedures and functions). Add a benefit: each module is easier to write, test and debug.
2. **Apply it to the scenario.** Name modules that belong to the situation in the question (for a shop's stock system: "record a sale", "reorder an item when stock is low"). Marks are capped if your answer never mentions the scenario.
3. **Name sub-modules:** give 3 or 4, each with what it does in this system ("SendReminder: emails a member whose loan is overdue").

6. Abstraction: pick the essential data and operations (8 questions)

1. **Purpose or benefit:** it filters out details that aren't needed, giving a simpler model, so less data is stored and the solution is easier and quicker to design and test.
2. **Choose the items** a new module needs from the list given. For each, **justify by the task:** say *why the module needs it* ("the mobile number, because the module sends text messages"), not just what it is.
3. **Items not needed:** pick ones that play no part in the task (an address, for a module that sends texts) and say why.
4. **Operations on the data:** add, delete or update a record; search for a record; sort the records.
5. Variants: tick "essential" or "not essential", or name the skill being used (abstraction).

7. State-transition diagram: complete or read (8 questions)

1. **Draw one circle per state** and label them. Draw the START arrow into the first state.
2. **Go through the table one row at a time.** For each row draw an arrow from the current state to the next state and label it `input | output` (just the input if there is no output). Tick the row off.
3. **Same current and next state:** draw a loop from the circle back to itself.
4. **Two arrows between the same pair of states** (one each way): curve them apart so each label is clearly on its own arrow.
5. **Reading a diagram:** start at the START state; for each input, follow the arrow with that input from the current state, write down its output and the new state.

8. Structure chart: draw, complete or read (8 questions)

1. **Boxes:** one per module, in the right hierarchy: the module that calls goes above the one it calls.
2. **Turn each header into arrows.** Each parameter is an arrow **down** to that module, labelled with its name. RETURNS is an arrow **up**. A BOOLEAN being returned is a flag (filled circle). A BYREF parameter is a **two-headed** arrow, not a separate down arrow and up arrow.
3. **"Calls either ... or ..."** means a **diamond** where the lines leave the calling box.
4. **"Repeatedly calls"** means a **curved arrow** drawn across the lines to the modules that are repeated.
5. **Reading a chart:** write the header for a module from its arrows (down arrows are the parameters, an up arrow means it is a FUNCTION with that RETURNS type), or explain the diamond (the module calls only one of them, depending on a condition).

9. Name the constructs or choose a loop type (7 questions)

1. **Tick the constructs** in each extract: `←` is **assignment**; IF and CASE are **selection**; FOR, WHILE and REPEAT are **iteration**. An extract can have more than one.
2. **Choose a loop:** a count-controlled loop when the number of repeats is fixed; a conditional (pre- or post-condition) loop when it must stop as soon as something happens. Justify by the task: "the loop can stop as soon as the first match is found, so fewer comparisons are made".
3. **Variant:** rewrite a CASE as nested IF ... ELSE ... ENDIF, keeping the OTHERWISE as the final ELSE.

10. Convert a flowchart or structured English to pseudocode (5 questions)

1. Go through the flowchart **box by box** (or the structured English step by step): each box becomes one statement.
2. A decision whose arrow goes **back up** is a loop: if the test comes after the boxes being repeated, use REPEAT ... UNTIL; if before, WHILE ... ENDWHILE. A decision that skips forward is an IF.
3. A subroutine box becomes CALL Name (...); if it gives a value, assign it: Average ← GetAverage(userID).

4. "Repeat from step 2" in structured English means a loop starting at step 2; the condition that ends it is the step that says "then end".

11. Stepwise refinement: describe it or apply it (5 questions)

1. **Describe it** (2 marks): break each step of the algorithm into smaller, more detailed steps, and keep going **until each step can be written directly as program code**.
2. **Apply it**: turn an outline such as "work out the weight band, then calculate the delivery charge" into about five steps, each close to one line of code: input the weight; if it is less than 2 kg set the charge to 4; otherwise, if it is 10 kg or less, set the charge to 7; otherwise set the charge to 12; output the charge.

12. Complete an identifier table (names and data types) (5 questions)

1. **Name**: meaningful, no spaces, not a keyword: `Category`, `DateSold`, `ItemCost`, `InStock`.
2. **Type** from the example value: "Fruit" is `STRING`, 20/02/2025 is `DATE`, 12.67 is `REAL`, `TRUE` is `BOOLEAN`, a whole count is `INTEGER`.
3. One mark per row needs **both** a good name and the right type.
4. **Variant**: spot a poor identifier (such as `X` or `Temp1`) and say why it is poor: it doesn't describe the data it holds, so the code is harder to understand and maintain.

13. Explain or correct a given algorithm (5 questions)

1. **Trace it** with a small example before you explain anything.
2. **Explain a part**: say what it tests and what happens. An initial value of `-1` for a position is chosen because it can never be a valid index, so it shows "not found". A nested IF that only stores the index while that value is still `-1` keeps the **first** match.
3. **Find operator errors**: `>` that should be `<` (a minimum needs `IF Num < Min`), `&` that should be `+` (adding to a count), `←` that should be `=` (a comparison in `UNTIL`). Give the corrected line.
4. **Logic errors in a flowchart**: trace one value from each range through it. If a range never reaches its box, the decisions are in the wrong order or a branch goes to the wrong place: swap the decisions or redirect the branch, and say which.

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 bike hire company is designing a program to manage its hires.

(a) Complete the identifier table. [4]

Example value	Explanation	Identifier	Data type
"BK0412"	the code on the bike		
14/03/2026	the date the bike was hired		
3.75	the cost per hour		
FALSE	whether the bike was returned damaged		

(b) Tick the constructs each extract contains. [3]

Extract	Assignment	Selection	Iteration
Cost $\leftarrow$ Hours * Rate			
IF Hours > 8 THEN Cost $\leftarrow$ 30 ENDIF			
WHILE Bikes[Index] <> "" ... ENDWHILE (no assignment inside)			

(c) Write a simpler version of the condition (Age > 17) OR (Age = 17). [1]

(d) Decomposition has been used to design the program. Describe decomposition, with reference to this program. [3]

(a) One mark per row, for a meaningful identifier **and** the right type:

Example value	Explanation	Identifier	Data type
"BK0412"	the code on the bike	BikeCode	STRING
14/03/2026	the date the bike was hired	HireDate	DATE
3.75	the cost per hour	HourlyRate	REAL
FALSE	whether the bike was returned damaged	IsDamaged	BOOLEAN

B1 B1 B1 B1

(b) One mark per row:

Extract	Assignment	Selection	Iteration
Cost $\leftarrow$ Hours * Rate	✓		
IF Hours > 8 THEN Cost $\leftarrow$ 30 ENDIF	✓	✓	
WHILE Bikes[Index] <> "" ... ENDWHILE (no assignment inside)			✓

B1 B1 B1

(c) Age $\geq$ 17 **B1**

(d) Decomposition breaks the hire problem down into smaller sub-problems, **B1** such as recording a new hire, working out the cost when a bike is returned and recording damage, **B1** each of which becomes a module (procedure or function) that can be written and tested on its own. **B1**

Example 2

Temp is a 1D array of 50 elements of type REAL. An algorithm will:

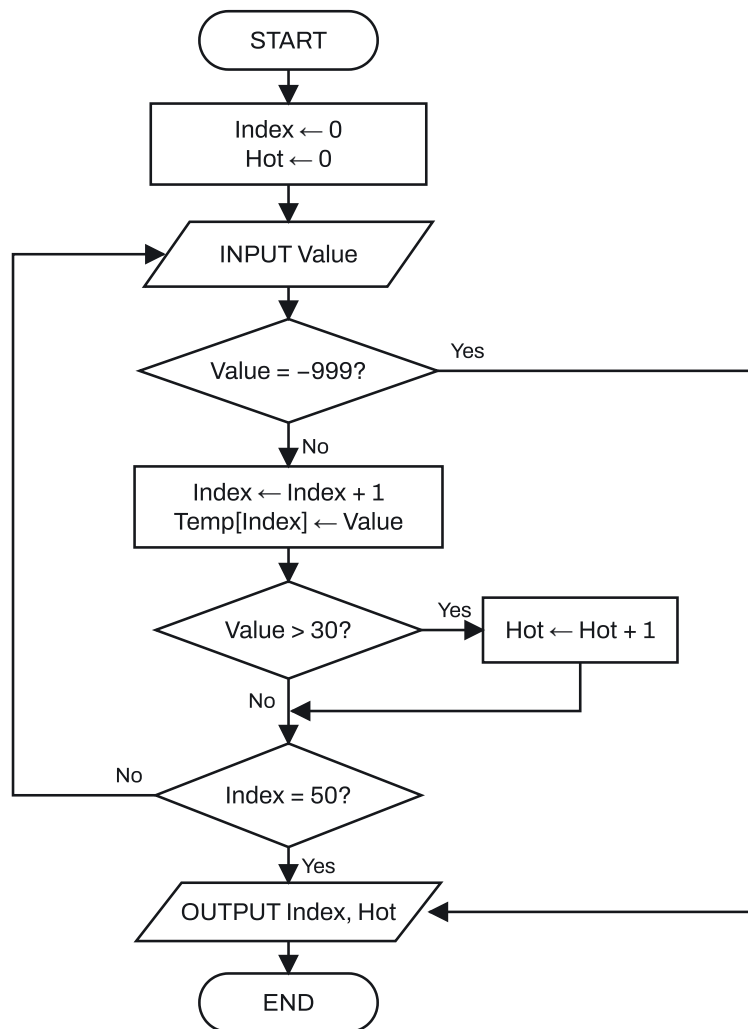
- input temperatures one at a time and store them in consecutive elements of Temp, starting at index 1;
- stop when -999 is input (-999 is not stored) or when the array is full;
- count the temperatures stored that are above 30;
- then output the number of temperatures stored and the number above 30.

(a) Draw a program flowchart for the algorithm. **[6]**

(b) Write the algorithm in pseudocode. Declarations are not needed. **[5]**

(c) Describe, without using pseudocode, how the algorithm could then find and output the highest temperature stored. **[4]**

(a)



B1 initialise `Index` and `Hot` before the loop; **B1** input inside the loop; **B1** test for `-999` and leave without storing; **B1** test for the array full (`Index = 50`) and leave; **B1** increment the index and store the value in a loop; **B1** count values above 30 and output both results after the loop.

With the inputs 28.5, 31, 33.5, 19, `-999` the output is 4 and 2.

(b)

`Index ← 0`

`Hot ← 0`

REPEAT

`INPUT Value`

 IF `Value <> -999` THEN

`Index ← Index + 1`

`Temp[Index] ← Value`

 IF `Value > 30` THEN

`Hot ← Hot + 1`

 ENDIF

 ENDIF

UNTIL Value = -999 OR Index = 50

OUTPUT Index, Hot

B1 initialisation; **B1** conditional loop with input inside; **B1** store only when the value is not -999, with the index incremented; **B1** both stopping conditions joined with OR; **B1** count above 30 and output after the loop.

(c)

1. Set a variable `Highest` to the value of the first element, `Temp[ 1 ]`. **B1**
2. Repeat for each index from 2 up to the number of temperatures stored (`Index`). **B1**
3. If the element at that index is greater than `Highest`, set `Highest` to that element. **B1**
4. After the loop has finished, output `Highest`. **B1**

With the same inputs the highest temperature is 33.5. (If nothing was stored, `Index` is 0 and there is no highest value, so check that first.)

Example 3

(a) A program has these modules:

```
PROCEDURE Checkout( )
FUNCTION ScanItem() RETURNS STRING
FUNCTION LookUpPrice(Code : STRING) RETURNS REAL
PROCEDURE PayCash(Total : REAL)
FUNCTION PayCard(BYREF Total : REAL) RETURNS BOOLEAN
```

`Checkout` repeatedly calls `ScanItem` then `LookUpPrice`. It then calls either `PayCash` or `PayCard`. Draw a structure chart for these modules. [5]

(b) Part of the program is described by this state table.

Current state	Input	Output	Next state
S1	E1	Z1	S2
S2	E2		S2
S2	E3	Z2	S3
S3	E1	Z1	S2
S2	E4		S1
S3	E4	Z3	S4

(i) Draw the state-transition diagram. S1 is the starting state. [4]

(ii) The inputs E1, E2, E3, E1, E3, E4 are received in that order, starting in S1. Give the outputs produced and the final state. [2]

(a) The second diagram in "Understand it" is the answer.

- **B1** all five boxes, with Checkout above the other four;
- **B1** the curved iteration arrow across the lines to ScanItem and LookUpPrice;
- **B1** the diamond on the lines to PayCash and PayCard;
- **B1** Code returned up from ScanItem, Code down to and Price up from LookUpPrice;
- **B1** Total down to PayCash, Total as a two-headed arrow to PayCard and the flag Approved up from it.

(b)(i) The third diagram in "Understand it" is the answer.

- **B1** four states and the START arrow into S1;
- **B1** S1 to S2 (E1 | Z1) and the loop on S2 (E2);
- **B1** S2 to S3 (E3 | Z2) and S3 to S2 (E1 | Z1), drawn as two separate arrows;
- **B1** S2 to S1 (E4) and S3 to S4 (E4 | Z3).

(b)(ii) Follow the arrows: S1 →(E1, output Z1) S2 →(E2, no output) S2 →(E3, Z2) S3 →(E1, Z1) S2 →(E3, Z2) S3 →(E4, Z3) S4.

The outputs are Z1, Z2, Z1, Z2, Z3 **B1** and the final state is S4. **B1**

Example 4

A shop adds a check digit to each product number using **weights**. Each digit is multiplied by its position counted from the left (1, 2, 3, ...), the products are added, and the check digit is the remainder when this total is divided by 7. The check digit is then added to the end of the number.

For example, for 5208: $5 \times 1 + 2 \times 2 + 0 \times 3 + 8 \times 4 = 41$, and $41 \bmod 7 = 6$, so the new number is 52086.

(a) Write pseudocode for a function `Generate ( )` that takes the original number as an integer parameter and returns the new number as an integer. **[7]**

(b) Write pseudocode for a function `IsValid ( )` that takes a new number and returns TRUE if its check digit is correct. It must use `Generate ( )`. **[2]**

(c) Show that this method detects the error of swapping the first two digits of 5208, and explain why a check digit made from a plain digit sum would not. **[2]**

(a)

```
FUNCTION Generate(Number : INTEGER) RETURNS INTEGER
    DECLARE NumString : STRING
    DECLARE Index, Total, Digit, CheckDigit : INTEGER
    NumString ← NUM_TO_STR(Number)
    Total ← 0
```

```

FOR Index ← 1 TO LENGTH(NumString)
    Digit ← STR_TO_NUM(MID(NumString, Index, 1))
    Total ← Total + Digit * Index
NEXT Index
CheckDigit ← Total MOD 7
RETURN Number * 10 + CheckDigit
ENDFUNCTION

```

- **B1** header with parameter, RETURNS INTEGER and ENDFUNCTION;
- **B1** all local variables declared, and Total set to 0 before the loop;
- **B1** convert the number to a string and loop for its length;
- **B1** extract one digit with MID in the loop and convert it to a number;
- **B1** multiply each digit by its position and add it to the total;
- **B1** check digit from Total MOD 7;
- **B1** return the number with the check digit added to the end.

(b)

```

FUNCTION IsValid(NewNumber : INTEGER) RETURNS BOOLEAN
    RETURN Generate(NewNumber DIV 10) = NewNumber
ENDFUNCTION

```

B1 NewNumber DIV 10 removes the check digit to give the original number; **B1** compare the result of Generate() with NewNumber and return the BOOLEAN.

(c) With the first two digits swapped, 52086 is typed as 25086. The check digit worked out from 2508 is $2 \times 1 + 5 \times 2 + 0 \times 3 + 8 \times 4 = 44$, and $44 \bmod 7 = 2$. This is not the 6 on the end, so 25086 is rejected. **B1** A plain digit sum is $2 + 5 + 0 + 8 = 15$ for both orders, so swapped digits give the same check digit and the error is not detected. **B1**

Common mistakes

- **Not initialising a total or counter, or setting it to 0 inside the loop**, so it is reset on every pass. Initialise it once, just before the loop.
- **Treating a string like an array**. NumString[3] isn't allowed; use MID(NumString, 3, 1).
- **Using functions that aren't in the insert**, such as Python's len() or split(). Use only the insert's functions.
- **Getting RAND wrong**. RAND(6) gives a real number from 0 up to 6, so INT(RAND(6)) + 1 is needed for 1 to 6. A parameter of 7, or a missing INT, loses the mark.
- **Calling a function and ignoring what it returns**. IsPrime(Num) on a line of its own does nothing useful; write IF IsPrime(Num) = TRUE THEN or assign it.

- **Dividing by the wrong number** for an average: divide by how many values were actually used, not the array size.
- **Flowcharts:** writing a whole loop in one box ("repeat while Index < 51"), a decision with only one exit or unlabelled exits, and arrows that lead nowhere or merge an input line with an output line. One action per box; two labelled exits per diamond.
- **Decomposition:** saying it breaks down an existing *program* (it breaks down a *problem*), using vague words such as "easy", or never mentioning the scenario.
- **Abstraction:** confusing it with decomposition or with hiding data from users. For "justify" marks, say why the module needs the item, not what the item is.
- **Structure charts:** forgetting the diamond when the notes say "either", and drawing a BYREF parameter as a parameter plus a separate return value.
- **Identifier names that are keywords**, such as `DECLARE String : STRING.`
- **Describing an algorithm by repeating the question.** "Compare with the previous numbers" is too vague: say which variable, which loop, which test.

How to get the marks

- **Pseudocode is marked by feature.** Each mark point is one feature (header, declarations, initialisation, loop, the key operation, return). A program that doesn't quite work can still earn most of the marks, so always write something for every bullet in the question.
- **Use the given identifiers** and the exact module name and parameters. Keep `THEN`, `ENDIF`, `NEXT`, `ENDWHILE`, `ENDFUNCTION` and `RETURNS`, and use `←` for assignment.
- **"All variables used must be declared"** means a `DECLARE` with a type for every local variable, including the loop counter.
- **"Do not use pseudocode"** means plain-English steps: "repeat for each line of the file", not `WHILE NOT EOF`. Statements in pseudocode can lose the marks.
- **"Describe"** needs the steps or features in order; **"Explain"** needs the reason too (why -1, why a conditional loop); **"Justify"** needs a reason that comes from the scenario in the question.
- **"State" or "Identify"** needs a short answer. If one item is asked for, only the first answer is marked.
- **Flowcharts** are marked on functional groups: initialise, input in a loop, each stopping test, store and increment, outputs in the right place.
- **Structure and state-transition diagrams** are marked in groups too: all boxes or states, then groups of correctly labelled arrows. Extra arrows can cost marks.
- **Decomposition and abstraction answers** should name parts of the given system; generic answers are capped.

Quick check

1. `A ← 7, B ← 12` and `Done ← FALSE`. Evaluate: (i) `A > 5 AND NOT Done`; (ii) `NOT (A > 5 OR B < 10)`; (iii) `B MOD A = 5 OR Done`.

2. Write an expression that gives a random whole number from 20 to 45 inclusive.
3. Simplify NOT (Speed < 30 OR Speed > 50) so that it has no NOT.
4. What does this pseudocode output, and how many times does the loop run?

```
Num ← 4072
Total ← 0
WHILE Num > 0
    Total ← Total + Num MOD 10
    Num ← Num DIV 10
ENDWHILE
OUTPUT Total
```

5. A cinema stores for each customer: name, email address, date of birth, favourite film genre, card number and preferred seat. A new module emails customers about new horror films. Identify two items the module needs and one it doesn't, with reasons.

Answers

1. (i) `TRUE AND TRUE` is `TRUE`. (ii) `A > 5` is `TRUE`, so the bracket is `TRUE` and `NOT` makes it `FALSE`. (iii) `12 MOD 7` is 5, so `TRUE OR FALSE` is `TRUE`.
2. `INT(RAND(26)) + 20`. There are $45 - 20 + 1 = 26$ values; `INT(RAND(26))` gives 0 to 25, and adding 20 gives 20 to 45.
3. `Speed >= 30 AND Speed <= 50`.
4. The loop adds 2, 7, 0, 4 (the digits from the right), so it outputs 13. It runs 4 times, once per digit; Num goes 4072, 407, 40, 4, 0.
5. Needed: the **email address**, because the module sends emails; the **favourite genre**, because only customers who like horror should be sent the email (the name is also fine, to personalise the email). Not needed: the **card number** (or date of birth, or preferred seat), because no payment is taken when sending the email.

Past questions to try

- 9618/21/M/J/25 Q3: a guessing game, from structured English to pseudocode, with random numbers.
- 9618/22/M/J/25 Q4: a flowchart that stores 100 values and outputs them in reverse.
- 9618/21/M/J/24 Q7: abstraction for a fitness club, then a structure chart.
- 9618/23/O/N/23 Q4: random numbers in ascending order, and simplifying a condition.