

Standard methods of solution

Read online at pastopic.com/cambridge/computer-science-0478/notes/standard-methods-of-solution

Last checked 30 September 2026

Read first: [Program development and design](#)

© 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 understand, use, recognise and write the five standard methods of solution in the syllabus:

1. **Linear search:** look at each item of an array in turn until the item you want is found, or the end of the array is reached.
2. **Bubble sort:** put the items of an array into ascending or descending order by comparing neighbouring items and swapping them when they are the wrong way round, pass after pass.
3. **Totalling:** keep a running total by adding each value to a variable that starts at zero.
4. **Counting:** add one to a variable (that starts at zero) each time something happens.
5. **Finding the maximum, the minimum and the average** of a set of values.

In Paper 2 you meet them inside algorithms given as pseudocode or flowcharts, and you write them yourself, in pseudocode, a flowchart, or (in the 15-mark question) program code. Arrays, loops and IF statements belong to topic 8, Programming; tracing, test data and finding errors are in **Testing, trace tables and errors**, and validation checks in **Validation and verification**. This note uses them only as far as the standard methods need them.

Understand it

The same few jobs turn up in almost every program: add things up, count things, find the biggest or smallest, find an item, put items in order. Each has a standard method, a pattern of a few lines that always works. Learn the patterns and you can build most Paper 2 answers out of them.

The examples below use an array of five quiz marks:

Marks[1] ← 14 Marks[2] ← 9 Marks[3] ← 17 Marks[4] ← 11 Marks[5] ← 20

Totalling

A **total** is kept in a variable that starts at **0** before the loop. Inside the loop, each value is **added to the total**. After the loop, the total holds the sum.

```

Total ← 0
FOR Index ← 1 TO 5
    Total ← Total + Marks[Index]
NEXT Index
OUTPUT "Total mark: ", Total

```

For the five marks this outputs 71. The key line is `Total ← Total + Marks[Index]`: the new total is the old total **plus the value**. Writing `Total ← Total + Index` (adding the loop counter) or forgetting to set `Total ← 0` first are the classic errors.

Counting

A **count** also starts at **0**, but each time the thing you are counting happens, you **add 1**. Usually an **IF** decides whether this value counts.

```

Passed ← 0
FOR Index ← 1 TO 5
    IF Marks[Index] >= 12
        THEN
            Passed ← Passed + 1
        ENDIF
NEXT Index
OUTPUT Passed, " students scored 12 or more"

```

This outputs 3 (for 14, 17 and 20). So the two statements to tell apart are:

Statement	Method
<code>Passed ← Passed + 1</code>	counting (adds one)
<code>Total ← Total + Marks[Index]</code>	totalling (adds the value)

A loop counter such as `Count ← Count + 1` is also counting: it counts the times round the loop.

Finding the maximum and minimum

Keep the largest value seen so far in a variable, say `Highest`. Compare every value with it; when a value is **bigger**, it becomes the new `Highest`. For the minimum, keep `Lowest` and replace it when a value is **smaller**.

```

Highest ← Marks[1]
Lowest ← Marks[1]
TopPlace ← 1
FOR Index ← 2 TO 5
    IF Marks[Index] > Highest
        THEN
            Highest ← Marks[Index]
            TopPlace ← Index

```

```

ENDIF
IF Marks[Index] < Lowest
    THEN
        Lowest ← Marks[Index]
    ENDIF
NEXT Index

```

This gives `Highest = 20` at position 5, and `Lowest = 9`.

The starting value matters. There are two safe ways to start:

- with the **first value** (as above), then compare the rest; or
- with a value that any real value will beat: the maximum starts **lower** than any possible value (for marks, 0 or -1), the minimum starts **higher** than any possible value (for marks out of 20, say 999).

Starting `Lowest` at 0 is a common error: no mark is below 0, so `Lowest` stays 0 and the real minimum is never found.

Which item, not just the value. When you need the name of the winner, also store the **index** where the maximum was found (`TopPlace` above). With parallel arrays (a name in `Names [ 5 ]` and its mark in `Marks [ 5 ]`), `Names [ TopPlace ]` is the winner's name.

Ties. If two items can share the highest value, finding the index finds only one of them. To output all of them, find the maximum in one loop, then use a **second loop** to output every item equal to it.

Finding the average

The **average** (mean) is the total divided by how many values there are:

$$\text{average} = \frac{\text{total}}{\text{number of values}}$$

Work it out **after** the loop, once the total and count are complete. For the marks, $71 \div 5 = 14.2$.

To round, use Cambridge's pseudocode function `ROUND(value, places)`: `OUTPUT ROUND(Total / 5, 1)` outputs the average to 1 decimal place. When the number of values isn't fixed, divide by the **count** you kept, not by the size of the array.

Linear search

A **linear search** looks at the items one at a time, from the first, comparing each with the **target**. It stops when it finds the target, or when it reaches the end of the array without finding it.

Target = 31

	1	2	3	4	5	6
Codes	48	15	62	31	77	20
	▲	▲	▲	▲		
	no	no	no	yes	never checked	

4 comparisons, output 4. Searching for 50 checks all 6, output -1.

```

OUTPUT "Enter the mark to find"
INPUT Target
Found ← FALSE
Index ← 1
WHILE Found = FALSE AND Index ≤ 5 DO
    IF Marks[Index] = Target
        THEN
            Found ← TRUE
        ELSE
            Index ← Index + 1
        ENDIF
    ENDWHILE
IF Found = TRUE
    THEN
        OUTPUT "Found at position ", Index
    ELSE
        OUTPUT "Not found"
    ENDIF

```

For target 17 it outputs "Found at position 3" after 3 comparisons; for 15 it checks all 5 and outputs "Not found". The parts to get right:

- the **index starts at 1** (the first item) and goes up by 1 each time;
- the loop has **two ways to stop**: found, **or** past the last item. So the condition to keep going joins them with **AND**: not found yet **and** still inside the array. Without the end test, a missing item makes the index run past the end of the array: the program crashes or loops for ever;
- a **flag** (Found) or a position variable set to -1 at the start (-1 means "not found yet", since no real position is -1) tells you afterwards whether the item was found;
- the "not found" message is output **once, after the loop**, not every time an item doesn't match.

A linear search works on unsorted data. It can stop at the first match, as here, or carry on to the end to **find every match** (count them, or copy each one to a results array): then use a **FOR** loop over the whole array and no flag.

Bubble sort

A **bubble sort** puts an array in order. It goes through the array comparing each item with the **next** one, and **swaps** them if they are the wrong way round. One trip through the array is a **pass**. After the first pass the largest value (for ascending order) has "bubbled" to the end; after each further pass, one more value is in its final place. Passes repeat until a pass makes **no swaps**.

	[1]	[2]	[3]	[4]	[5]	
Start	7	2	9	4	1	
After pass 1	2	7	4	1	9	4 comparisons
After pass 2	2	4	1	7	9	3 comparisons
After pass 3	2	1	4	7	9	2 comparisons
After pass 4	1	2	4	7	9	1 comparison

```

Last ← 5
REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO Last - 1
        IF Marks[Index] > Marks[Index + 1]
            THEN
                Temp ← Marks[Index]
                Marks[Index] ← Marks[Index + 1]
                Marks[Index + 1] ← Temp
                Swapped ← TRUE
            ENDIF
    NEXT Index
    Last ← Last - 1
UNTIL Swapped = FALSE OR Last = 1

```

How each part works:

- **Compare neighbours:** Marks[Index] with Marks[Index + 1]. The inner loop goes only to Last - 1, because the last item has no neighbour after it. With n items, the first pass makes $n - 1$ comparisons.
- **Swap using a temporary variable:** three assignments. Temp keeps a copy of the first item before it is overwritten; without it, both items end up equal.
- **The order:** > in the comparison gives **ascending** order (smallest first); < gives **descending** order (largest first). For strings, > compares alphabetically, so the same code sorts names A to Z.

- **The flag Swapped** is set to FALSE at the start of each pass and to TRUE whenever a swap is made. If a whole pass makes no swaps, the array is in order and the sort stops early.
- **The shrinking pass:** $\text{Last} \leftarrow \text{Last} - 1$ after each pass, because the end item of each pass is already in place and doesn't need checking again.

The last two are what make this bubble sort **efficient**: fewer passes, and fewer comparisons in each pass. At most $n - 1$ passes are ever needed.

Parallel arrays. When names are in `Names [ ]` and their scores in `Scores [ ]` at the same index, a sort by score must **swap both arrays at the same time**, using the same index, so that each name stays with its score. With a 2D array, swap every column of the two rows.

Putting methods together

Most real questions use several methods in one loop. The pattern is always:

1. **Before the loop:** set every total and count to 0, and the maximum or minimum to its starting value.
2. **Inside the loop:** input or read the value; add it to the total; if it meets the condition, add 1 to the count; compare it with the maximum and minimum.
3. **After the loop:** work out the average; output the results with messages.

When a total or count is needed **for each** of several things (each student, each day), reset it to 0 at the start of the **outer** loop, not once at the very beginning.

Key facts and formulas

There are no formulas given in the exam; learn these.

Fact	Status
Totalling: $\text{Total} \leftarrow 0$ before the loop; $\text{Total} \leftarrow \text{Total} + \text{Value}$ inside it	Learn it
Counting: $\text{Count} \leftarrow 0$ before the loop; $\text{Count} \leftarrow \text{Count} + 1$ inside it (usually inside an IF)	Learn it
Maximum: start with the first value or a very low value; replace when $\text{Value} > \text{Highest}$	Learn it
Minimum: start with the first value or a very high value; replace when $\text{Value} < \text{Lowest}$	Learn it
$\text{average} = \frac{\text{total}}{\text{number of values}}$, worked out after the loop	Learn it
<code>ROUND(Value, Places)</code> rounds to that many decimal places	Learn it
Linear search: check each item in turn from the first; stop when found or past the last item	Learn it
Bubble sort: compare neighbours, swap with a temporary variable, repeat passes until no swaps	Learn it
n items: the first pass makes $n - 1$ comparisons; at most $n - 1$ passes	Learn it
$>$ in the comparison gives ascending order; $<$ gives descending order	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (36 questions, 2021–2025), the standard methods came up in these types. 12 of the 36 questions mix more than one type, so the counts add up to 53.

Question type	Questions
Write a full 15-mark program that uses the standard methods	14
Find and correct errors in an algorithm	7
Identify or name the standard method used	7
Trace an algorithm that uses a standard method	6
Write a short algorithm for one standard method	5
Explain how a search or sort works, why it is efficient, or its limits	4
Extend an algorithm to find a maximum or minimum, or add validation	4
Complete a partly written algorithm	3
Output an average rounded to a number of decimal places	3

1. Write a full 15-mark program (14 questions)

The last question of Paper 2. A scenario gives named arrays (often a 1D array of names and a 2D array of scores, at the same index), and a bulleted list of requirements. It is marked in bands: up to 9 marks (AO2) for using the right techniques and data structures to cover **every** requirement, and up to 6 (AO3) for comments, meaningful names and a logical, accurate solution.

1. **Underline** every array and variable name given. Use them exactly. Don't declare them if the question says you needn't.
2. **Turn each bullet into a section** of code, in order, with a comment above it (`// validate each score`).
3. **Input with validation**: a prompt, then a `REPEAT ... UNTIL` loop (or `WHILE`) that asks again until the value is in range, not a single `IF` (that gives only one retry). Nest it inside the loops that go through the items: `FOR` each competitor, `FOR` each round.
4. **Build each result from the standard methods**: totals, counts, maximum and minimum with their index, averages after the loop, a bubble sort that swaps all the parallel arrays together, or a linear search.
5. **Reset per-item totals** (and each item's highest and lowest) inside the outer loop.
6. **Output** each result with a message. Output names, not just numbers, when the question asks "which" or "who".
7. **Comment** every section and use meaningful names (`RoundScore`, not `x`).

Variations you will meet:

- **Drop the highest and lowest**: total all the values, find the highest and lowest in the same loop, then
 $\text{Overall} \leftarrow \text{Total} - \text{Highest} - \text{Lowest}.$

- **Bands** (qualify / reserve / fail): IF ... THEN ... ELSE IF ..., each branch outputting a message and adding 1 to its own counter.
- **Ties for the top**: find the maximum first, then a second loop outputs every name whose value equals it.
- **Sort, then take the top few**: bubble sort (descending for "best is highest", ascending for times) swapping names too, then output items 1 to 3.
- **Search a 2D array and copy matches**: loop through every row, and for each match copy its columns into the next free row of a results array; output "not found" if none.
- **A menu**: a REPEAT ... UNTIL Choice = 3 loop around the whole program, with the choice validated.

2. Find and correct errors (7 questions)

You are given numbered lines and told how many errors there are. Give the **line number** and the **whole corrected line** for each.

1. Read the description above the code: it tells you what each variable should do.
2. Check the declarations: a name stored as REAL, or a loop counter as STRING or CHAR, is wrong.
3. Check the starting values: totals and counts start at 0; a maximum low, a minimum high; a flag the right way round.
4. Check each standard-method line: `Total ← Total + Value` (not `+ Count`, not `*`); the maximum compared with `Highest` (not with `Total`); the right array index.
5. Check the loops: FOR ends with NEXT, REPEAT with UNTIL, WHILE with ENDWHILE, IF with ENDIF (not ENDCASE); bubble sort's inner loop goes to $n - 1$; AND and OR in the loop condition.
6. Check the swap: the third line must write Temp back into the **second** item.
7. Check the outputs: the right variable, and an average actually divided.

In a flowchart, look for a **logic error that lets invalid data through**: for example a decision that should reject negative values is missing (or tests the wrong way), so a negative value is accepted and added to the total. Say which box is wrong, and fix it with a decision straight after the input that sends a bad value back to the input (with an error message) before it can be totalled or counted.

3. Identify or name the standard method (7 questions)

1. Dry-run a few steps if you need to, then name the method: **linear search**, **bubble sort**, **totalling**, **counting**, **finding the maximum / minimum / average**.
2. For a sort, add the **order**: "a bubble sort into descending order" (the order often earns its own mark). Look at the comparison and which way the swap goes.
3. For "give the line numbers", point to the one line that does it: `+ 1` is counting, `+ Value` is totalling, `/` is the average.

Don't answer "a flowchart" or "an array": those are how the method is written, not the method.

4. Trace an algorithm that uses a standard method (6 questions)

1. One column per variable, one per array cell, and one for output. Write a value only when it **changes**.
2. For a **bubble sort**: work comparison by comparison. At each swap, write the new values of the two cells, the temporary variable and the flag. At the end of a pass, the index goes back to 1 and the flag back to FALSE.
3. For a **linear search**: the output is the position of the first match, or the "not found" value (such as -1) when every item has been checked.
4. For **totalling with a rogue value**: the rogue value (for example -1) stops the loop and is **not** added or counted.

5. Write a short algorithm for one standard method (5 questions)

Each mark point is one of: set up (total or count to 0, index to 1), a correct loop with its end condition, the update line, the condition (for counting or search), and the output **after** the loop.

- **Total with a rogue value**: input before the loop; `WHILE Value <> rogue` add it and input the next value inside the loop; output after.
- **Count values over a limit**: the same shape, with `IF Value > limit THEN Count ← Count + 1`.
- **Linear search**: flag or position set first, a loop that stops when found or past the end, output of the position or "not found" after the loop.
- **Bubble sort**: the pattern in "Understand it", with the right comparison for the order asked.
- **As a flowchart**: terminators, a prompt and input, process boxes for the total and counter, a decision for the loop test, output after the loop.

6. Explain how a search or sort works, why it's efficient, or its limits (4 questions)

- **Why a bubble sort is efficient**: a flag records whether any swap was made in a pass, and the sort stops as soon as a pass makes none; and the inner loop gets one shorter each pass, because the end items are already sorted. Both cut the number of comparisons.
- **Limits of a linear search on a key such as a first name**: the key may not be unique, so a search that stops at the first match can give the wrong person's data; if the key is missing there may be no output, or the search may run off the end. Improve it with a more specific key (surname, ID number), by searching the whole array for every match, and with a "not found" message.
- **How the index is used**: the index starts at the first position, the item at that index is compared with the target, and the index is increased by 1 until a match or the end.

7. Extend an algorithm to find a maximum or minimum, or add validation (4 questions)

You describe the change; you don't rewrite the algorithm. One mark per point, each saying **what** and **where**:

1. **Declare** a new variable (for example `Tallest`) at the start.
2. **Initialise** it before the loop: a maximum to a low value such as 0, a minimum to a high value (or to the first input).

3. **Inside the loop, after the input**, add an **IF** comparing the input with it.
4. If the input is bigger (smaller), **replace** the variable with it.
5. **After the loop**, output it.

For **validation** of an input between two limits: after the input, a **WHILE** loop that runs while the value is out of range, outputs an error message and inputs again; or a **REPEAT** loop around the prompt and input, **UNTIL** the value is in range.

8. Complete a partly written algorithm (3 questions)

1. Read what the algorithm is for, then see which standard-method line is missing from each gap: the starting value, the update ($+ 1$ or $+ A[C]$), the condition, or the loop's end (**NEXT C**, **ENDWHILE**).
2. When an array ends with a 0 "no more data" marker and the index starts at 1, the index is **one past the last value** when the loop stops, so the number of values is $C - 1$, and the average is $Total / (C - 1)$.
3. Output each result with a message.
4. **Suggest meaningful identifiers** for single-letter names: one that says what each holds, from how it is used. For example, an array **A** of numbers $\rightarrow$ **Values**; **C** used as the array index $\rightarrow$ **Index**; **W** that has each value added to it $\rightarrow$ **Total**; **T** that has 1 added $\rightarrow$ **Counter**; **X** set to a total divided by a count $\rightarrow$ **Average**. Usually one mark for the array and up to two for the variables.

9. Output an average rounded to a number of places (3 questions)

One statement: `OUTPUT ROUND( Total / 500, 1 )`. One mark for the correct average (total divided by the right number), one for **ROUND** with the right number of places, and the statement must include **OUTPUT**.

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

An online shop keeps the number of units in stock of each of its 50 products in the array `stock[1:50]`. A product needs reordering when fewer than 10 units are left.

Write pseudocode that finds and outputs, with messages:

- the product number with the fewest units in stock, and that number of units;
- how many products need reordering;

- the total number of units in stock, and the mean number of units per product, rounded to the nearest whole number. **[7]**

```

Fewest ← 100000                                // higher than any real stock level
FewestItem ← 0
Reorder ← 0
Units ← 0
FOR Item ← 1 TO 50
    Units ← Units + Stock[Item]
    IF Stock[Item] < 10
        THEN
            Reorder ← Reorder + 1
    ENDIF
    IF Stock[Item] < Fewest
        THEN
            Fewest ← Stock[Item]
            FewestItem ← Item
    ENDIF
NEXT Item
OUTPUT "Product ", FewestItem, " has the fewest units: ", Fewest
OUTPUT "Products to reorder: ", Reorder
OUTPUT "Units in stock: ", Units
OUTPUT "Mean units per product: ", ROUND(Units / 50, 0)

```

- Total, count, minimum and its product number all given starting values before the loop (the minimum high).

B1

- A loop through all 50 products. **B1**
- Totalling: `Units ← Units + Stock[Item]`. **B1**
- Counting inside an `IF ... < 10`. **B1**
- The minimum replaced when a stock level is smaller, **and** its product number stored. **B1**
- The mean worked out after the loop and rounded with `ROUND(..., 0)`. **B1**
- All the outputs with messages. **B1**

Dry run, pretending the shop had only 6 products (loop to 6, divide by 6) with stock levels 34, 8, 51, 3, 17, 9: product 4 has the fewest (3 units); 3 products need reordering (8, 3 and 9); there are 122 units, so the mean is $122 \div 6 = 20.33 \dots$, output as 20.

Example 2

```

01 Last ← 4
02 REPEAT
03     Swapped ← FALSE
04     FOR Index ← 1 TO Last - 1
05         IF Names[Index] > Names[Index + 1]

```

```

06         THEN
07             Temp ← Names[ Index ]
08             Names[ Index ] ← Names[ Index + 1 ]
09             Names[ Index + 1 ] ← Temp
10             Swapped ← TRUE
11         ENDIF
12     NEXT Index
13     Last ← Last - 1
14 UNTIL Swapped = FALSE OR Last = 1

```

Names[1 : 4] holds "Kofi", "Ana", "Lena", "Dev".

(a) Write the contents of the array after each pass. **[3]**

(b) State the total number of comparisons made in line 05. **[1]**

(c) State the change needed to sort the names into reverse alphabetical order. **[1]**

(d) The array starts as "Ana", "Dev", "Kofi", "Lena". State how many comparisons the algorithm makes, and explain why. **[2]**

(a)

- Pass 1: "Kofi" > "Ana", swap; "Kofi" < "Lena", no swap; "Lena" > "Dev", swap. Ana, Kofi, Dev, Lena **B1**
- Pass 2 (to Last - 1 = 2): Ana, Kofi, no swap; "Kofi" > "Dev", swap. Ana, Dev, Kofi, Lena **B1**
- Pass 3 (to 1): Ana, Dev, no swap. Ana, Dev, Kofi, Lena; no swaps, so the sort stops **B1**

(b) $3 + 2 + 1 = 6$ comparisons. **B1**

(c) In line 05, change > to <. **B1**

(d) 3 comparisons **B1**: the first pass compares each neighbouring pair once, makes no swaps, so Swapped stays FALSE and the UNTIL condition ends the sort after one pass **B1**.

Example 3

This algorithm should search the array Codes [1 : 200] for a product code that is input, and output its position or a message if it isn't there. It contains four errors.

```

01 DECLARE Codes : ARRAY[1:200] OF STRING
02 DECLARE Index : INTEGER
03 DECLARE Found : BOOLEAN
04 DECLARE Wanted : STRING
05 OUTPUT "Enter the product code"
06 INPUT Wanted
07 Found ← TRUE
08 Index ← 0
09 WHILE Found = FALSE OR Index <= 200 DO

```

```

10     IF Codes[Index] = Wanted
11     THEN
12         Found ← TRUE
13     ELSE
14         Index ← Index + 1
15     ENDIF
16 ENDWHILE
17 IF Found = TRUE
18 THEN
19     OUTPUT "Found at position ", Wanted
20 ELSE
21     OUTPUT "Not found"
22 ENDIF

```

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

(b) Some codes appear more than once. Explain how you would change the corrected algorithm so that it outputs every position where the code is found, and how many times it was found. **[4]**

(a)

- Line 07: Found ← FALSE (nothing has been found yet). **B1**
- Line 08: Index ← 1 (the array starts at 1, so Codes [0] doesn't exist). **B1**
- Line 09: WHILE Found = FALSE AND Index <= 200 DO (keep going only while both are true). **B1**
- Line 19: OUTPUT "Found at position ", Index (the position, not the code). **B1**

With these corrections, and Codes holding "TB-12", "RK-40", "TB-77", "RK-40", "LM-03" in its first five places, searching for "RK-40" outputs "Found at position 2".

(b) Any four:

- Declare a counter Matches and set it to 0 before the loop. **B1**
- Replace the WHILE loop with FOR Index ← 1 TO 200, so the whole array is searched instead of stopping at the first match (the flag is no longer needed). **B1**
- Inside the loop, when Codes [Index] = Wanted, output Index **B1** and add 1 to Matches. **B1**
- After the loop, if Matches = 0 output "Not found", otherwise output Matches with a message. **B1**

For the data above, "RK-40" now gives positions 2 and 4, found 2 times.

Example 4

A game keeps its ten best scores in the array HighScores [1 : 10], in no particular order. Scores are whole numbers from 0 to 5000.

Write a program that:

- inputs the score from a new game, validating that it is from 0 to 5000 inclusive
- finds the lowest score in the table; if the new score is higher, it replaces the lowest one, otherwise a message says the score is not high enough
- sorts the table into descending order
- if the new score went into the table, searches the sorted table for it and outputs its rank (its position)
- outputs the sorted table, numbered, and the mean of the ten scores rounded to 1 decimal place.

You must use pseudocode or program code and add comments. You don't need to declare the array or variables. All inputs and outputs must have suitable messages. **[15]**

```
// R1: input and validate the new score
REPEAT
    OUTPUT "Enter your score (0 to 5000)"
    INPUT NewScore
    IF NewScore < 0 OR NewScore > 5000
        THEN
            OUTPUT "The score must be from 0 to 5000, please try again"
        ENDIF
UNTIL NewScore >= 0 AND NewScore <= 5000

// R2: find the lowest score and where it is (minimum with its index)
Lowest ← HighScores[1]
LowPlace ← 1
FOR Index ← 2 TO 10
    IF HighScores[Index] < Lowest
        THEN
            Lowest ← HighScores[Index]
            LowPlace ← Index
        ENDIF
NEXT Index
IF NewScore > Lowest
    THEN
        HighScores[LowPlace] ← NewScore           // the new score replaces the lowest
        OnTable ← TRUE
    ELSE
        OUTPUT "Sorry, your score is not high enough for the table"
        OnTable ← FALSE
ENDIF

// R3: bubble sort into descending order
Last ← 10
REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO Last - 1
```

```

    IF HighScores[Index] < HighScores[Index + 1]
        THEN
            Temp ← HighScores[Index]
            HighScores[Index] ← HighScores[Index + 1]
            HighScores[Index + 1] ← Temp
            Swapped ← TRUE
        ENDIF
    NEXT Index
    Last ← Last - 1
UNTIL Swapped = FALSE OR Last = 1

// R4: linear search of the sorted table for the new score
IF OnTable = TRUE
    THEN
        Rank ← 1
        WHILE HighScores[Rank] <> NewScore AND Rank < 10 DO
            Rank ← Rank + 1
        ENDWHILE
        OUTPUT "Well done, you are ranked ", Rank
    ENDIF

// R5: output the table and total it for the mean
Total ← 0
FOR Index ← 1 TO 10
    OUTPUT Index, ". ", HighScores[Index]
    Total ← Total + HighScores[Index]
NEXT Index
OUTPUT "Mean of the top ten scores: ", ROUND(Total / 10, 1)

```

How it earns the marks:

- **AO2 (up to 9):** every requirement is met with a suitable technique: a validation loop for the input; the minimum found together with its index, so the right element can be replaced; a bubble sort with a flag and a shrinking pass, using < for descending order; a linear search that stops at the match; totalling, and the mean after the loop with ROUND. It uses the array named in the question.
- **AO3 (up to 6):** comments on every section, meaningful names (LowPlace, OnTable, Rank), and a logical order: input, replace, sort, search, output.

A dry run with the table 3200, 4100, 950, 2780, 4100, 1500, 3620, 2050, 1200, 3999 and a new score of 2600: the lowest score is 950 (position 3), so 2600 replaces it. Sorted, the table is 4100, 4100, 3999, 3620, 3200, 2780, 2600, 2050, 1500, 1200, so the new score is ranked 7. The ten scores total 29 149, so the mean is output as 2914.9. A new score of 900 is lower than 950, so it gets the "not high enough" message and the table is output unchanged, apart from being sorted.

Common mistakes

- **Wrong starting value for a minimum.** Examiners often saw the smallest value set to 0 before the loop (`Small ← 0`), so it never changed. Start a minimum high (or at the first value), a maximum low.
- **Not initialising totals and counters.** Forgetting `Total ← 0`, or setting it inside the loop so it resets every time, loses marks in traces and in the 15-mark program. Reset per-item totals only in the outer loop.
- **Calling counting "totalling".** `Number ← Number + 1` is counting; totalling adds a value. Candidates regularly mixed them up.
- **Only one winner.** In questions with a highest score, most candidates stored and output one name even when several competitors shared it. Use a second loop to output every item equal to the maximum.
- **Outputting the number instead of the name.** When asked which team or person had the highest score, output the name (using the stored index), not the score.
- **Not naming the method.** When a flowchart is a linear search, "a flowchart" earns nothing. For a sort, give the order as well as "bubble sort": only candidates who said ascending (or descending) got both marks.
- **Missing the end-of-array test.** A search with no end test runs for ever, or crashes, when the item isn't there.
- **Leaving out OUTPUT or ROUND.** A rounded average must be output in the same statement, and averages the question asks to round must actually be rounded; both were often missed.
- **Validating with a single IF.** That allows only one re-entry; use a loop that repeats until the value is valid.
- **Answering in the wrong form.** Where pseudocode is asked for, program code or a flowchart earns nothing (the 15-mark question is the only one where program code is allowed).

How to get the marks

- **Name the method exactly:** linear search, bubble sort, totalling, counting, finding the maximum / minimum / average. For a sort, say ascending or descending.
- **Short algorithms are marked point by point:** starting values before the loop, a correct loop and end condition, the update line, the condition, the output after the loop. Write each clearly on its own line.
- **Precision:** `Value > 100`, not "value is over 100"; "between 1 and 500 inclusive" is `Value >= 1 AND Value <= 500`.
- **Error questions:** one mark per error, for the line number with the fully corrected statement. Write the whole line, not "change + to -".
- **"Explain how you would change ...":** one mark per point, each saying what and where: declare, initialise before the loop, compare inside the loop after the input, replace, output after the loop. You needn't rewrite the algorithm.
- **The 15-mark program:** marked as a whole in bands. Cover **every** bullet with the right technique, use the given array names, put a message on every input and output, comment each section, and use meaningful names. Minor syntax slips are ignored; missing requirements are not.
- **Trace tables:** fill a column only when the value changes, and copy outputs exactly, messages included.

Quick check

1. Name the standard method in each fragment: (i) `IF Speed[i] > Top THEN Top ← Speed[i];` (ii) `Laps ← Laps + 1;` (iii) `Bill ← Bill + Price[i];` (iv) a loop comparing `Seat[i] = Wanted` for each `i` until a match is found.
2. The array `Pins[1:6]` holds 4021, 7750, 1308, 7750, 9902, 2566. A linear search stops at the first match. For the target 7750, what position is output and how many comparisons are made? What happens for the target 5000?
3. A bubble sort into **descending** order swaps neighbours when the first is smaller than the second. Write the array 8, 3, 9, 1, 6 after the first pass.
4. An algorithm inputs four temperatures and sets `Lowest ← 0` before the loop, then replaces `Lowest` whenever a temperature is smaller. For 18, 22, 15, 19 it outputs 0. Explain why, and give a correct starting value.
5. `Total` is 47 and `Count` is 6. Write one statement to output the average to 2 decimal places, and state what it outputs.

Answers

1. (i) Finding the maximum; (ii) counting; (iii) totalling; (iv) linear search.
2. Position 2, after 2 comparisons (it stops at the first 7750, so position 4 is never reached). For 5000, all 6 items are compared, none matches, and it outputs "not found" (or -1).
3. 8 and 3: no swap; 3 and 9: swap → 8, 9, 3, 1, 6; 3 and 1: no swap; 1 and 6: swap. After the first pass: **8, 9, 3, 6, 1**. The smallest value, 1, is now at the end. (Fully sorted it would be 9, 8, 6, 3, 1.)
4. No temperature is below 0, so the comparison is never true and `Lowest` keeps its starting value. Start it higher than any possible temperature (for example `Lowest ← 1000`), or at the first temperature input; it then outputs 15.
5. OUTPUT "Average: ", `ROUND(Total / Count, 2)`, which outputs 7.83 ($47 \div 6 = 7.833\dots$).

Past questions to try

- **0478/22/F/M/25 Q6**: correct five errors in a totalling and maximum algorithm, round the average, add a minimum.
- **0478/22/F/M/24 Q3**: write a linear search and a bubble sort in pseudocode.
- **0478/21/M/J/23 Q7**: trace a search flowchart, name the method and explain what goes wrong with a missing item.
- **0478/21/O/N/24 Q11**: a 15-mark program with verification, a bubble sort of parallel arrays and counting.