

Algorithm design and problem-solving

IGCSE Computer Science 0478 · Topic 7 revision notes

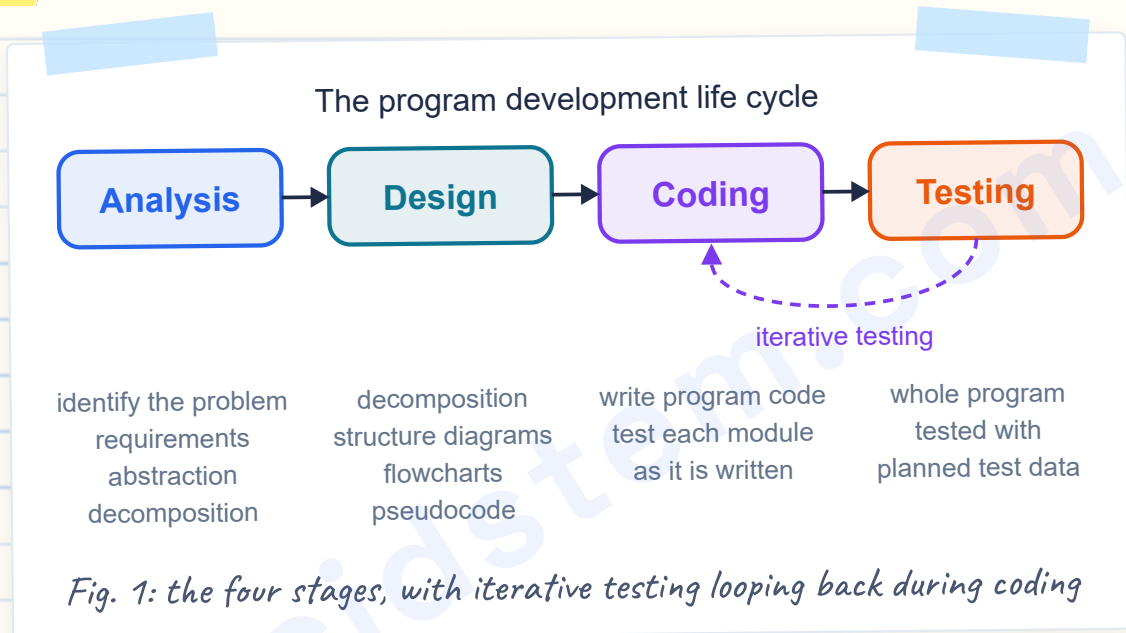
Everything from the 9 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

1. The program development life cycle
2. Decomposition, structure diagrams, flowcharts and pseudocode
3. Standard methods: linear search and bubble sort
4. Standard methods: totalling, counting, maximum, minimum and average
5. Validation and verification checks
6. Normal, abnormal, extreme and boundary test data
7. Dry-running algorithms with trace tables
8. Finding and correcting errors in algorithms
9. Writing and amending algorithms

1 The program development life cycle

Syllabus 7 (1-2)

The **program development life cycle** has four stages. **Analysis**: identify the problem and its **requirements**, using **abstraction** (keep the important details, remove the rest) and **decomposition** (break it into smaller parts). **Design**: plan the solution with **structure diagrams**, **flowcharts** and **pseudocode**. **Coding**: write the program code, with **iterative testing** of each part as it is written. **Testing**: test the finished program with planned **test data**.



Worked example: Abstraction

A program will plan the fastest bus route between two stops. Give two details to keep and one detail to remove.

1. Ask: does this detail change the answer to the problem?
2. Keep: the stops on each route and the journey time between stops (also: departure times).
3. Remove: the colour of the buses or the names of the drivers; they do not affect the fastest route.

Answer: Keep stops and journey times; remove details such as bus colour

Name the stage, then the task.

- ✓ **Analysis:** identify the problem and requirements; abstraction; decomposition.
- ✓ **Design:** decomposition, structure diagrams, flowcharts, pseudocode.
- ✓ **Coding:** write the program code; iterative testing of each part.
- ✓ **Testing:** test the whole program with test data.

Watch out!

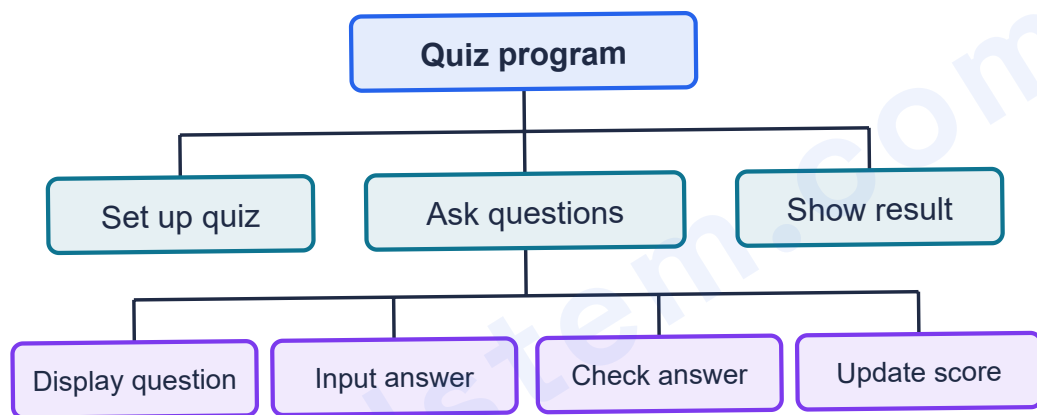
Abstraction is not the same as decomposition. Abstraction removes unnecessary detail; decomposition splits the problem into smaller parts. Questions often ask you to describe one of them, so do not mix them up.

Iterative testing belongs to the **coding** stage: small parts are tested and corrected repeatedly while they are being written. The **testing** stage is when the whole program is tested with test data.

2 Decomposition, structure diagrams, flowcharts and pseudocode

Syllabus 7 (1-2)

Every computer system is made up of **sub-systems**, which are made up of further sub-systems. **Decomposition** breaks a problem into its component parts: **inputs**, **processes**, **outputs** and **storage**. A solution can be designed with a **structure diagram** (the hierarchy of sub-systems), a **flowchart** (the steps, using standard symbols) or **pseudocode** (the steps written as structured code).



each level breaks a task into smaller sub-tasks (sub-systems)

Fig. 1: a structure diagram for a quiz program, decomposed into two levels of sub-systems

Worked example: Decompose a problem

A car park barrier lets a car in when a valid ticket is inserted. Identify the inputs, processes, outputs and storage.

1. Inputs: the ticket code read by the scanner; a sensor showing a car is at the barrier.
2. Processes: check the ticket code is valid and not already used; count the cars inside.
3. Outputs: raise the barrier; show "Car park full" on a display.
4. Storage: the list of valid ticket codes and the number of free spaces.

Answer: Inputs, processes, outputs and storage each identified

What, then how.

- ✓ **Decompose:** inputs, processes, outputs, storage.
- ✓ **Structure diagram:** the hierarchy of sub-systems; no order or decisions.
- ✓ **Flowchart:** terminator, process, input/output, decision, subroutine, flow lines.
- ✓ **Purpose:** say what the algorithm is for in its context, then describe the processes.

Watch out!

A structure diagram is not a flowchart. It has no decisions, loops or arrows showing order: it only shows how the system is divided into sub-systems.

Every decision diamond needs exactly two exits, and both must be labelled (Yes/No or True/False). Missing labels lose marks.

"It inputs numbers and outputs a number" is not a purpose. State what the algorithm is for, using the context: counting freezing days, finding the highest score, and so on.

3 Standard methods: linear search and bubble sort

Syllabus 7 (3-4)

A **linear search** checks each item of a list **in turn**, from the first, until the item is found or the end of the list is reached. A **bubble sort** compares each pair of **adjacent** items and **swaps** them if they are in the wrong order. One **pass** moves the largest remaining item to the end. Passes repeat until a whole pass makes **no swaps**, so the list is sorted.

Bubble sort: the first pass through 7, 3, 9, 1, 5



Fig. 1: in the first pass the largest value, 9, bubbles along to the end of the list

Worked example: Search for Dev

Using the algorithm above, how many names are compared when the user enters Dev, and what is output?

1. Index 1: Ali is not Dev, so Index becomes 2.
2. Index 2: Bea is not Dev; Index 3: Chen is not Dev; Index becomes 4.
3. Index 4: Dev matches, so Found becomes TRUE and the loop stops.
4. 4 comparisons were made, and Index is still 4.

Answer: 4 comparisons; output "Dev found at position 4"

Two methods, learnt by heart.

- ✓ **Linear search:** start at item 1; compare; move on; stop when found or at the end.
- ✓ **Bubble sort:** compare neighbours; swap if in the wrong order (using Temp).
- ✓ **Pass:** one run through the list; the largest remaining value reaches its place.
- ✓ **Stop:** when a whole pass makes no swaps.

Watch out!

Do not stop the search with only one condition. The loop needs both: stop when the item is found or when the end of the list is reached, otherwise a missing item makes Index run past the end of the array.

Swapping with `Values[Index]`
`← Values[Index + 1]` followed
by `Values[Index + 1] ←`
`Values[Index]` does not swap:
both end up as the same value.
Always use a temporary variable.

The list can be sorted before the algorithm knows it. After pass 3 it is in order, but one more pass is needed to make no swaps and confirm it.

4 Standard methods: totalling, counting, maximum, minimum and average

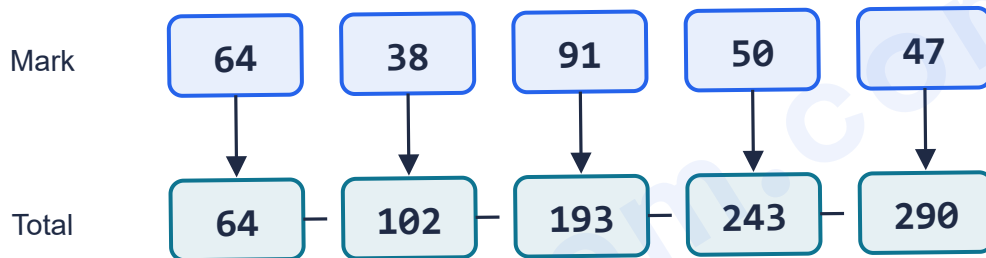
Syllabus 7 (3-4)

Totalling: set $\text{Total} \leftarrow 0$ before the loop, then $\text{Total} \leftarrow \text{Total} + \text{Value}$ inside it.

Counting: set $\text{Count} \leftarrow 0$, then $\text{Count} \leftarrow \text{Count} + 1$ each time the condition is met.

Maximum and **minimum**: start from a value that will certainly be replaced (or from the first item), then replace it when a bigger (or smaller) value is found. **Average**: $\text{total} \div \text{number of values}$, calculated **after** the loop.

Totalling: add each mark to a running total



Total 290, Average $290 / 5 = 58$

Highest 91, Lowest 38, Count of marks ≥ 50 : 3

Fig. 1: five marks added one at a time to a running total

Worked example: Average with a count

Values are input until -1 is entered. The inputs are 12, 15, 20, 9, -1. Find the total, the count and the average.

1. The rogue value -1 stops the loop; it is not added or counted.
2. $\text{Total} = 12 + 15 + 20 + 9 = 56$.
3. $\text{Count} = 4$ values.
4. $\text{Average} = 56 / 4 = 14$.

Answer: Total 56, count 4, average 14

Initialise, update, finish.

- ✓ **Before the loop:** $Total \leftarrow 0$, $Count \leftarrow 0$, Highest low, Lowest high (or both the first value).
- ✓ **Inside the loop:** $Total \leftarrow Total + Value$; $Count \leftarrow Count + 1$ when the condition is true.
- ✓ **Max/min:** IF $Value > Highest$ THEN $Highest \leftarrow Value$ (and the reverse for Lowest).
- ✓ **After the loop:** $Average \leftarrow Total / \text{number of values}$; then output.

Watch out!

Totalling adds the value ($Total \leftarrow Total + Mark$); counting adds 1 ($Count \leftarrow Count + 1$). Swapping them is one of the most common errors examiners set.

Starting Lowest at 0 is wrong whenever all the values could be above 0: the minimum would never change. Start the minimum high and the maximum low, or start both at the first value.

Do not calculate the average inside the loop, and do not divide by a loop counter whose value after the loop you are unsure of. Divide by a known number, or by a Count variable you have kept yourself.

5 Validation and verification checks

Syllabus 7 (5-6)

Validation is an **automatic** check by the computer that input data is **sensible and allowed** before it is accepted: **range**, **length**, **type**, **presence**, **format** and **check digit** checks. **Verification** checks that data has been **entered accurately**, matching the original: a **visual check** or a **double entry check**. Validation cannot prove data is correct, only that it is reasonable.

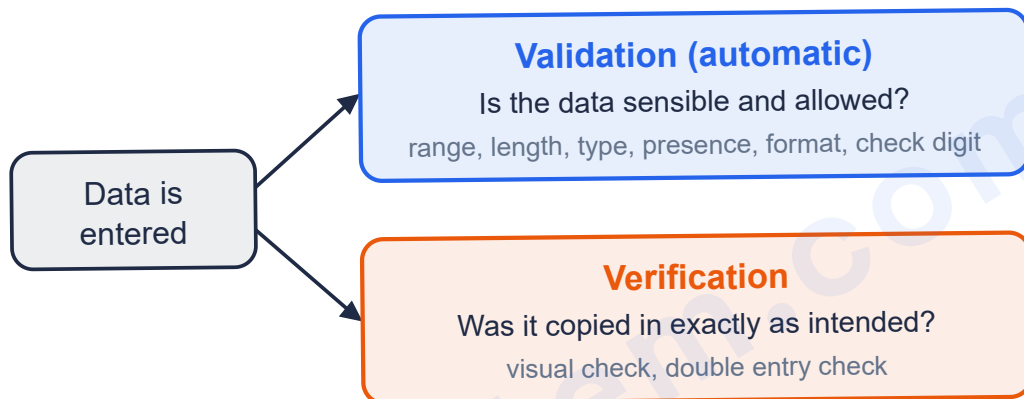


Fig. 1: validation asks whether data is allowed; verification asks whether it was entered as intended

Worked example: ISBN-13 check digit

Calculate the check digit for the first 12 digits of an ISBN: 978123456789.

1. Multiply the digits alternately by 1 and 3: 9×1 , 7×3 , 8×1 , 1×3 , 2×1 , 3×3 , 4×1 , 5×3 , 6×1 , 7×3 , 8×1 , 9×3 .
2. Products: 9, 21, 8, 3, 2, 9, 4, 15, 6, 21, 8, 27. Their sum is 133.
3. Divide by 10 and take the remainder: $133 \text{ MOD } 10 = 3$.
4. Subtract from 10: $10 - 3 = 7$. (If the remainder is 0, the check digit is 0.)

Answer: Check digit 7, so the full ISBN is 9781234567897

Allowed, or as intended?

- ✓ **Validation (automatic):** range, length, type, presence, format, check digit.
- ✓ **Verification:** visual check, double entry check.
- ✓ **Describe a check:** say what it tests and give an example from the question's context.
- ✓ **Write a check:** input, test the rule, error message, input again until valid.

Watch out!

A range check is about the *value* (13 is too big); a length check is about the *number of characters* ("abc" is too short). Name the check that matches the rule in the question.

A check digit is a **validation** check (it is calculated automatically), even though it catches typing mistakes. It cannot tell you whether the number is the right book, only that the digits are consistent.

Validation and verification are not interchangeable. Validation: is the data *reasonable*? Verification: is it *the same as the original*? Neither guarantees the data is true.

6 Normal, abnormal, extreme and boundary test data

Syllabus 7 (5-6)

Normal data is typical data that should be **accepted**. **Abnormal** (erroneous) data should be **rejected**: out of range or the wrong type. **Extreme** data is the **largest or smallest acceptable** value. **Boundary** data is the largest or smallest acceptable value **and** the corresponding smallest or largest **rejected** value, testing both sides of each limit. Each test has an **expected result**.

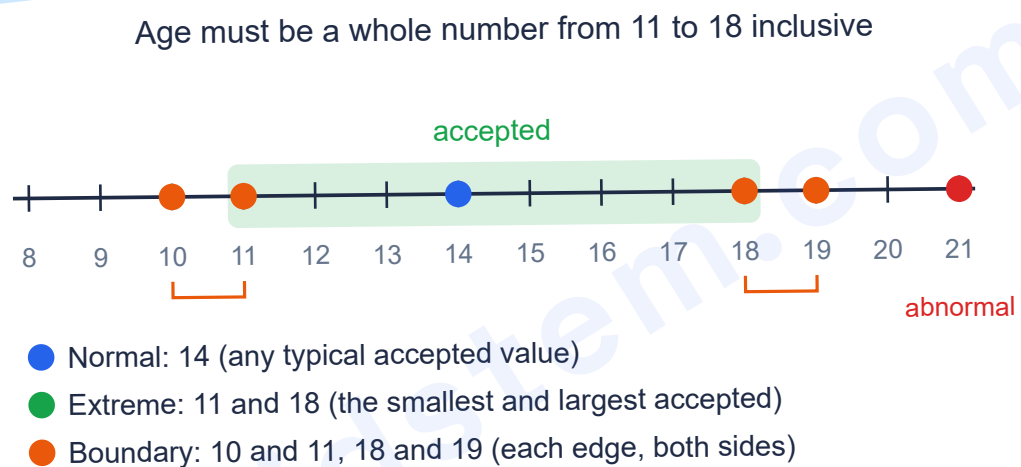


Fig. 1: for ages 11 to 18, the four kinds of test data sit in different places on the number line

Worked example: A percentage

A program accepts whole-number percentages from 0 to 100 inclusive. Give one example of each type of test data.

1. Normal: a typical value such as 65 (accepted).
2. Abnormal: a value outside the range or of the wrong type, such as 150 or "sixty" (rejected).
3. Extreme: the smallest or largest accepted value, 0 or 100 (accepted).
4. Boundary: at the top, 100 (accepted) and 101 (rejected); at the bottom, 0 (accepted) and -1 (rejected).

Answer: Normal 65; abnormal 150; extreme 0 or 100; boundary 100 and 101 (or 0 and -1)

Find the limits first.

- ✓ **Normal:** typical, accepted.
- ✓ **Abnormal:** out of range or wrong type, rejected.
- ✓ **Extreme:** the smallest or largest accepted value.
- ✓ **Boundary:** the last accepted value and the first rejected value at a limit.

Watch out!

Extreme data is always accepted: it is the very edge of what is allowed. Boundary data comes in pairs, one accepted and one rejected, at each limit.

Read "more than 10" carefully: 10 itself is rejected, so the smallest acceptable whole number is 11. The boundary pair is then 10 (rejected) and 11 (accepted).

An expected result must be specific: "rejected, error message shown" or "accepted", not just "it works".

7 Dry-running algorithms with trace tables

Syllabus 7 (7-9)

A **dry run** is working through an algorithm by hand with given test data. A **trace table** has a **column for each variable** and one for **OUTPUT** (including user prompts). Work line by line: write a new value in the column **only when it changes**, move down a row as the algorithm moves on, and follow every condition and loop exactly as written, not as you think the programmer meant.

Trace table for the digit-sum algorithm, input 352

Number	Digit	Sum	OUTPUT
352		0	
35	2	2	
3	5	7	
0	3	10	
			10

one new row each time round the loop; the highlighted row is the third pass

Fig. 1: a completed trace table for the digit-sum algorithm with input 352

Worked example: Trace with input 6

Complete the trace for the algorithm above when the input is 6.

1. Number = 6. Count = 1: 6 is even, so Number = $\text{DIV}(6, 2) = 3$. Output 3.
2. Count = 2: 3 is odd, so Number = $3 \times 3 + 1 = 10$. Output 10.
3. Count = 3: 10 is even, so Number = 5. Output 5.
4. Count = 4: 5 is odd, so Number = $3 \times 5 + 1 = 16$. Output 16. The loop ends after Count = 4.

Answer: Outputs 3, 10, 5, 16

Follow the code, not your guess.

- ✓ **Columns:** one per variable, plus OUTPUT (prompts too).
- ✓ **Rows:** move down as the algorithm moves on; record a value only when it changes.
- ✓ **Conditions:** test each one with the current values, every time.
- ✓ **Loops:** WHILE tests first (may run 0 times); REPEAT tests last (runs at least once).

Watch out!

The rogue value -1 is input and recorded in the trace table, but the loop ends before it is processed. Do not add it to the total.

Use the new value of a variable on the next line. After Number becomes 3, the OUTPUT on the same pass shows 3, not 6.

A REPEAT loop always runs at least once, because the condition is tested at the end. With input 0 this algorithm still runs once and outputs 0.

8 Finding and correcting errors in algorithms

Syllabus 7 (7-9)

To find errors, first work out the algorithm's **intended purpose**, then **dry-run** it with simple test data and compare the actual result with the **expected result**. Common errors are a **wrong operator** (< for >), **counting instead of totalling**, a **wrong initial value** or one set inside the loop, **wrong loop bounds**, a **missing input inside a loop** and a **swap without a temporary variable**. For each error give the line number and the **corrected line of pseudocode**.

Intended: output the total of 10 numbers

```
1 Total ← 0
2 FOR Count ← 1 TO 10
3 INPUT Number
4 Total ← Total + 1
5 NEXT Count
6 OUTPUT Total
```

→ Total ← Total
+ Number

line 4 counts instead of adding

Fig. 1: line 4 adds 1 instead of the number input, so the algorithm counts instead of totalling

Worked example: Find and correct the errors

Identify the three errors and write the corrected lines.

1. Dry-run with simple data, e.g. ten 5s: Total ends as 10 (should be 50) and Highest stays 0 (should be 5).
2. Line 05 counts instead of totalling. Correct: Total ← Total + Number
3. Line 06 tests for a smaller number, so Highest never changes from 0. Correct: IF Number > Highest
4. Line 11 divides by 9, but there are 10 numbers. Correct: Average ← Total / 10

Answer: Lines 05, 06 and 11, corrected as shown

Purpose, dry run, fix.

- ✓ **Purpose:** read what the algorithm is meant to do.
- ✓ **Dry run:** use small, simple test data and compare actual with expected results.
- ✓ **Check the usual suspects:** operators, initial values, loop bounds, inputs in loops, swaps, divisors.
- ✓ **Answer:** the line number plus the full corrected statement.

Watch out!

"Line 5 is wrong" earns no mark on its own. Give the line number and the full corrected statement in pseudocode.

Do not rewrite the whole algorithm. Change only the faulty lines, and keep the original variable names.

A loop condition that uses a variable never changed inside the loop is a warning sign: either the loop never runs, or it never ends.

9 Writing and amending algorithms

Syllabus 7 (7-9)

An algorithm must be **precise**: write `Score > Target`, not "score is bigger than the target". Build a solution from the **standard parts**: **initialise** variables, **prompt and input**, **validate**, process inside a suitable **loop** (totalling, counting, max/min), then **output** clear results. Use **meaningful identifiers**. To **amend** an algorithm, keep what works and add or change only the lines needed.

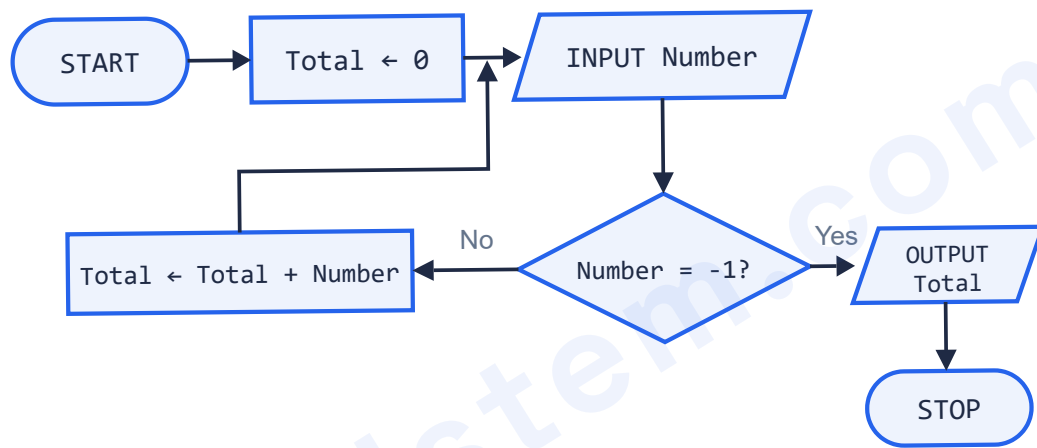


Fig. 1: a flowchart that totals numbers until -1 is entered

Worked example: A ticket scenario

Each of 30 customers buys 1 to 4 concert tickets at \$5.50 each. Write an algorithm that validates each input, then outputs the total money taken and the number of customers who bought 4 tickets.

1. Initialise: `TotalTickets ← 0` and `FourCount ← 0` before any loop; declare the constant `Price`.
2. Count-controlled loop for the 30 customers: `FOR Customer ← 1 TO 30 ... NEXT Customer`.
3. Inside it, validate with a REPEAT loop: prompt, `INPUT Tickets`, error message if `Tickets < 1 OR Tickets > 4`, UNTIL the value is valid.
4. Total the tickets and count the customers who bought 4; after the loop calculate `Takings ← TotalTickets * Price` and output both results.

Answer: The full pseudocode is shown above

Initialise, input, process, output.

- ✓ **Initialise:** totals and counts to 0; max low, min high; declare constants.
- ✓ **Input:** prompt, then INPUT; validate with a loop if the question asks.
- ✓ **Process:** choose FOR (known number), WHILE or REPEAT (until a condition).
- ✓ **Output:** results with clear messages, after the loop when they are final.

Watch out!

Describing the algorithm in English sentences earns little or nothing. Write statements: INPUT, OUTPUT, $\leftarrow$, IF, loops.

Put the validation loop *inside* the customer loop, around the INPUT. Validating once before the main loop checks only the first customer.

When amending, keep the original identifiers and structure. Adding a whole new loop, or renaming variables, risks breaking the parts that already worked.