

Testing, trace tables and errors

Read online at pastopic.com/cambridge/computer-science-0478/notes/testing-and-trace-tables

Last checked 30 September 2026

Read first: [Program development and design](#), [Standard methods of solution](#)

© 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. **Suggest and apply suitable test data** for a program, using the four types in the syllabus:
 - **normal** data: sensible values the program should accept;
 - **abnormal** data (also called erroneous): values the program should reject, because they are outside the allowed limits or the wrong type;
 - **extreme** data: the largest and the smallest value the program should accept;
 - **boundary** data: the largest or smallest acceptable value **together with** the value just beyond it that should be rejected.
2. **Complete a trace table to record a dry run of an algorithm**: work through a flowchart or pseudocode by hand, one step at a time, writing down every change to a variable, every output and every prompt shown to the user.
3. **Identify errors in a given algorithm and suggest how to correct them**, line by line.

Questions on these nearly always also ask you to **state the purpose** of the algorithm you traced or **describe the processes** in it (syllabus 7.3, in Program development and design), so this note covers that too. Writing whole algorithms and the standard methods themselves (totalling, counting, maximum, linear search, bubble sort) are in **Standard methods of solution**; validation checks are in **Validation and verification**.

Understand it

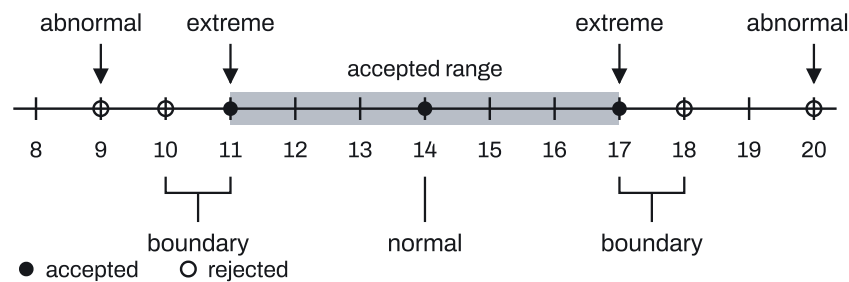
Why programs are tested

A program can look finished and still give wrong answers for some inputs. **Testing** is the last stage of the program development life cycle: you run the program with chosen inputs, called **test data**, and compare what it does with what it **should** do (the **expected outcome**). If they differ, there is an error to find and fix.

Good test data is chosen on purpose, to try every kind of input the program might meet: typical values, values at the edges of what is allowed, and values that should be refused.

The four types of test data

Take a junior sports club whose program accepts a member's age only if it is a **whole number from 11 to 17 inclusive**.



Type	Example for ages 11 to 17	Expected outcome
Normal	14	accepted
Abnormal	25, -3, 13.5, "twelve"	rejected
Extreme	11 and 17	accepted
Boundary	10 and 11; 17 and 18	10 and 18 rejected, 11 and 17 accepted

- **Normal** data checks that the program does its ordinary job with ordinary input.
- **Abnormal** data checks that bad input is refused. It can be out of range (25), the wrong type ("twelve"), or break another rule (13.5 is not a whole number).
- **Extreme** data sits exactly on the edge but still inside: it checks that the end values are **accepted**. A very common bug is writing `>` where `>=` was needed, which wrongly rejects 11.
- **Boundary** data tests **both sides** of an edge: the last value that should pass and the first that should fail. That catches a limit that is off by one in either direction.

So every extreme value is also half of a boundary pair, but a boundary pair always has a rejected value as well.

Decimals. "Just beyond" depends on the precision allowed. If a price must be from 0.50 to 20.00 with 2 decimal places, the boundary pairs are 0.49 and 0.50, and 20.00 and 20.01.

Lengths. For a rule about the number of characters, the test data is a string of that length. For "a username of 6 to 12 characters", extreme data is a 6-character and a 12-character username; boundary data adds a 5-character and a 13-character one; abnormal data could be a 2-character username. For "exactly 7 characters", the 7-character string is both the smallest and the largest acceptable length, and 6 and 8 characters are the rejected neighbours.

A test plan lists each piece of test data with its type and its expected outcome, so that the actual result can be compared with it.

Dry runs and trace tables

A **dry run** means working through an algorithm by hand, doing exactly what each step says, without a computer. A **trace table** records it: one column for each variable (and each array element that changes), and one column for **OUTPUT**. Reading down a column shows every value that variable held, in order.

This algorithm uses the pseudocode functions MOD (the remainder after whole-number division) and DIV (the whole-number part of a division). For example, MOD(17, 5) returns 2 and DIV(17, 5) returns 3.

```
OUTPUT "Enter a whole number"
INPUT Number
Sum ← 0
WHILE Number > 0 DO
    Digit ← MOD(Number, 10)
    Sum ← Sum + Digit
    Number ← DIV(Number, 10)
ENDWHILE
OUTPUT "Digit sum ", Sum
```

The dry run for the input 4072:

Number	Digit	Sum	OUTPUT
			Enter a whole number
4072		0	
407	2	2	
40	7	9	
4	0	9	
0	4	13	
			Digit sum 13

How it was filled in:

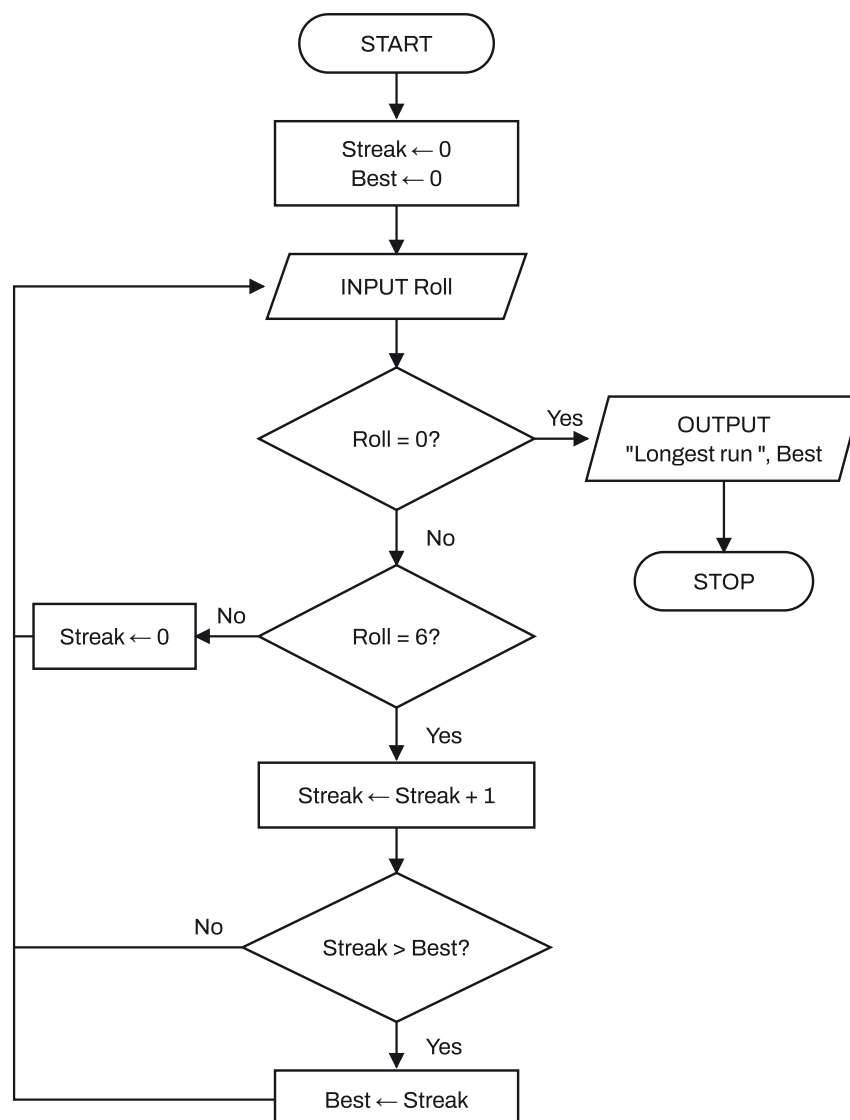
- The **prompt** is an output, so it goes in the OUTPUT column, in its place in the order.
- **Starting values** (here $\text{Sum} \leftarrow 0$, and the input) go on the first rows.
- Each time round the loop, MOD takes off the last digit, it is added to Sum, and DIV removes it from Number. Only values that change are written in.
- Before every repeat, check the loop condition with the **current** values. When Number reaches 0, $\text{Number} > 0$ is false, so the loop ends and the final output is written.
- Outputs are written exactly as they would appear on screen, text and value together, **without** quotation marks.

The exact row a value goes on matters less than the **order of values down each column**, which is what the marks are for. A tidy habit is to start a new row whenever the next value would go in a column that is already filled on the current row.

Because the algorithm adds up the digits of a positive whole number, that is its **purpose**. Tracing first and then asking "what did it do to my data?" is the easiest way to find the purpose of an unfamiliar algorithm.

Tracing flowcharts

A flowchart is traced the same way: follow the arrows from START, do each process box, and at each decision diamond work out the condition with the current values to choose the Yes or No arrow. The flowchart below is used in Worked example 2. Its loop goes back up to INPUT Roll until the rogue value 0 is entered.



Rogue values. Many algorithms stop when a special value such as 0, -1 or 999 is input. The rogue value ends the loop and is **not** processed as data, and any inputs listed after it are never read.

Finding errors

An algorithm with an error either will not run (a keyword or structure is wrong) or runs and gives the wrong result (its logic is wrong). Exam questions give numbered lines, say what the algorithm should do and how many errors there are. The quickest way to find them is to compare every line with the description, part by part:

- **Declarations:** does each data type suit what is stored (text in `STRING`, a whole-number counter in `INTEGER`, money or measurements in `REAL`)?
- **Starting values:** does each total, counter, highest or lowest value, and flag start where it should?
- **Loops:** the right start and end values, the right closing keyword (`FOR ... NEXT`, `REPEAT ... UNTIL`, `WHILE ... ENDWHILE`), and a condition that can end the loop.
- **Conditions:** `<` or `>`, `>=` or `<=`, `AND` or `OR`, and the right variable being compared.
- **Calculations and assignments:** the right operator (`+`, `*`, `/`), the right variable on each side, the right array index.
- **Inputs and outputs:** `INPUT` where a value is read in, `OUTPUT` where one is shown, and the right variable or message output.

A dry run with a few simple values confirms each suspicion: if the algorithm gives a different answer from the one you worked out by hand, the line where the values first go wrong holds an error.

Key facts and formulas

There is no formula list for this topic; learn these.

Fact	Status
Normal data: inside the allowed limits, should be accepted	Learn it
Abnormal (erroneous) data: outside the limits or the wrong type, should be rejected	Learn it
Extreme data: the largest and the smallest acceptable values	Learn it
Boundary data: the largest or smallest acceptable value and the nearest rejected value beyond it	Learn it
A trace table has a column for each variable, each changing array element and <code>OUTPUT</code> ; prompts are outputs	Learn it
A rogue value ends the loop and is not processed; later inputs are never read	Learn it
$\text{DIV}(a, b)$ is the whole-number part of $a \div b$; $\text{MOD}(a, b)$ is the remainder: $\text{DIV}(17, 5) = 3$, $\text{MOD}(17, 5) = 2$	Learn it

Fact	Status
ROUND(value, places), LENGTH(string), SUBSTRING(string, start, length) (first character is position 1), UCASE, LCASE	Learn it
Loop closers: FOR ... NEXT, REPEAT ... UNTIL, WHILE ... ENDWHILE; IF ... ENDIF, CASE ... ENDCASE	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (79 questions, 2021–2025), the types came up this often. 34 of the 79 questions mix more than one type, so the counts add up to 117.

Question type	Questions
Complete a trace table (dry-run a flowchart or pseudocode)	34
Find errors in given pseudocode or a flowchart and correct them	25
State the purpose of an algorithm or describe its processes	19
Choose test data for a stated rule	17
Explain a problem a dry run shows, and fix it	12
Name, define or match the types of test data	10

1. Complete a trace table (34 questions)

- Read the headings** you are given: they tell you which variables, array elements and outputs to follow. Fill in any starting values on the first row (a pre-filled first row, such as an array's current contents, is already done for you).
- Work one step at a time**, in the order the algorithm runs. At each assignment, write the new value in that variable's column. Don't write a value that hasn't changed.
- At each decision or loop test**, work out the condition with the current values before choosing the route.
- Write every output** (prompts included) exactly as it would appear, without quotation marks.
- Stop when the algorithm stops**: at the rogue value, when a loop condition ends it, or at STOP. Leave the remaining input unused.

Variations you will meet:

- Array columns**: one column per element, such as Store[1] to Store[8]; write an element only when it changes. For a bubble sort, follow each comparison and swap, the temporary variable and the flag.
- String functions**: SUBSTRING(Word, 3, 1) is the third character; LENGTH counts the characters. Trace them letter by letter.
- Two or more variables that control one loop**, such as a counter and a flag: update each one exactly where its line is, and test the whole loop condition with both current values.

- **DIV, MOD and ROUND:** work them out exactly as defined, for example $\text{MOD}(23, 4) = 3$.
- **Several short tables**, one for each input value; or a table of "input → output" only.
- **Nested IF or CASE:** follow only the branch the condition chooses; an ELSE branch is skipped when the IF is true.

2. Find and correct errors (25 questions)

1. **Read the description** first and note what each variable is for.
2. **Go through the checklist** in "Understand it": declarations, starting values, loops, conditions, calculations, inputs and outputs.
3. **Test each suspect line** against a tiny dry run if unsure.
4. **Write each answer** as the line number and the **whole corrected line**, for example "Line 11: `Fine ← DaysLate * 0.20`". Give exactly the number of errors asked for.

Variations you will meet:

- **An undeclared variable:** the fix is a DECLARE line (with a sensible type) placed before it is used.
- **Errors in a flowchart:** name the box and write what it should say, such as the right condition in a decision.
- **Follow-up parts** on the corrected algorithm. A rounded output is one statement, such as `OUTPUT ROUND(Distance / Laps, 2)`. An extension (a new result the algorithm must also produce) always needs the same four things, each said with **where** it goes: a new variable declared; a starting value set before the loop; an update inside the loop; an output after the loop. For example, to also count the inputs over 100: declare `Over` as an integer, set `Over ← 0` before the loop, add `IF Value > 100 THEN Over ← Over + 1` inside the loop after the input, and `OUTPUT "Over 100: ", Over` after the loop.

3. State the purpose or describe the processes (19 questions)

1. **Trace first** (you usually just have), then look at what the algorithm did to the data.
2. **"State the purpose"** (1 mark): one precise phrase naming the job, such as "adds up the digits of a whole number", "finds the longest run of sixes", "reverses a word". "It inputs numbers and outputs a result" earns nothing.
3. **"Describe the processes"** (usually 3 marks): list the separate steps in order, one mark each: what is input, what is checked or calculated, what is stored, when it stops and what is output.

Variations you will meet:

- **"Explain how the algorithm knows when to stop":** point to the loop's condition and say which variable in it changes each time round, and how, until the condition ends the loop. For example, in the digit-sum algorithm in "Understand it", `DIV` removes one digit from `Number` on every pass, so `Number` eventually reaches 0 and `Number > 0` becomes false.
- **"State the rules the data must meet" or "the range being tested":** read them off the conditions, for example `Temp >= 5 AND Temp <= 30` tests 5 to 30 inclusive.

4. Choose test data for a rule (17 questions)

1. **Find the limits** and the precision: whole numbers, 2 decimal places, or a number of characters.
2. **Normal**: a value comfortably inside.
3. **Abnormal**: a value well outside, or of the wrong type (text for a number).
4. **Extreme**: the smallest or largest acceptable value itself.
5. **Boundary**: the acceptable limit and the first value beyond it at the stated precision (for -50 to 50 whole numbers: 50 and 51 , or -50 and -51).
6. **Give the reason or outcome** if asked: say whether it should be accepted or rejected, and why ("it is the largest value that should be accepted").

Variations you will meet:

- **A table to complete** with test data, type and purpose: fill each gap so that the three agree. A purpose that says "rejects data outside the limits" goes with abnormal data; "accepts data that is in range" with normal data; the boundary row's purpose is about the value on one side of the edge, accepted or rejected.
- **Length rules**: give actual strings of the right lengths ("at most 15 characters": a 15-character string is accepted, a 16-character one is rejected).
- **Format rules** (an email address, a date): normal data follows the format; abnormal data breaks it (no @, month 14).
- **A full set of test data** for an algorithm: at least one of each type, with expected outcomes.

5. Explain a problem a dry run shows, and fix it (12 questions)

1. **Say what goes wrong**, for which input, and why, pointing to the step responsible.
2. **Give a concrete change** (and explain any code): what to add, remove, move or reorder, and where.

Variations you will meet:

- **Two checks joined by ELSE that should both run**: when the first check is true the second is skipped, so a value that should update both only updates one. Make them two separate `IF` statements.
- **Range tests in the wrong order**: if the lowest threshold is tested first, every higher value also passes it and never reaches the later tests. Test the highest threshold first (or give each test both limits), and match each result to the right test.
- **A total or counter that is not reset**: set it back to 0 at the start of each outer-loop pass.
- **Upper and lower case**: `Answer = 'Y'` fails for 'y'. Convert the input with `UCASE` first, or test both with `OR`.
- **An input the algorithm never expected**: ask "what could a user type here that the designer didn't plan for?" For a program that shares a bill between `People` diners, `People = 0` makes the division fail and a negative number gives a nonsense share. The fix is a check straight after the input that refuses such values and asks again, with a message.

- **An algorithm that gives up too early:** a loop that ends as soon as one condition is met can miss later data that also matters. For example, a check for "any value over 40" that stops at the first one cannot also report how many there were; let the loop run to the end and count instead.

6. Name, define or match the types of test data (10 questions)

- **List three types:** any three of normal, abnormal, extreme, boundary.
- **Define** each in one sentence using "accepted" or "rejected" (see the table in "Understand it").
- **Match or tick:** boundary covers both the accepted edge value and the rejected one next to it; extreme is accepted only; abnormal is always rejected.
- **The purpose of test data:** to check that a program behaves as intended by comparing actual results with expected results, using each type of data. Give an example type if asked.
- **Testing a long loop** (such as 500 inputs) faster: change the loop limit to a small number while testing, then set it back.

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 smart thermostat program accepts a target temperature, a whole number of degrees Celsius from 12 to 28 inclusive.

(a) Complete a test plan with one piece of **normal**, **abnormal**, **extreme** and **boundary** test data. For each, state the expected outcome. For the boundary data, give both values of one pair. **[5]**

(b) A second input is a room code, which must be exactly 7 characters long. Suggest test data at this rule's boundary, and explain your choices. **[3]**

(a)

Test data	Type	Expected outcome
21	Normal	accepted
40 (or "cold", or 19.5)	Abnormal	rejected
28 (or 12)	Extreme	accepted
28 and 29 (or 11 and 12)	Boundary	28 accepted, 29 rejected

B1 normal, **B1** abnormal, **B1** extreme, **B1** a correct boundary pair, **B1** all four expected outcomes right.

(b) "KT-4081" (7 characters) should be accepted **B1**; "KT-408" (6 characters) and "KT-40811" (8 characters) should be rejected **B1**. With an exact length, 7 is both the smallest and the largest acceptable length, so the values just past it on both sides test that the check rejects anything shorter or longer **B1**.

Example 2

This algorithm is the flowchart in "Understand it" (Streak, Best and Roll). It reads dice rolls until 0 is entered.

(a) Complete the trace table for the input data: 6, 6, 2, 6, 6, 6, 1, 0, 6 **[4]**

(b) State the purpose of the algorithm. **[1]**

(a)

Roll	Streak	Best	OUTPUT
	0	0	
6	1	1	
6	2	2	
2	0		
6	1		
6	2		
6	3	3	
1	0		
0			Longest run 3

B1 for each correct column. Notes on the working:

- After the second 6, **Streak** (2) is greater than **Best** (1), so **Best** becomes 2. After the 2, **Streak** goes back to 0, but **Best** keeps 2.
- In the second run, **Streak** = 1 and 2 are **not** greater than **Best** = 2, so **Best** doesn't change until **Streak** reaches 3.
- The 0 is the rogue value: the Yes route outputs the result and stops, so the final 6 is never read.

(b) It finds the longest run of consecutive sixes in the rolls entered. **B1**

Example 3

This algorithm should input the number of days late for each of 25 returned library books. Each book is fined 0.20 per day late, up to a maximum of 5.00. It outputs the total of the fines and the number of books returned on time. There are four errors.

```

01 DECLARE DaysLate : INTEGER
02 DECLARE Fine : REAL
03 DECLARE TotalFines : REAL
04 DECLARE OnTime : INTEGER
05 DECLARE Book : INTEGER
06 TotalFines ← 0
07 OnTime ← 0
08 FOR Book ← 1 TO 25
09     OUTPUT "Days late for book ", Book
10     INPUT DaysLate
11     Fine ← DaysLate + 0.20
12     IF Fine < 5.00
13         THEN
14             Fine ← 5.00
15     ENDIF
16     TotalFines ← TotalFines + Fine
17     IF DaysLate > 0
18         THEN
19             OnTime ← OnTime + 1
20     ENDIF
21 NEXT DaysLate
22 OUTPUT "Total fines: ", TotalFines
23 OUTPUT "Books returned on time: ", OnTime

```

(a) Identify the line number of each error and write the corrected line. [4]

(b) The corrected algorithm is run for 3 books instead of 25, with the inputs 0, 12 and 40. State the two values output. [2]

(c) Explain why 25 and 26 are useful test values for DaysLate. [2]

(a)

- Line 11: $\text{Fine} \leftarrow \text{DaysLate} * 0.20$ (the fine is the days multiplied by the daily rate). **B1**
- Line 12: $\text{IF Fine} > 5.00$ (only a fine **above** the maximum is cut down; as written, every smaller fine is raised to 5.00). **B1**
- Line 17: $\text{IF DaysLate} = 0$ (a book with no days late is the one returned on time). **B1**
- Line 21: NEXT Book (a FOR loop is closed with its own counter). **B1**

(b) The fines are $0 \times 0.20 = 0$, $12 \times 0.20 = 2.40$ and $40 \times 0.20 = 8.00$, which is capped at 5.00. Total fines: 7.40 **B1**. Books returned on time: 1 **B1**.

(c) 25 days gives exactly $25 \times 0.20 = 5.00$, the largest fine that is not changed; 26 days gives 5.20, the first fine that must be cut to 5.00 **B1**. Testing both sides of the point where the cap starts checks that the comparison in line 12 is right, like boundary data for a range **B1**.

Example 4

This algorithm should output the mean of three test scores for each of two students.

```

01 Total ← 0
02 FOR Student ← 1 TO 2
03     FOR Test ← 1 TO 3
04         INPUT Score
05         Total ← Total + Score
06     NEXT Test
07     OUTPUT "Average: ", Total / 3
08 NEXT Student

```

(a) Complete the trace table for the input data: 12, 15, 18, 9, 11, 16 [4]

(b) Explain why the second output is wrong, and how to correct the algorithm. [2]

(a)

Student	Test	Score	Total	OUTPUT
			0	
1	1	12	12	
	2	15	27	
	3	18	45	
				Average: 15
2	1	9	54	
	2	11	65	
	3	16	81	
				Average: 27

B1 Student and Test columns, **B1** Score, **B1** Total, **B1** OUTPUT.

(b) Total is set to 0 only once, before the outer loop, so the second student's scores are added on to the first student's 45: the output is $81 \div 3 = 27$ instead of $36 \div 3 = 12$ **B1**. Move Total ← 0 inside the outer loop, straight after line 02, so it is reset for each student **B1**.

Common mistakes

- **Carrying on after the algorithm has stopped.** When the rogue value is input or the loop condition is met, the trace ends; values listed after it are never read. Examiners regularly see extra rows traced for them.
- **Filling columns by guessing the pattern.** Work every step in order and write each value as it happens. Candidates who jump ahead to what they think the algorithm does often fill in wrong values.

- **Getting the OUTPUT column wrong.** Leaving out prompts, putting quotation marks around the text, or writing only the number without its message all lose the mark.
- **Mixing up extreme and boundary data.** Extreme is only the edge value that is accepted; boundary needs the edge value **and** the rejected neighbour. An "extreme" value that the program would reject is not extreme.
- **Giving only one boundary value,** or a rejected neighbour at the wrong precision (for 2 decimal places, the value just past 20.00 is 20.01).
- **Vague error corrections.** "Line 12 is wrong" or "change the sign" is not enough: give the line number with the whole corrected statement.
- **Rewriting the given algorithm unchanged** when asked to improve it. Marks come only for the new or changed parts, so say exactly what is added and where.
- **Purposes that describe the code, not its job.** "It uses a loop and an IF" is not a purpose; say what the algorithm achieves for its user.
- **Using English instead of precise pseudocode** in a correction: write `Count > 10`, not "count is more than 10".

How to get the marks

- **Trace tables** are usually marked **one mark per column** (or group of columns). A column is marked on the order of its values, so a small slip in one column doesn't cost the others. Put starting values first and write clearly: a crossed-out value should be clearly crossed out.
- **"Identify" errors:** line number plus the corrected line, one mark each. Where the error is a missing line, say where the new line goes.
- **"State"** the purpose: one short, specific phrase. **"Describe"** the processes: separate points in order, as many as there are marks.
- **"Suggest"** test data: an actual value (or string) for each type, and the outcome or reason when asked. For boundary data, both values of the pair are safest.
- **"Explain"** a problem: the cause **and** its effect for a specific input, then the change. "Any code used must be fully explained" means each statement you add needs a sentence saying what it does.
- **Precision:** conditions written as code (`Score >= 40`), and inclusive limits written with `>=` and `<=`.

Quick check

1. A lift program accepts a floor number, a whole number from -2 to 15 inclusive. Give one normal value, one abnormal value, both extreme values and one boundary pair.
2. (a) Explain the difference between extreme data and boundary data. (b) A program accepts a reading from 0.5 to 9.5 with one decimal place. Give the extreme data and the boundary pair at the top end.
3. Complete a trace table (X, Y, OUTPUT) for this algorithm.

```
X ← 1
Y ← 0
REPEAT
    Y ← Y + X
    X ← X * 3
UNTIL X > 50
OUTPUT Y
```

4. This should count how many of the ten temperatures in `Temps [ 1 : 10 ]` are below 0. Identify the two errors and correct them.

```
01 Freezing ← 1
02 FOR Index ← 1 TO 10
03     IF Temps[Index] <= 0
04         THEN
05             Freezing ← Freezing + 1
06     ENDIF
07 NEXT Index
```

5. State the purpose of this algorithm, and its output for the inputs 3 then 4, and for 10 then 3.

```
INPUT B
INPUT E
P ← 1
C ← 0
WHILE C < E DO
    P ← P * B
    C ← C + 1
ENDWHILE
OUTPUT P
```

Answers

1. Normal: 7 (any whole number from -1 to 14). Abnormal: 20, or "ground", or 3.5. Extreme: -2 and 15 . Boundary: -3 and -2 , or 15 and 16 (-3 and 16 rejected, -2 and 15 accepted).
2. (a) Extreme data is the largest or smallest value that should be accepted. Boundary data is that value together with the nearest value beyond it, which should be rejected, so it tests both sides of the limit. (b) Extreme: 0.5 and 9.5 . Top boundary pair: 9.5 (accepted) and 9.6 (rejected).
3. X: 1, 3, 9, 27, 81. Y: 0, 1, 4, 13, 40. After each pass the UNTIL test is checked: 3, 9 and 27 are not greater than 50, but 81 is, so the loop ends. OUTPUT: 40.
4. Line 01: Freezing $\leftarrow 0$ (a count starts at 0). Line 03: IF Temps[Index] < 0 (0 itself is not below 0). For example, with temperatures 3, -1 , 0, -4 , 5, 2, -2 , 7, 0, 1 the corrected algorithm outputs 3; the original gives 6.
5. It raises B to the power E (it multiplies B by itself E times). For 3 then 4 it outputs $3 \times 3 \times 3 \times 3 = 81$. For 10 then 3 it outputs $10 \times 10 \times 10 = 1000$.

Past questions to try

- [0478/21/O/N/25 Q7](#): trace a flowchart for two different inputs and state its purpose.
- [0478/22/M/J/25 Q5](#): five errors in an algorithm for names and heights, then round an average.
- [0478/22/M/J/25 Q3](#): normal, abnormal, extreme and boundary data for a range check.
- [0478/22/F/M/25 Q7](#): trace a calculator flowchart and fix its problems with lower-case input.