

Validation and verification

Read online at pastopic.com/cambridge/computer-science-0478/notes/validation-and-verification

Last checked 30 September 2026

Read first: [Methods of error detection](#), [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. **Explain why input data needs validation checks.** The program itself tests every value it is given against rules, and refuses any value that breaks a rule, asking for it again.
2. **Name, describe and use the six validation checks** in the syllabus: **range check**, **length check**, **type check**, **presence check**, **format check** and **check digit**. Pick the right one for a rule, say what it would do with the data in a given situation, and write it in pseudocode.
3. **Explain why input data needs verification checks.** They find slips made while copying or typing, so that what is stored is a faithful copy of the original.
4. **Describe the two verification checks**, the **visual check** and the **double entry check**, say what each is for, and write an algorithm for each one.

Choosing test data (7.6) and writing, tracing and correcting algorithms (7.7–7.9) belong to **Testing, trace tables and errors**. Exam questions often join them to validation, so "How to do it" shows how they fit here.

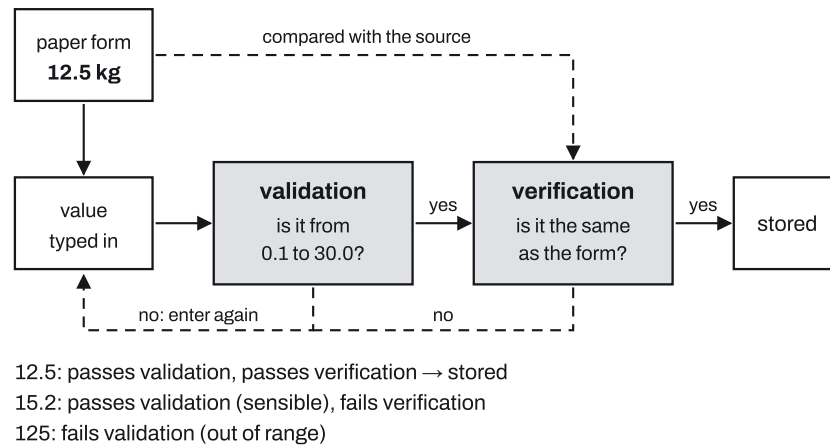
Understand it

Two different questions about one value

When a person types a value into a computer, two different things can go wrong. The value can be **impossible**: a price of –4, a month of 13, a surname left blank. Or it can be **possible but not what was meant**: the form says 12.5 kg and the clerk types 15.2 kg. Validation and verification deal with these two problems.

- **Validation** asks: *does this value obey the rules?* The **program** tests it against rules it has been given, with no person involved, and refuses a value that breaks them.
- **Verification** asks: *is this the value we were meant to get?* It compares what arrived in the computer with where it came from (a paper form, a label, what the user intended) to catch a slip made while it was being keyed in.

Neither of them proves the data is **true**. A value can obey every rule and still be the wrong value (15.2 is a perfectly reasonable weight); a value can be copied perfectly from a source that was itself wrong. That is why systems use both.



The six validation checks

Each check tests one kind of rule. These examples come from an online gym sign-up form.

Check	What it tests	Example rule	Pseudocode test for a bad value
Range check	a number is not below a lowest value or above a highest value	age from 16 to 99 inclusive	<code>Age < 16 OR Age > 99</code>
Length check	how many characters there are (exactly, no fewer than, or no more than)	username 6 to 15 characters	<code>LENGTH(Username) < 6 OR LENGTH(Username) > 15</code>
Type check	the value is the kind of data expected, such as a whole number rather than text or a decimal	number of guest passes is a whole number	<code>Passes <> ROUND(Passes, 0)</code>
Presence check	the field is not empty	surname must be filled in	<code>Surname = ""</code>
Format check	the characters follow a fixed pattern	class start time as hh:mm	<code>SUBSTRING(Start, 3, 1) <> ':'</code>
Check digit	an ID number is keyed in without a slip, using an extra digit calculated from the others	5-digit member number ending in a check digit	recalculated digit $\neq$ entered digit

A few points about each:

- **Range** needs a test at **each** end unless the rule has only one end ("under 1000", "1 or more"). "From 16 to 99 inclusive" means $16 \leq \text{Age} \leq 99$, so a bad value is `Age < 16 OR Age > 99`.
- **Length** counts characters, not how big a number is. It suits strings: usernames, codes, phone numbers, text boxes with a size limit.
- **Type** is usually "is it a whole number?" A whole number doesn't change when it is rounded to 0 decimal places, so `Passes = ROUND(Passes, 0)` is true only for a whole number.
- **Presence** only notices an empty field. Anything at all typed in passes it.

- **Format** is about **which kind of character sits in which place**: two letters then four digits, a date as dd/mm/yyyy, a colon in the third place, an @ in an email address. Take out characters with `SUBSTRING(String, Start, Length)`.
- **Check digit** is the extra digit at the end of an ID number (a barcode, an ISBN, a library card), worked out from the other digits. It is explained below.

Validation in pseudocode: loop until it's valid

A bad value has to be asked for again, and the replacement can be bad too, so validation always needs a **loop** with the input **inside** it. A single **IF** gives the user only one more try. There are two standard shapes, and either is fine.

WHILE loop: input once, then keep asking while the value is **bad**.

```
OUTPUT "Enter your age (16 to 99)"
INPUT Age
WHILE Age < 16 OR Age > 99 DO
    OUTPUT "Age must be from 16 to 99, please try again"
    INPUT Age
ENDWHILE
```

REPEAT loop: keep asking **until** the value is **good**. Put the error message in an **IF** inside the loop, so it only appears after a bad value.

```
REPEAT
    OUTPUT "Choose a username of 6 to 15 characters"
    INPUT Username
    IF LENGTH(Username) < 6 OR LENGTH(Username) > 15
        THEN
            OUTPUT "That username is the wrong length"
        ENDIF
UNTIL LENGTH(Username) >= 6 AND LENGTH(Username) <= 15
```

Look at the two conditions. The **WHILE** condition describes a **bad** value, and its parts are joined with **OR** (breaking any one rule is enough to be bad). The **UNTIL** condition describes a **good** value, joined with **AND** (every rule must hold). Swapping them is the most common logic error in this topic.

One loop can hold several checks on the same input: `WHILE Username = "" OR LENGTH(Username) > 15` `DO` is a presence check and a length check together.

Checking the characters of a string

Some rules are about what is inside a string: it must start with a certain character, include a symbol, or be made only of digits. `LENGTH` tells you how many characters there are, and `SUBSTRING(Text, Position, 1)` gives you one character at a time. Characters can be compared, so `Ch >= '0' AND Ch <= '9'` is true for a digit and `Ch >= 'A' AND Ch <= 'Z'` for a capital letter.

A photo app saves pictures under a name the user types, and the name must have **exactly one full stop** (before the file type, as in `beach . jpg`). Visit every character in turn and count the full stops:

```

REPEAT
    OUTPUT "Enter a file name, for example beach.jpg"
    INPUT FileName
    Dots ← 0
    FOR Position ← 1 TO LENGTH(FileName)
        IF SUBSTRING(FileName, Position, 1) = '.'
            THEN
                Dots ← Dots + 1
            ENDIF
    NEXT Position
    IF Dots <> 1
        THEN
            OUTPUT "A file name needs exactly one full stop"
        ENDIF
UNTIL Dots = 1

```

"beach.jpg" has 1 full stop and is accepted; "beach" (0) and "my.beach.jpg" (2) are refused. Change what the `IF` compares with and the same shape looks for any character. Change the final test and it answers other questions: `Dots >= 1` for "includes at least one", `Dots = 0` for "must not include".

Sometimes you only need to know **whether** a character appears, and can stop looking once it has. Then use a condition-controlled loop instead of the `FOR` loop: a Boolean variable starts `FALSE` and becomes `TRUE` when the character turns up, and the loop keeps going while it is still `FALSE` **and** there are characters left to look at. After the loop, that variable tells you which message to show. To check that **every** character is a digit, turn the idea round: the Boolean starts `TRUE` and becomes `FALSE` at the first character that is **not** a digit.

Check digits

When an ID number is first created, a **check digit** is calculated from its other digits by a fixed method and put on the end. Each time the number is typed or scanned later, the computer does the **same calculation again** on the other digits and compares its answer with the last digit it received. If they differ, something was keyed wrongly and the number is refused.

The exam always tells you the method. A common kind multiplies the digits by **weights**, adds up the results and takes a remainder. For example, with weights 3, 1, 3, 1 and the check digit = $(10 - \text{total} \bmod 10) \bmod 10$, the number 6392 gives

$$6 \times 3 + 3 \times 1 + 9 \times 3 + 2 \times 1 = 18 + 3 + 27 + 2 = 50,$$

$50 \bmod 10 = 0$, and $(10 - 0) \bmod 10 = 0$, so the full number is 63920.

A check digit catches one digit keyed wrongly. If the digits are **weighted**, it also catches most cases of two neighbouring digits typed the wrong way round (a **transposition**). A check digit made by just **adding** the digits can't catch that, because the order doesn't change the total: $1 + 7 = 7 + 1$. With weights, neighbours count differently: $1 \times 3 + 7 \times 1 = 10$ but $7 \times 3 + 1 \times 1 = 22$, so the swap usually changes the check digit (Example 4(c) shows one being caught).

A check digit is used when data is **entered**. Checking data after it has been sent over a network is a different job, done by parity checks and checksums (see Methods of error detection).

Verification: visual check and double entry

Visual check. After typing, a person reads the value on the screen against the original document, symbol by symbol, and either confirms it or puts it right. It is quick and needs no extra software, but people miss slips, especially in long strings of digits.

A visual check in pseudocode **displays the value that was typed**, asks the user whether it is right, and repeats the input until they say yes:

```
REPEAT
    OUTPUT "Enter the delivery postcode"
    INPUT Postcode
    OUTPUT "You entered ", Postcode, ". Is this correct? (Y/N)"
    INPUT Answer
UNTIL Answer = "Y"
```

If the user notices "TN4 8DQ" where they meant "TN4 8QD", they answer N and type it again. The computer only shows the value; the **person** does the checking.

Double entry check. The value is keyed in a **second time**, and the **program** compares the two copies. If they agree it goes on; if not, one copy must contain a slip, so the user is told and has to type it again. It is used when the original can't be read back on screen, such as a new password shown as dots, or when one wrong digit would be costly, such as a bank account number.

A double entry check in pseudocode needs two inputs into two different variables, a comparison, and a loop that repeats until the copies agree:

```
REPEAT
    OUTPUT "Enter the account number for refunds"
    INPUT Account
    OUTPUT "Enter the account number again"
    INPUT Account2
    IF Account <> Account2
        THEN
            OUTPUT "The two account numbers do not match, please start again"
        ENDIF
UNTIL Account = Account2
```

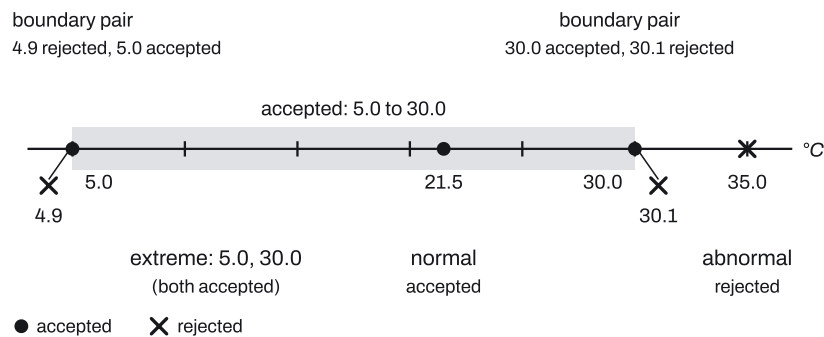
Both verification checks have the **same purpose**: to catch a copying or typing slip, so that the stored value matches its source. Neither can tell whether the value makes sense: typing "00000000" twice passes a double entry check.

Testing a validation check

To show a validation check works, you try it with values chosen to prove that it lets through what it should and stops what it should. For a rule "a whole number from 20 to 60 inclusive":

Test data type	Meaning	Example	Expected result
Normal	an ordinary value that obeys the rule	35	accepted

Test data type	Meaning	Example	Expected result
Abnormal (erroneous)	a value that breaks the rule: out of range, wrong type, wrong format	85, or "forty"	rejected
Extreme	the largest or smallest value that is accepted	20 and 60	accepted
Boundary	at each end, the last value let in and the first value kept out	20 and 19; 60 and 61	20, 60 accepted; 19, 61 rejected



Key facts and formulas

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

Fact	Status
Validation: the program tests each value against rules and refuses it if it breaks one	Learn it
Verification: a check for slips made while copying or typing , so the stored value matches its source	Learn it
Range check: lowest and highest allowed values; a bad value is $X < \text{Low}$ OR $X > \text{High}$	Learn it
Length check: the number of characters, <code>LENGTH(X)</code>	Learn it
Type check: the right kind of data, e.g. whole number: $X = \text{ROUND}(X, 0)$	Learn it
Presence check: the field is not empty, $X \neq ""$	Learn it
Format check: the pattern of characters, <code>SUBSTRING(X, Start, Length)</code>	Learn it
Check digit: worked out again from the other digits on entry and compared with the last digit	Learn it
Visual check: a person reads the screen against the original	Learn it

Fact	Status
Double entry check: keyed twice, the program compares the two copies	Learn it
WHILE loop: condition for a bad value (OR); REPEAT ... UNTIL: condition for a good value (AND)	Learn it
Test data: normal (accepted), abnormal (rejected), extreme (largest/smallest accepted), boundary (accepted and rejected pair at an end)	Learn it
Check digit methods are always given in the question	Given

How to do it

In the Paper 2 questions we have tagged for this topic (56 questions, 2021–2025), the types came up this often. 21 of the 56 questions mix more than one type, so the counts add up to 86.

Question type	Questions
Write or complete pseudocode that validates an input	17
Describe a verification method and why it is used	14
Name the right validation check for a rule	12
Explain what a validation check does, in context	10
Validate inputs inside a 15-mark program	9
Choose test data for a validation rule, with reasons	8
Pre-release task questions on validation (2021–2022 only)	7
Trace, correct or apply a check algorithm	5
Write pseudocode for a double entry check	2
Classify statements as validation, verification, both or neither	2

1. Write or complete pseudocode that validates an input (17 questions)

- Turn the rule into a check.** Two ends → range; a number of characters → LENGTH; whole number → ROUND or MOD; certain characters in certain places → SUBSTRING.
- Write the loop** with the prompt and the input **inside** it (for a WHILE loop, also once before it).
- Write the condition precisely.** Read "no fewer than", "at most", "inclusive" slowly. A comment box that allows "no more than 40 characters" is good when `LENGTH (Comment) <= 40`, so a bad value is `LENGTH (Comment) > 40`. In a WHILE, describe the bad value with OR; in an UNTIL, the good value with AND.
- Tell the user what went wrong**, with a message that appears only after a bad value and says what is allowed.
- Dry-run it** with one bad value then one good value: does a second bad value get caught, and does the loop stop at the first good one? Use the variable names the question gives.

Variations you will meet:

- **"Explain how you would change the algorithm"**: describe in words where the new lines go (next to the input they check) and what they do. Code on its own, without explanation, isn't enough.
- **Look for one character in a string**: the flag-and-position search in "Understand it", with the test changed to the character you need, and a message after the loop for whichever way it turned out.
- **Check the first characters**: take them with `SUBSTRING (Code, 1, n)` and compare the result with the required text.
- **Fill in the gaps** in a given validation algorithm: work out what each blank must be from the rule and from the lines around it (a missing limit, a missing re-input, a missing update).
- **A flowchart instead of code**: a decision box right after the input; its "no" branch goes to an error output and back to the input.

2. Describe a verification method and why it is used (14 questions)

1. **Name** one of the two verification methods: **visual check** or **double entry check**. A range, type or other validation check is never the answer here.
2. **Describe** how it works in your own words, covering who does what: for a visual check, a *person* reads the screen against the original and puts right any difference; for double entry, the value is *keyed twice* and the *program* compares the copies, asking again if they differ.
3. **Give the purpose** when asked "why": to catch a slip made while the data was typed or copied, so that what is stored matches the original. Don't say it makes the data "correct" or "sensible"; that is not what verification does.
4. **Write a visual check** if an algorithm is asked for: input, display the value back with a yes/no question, input the answer, and repeat the whole thing until the answer is yes.

Fit the method to the situation. A password typed as dots can't be read back, so double entry suits it. Data copied from a printed sheet can be read back against the sheet, so a visual check works.

3. Name the right validation check for a rule (12 questions)

1. Look at **what the rule is about**: the size of a number → **range**; how many characters → **length**; the kind of data (whole number, no letters) → **type**; not left empty → **presence**; which character goes where → **format**; an ID number ending in a calculated digit → **check digit**.
2. In a matching question there is usually a spare check. Do the ones you are sure of first, and check every line before you draw it.
3. Don't repeat a check that the question has already named, or one you have used for another part when it says "each must be different".

Some rules need **more than one** check, and some need fewer than you might think. A delivery date that must be filled in and be no more than 30 days away needs a **presence** check and a **range** check. An optional nickname of at most 12 characters needs only a **length** check: it may be left empty, so a presence check would be wrong.

4. Explain what a validation check does, in context (10 questions)

1. **Name** the check.
2. **Say what it would do with this data**, using the question's own item and limits: "a range check on the rating, refusing anything under 1 or over 5", not "it checks the value is in range".

3. For "State the purpose of a ... check", one short sentence in your own words is enough, from the table in "Understand it".
4. For "Describe validation", say that the program tests the input against rules and refuses what breaks them. Add an example check if there are two marks.

Describe a check for the input the question asks about, not a different one from the same program.

5. Validate inputs inside a 15-mark program (9 questions)

The long program question at the end of Paper 2 includes "input and validate ..." among its requirements, and validation is one of the techniques the levels marking looks for.

1. Find **every input the question says to validate**, and any menu choice. Give each its own REPEAT ... UNTIL (or WHILE) loop, with the prompt, the input and an error message inside.
2. Work out each rule from the scenario: whatever limits, sizes or menu options it gives for that input.
3. **Checks that use other values**: sometimes a value is only valid compared with something else already entered, such as parts that must add up to a whole. Put that comparison in the loop condition, for example UNTIL Home + Away = Total.
4. When many items are entered, put the validation loop **inside** the loop that goes through the items, so every one is checked.
5. Comment it, for example // keep asking until the rating is 1 to 5.

6. Choose test data for a validation rule, with reasons (8 questions)

1. Write down the rule exactly, with both ends and whether they are included.
2. Choose one value for each type asked for, all **different**: **normal** (comfortably inside the rule), **abnormal** (clearly breaks it), **extreme** (the smallest or largest value that is **let in**), **boundary** (at one end, the value let in **and** the first value kept out).
3. Give each value a **reason** that says which kind it is **and** whether the program should **accept or reject** it, for example "the top of the allowed range, so the program should let it through".

For decimals, "the first value kept out" depends on the number of places: with a lowest price of 0.50 to 2 decimal places, the pair is 0.50 and 0.49. For strings, test data is about the length: for a username of 4 to 10 characters, a 10-character name is accepted and an 11-character one rejected. Abnormal data must really **break the rule**; a value that obeys the rule but is unusual is still normal data.

7. Pre-release task questions on validation (7 questions, 2021–2022 only)

Before 2023, Paper 2 Question 1 was about a program candidates had written in advance. These questions no longer appear, but the skills do. To **describe how an input is validated**, name the check with its rule for that input, then explain the code: the test for a bad value, the message, the re-input, and the loop that keeps going until the value is acceptable. To **describe how you would test that validation**, name the kinds of test data you would use and what the program should do with each (accept or reject).

8. Trace, correct or apply a check algorithm (5 questions)

Apply a check digit method:

1. Follow the steps in the order given, writing down each intermediate result (every product, the total, the remainder from MOD, then whatever the last step is).
2. **To test a whole number**, work out the check digit again from every digit except the last, and compare.
3. **Questions about what a method can or can't detect**: try it. Make a small change to a number of your own (one digit altered, two digits exchanged) and see whether the check digit changes. "Understand it" explains which kinds of method notice which slips.

Find and correct errors in a given algorithm. Read the description above the code first. Then go through the code line by line and ask, for each line:

- **Declarations**: does each data type suit what is stored in it?
- **Starting values**: is every total, counter and flag set before it is used, to the right value?
- **Loops**: does each loop start and stop where the description says, so it handles exactly the right items?
- **Calculations**: does each one carry out the step described, with the right operator and the right values?
- **Conditions**: is the right thing tested, with the right comparison (< or <=, = or >) and the right AND/OR?
- **Array indexes**: is the right element used?
- **Outputs**: is the right variable or message shown, at the right time?

Give each correction as the whole corrected line, not a description.

Trace an algorithm and state its rules:

1. Make one column per variable and one for output, and write a new value only when it changes. A flag that starts TRUE and is set to FALSE by any broken rule stays FALSE. A search loop stops as soon as it finds what it is looking for.
2. To **state the rules**, turn each IF into a sentence. For example, if a line that sets the flag to FALSE is guarded by `IF LENGTH(Nickname) > 12`, the rule is "a nickname may have at most 12 characters"; if it is guarded by `IF SUBSTRING(Colour, 1, 1) <> '#'`, the rule is "a colour code must begin with #".

9. Write pseudocode for a double entry check (2 questions)

1. Input the value, then input it a second time into a **different** variable.
2. Compare the two. If they differ, output a message (showing both values if asked) and ask again.
3. Loop until they agree, and make sure the agreed value ends up where the question wants it stored.
4. **For many values in an array**, the idea is the same, applied item by item: the second copy of each item is compared with the element in the same position. These questions give a bulleted specification; turn each bullet into code in the order given, and check each one is done once per item or once in total, as it says.

Only write what the question asks for; adding range or type checks to a double entry answer wastes time.

10. Classify statements as validation, verification, both or neither (2 questions)

- If the statement describes the program testing a value against a rule of its own, it is **validation**.

- If it describes comparing the value with where it came from, it is **verification**.
- If it would be true of either kind of check, tick **both**.
- If it describes something no check can do, or has nothing to do with checking input, it is **neither**. Decide what each statement is really claiming before you tick.

Worked examples

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

- **M1** method mark: for a correct method, even if the numbers slip.
- **A1** accuracy mark: for a correct answer; it needs the M mark before it.
- **B1** mark for a correct result on its own, with no method needed.

Example 1

A courier company's booking form has these inputs:

Input	Rule
Parcel weight (kg)	from 0.1 to 30.0
Number of parcels	a whole number
Delivery postcode	two letters, a digit, a space, a digit, two letters, such as TN4 8QD
Tracking number	eight digits; the last is calculated from the first seven
Delivery instructions	no more than 100 characters

(a) For each input, name the most suitable validation check and state how it would be used. Each check must be different. **[10]**

(b) A clerk types a weight of 12.5 kg from a paper form as 15.2 kg. Explain why validation does not stop this error, and describe a check that would. **[3]**

(a) One mark for each check, one for how it is used with this input:

- Parcel weight: **range check B1**, refusing any weight under 0.1 kg or over 30.0 kg **B1**.
- Number of parcels: **type check B1**, refusing text or a decimal so only a whole number gets through **B1**.
- Postcode: **format check B1**, testing each place holds the right kind of character: letter, letter, digit, space, digit, letter, letter **B1**.
- Tracking number: **check digit B1**: the computer works out the last digit again from the first seven and refuses the number if its answer is different **B1**.
- Instructions: **length check B1**, refusing more than 100 characters **B1**.

(b) 15.2 lies between 0.1 and 30.0, so it passes the range check: validation only tests the rules, and a wrong but reasonable value obeys them **B1**. A **visual check B1**: before saving, the clerk reads the weight on the screen against the paper form and corrects it **B1** (or a **double entry check**: the weight is keyed a second time and the program only

goes on if both copies agree). The first diagram in "Understand it" shows this case.

Example 2

A smart thermostat accepts a target temperature from 5.0 °C to 30.0 °C inclusive, to one decimal place.

(a) Write pseudocode to input the target temperature into the variable `Target`, re-inputting it until it is in the accepted range, with suitable messages. **[4]**

(b) Complete a test plan for the check with one piece of normal data, one of abnormal data, both pieces of extreme data and one boundary pair at the lower end. For each, give the expected result. **[5]**

(a)

```

OUTPUT "Enter the target temperature (5.0 to 30.0)"
INPUT Target
WHILE Target < 5.0 OR Target > 30.0 DO
    OUTPUT "Temperature must be from 5.0 to 30.0, please try again"
    INPUT Target
ENDWHILE
OUTPUT "Thermostat set to ", Target

```

- A condition-controlled loop. **B1**
- The condition with both ends, `< 5.0 OR > 30.0`. **B1**
- An error message inside the loop. **B1**
- Re-input inside the loop, so the loop ends when a valid value is entered. **B1**

(b)

Test data	Type	Expected result
21.5	normal	accepted B1
35.0 (or "warm")	abnormal	rejected B1
5.0 and 30.0	extreme	both accepted B1
5.0	boundary	accepted B1
4.9	boundary	rejected B1

The last two rows are the boundary pair, one mark each: 5.0 is the smallest value let in and 4.9 is the next value down at one decimal place. The second diagram in "Understand it" shows every value on a number line.

Example 3

A kayak club takes bookings online. Each booking needs:

- the number of the boat: two capital letters, a hyphen, then two digits, for example KW-07;
- the length of the hire in hours: from 0.5 to 8.0 (half hours are allowed).

Write pseudocode that inputs and validates both values, re-inputting each until it is valid, with suitable messages, then outputs the booking. You do not need to declare variables. **[7]**

```
// format check on the boat number
REPEAT
    OUTPUT "Enter the boat number, for example KW-07"
    INPUT Boat
    Valid ← TRUE
    IF LENGTH(Boat) <> 5
        THEN
            Valid ← FALSE
        ELSE
            FOR Position ← 1 TO 5
                Ch ← SUBSTRING(Boat, Position, 1)
                IF Position <= 2 AND (Ch < 'A' OR Ch > 'Z')
                    THEN
                        Valid ← FALSE
                    ENDIF
                IF Position = 3 AND Ch <> '-'
                    THEN
                        Valid ← FALSE
                    ENDIF
                IF Position >= 4 AND (Ch < '0' OR Ch > '9')
                    THEN
                        Valid ← FALSE
                    ENDIF
            NEXT Position
        ENDIF
    IF Valid = FALSE
        THEN
            OUTPUT "The boat number must be two capital letters, a hyphen and two digits"
        ENDIF
UNTIL Valid = TRUE
// range check on the hours
REPEAT
    OUTPUT "How many hours (0.5 to 8.0)?"
    INPUT Hours
    IF Hours < 0.5 OR Hours > 8.0
        THEN
            OUTPUT "Hours must be from 0.5 to 8.0"
        ENDIF
UNTIL Hours >= 0.5 AND Hours <= 8.0
OUTPUT "Kayak ", Boat, " booked for ", Hours, " hours"
```

- A loop for the boat number that repeats until it is valid, with the input inside. **B1**
- LENGTH(Boat) compared with 5. **B1**
- SUBSTRING used to take each character in turn. **B1**

- Letters, hyphen and digits each tested in the right places. **B1**
- A separate loop for the hours, with the input inside. **B1**
- Both ends of the range, 0.5 and 8.0, included. **B1**
- Error messages for both inputs and the final output. **B1**

Dry run: "kw-07" is rejected (lower-case letters), "KW-7" is rejected (4 characters), "KW-07" is accepted. Then 9 is rejected (too long), 0.25 is rejected (too short), 2.5 is accepted and "Kayak KW-07 booked for 2.5 hours" is output.

Example 4

A library card number has seven digits. The last digit is a check digit, calculated from the first six like this:

- multiply the digits by 3, 1, 3, 1, 3, 1 in turn and add the results;
- find the remainder when the total is divided by 10;
- the check digit is 10 minus the remainder, or 0 if the remainder is 0.

(a) Calculate the check digit for the first six digits 204517. **[2]**

(b) The seven digits of a card number are input one at a time into the array `Digit[1:7]`. Write pseudocode to input the digits, work out the check digit again from the first six, and output whether the card number is valid. You do not need to declare any variables. **[5]**

(c) A librarian means to type the card number from (a) but swaps two digits and types 2054177. State, with working, what your algorithm outputs. **[2]**

(d) A trainee wrote this version of the checking part. Identify the three lines with errors and write each one correctly. **[3]**

```

01 Total ← 0
02 FOR Position ← 1 TO 6
03     IF MOD(Position, 2) = 0
04         THEN
05             Total ← Total + Digit[Position] * 3
06         ELSE
07             Total ← Total + Digit[Position]
08     ENDIF
09 NEXT Position
10 Check ← 10 - MOD(Total, 10)
11 IF Check = Digit[6]
12     THEN
13         OUTPUT "Card number valid"
14     ELSE
15         OUTPUT "Card number not valid"
16 ENDIF

```

(a) $2 \times 3 + 0 \times 1 + 4 \times 3 + 5 \times 1 + 1 \times 3 + 7 \times 1 = 6 + 0 + 12 + 5 + 3 + 7 = 33$. **M1**

$33 \text{ MOD } 10 = 3$, and $10 - 3 = 7$, so the check digit is **7** and the card number is 2045177. **A1**

(b)

```

FOR Position ← 1 TO 7
    OUTPUT "Enter digit ", Position
    INPUT Digit[Position]
NEXT Position
Total ← 0
FOR Position ← 1 TO 6
    IF MOD(Position, 2) = 1
        THEN
            Total ← Total + Digit[Position] * 3           // odd positions weigh 3
        ELSE
            Total ← Total + Digit[Position]               // even positions weigh 1
        ENDIF
NEXT Position
Check ← MOD(10 - MOD(Total, 10), 10)
IF Check = Digit[7]
    THEN
        OUTPUT "Card number valid"
    ELSE
        OUTPUT "Card number not valid, please enter it again"
ENDIF

```

- All seven digits input into the array in a loop. **B1**
- Total set to 0 and a loop over the first six digits only. **B1**
- Weights 3 and 1 applied in turn. **B1**
- The check digit found with MOD, including the case where the remainder is 0 (the outer MOD(..., 10) turns 10 into 0). **B1**
- The worked-out digit compared with Digit[7], with an output for both results. **B1**

(c) From 205417: $6 + 0 + 15 + 4 + 3 + 7 = 35$, $35 \text{ MOD } 10 = 5$, so Check is $10 - 5 = 5$ **M1**. That is not the 7 entered, so it outputs "Card number not valid, please enter it again": the swap has been caught. **A1**

(d) One mark for each line and its correction:

- Line 03: the weight of 3 belongs on the odd positions, so IF MOD(Position, 2) = 1. **B1**
- Line 10: a remainder of 0 would give 10, not 0, so Check ← MOD(10 - MOD(Total, 10), 10). **B1**
- Line 11: the check digit is the seventh digit, so IF Check = Digit[7]. **B1**

Common mistakes

- **Answering a verification question with validation checks.** Examiner reports mention this almost every year. If the question says verification, the answer is a visual check or double entry, nothing else.
- **Saying a check makes data "correct".** Validation tests rules and verification catches copying slips. Neither can prove that a value is true, and answers that claim it lose the mark.

- **A check with no link to the data.** A bare name such as "type check" usually earns only half the marks. Say what it would do with **this** input, using its limits or pattern.
- **Describing a check for the wrong input.** If the question is about one field, a check on a different field from the same scenario earns nothing.
- **A range check with one end missing**, or with $<$ where $\leq$ is needed. Reports note that full marks go to answers with **both** ends right.
- **One IF instead of a loop**, or the input outside the loop. The user must be asked again for as long as the value is bad.
- **Mixing up AND and OR.** A WHILE condition that describes a good value keeps asking for more after a valid value. Say the condition aloud with a test value.
- **Confusing extreme and boundary data.** Extreme values are always accepted; a boundary pair has one value that is accepted and one that is rejected.
- **Test data reasons that don't say what happens.** Every reason should say whether the program accepts or rejects the value.
- **Only code when an explanation is asked for.** "Explain how you would change the algorithm" needs words; code without explanation scores nothing in these parts.
- **Adding extras nobody asked for.** Range checks bolted onto a double entry algorithm, or declarations the question says aren't needed, cost time and can break the logic.
- **Calling a check digit a transmission check.** It is used on **input**; parity and checksums are for data that has been sent.

How to get the marks

- **"State the purpose of ..."**: one plain sentence in your own words on what the check stops getting through. No example is needed unless asked.
- **"Identify ... and describe"**: the name earns one mark; the rest are for how it works, **in the question's situation**.
- **"Explain why verification is used"**: link it to typing or copying. People make slips when they key data in from somewhere else, and verification exists to catch them.
- **Pseudocode** is marked on what it does, point by point. A marker checks that a bad value can't get through, that the user is asked again, that they are told why, and that a good value ends the loop. Use the variable names given, and write every condition exactly (`Mark >= 0`, not "mark is not negative"); the syllabus asks for this precision.
- **"Explain how you would change the algorithm"**: say where the change goes and what it does; any code you give must be explained.
- **Test data**: each value different, each **type** named if asked, and a reason that says **accepted** or **rejected**. Boundary data comes in pairs.
- **Check digit calculations**: show every product and the total, so the method mark is safe even if the last step slips.
- **Correcting errors**: give the line number and the whole corrected line.
- **Tick grids and matching**: one tick per row and one line per item, as the question says; extra ticks or lines lose the mark.

- **Pseudocode only:** outside the 15-mark question, program code or a flowchart earns nothing when pseudocode is asked for.

Quick check

1. Say whether each is **validation**, **verification**, **both** or **neither**: (i) a booking form refuses a return date that is before the outward date; (ii) a nurse reads the dose shown on screen back against the prescription; (iii) making sure a customer has given their real name; (iv) a new email address is typed into two boxes and compared.
2. Name the most suitable validation check for: (i) a percentage mark from 0 to 100; (ii) a car registration in the pattern two letters, two digits, a space, three letters; (iii) a 13-digit barcode whose last digit is worked out from the other twelve; (iv) an emergency contact name that cannot be left empty.
3. A café's app accepts the number of guests at a table as a whole number from 2 to 12 inclusive. Give the boundary pair at the upper end and one piece of abnormal data, with the expected result for each.
4. Using the check digit method from Example 4, say whether each card number is valid: (a) 3815209; (b) 5102648.
5. A student validates a ten-pin bowling score from 0 to 300 with `INPUT Score` followed by `IF Score < 0 OR Score > 300 THEN INPUT Score ENDIF`. Explain the problem and write a correct version.

Answers

- (i) Validation (the program tests a rule comparing two dates). (ii) Verification (a visual check against the source). (iii) Neither (no check can prove data is true). (iv) Verification (a double entry check).
- (i) Range check; (ii) format check; (iii) check digit; (iv) presence check.
- Boundary pair: 12, accepted (the largest number let in) and 13, rejected (the first number past the end).
Abnormal: for example 40 or "six", rejected (out of range / not a number).
- (a) $3 \times 3 + 8 + 1 \times 3 + 5 + 2 \times 3 + 0 = 9 + 8 + 3 + 5 + 6 + 0 = 31$; $31 \text{ MOD } 10 = 1$; $10 - 1 = 9$, which matches the last digit, so **valid**. (b) $15 + 1 + 0 + 2 + 18 + 4 = 40$; $40 \text{ MOD } 10 = 0$, so the check digit should be 0, not 8: **invalid**.
- The IF allows only one re-entry, so a second bad score is accepted. Use a loop:

```
INPUT Score
WHILE Score < 0 OR Score > 300 DO
    OUTPUT "Score must be from 0 to 300, please re-enter"
    INPUT Score
ENDWHILE
```

Past questions to try

- 0478/22/M/J/24 Q4**: three validation checks for a product code, then LENGTH and SUBSTRING pseudocode.
- 0478/22/M/J/23 Q4**: checks for a measurement, then a double entry check in pseudocode.
- 0478/23/O/N/22 Q3**: calculate and test check digits, and find an error a simple sum misses.
- 0478/23/O/N/24 Q10**: trace a password checker and state its rules.