

Arrays

Read online at pastopic.com/cambridge/computer-science-0478/notes/arrays

Last checked 30 September 2026

Read first: [Programming concepts](#)

© 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. **Declare and use one-dimensional (1D) and two-dimensional (2D) arrays:** write the `DECLARE` line with the array's name, its index range (or ranges) and the data type of its elements, then store values in it and read them back.
2. **Understand why and how arrays are used**, including using a **variable as the index**, so that one line of code can reach a different element each time it runs.
3. **Write values into an array and read values out of it with loops**, including **nested loops** for 2D arrays. The first index of an array can be **0** or **1**; the declaration says which.

Loops, totalling, counting, validation and string handling are in **Programming concepts**; linear search and bubble sort are in **Standard methods of solution**. This note shows how they work on arrays.

Understand it

Why arrays?

Suppose a program needs the number of pages a student read on each of six days. With ordinary variables you would need `Day1`, `Day2`, ... `Day6`, and six separate `INPUT` lines. For a whole year that is 365 variables, and a loop can't help, because each variable has a different name.

An **array** solves this. Picture a row of numbered boxes: each box is an **element**, and its number is its **index**. The whole row goes by one name, it has a fixed number of boxes, and every box holds the same data type. You pick an element by writing the index in square brackets after the name: `Pages [ 4 ]` is the element at index 4.

Because the index can be a **variable**, a loop can visit every element with a single line of code. The loop changes the variable, and the same line then refers to a different element each pass.

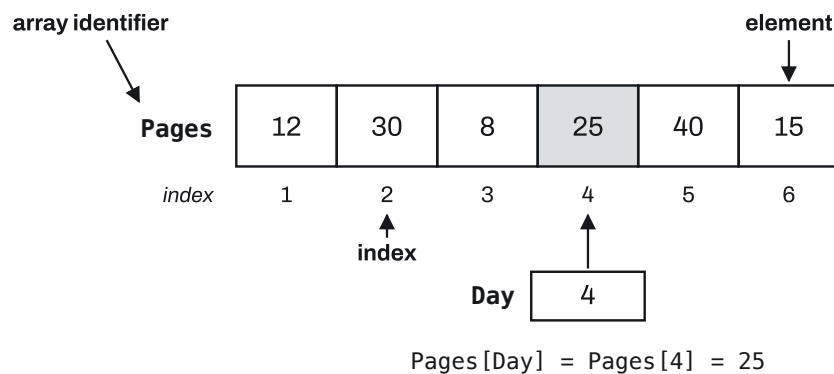
Declaring a 1D array

```
DECLARE Pages : ARRAY[1:6] OF INTEGER
```

Read this as: "Pages is an array with indexes 1 to 6, and every element is an integer". The parts are:

- the **identifier** (Pages): a meaningful name, with no spaces or punctuation;
- the **lower and upper bound** of the index (1 : 6), which fix the number of elements: $6 - 1 + 1 = 6$;
- the **data type** of every element (INTEGER, REAL, STRING, CHAR or BOOLEAN). Names go in STRING arrays, prices in REAL arrays, and so on. An array cannot mix types.

The first index does not have to be 1. `DECLARE Pages : ARRAY[ 0 : 5 ] OF INTEGER` also has six elements, numbered 0 to 5. Whatever the first index is, use the declared range everywhere: a loop over this version runs `FOR Day ← 0 TO 5`. Python lists always start at index 0.



Writing and reading elements

An element is used exactly like a variable:

```
Pages[4] ← 25           // store 25 in the element at index 4
OUTPUT Pages[4]         // read it back
INPUT Pages[1]          // input straight into an element
Pages[2] ← Pages[1] + 10 // read one element, store into another
```

The index can be a variable or a whole expression. If `Day` holds 2, then `Pages[ Day ]` is `Pages[ 2 ]` and `Pages[ Day + 1 ]` is `Pages[ 3 ]`. The expression is worked out first, then that element is used.

Two things to keep apart:

- the **index** says **where** (a position: 1, 2, 3, ...);
- the **element** is **what is stored there** (8, 25, "Maze", ...).

An index outside the declared range, such as `Pages[ 7 ]` or `Pages[ 0 ]` for `ARRAY[ 1 : 6 ]`, doesn't exist. Using one is a run-time error, so check loop limits against the bounds.

Loops and arrays

To fill or process every element, loop the index through the whole range. A `FOR` loop fits because the number of elements is known.

```

FOR Day ← 1 TO 6
    OUTPUT "Pages read on day ", Day
    INPUT Pages[Day]
NEXT Day

```

Once the data is in the array, other loops can go back over it as many times as you like, which ordinary input into one variable can't do. With the values in the diagram (12, 30, 8, 25, 40, 15):

```

Total ← 0
Most ← Pages[1]
MostDay ← 1
FOR Day ← 1 TO 6
    Total ← Total + Pages[Day]
    IF Pages[Day] > Most
        THEN
            Most ← Pages[Day]
            MostDay ← Day
        ENDIF
NEXT Day
OUTPUT "Total pages: ", Total
OUTPUT "Most pages were read on day ", MostDay

```

This outputs a total of 130 and day 5. Notice that the loop keeps the **index** of the largest element (MostDay) as well as its value. The index is what lets you find anything else about that element later.

Other patterns you will use:

- **Setting every element to a start value** (initialising): `FOR Day ← 1 TO 6` with `Pages[Day] ← 0` inside.
- **A loop that stops early**: `WHILE` or `REPEAT ... UNTIL` with the index increased inside the loop, for example to stop at the first empty element (" ") or the first 0 that marks the end of the data. You must add 1 to the index yourself.
- **The input value as the index** (a frequency table): to count star ratings from 1 to 5, use `RatingCount : ARRAY[1:5] OF INTEGER`, set all five to 0, then for each rating `Stars` do `RatingCount[Stars] ← RatingCount[Stars] + 1`. The ratings 5, 3, 5, 4, 1, 5, 4 leave the counts 1, 0, 1, 2, 3 for 1 to 5 stars.

2D arrays

A **2D array** is a table: rows and columns, every element the same type. It has two index ranges.

```

DECLARE Tickets : ARRAY[1:4, 1:3] OF INTEGER

```

This has $4 \times 3 = 12$ elements. By convention the first index is the **row** and the second is the **column**, so `Tickets[3, 2]` is the element in row 3, column 2.

Exam questions very often pair a 1D array with a 2D one: here `Film[1:4]` holds four film titles, and `Tickets[1:4, 1:3]` holds the tickets sold at each film's three showings. They are **parallel arrays**: the **same row index means the same film** in both. That is how you get from a number in `Tickets` back to the film's title.

Film		Tickets			
		row	column (showing)		
			1	2	3
1	Comet	1	45	60	82
2	Harbour	2	30	25	51
3	Maze	3	70	88	95
4	Echo	4	12	40	33

`Film[3] = "Maze"`
`Tickets[3, 2] = 88`

Row 3 is the same film in both arrays

Nested loops over a 2D array

To visit every element of a 2D array, put one loop inside another. The outer loop picks the row; the inner loop runs through every column of that row before the outer loop moves on.

```

FOR Row ← 1 TO 4
    RowTotal ← 0                                // reset for each film
    FOR Showing ← 1 TO 3
        RowTotal ← RowTotal + Tickets[Row, Showing]
    NEXT Showing
    OUTPUT Film[Row], " sold ", RowTotal, " tickets"
NEXT Row

```

The inner line runs $4 \times 3 = 12$ times. The film totals come out as 187, 106, 253 and 85, so Maze sold the most. Where each line sits matters:

- `RowTotal ← 0` goes **inside the outer loop but before the inner loop**, so each film starts from 0;
- the `OUTPUT` goes **after the inner loop**, once the row is finished;
- `Film[Row]` uses the **outer** index, because it belongs to the whole row.

To work down a **column** instead (for example the total for showing 2 across all films: $60 + 25 + 88 + 40 = 213$), swap the loops so the column is on the outside. Inputting a 2D array works the same way, with `INPUT Tickets[Row, Showing]` as the inner line.

Key facts and formulas

There is no formula sheet for Paper 2; learn these.

Fact	Status
1D: DECLARE Name : ARRAY[Lower:Upper] OF TYPE	Learn it
2D: DECLARE Name : ARRAY[Lower1:Upper1, Lower2:Upper2] OF TYPE; first index row, second column	Learn it
Number of elements: Upper – Lower + 1 in each dimension; a 2D array has rows × columns	Learn it
Every element has the same data type; the first index may be 0 or 1	Learn it
Name[i] and Name[r, c] are used like variables; the index can be a variable or an expression	Learn it
An index outside the declared bounds is an error	Learn it
Fill or process every element with FOR over the whole index range; nested FOR loops for 2D (outer row, inner column)	Learn it
Parallel arrays: the same index in each array belongs to the same item	Learn it

How to do it

54 Paper 2 questions from 2021–2025 are tagged with this topic. One of them, **0478/22/F/M/22 Q2**, only counts values and uses no array, so the table counts the other 53. 26 of the 53 mix more than one type, so the counts add up to 87.

Question type	Questions
Write a full program using given 1D and 2D arrays (15 marks)	20
Pre-release task algorithm using arrays (syllabus before 2023)	14
Describe an array: name, data type, size, sample data, use	11
Explain or write changes to an array algorithm	10
Find and correct errors in an array algorithm	6
Output array contents or results with messages	6
Complete an incomplete array algorithm	5
Describe what an array algorithm does	4

Question type	Questions
Complete a trace table for an algorithm using an array	3
Declare and initialise an array with a loop	2
Write or explain a short search or minimum loop over a 1D array	2
Suggest meaningful identifiers for an array and its variables	2
State or describe what an array is	2

1. Write a full program using given 1D and 2D arrays (20 questions)

The last question on every Paper 2 since 2023. A scenario names the arrays (usually a 1D array of names and a 2D array of numbers for the same people or things) and lists the requirements as bullets. It is marked as a whole in bands: up to 9 for choosing suitable techniques and data structures and covering every requirement, and up to 6 for comments, meaningful names, and a solution that is logical and works. Pseudocode, Python, Java or Visual Basic are all accepted.

- 1. Read the scenario to the end.** Write down every array name exactly as given, what each index means ("first index is the member, second is the week") and the sizes. Check whether you are told to declare things; if it says you don't need to, don't spend time on it.
- 2. Turn each bullet into a block of code**, in the order the bullets come, with a comment above each block saying which requirement it meets.
- 3. Input into the 2D array with nested loops:** outer loop over the people or items, inner loop over the columns. Show whose data is being entered in the prompt, for example `OUTPUT "Week ", Week, "`
`for ", Member[M].`
- 4. Validate with a loop, not an IF:** `WHILE Value < 0 OR Value > 30 DO ... error message, INPUT again ... ENDWHILE`. An IF only gives one second chance.
- 5. Process each row after its inner loop.** Reset the row's total, highest and lowest at the start of each row; highest and lowest can start at the row's first element, or at a value outside the valid range.
- 6. Count into categories** with `IF ... ELSE IF` on each row's result, using counters set to 0 before the loop, and output the counts after the loop.
- 7. Output names, not just numbers**, by using the row index in the 1D names array.
- 8. Tidy up:** every input has a prompt, every output a message, and every name you invented says what it holds.

Techniques that keep coming up:

- **Store a result per row** in a third 1D array (`Total[M] ← ...`) when the question names one, so it can be sorted or searched later.
- **Highest with ties:** first loop to find the highest value, second loop to output **every** name whose value equals it. Outputting only one name loses marks when several share the top score.
- **Discard the highest and lowest:** total the row, find its highest and lowest, and subtract both from the total.

- **Sort parallel arrays** with a bubble sort that compares the values but swaps the matching elements of **every** parallel array, each through its own temporary variable (see Example 4). Descending order means swap when the first is **smaller** than the next.
- **Linear search that copies matches**: go through every row, and whenever the key matches, copy that row into the next free row of a results array, adding 1 to a results counter. Afterwards output the results, or a "not found" message if the counter is still 0.
- **First free element**: start at index 1 and move on while the element is not empty (" ") and the index is within the bounds; store the new data there.
- **A menu**: a REPEAT ... UNTIL Choice = 3 loop around a validated choice and an IF (or CASE) for each option.
- **Random whole numbers** into an array: $\text{Roll}[I] \leftarrow \text{ROUND}(\text{RANDOM}() * 5, 0) + 1$ gives 1 to 6 (see **Procedures, functions and library routines**).

2. Pre-release task algorithm using arrays (14 questions)

Papers before 2023 had a 20-mark section on a task set in advance; the current syllabus has no pre-release material. These parts asked you to explain, step by step, how your program looped through its arrays to count, total or pick out values. The skill still pays: when asked to **explain** an array algorithm, describe each step in order and say which array and index each step uses ("the index goes from 1 to 30; each element is compared with the highest so far ..."), rather than a general summary.

3. Describe an array: name, data type, size, sample data, use (11 questions)

Give each feature asked for, one per mark, and make them fit together:

- a **meaningful identifier**, with no spaces or punctuation: BusFare;
- the **data type** of its elements, named exactly: REAL, not "number";
- the **size**: ARRAY [1 : 20], 20 elements;
- **sample data**: 2.50, 3.75;
- its **use**: "holds the fare for each of the 20 routes, route number = index".

A full answer can be written as one declaration plus a sentence: `DECLARE BusFare : ARRAY[ 1:20 ] OF REAL`, storing the fare for each route, such as 2.50.

4. Explain or write changes to an array algorithm (10 questions)

Each step you describe earns a mark only if it is precise: say **what** is added, **where** in the algorithm, and **what it does**. Any code you write must also be explained.

1. **Also find the largest (or smallest) value**. Declare a variable for it; before the loop give it a start value that any real value will beat (below the lowest possible for a largest, above the highest possible for a smallest, or the first element); inside the loop, after each value arrives, compare it with the variable and replace the

variable when it beats it; after the loop, output it. To report **who** it belongs to, also store the index when you replace the value.

2. **Keep every input instead of only a running total.** A running total holds just the sum, so each number is lost as soon as the next one arrives, and nothing can be listed, sorted or checked afterwards. Give each number a home of its own: an array with one element per input, filled inside the same loop, for example `INPUT Reading[N]` inside `FOR N ← 1 TO 40`. Because the numbers survive the loop, a later loop can go back and print `Reading[1]` to `Reading[40]`.

3. **Stop the array overflowing.** Make the loop unable to run once the last element is filled:

```
Free ← 1
More ← 'Y'
WHILE More = 'Y' AND Free <= 80 DO
    OUTPUT "Guest name: "
    INPUT Guest[Free]
    Free ← Free + 1
    OUTPUT "Another guest? (Y/N)"
    INPUT More
ENDWHILE
```

The loop only runs while an empty element is left, so index 81 is never used, whatever the user answers.

4. **Work for any number of items.** Input how many there are first, and use that variable as the loop's end value (no larger than the array's size).
5. **Improve a search.** Ask what the search key can't promise. It may match nothing, so the user needs a message for that case after the loop. It may match several records (a surname, a town), and a search that ends at its first hit can then report the wrong one: either carry on to the last element and show every hit, or make the key unique (a membership number) or pair it with a second field.

5. Find and correct errors in an array algorithm (6 questions)

Usually 4 or 5 errors, one mark each for the line number with a corrected line.

1. **Read the description first** and write down what each variable and array should hold.
2. **Check the declarations:** each array's data type matches what it stores, its size matches the description, and every variable used is declared.
3. **Check every index:** is it the loop variable of the loop it is in? Is it the right index in the right position for a 2D array (`[Row, Column]`, not swapped, and not a fixed number where a variable is needed)? Is it inside the bounds (`FOR I ← 0 TO 50` for `ARRAY[1:50]` fails)?
4. **Check the start values:** totals and counters usually start at 0, a largest-so-far below any real value, an index at the first element.
5. **Check each calculation and comparison:** totals add the **element**, not the index; counters add 1, not the value; a comparison uses the variable the description names.

6. **Check the loop structure:** keywords match (REPEAT ... UNTIL, WHILE ... DO ... ENDWHILE, FOR ... NEXT), a WHILE loop increases its index, and resets sit in the right loop.
7. **Check the inputs, outputs and flags:** data that the user types needs INPUT, not OUTPUT; an output shows one element or variable (not the whole array name) with the value the description asks for; a Boolean flag that should end a loop is set to the value that actually ends it.
8. Write each correction as the **full corrected line** with its line number.

Example 2 below works through a 2D version.

6. Output array contents or results with messages (6 questions)

- **Every element:** a FOR loop over the range, outputting `Name [ I ]` (and the matching element of each parallel array) with a message.
- **Only the elements in use:** if a counter recorded how many were stored, loop from 1 to that count; if not, use a condition-controlled loop that ends at the first unused element and can never go past the last index. For a 2D array, output every column of a row before moving to the next row.
- **Results worked out from the array:** output each value with a message saying what it is. Rounded values use `ROUND: OUTPUT "Mean: ", ROUND(Total / Count, 2)`. When a loop ran on past the last value (a sentinel loop), the number of values is one less than the final index; see type 7.

7. Complete an incomplete array algorithm (5 questions)

Fill each gap so the algorithm does what the description says. Read the lines around each gap: they tell you which variable is the index, which is the total and which is the counter.

- **Start of the index:** the first element, for example `Index ← 1`, before a WHILE loop.
- **Totalling:** `Sum ← Sum + Data[Index]` (the element, not the index).
- **Sentinel end:** when a 0 or " " marks the end of the data, the loop test is `WHILE Data[Index] <> 0`. When it stops, `Index` points at the sentinel, so the number of real values is `Index - 1`, and their mean is `Sum / (Index - 1)`.
- **Frequency table:** the value itself is the index: `Count[Value] ← Count[Value] + 1`.
- **Loop ends:** `NEXT Index` to close a FOR; the upper bound as the end value.
- **Filling a 2D array inside a procedure:** it is the nested input loop from "Understand it", with the number of rows arriving as a parameter instead of being fixed. Writing and calling procedures is in **Procedures, functions and library routines**.

8. Describe what an array algorithm does (4 questions)

Trace one or two values to see what the algorithm achieves overall, and say that first. Then go through it in the order it runs, one step per mark, naming the arrays, variables and messages it actually uses; include what happens to a value that fails a test and what makes the algorithm stop. Naming a standard method ("a linear search", "a bubble sort into ascending order") can earn a mark when that is what it is.

9. Complete a trace table for an algorithm using an array (3 questions)

1. Give each array element its own column (`Box[ 1 ]`, `Box[ 2 ]`, ...) and copy in any starting values from the first row.
2. Work through one step at a time. Write a value in a column **only when it changes**; leave unchanged elements blank.
3. For each index expression, work out the index first, then read or write **that** element.
4. Put outputs in the output column exactly as the algorithm produces them, including messages.
5. Watch the ends: the last pass of a loop, and a value that fails an early check (it may never reach the array).

Example 1 below is a traced algorithm.

10. Declare and initialise an array with a loop (2 questions)

```
DECLARE Badge : ARRAY[1:8] OF STRING
DECLARE B : INTEGER
FOR B ← 1 TO 8
    Badge[B] ← ""
NEXT B
```

Why start every element at "" (strings) or 0 (numbers)? Until something is stored, an element's contents can't be relied on. An empty string gives later code a clear sign that an element is free, so a search or a list can tell where the data stops; a 0 lets an element be used as a total or a count and added to straight away. FOR suits the job, since the array's size fixes the number of passes before the loop begins. A 2D array needs a second FOR loop inside, for the columns.

11. Write or explain a short search or minimum loop over a 1D array (2 questions)

- **Smallest (or largest) element:** start with the first element (or a value outside the range), loop through the rest comparing and replacing, and output **once, after** the loop.
- **Searching using the index:** start the index at the first element, compare that element with the target, and add 1 to the index each time it doesn't match; stop when it matches or the index passes the last element, then say whether it was found (and at which index).

12. Suggest meaningful identifiers for an array and its variables (2 questions)

Work out what each single letter holds by reading the algorithm, then give a name that says so: an array holding readings might be `Readings`; a loop index `Index`; a running sum `Total`; a count `Count`; a mean `Average`. No spaces or punctuation.

13. State or describe what an array is (2 questions)

Start with positions: each value sits in a numbered slot, its **index**, and you reach it as `Goals[i]`. Then say the whole group shares **one name**, and that every slot must hold the **same kind of data** (all integers, say). If the question wants an example, show one being set up, for instance `Goals` holding 38 whole numbers: `DECLARE Goals : ARRAY[1:38] OF INTEGER`. A 2D array works the same way with a row number and a column number.

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

This algorithm uses the array `Tally`, whose indexes start at 0.

```

01 DECLARE Tally : ARRAY[0:4] OF INTEGER
02 DECLARE L : INTEGER
03 DECLARE P : INTEGER
04 FOR L ← 0 TO 4
05     Tally[L] ← L * 3
06 NEXT L
07 P ← 1
08 WHILE Tally[P] < 10 DO
09     Tally[P + 1] ← Tally[P + 1] + Tally[P]
10     P ← P + 1
11 ENDWHILE
12 OUTPUT P, " ", Tally[P]
```

- (a) State how many elements `Tally` has. **[1]**
- (b) Complete a trace table for the algorithm, with columns `L`, `P`, `Tally[0]` to `Tally[4]` and `OUTPUT`. **[5]**
- (c) Line 08 is changed to `WHILE Tally[P] < 100 DO`. Explain what goes wrong. **[2]**

(a) 5, indexes 0 to 4: $4 - 0 + 1 = 5$. **B1**

(b)

L	P	Tally[0]	Tally[1]	Tally[2]	Tally[3]	Tally[4]	OUTPUT
0		0					
1			3				
2				6			
3					9		
4						12	
	1			9			
	2				18		
	3						3 18

- L from 0 to 4. **B1**
- Tally[0] to Tally[4] set to 0, 3, 6, 9, 12. **B1**
- P goes 1, 2, 3. **B1**
- Tally[2] becomes $6 + 3 = 9$ and Tally[3] becomes $9 + 9 = 18$; Tally[4] is never changed. **B1**
- The loop stops because $18 \geq 10$, and the output is 3 18. **B1**

(c) The loop would keep going: Tally[4] becomes $12 + 18 = 30$ and P becomes 4. Since $30 < 100$, line 09 then tries to use Tally[5] **B1**, which is outside the bounds 0 to 4, so the program fails with an error **B1**. The condition should also check that P stays below 4.

Example 2

A fruit farm has 8 fields. The 1D array Field[1:8] already holds their names. This algorithm should input the kilograms of fruit picked in each field on each of 5 days into the 2D array Kilos[] (first index the field, second the day), then output each field's name and total, and finally output how many fields had a total of more than 500 kg.

```

01 DECLARE Kilos : ARRAY[1:8, 1:5] OF REAL
02 DECLARE FieldTotal : REAL
03 DECLARE Big : INTEGER
04 DECLARE F : INTEGER
05 DECLARE D : INTEGER
06 Big ← 0
07 FOR F ← 1 TO 8
08     FOR D ← 0 TO 5
09         INPUT Kilos[D, F]
10     NEXT D
11 NEXT F
12 FOR F ← 1 TO 8
13     FOR D ← 1 TO 5

```

```

14      FieldTotal ← 0
15      FieldTotal ← FieldTotal + Kilos[F, D]
16  NEXT D
17  OUTPUT Field[F], " picked ", FieldTotal, " kg"
18  IF FieldTotal < 500
19      THEN
20          Big ← Big + FieldTotal
21  ENDIF
22 NEXT F
23 OUTPUT Big, " fields picked more than 500 kg"

```

(a) Identify the line numbers of five errors and give a correction for each. **[5]**

(b) Explain how to change the corrected algorithm so that it also outputs the name of the field with the largest total. **[4]**

(a) One mark per line with a working correction:

- Line 08: the days are indexed 1 to 5, so `D ← 0` reaches an element that doesn't exist. `FOR D ← 1 TO 5` **B1**
- Line 09: the field is the first index. `INPUT Kilos[F, D]` **B1**
- Line 14: the total is reset on every day, so it only ever holds one day. Move `FieldTotal ← 0` to just before line 13, inside the field loop **B1**
- Line 18: the comparison is the wrong way round. `IF FieldTotal > 500` **B1**
- Line 20: Big counts fields, so it goes up by 1. `Big ← Big + 1` **B1**

(b)

- At the start, declare `Largest : REAL` and `LargestField : INTEGER`, and set `Largest ← 0`, below any real total. **B1**
- After line 16, where the field's total is complete, compare it: `IF FieldTotal > Largest.` **B1**
- If it is larger, store it with `Largest ← FieldTotal` and keep the field's index with `LargestField ← F.` **B1**
- After the field loop ends, `OUTPUT "Largest harvest: ", Field[LargestField]`, using the stored index to reach the name. **B1**

Example 3

A lost property office records items in the 1D array `Item[1:100]`. Unused elements hold the empty string "", and items are always stored from `Item[1]` onwards with no gaps. This incomplete algorithm stores a new item in the first empty element.

```

01 Slot ← 1
02 WHILE Slot <= 100 AND ..... DO
03     Slot ← .....

```

```

04 ENDWHILE
05 IF Slot > 100
06     THEN
07         OUTPUT "The log is full"
08     ELSE
09         OUTPUT "Describe the item"
10         INPUT .....
11 ENDIF

```

(a) Complete the algorithm. **[3]**

(b) The array holds "umbrella", "scarf" and "phone" in its first three elements, and "" in the rest. State the value of `Slot` when the `WHILE` loop ends. **[1]**

(c) When the `WHILE` loop ends (before line 05 runs), `Slot` is the index of the first empty element, or 101 if the log is full. Write pseudocode that could be placed at that point and uses `Slot` to output the items already stored, newest first, each with its number, for example 2: scarf. **[3]**

(a) Line 02: `Item[Slot] <> ""` **B1**. Line 03: `Slot + 1` **B1**. Line 10: `INPUT Item[Slot]` **B1**.

(b) 4 **B1** (`Item[4]` is the first empty element).

(c)

The newest item is the one just before the first empty element, so the loop counts down from `Slot - 1` to 1:

```

FOR Number ← Slot - 1 TO 1 STEP -1
    OUTPUT Number, ": ", Item[Number]
NEXT Number

```

- The loop starts at `Slot - 1`, the last stored item. **B1**
- It counts down to 1, using `STEP -1`. **B1**
- Each item is output with its number. **B1**

With the data in (b), `Slot` is 4 when the loop ends, and this outputs 3: phone, 2: scarf and 1: umbrella. If the log was full, `Slot - 1` is 100, so the loop still starts at the last element.

Example 4

A bike-hire scheme has 12 docking stations. The 1D array `Station[1:12]` holds the station names. The 2D array `Hires[1:12, 1:7]` is to hold the number of bikes hired at each station on each day of a week; its first index is the same as the station's index in `Station[ ]`. The 1D array `WeekTotal[1:12]` is to hold each station's total for the week.

Write a program that:

- inputs the number of hires for each station on each day, accepting only whole numbers from 0 to 99;

- calculates each station's total for the week and stores it in `WeekTotal[ ]`;
- sorts `Station[ ]` and `WeekTotal[ ]` into descending order of total;
- outputs the names and totals of the three busiest stations;
- counts and outputs how many stations had at least one day with no hires.

You must use pseudocode or program code and add comments to explain how your code works. You do not need to declare any arrays or variables. All inputs and outputs must contain suitable messages. **[15]**

```
// Requirement 1: input and validate the hires for every station and day
FOR S ← 1 TO 12
    FOR Day ← 1 TO 7
        OUTPUT "Hires at ", Station[S], " on day ", Day, ": "
        INPUT Count
        WHILE Count < 0 OR Count > 99 OR Count <> ROUND(Count, 0) DO
            OUTPUT "Enter a whole number from 0 to 99: "
            INPUT Count
        ENDWHILE
        Hires[S, Day] ← Count
    NEXT Day
NEXT S

// Requirement 2 and 5: total each station, and note any day with no hires
QuietStations ← 0
FOR S ← 1 TO 12
    WeekTotal[S] ← 0 // start each station from 0
    ZeroDay ← FALSE
    FOR Day ← 1 TO 7
        WeekTotal[S] ← WeekTotal[S] + Hires[S, Day]
        IF Hires[S, Day] = 0
            THEN
                ZeroDay ← TRUE
            ENDIF
    NEXT Day
    IF ZeroDay = TRUE // count the station once only
        THEN
            QuietStations ← QuietStations + 1
        ENDIF
NEXT S

// Requirement 3: bubble sort, largest total first,
// swapping the names with the totals so they stay matched
Swapped ← TRUE
WHILE Swapped = TRUE DO
    Swapped ← FALSE
```

```

FOR S ← 1 TO 11
    IF WeekTotal[S] < WeekTotal[S + 1]
        THEN
            TempTotal ← WeekTotal[S]
            WeekTotal[S] ← WeekTotal[S + 1]
            WeekTotal[S + 1] ← TempTotal
            TempName ← Station[S]
            Station[S] ← Station[S + 1]
            Station[S + 1] ← TempName
            Swapped ← TRUE
        ENDIF
    NEXT S
ENDWHILE

// Requirement 4: after sorting, the busiest three are in positions 1 to 3
OUTPUT "The three busiest stations this week:"
FOR S ← 1 TO 3
    OUTPUT S, ". ", Station[S], " with ", WeekTotal[S], " hires"
NEXT S

// Requirement 5: output the count
OUTPUT QuietStations, " stations had at least one day with no hires"

```

How this earns the marks:

- **Techniques and data structures (up to 9):** nested loops over stations and days; a validation loop with an error message; totals stored in the named array and reset for each station; a flag so a station with several zero days is counted once; a bubble sort that compares totals and swaps names and totals together; output of the top three from the sorted arrays; the count output with a message. All three named arrays are used as described.
- **Comments, names, logic, accuracy (up to 6):** a comment for every requirement and for the less obvious lines, meaningful names (`WeekTotal`, `QuietStations`, `TempName`), and blocks in a logical order that together do every task.

The sort only moves `Station[ ]` and `WeekTotal[ ]`. That is fine here, because `Hires[ ]` isn't used after the totals are worked out; if it were, its rows would have to be swapped as well.

Common mistakes

- **Storing values in a single variable when an array is needed.** Each new input overwrites the last one, so nothing is left to sort, search or output later. When the question names an array, put the data in it.
- **Ignoring the arrays the question gives.** Rename them, or invent your own instead, and the data-structure credit goes. Copy the names exactly, and use the same name all the way through.

- **Names with spaces or punctuation**, such as `Test Score[ ]` or `score 's`. An identifier must be something a program could actually use: `TestScore`.
- **Vague data types**. "Number" or "text" is not a data type; write `INTEGER`, `REAL`, `STRING`, `CHAR` or `BOOLEAN`.
- **Mixing up the index and the element**. `Total ← Total + I` adds the position, not the value stored there. Use `Total ← Total + Data[I]`.
- **Swapping the two indexes of a 2D array**, or writing a fixed number where the loop variable belongs. Keep `[Row, Column]` in the order the question defines, and use the loop variables.
- **Loop limits that miss the bounds**: starting at 0 for an array declared from 1, or running one element past the end. Compare every `FOR` line with the declaration.
- **Forgetting to reset per row**. A total, highest or lowest set once before the outer loop carries over from one person to the next. Reset it inside the outer loop, before the inner one.
- **Not setting totals and counters to 0** before the loop that uses them. Examiners see this every year.
- **Validating with an IF**, which only rechecks once. Use a loop that keeps asking until the value is acceptable.
- **Outputting only one winner**. When several people share the top value, the question expects all of them. Find the top value first, then output every match.
- **Outputting a number when the question asks for the name**. Use the index to look the name up in the parallel 1D array.
- **Sorting one parallel array and not the others**, so names end up next to other people's scores. Swap the matching elements of every parallel array together.
- **A WHILE loop that never moves on**. Unlike a `FOR` loop, it needs its own `Index ← Index + 1` inside.
- **Filling unchanged cells in a trace table**, or copying values down every row. Only write a value when it changes.
- **Declaring everything when the question says you needn't**. It wastes time on the longest question; just use the given names.

How to get the marks

- **Line-by-line marks** (errors, gaps, short algorithms): each mark is for one precise thing, such as the right index, the right start value or the right loop end. Write the complete line, not a description of it.
- **"Identify the line number ... and give a correction"**: one mark per error needs **both** the line number and a corrected line that works. An explanation alone is not a correction.
- **"Explain how you would change ..."**: say what you add, where it goes (before the loop, inside it after the input, after it) and what it does. Include short statements if they help, but explain them.
- **"Describe the processes"**: steps in the order they happen, each tied to the given algorithm, not a general comment on what arrays do.
- **"Describe an array"**: name, data type, size, sample data and use are separate marks; give each one that is asked for.

- **Trace tables:** each column is usually marked on its own, so an error in one column doesn't cost the rest. Keep going carefully.
- **"Write the pseudocode" means pseudocode**, with `←`, `NEXT`, `ENDWHILE` and the other keywords; program code there can lose marks. The 15-mark program alone accepts program code.
- **The 15-mark program:** every requirement needs a matching block of code, and every array named in the question must hold the data it is meant to. Comment each block and choose names that say what they hold; small syntax slips are ignored, but missing parts are not.
- **Outputs with messages:** when asked for "suitable messages", each value needs its own label.

Quick check

1. The array `Locker` is declared as `DECLARE Locker : ARRAY[1:6, 1:15] OF CHAR`. (i) How many elements does it have? (ii) Write a statement that stores 'F' in the element in row 4, column 11.
2. The array `Val[1:5]` holds 4, 1, 5, 2, 3 (in index order), and `K` holds 2. State the value of (i) `Val[K + 3]` (ii) `Val[Val[K]]` (iii) `Val[Val[K] + Val[4]]`.
3. The array `Depth[1:50]` holds 50 river depth readings. Write pseudocode that outputs how many of the readings are greater than the mean of all 50.
4. The 2D array `Score[1:3, 1:4]` holds these values:

	Column 1	Column 2	Column 3	Column 4
Row 1	7	2	9	4
Row 2	5	8	1	6
Row 3	3	3	10	2

Write pseudocode that stores the total of each column in `ColTotal[1:4]`, and state the value of `ColTotal[3]`.

5. `Product[1:30]` and `Price[1:30]` are parallel arrays. A program sorts `Price[ ]` into ascending order but doesn't change `Product[ ]`. Explain what goes wrong.

Answers

1. (i) $6 \times 15 = 90$. (ii) `Locker[4, 11] ← 'F'`.

2. (i) `Val[5] = 3`. (ii) `Val[K] = Val[2] = 1`, so `Val[1] = 4`. (iii) $1 + 2 = 3$, so `Val[3] = 5`.

3. The mean isn't known until every reading has been added, so two loops are needed:

```
Total ← 0
FOR R ← 1 TO 50
    Total ← Total + Depth[R]
NEXT R
Mean ← Total / 50
Above ← 0
FOR R ← 1 TO 50
    IF Depth[R] > Mean
        THEN
            Above ← Above + 1
    ENDIF
NEXT R
OUTPUT Above, " readings are above the mean"
```

4. The column goes in the outer loop, so each total is finished before the next begins:

```
FOR C ← 1 TO 4
    ColTotal[C] ← 0
    FOR R ← 1 TO 3
        ColTotal[C] ← ColTotal[C] + Score[R, C]
    NEXT R
NEXT C

ColTotal[3] = 9 + 1 + 10 = 20 (the four totals are 15, 13, 20 and 12).
```

5. The arrays are linked only by index: `Price[i]` is the price of `Product[i]`. After sorting only `Price[ ]`, the prices have moved but the product names haven't, so most products are now next to another product's price. Every swap in `Price[ ]` must also swap the same two elements of `Product[ ]`.

Past questions to try

- **0478/23/M/J/24 Q4**: errors in a 2D array input routine, output that stops at the first empty row, and stopping the array overflowing.
- **0478/21/O/N/24 Q6**: complete an algorithm that counts the zeros in an array and totals the rest, then name its variables.
- **0478/22/O/N/25 Q8**: a 15-mark program that counts frequencies into a 2D array and sorts it.

- **0478/23/M/J/25 Q8**: a 15-mark program with a menu, the first free row and a search that copies matches into a results array.