Exercise walk-through

10.2.1 Bubble sort, array bounds and the standard array routines

A-Level Computer Science

You must be able to

- use the technical terms associated with arrays, including **index 索引**, **lower bound 下界** and **upper bound 上界**
- select a suitable data structure, a 1-D or a 2-D array, for a given task
- write pseudocode for 1-D and 2-D arrays
- write pseudocode to process array data: sort using a bubble sort, search using a linear search

1 The method

Method — Bounds, and how many elements there are

In `DECLARE Data : ARRAY[1:120] OF REAL`, the **lower bound** is 1 and the **upper bound** is 120. The count is

$$\text{upper bound} - \text{lower bound} + 1$$

so this array holds **120** elements. The "+1" is what catches people out: `ARRAY[0:8]` holds **9**, not 8, and `ARRAY[5:14]` holds **10**.
For a 2-D array, multiply the two counts: `ARRAY[1:150, 1:2] OF STRING` is $150 \times 2 = \mathbf{300}$ elements, and `ARRAY[0:4, 0:5]` is $5 \times 6 = \mathbf{30}$.

- A declaration needs the **bounds and the data type**. Give both, or the mark is not there.

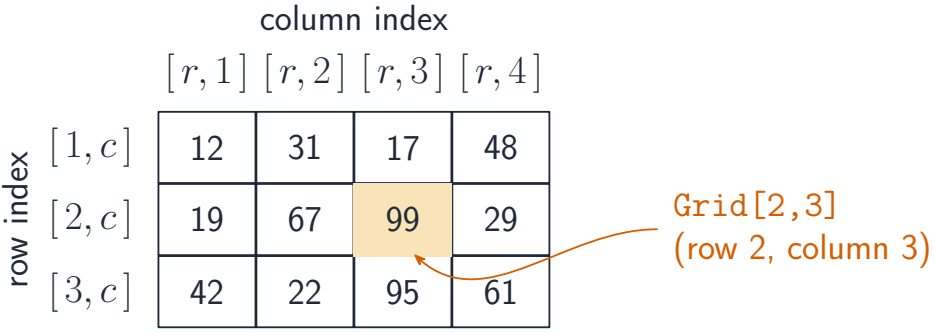
Method — 1-D or 2-D?

- Use **1-D** for a single sequence of values: the marks of 30 students, the readings from one sensor.
- Use **2-D** where the data has **two natural dimensions**: a grid, or rows against columns – a seating plan, a noughts-and-crosses board, monthly sales for each of several shops.
- The first index is the **row**, the second the **column**. Visit every cell with **nested** loops, the outer over rows and the inner over columns. To search one row only, fix the row index and loop over the column.

An array can also replace a chain of selection statements: `DaysInMonth[Month]` looks the answer up directly instead of twelve `IF` clauses, which is shorter to write, faster to read and easier to maintain.

Method — 1-D or 2-D?

An array can also replace a chain of selection statements: DaysInMonth[Month] looks the answer up directly...



A 2-D array (a table) with row and column indices

Method — The bubble sort, one pass at a time

A bubble sort compares **adjacent pairs** and swaps any that are out of order. After one pass, the largest value has reached the end – which is the fact the efficient version exploits.

Worked example. *Sort 7, 3, 9, 2.*

After pass	Array	Swaps in that pass
1	3, 7, 2, 9	2
2	3, 2, 7, 9	1
3	2, 3, 7, 9	1
4	2, 3, 7, 9	0 – so stop

Method — The bubble sort, one pass at a time

```
REPEAT
    Swapped <- FALSE
    FOR Index <- 1 TO Limit - 1
        IF Data[Index] > Data[Index + 1] THEN
            Temp <- Data[Index]
            Data[Index] <- Data[Index + 1]
            Data[Index + 1] <- Temp
            Swapped <- TRUE
        ENDIF
    NEXT Index
    Limit <- Limit - 1
UNTIL Swapped = FALSE
```

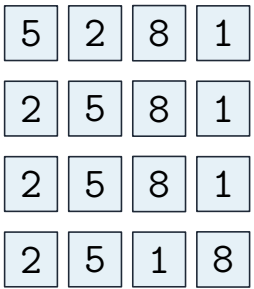
Method — The bubble sort, one pass at a time

The marks are given for four separate things, so make each one visible:

1. the **outer loop** that repeats until a pass makes no swap,
2. the `Swapped` **flag**, set inside the `IF`,
3. the **three-line swap** through a temporary variable – `Data[Index] <- Data[Index + 1]` on its own destroys a value,
4. the **shrinking limit**, because after each pass one more value is already in place.

Method — The bubble sort, one pass at a time

A bubble sort compares adjacent pairs and swaps any that are out of order.



Method — The same sort, in steps

When a question says *do not use pseudocode*, write it as steps:

1. Repeat steps 2 to 4 until a pass makes no swaps.
2. Compare each adjacent pair of elements in turn, from the start.
3. If a pair is out of order, swap the two values.
4. After each pass the largest unsorted value has reached the end, so the next pass can stop one place earlier.

Method — The routines that move data about

- **Position of the largest.** Set `Largest` to the first element and `Position` to 1; for each remaining element, if it is greater than `Largest`, store both the value and the index. Using a strict > keeps the **first** of two equal maxima; >= would keep the last.
- **Linear search returning a position.** Set `FoundAt <- -1` before the loop – a value that can never be a valid index – so after the loop `FoundAt = -1` means "not found". Leave the loop as soon as it matches.
- **Remove the element at index `k`.** Move every later element one place **towards the start**, so the gap closes, then mark the last element as unused. Removing index 2 from `4, 9, 2, 7, 5` gives `4, 2, 7, 5, unused`.

Method — The routines that move data about

- **Insert into a sorted array.** Find the first index whose element is **larger** than the new value, move that element and every later one one place **towards the end**, and store the new value in the gap. Inserting 6 into `2, 4, 8, 9`: the first larger element is 8 at index 3, so the result is `2, 4, 6, 8, 9`.
- For a remove, copy **forwards** through the array; for an insert, copy **backwards** from the end. Going the wrong way overwrites the values you have not moved yet.

2 Practice

Practice 2.1 ▪ 3 marks

State how many elements each of these arrays holds.

Practice 2.1 (a)

```
DECLARE Data : ARRAY[1:120] OF REAL
```

Solution 2.1 (a)

Method: Bounds, and how many elements there are — p.4

Worked steps

$120 - 1 + 1 = \mathbf{120}$ B1

Practice 2.1 (b)

```
DECLARE Flags : ARRAY[0:8] OF BOOLEAN
```

Solution 2.1 (b)

Worked steps

$8 - 0 + 1 = \mathbf{9}$ – not 8 **B1**

Practice 2.1 (c)

```
DECLARE Table : ARRAY[1:150, 1:2] OF STRING
```

Solution 2.1 (c)

Worked steps

$150 \times 2 = \mathbf{300}$ B1

Practice 2.2 ▪ 2 marks

An array is declared as `ARRAY[5:14] OF INTEGER`. State its **lower bound**, its **upper bound**, and the number of elements.

Solution 2.2

Method: Bounds, and how many elements there are — p.4

Worked steps

- lower bound **5**, upper bound **14** B1
- $14 - 5 + 1 = \mathbf{10}$ elements B1

Practice 2.3 ▪ 2 marks

For each task, state whether a **1-D** or a **2-D** array is the better choice, and give a reason.

Practice 2.3 (a)

the marks of 30 students in one test

Solution 2.3 (a)

Method: 1-D or 2-D? — p.5

Worked steps

1-D – the marks are a single sequence of values, one per student B1

Practice 2.3 (b)

a noughts-and-crosses board

Solution 2.3 (b)

Worked steps

2-D – the board has two natural dimensions, three rows by three columns B1

Practice 2.4 ▪ 2 marks

An array holds `7, 3, 9, 2`. Write the contents of the array after **one complete pass** of a bubble sort.

Solution 2.4

Method: The bubble sort, one pass at a time — p.7

Worked steps

- compare and swap each adjacent pair in turn: `7, 3` swap; `7, 9` in order; `9, 2` swap M1
- after one pass: `3, 7, 2, 9` A1
- the largest value, 9, has reached the end

Practice 2.5 ▪ 2 marks

Before a linear search, a programmer sets `FoundAt <- -1` rather than `0`. Explain why.

Solution 2.5

Method: The routines that move data about — p.12

Worked steps

- a valid index can be 0 in an array whose lower bound is 0, so 0 could be mistaken for a genuine result **B1**
- `-1` can never be a valid index, so after the loop `FoundAt = -1` unambiguously means “not found” **B1**

3 Exam-style

Exam-style 3.1 ▪ 5 marks

A 1-D array `Data[1:n]` holds `n` integers. Write pseudocode to sort it into ascending order using a **bubble sort**.

Solution 3.1

> Method: The bubble sort, one pass at a time — p.7

Worked steps

- an outer loop that repeats the passes **B1**
- an inner loop over the adjacent pairs **B1**
- the comparison `Data[Index] > Data[Index + 1]` **B1**
- the swap through a **temporary** variable, all three lines **B1**
- the loop ends correctly, leaving the array sorted **B1**

Solution 3.1

```
DECLARE Index, Pass, Temp : INTEGER
FOR Pass <- 1 TO n - 1
    FOR Index <- 1 TO n - 1
        IF Data[Index] > Data[Index + 1] THEN
            Temp <- Data[Index]
            Data[Index] <- Data[Index + 1]
            Data[Index + 1] <- Temp
        ENDIF
    NEXT Index
NEXT Pass
```

Exam-style 3.2 ▪ 5 marks

The sort in 3.1 always makes the same number of passes, whatever the data.

Exam-style 3.2 (a) ▪ 3 marks

Rewrite it so that it stops as soon as a pass makes no swap, and so that each pass examines one fewer element.

Solution 3.2 (a)

Worked steps

a Swapped flag, set FALSE at the start of each pass and TRUE inside the IF **M1**

- the outer loop becomes REPEAT ... UNTIL Swapped = FALSE **B1**
- a Limit that decreases by one after each pass, with the inner loop running to Limit - 1 **B1**

```
Limit <- n
REPEAT
    Swapped <- FALSE
    FOR Index <- 1 TO Limit - 1
        IF Data[Index] > Data[Index + 1] THEN
            Temp <- Data[Index]
            Data[Index] <- Data[Index + 1]
            Data[Index + 1] <- Temp
            Swapped <- TRUE
        ENDIF
    NEXT Index
    Limit <- Limit - 1
UNTIL Swapped = FALSE
```

Exam-style 3.2 (b) ▪ 2 marks

Explain what **each** of those two changes saves.

Solution 3.2 (b)

Worked steps

the **flag** stops the sort as soon as the array is in order, so an already-sorted array costs one pass instead of $n - 1$ B1

- the **shrinking limit** avoids re-examining the values at the end, which each pass has already put in their final place B1

Exam-style 3.3 ▪ 4 marks

Describe a bubble sort as a series of steps. Do **not** use pseudocode.

Solution 3.3

Method: The same sort, in steps — p.11

Worked steps

any four, in a sensible order:

- repeat the following until a pass makes no swaps B1
- compare each adjacent pair of elements in turn, starting at the beginning B1
- if a pair is out of order, swap the two values B1
- after each pass the largest unsorted value has reached the end, so the next pass can stop one place earlier B1
- no pseudocode keywords used (`FOR`, `IF`, `<-` are pseudocode, so they lose the style mark)

Exam-style 3.4 ▪ 5 marks

A 1-D array `Stock[1:100]` holds product codes as strings. An unused element holds `""`.
Write pseudocode for a procedure `Remove(Position)` that deletes the element at `Position` by moving every later element one place towards the start, and marks the last element as unused.

Solution 3.4

> Method: The routines that move data about — p.12

Worked steps

- header with the parameter, and the loop counter declared **B1**
- loop from `Position` to the second-from-last element **M1**
- `Stock[Index] <- Stock[Index + 1]` – copying **forwards**, so nothing is overwritten before it is moved **B1**
- set the last element to `""` after the loop **B1**
- `ENDPROCEDURE`, with everything closed **B1**

Solution 3.4

```
PROCEDURE Remove(Position : INTEGER)
    DECLARE Index : INTEGER
    FOR Index <- Position TO 99
        Stock[Index] <- Stock[Index + 1]
    NEXT Index
    Stock[100] <- ""
ENDPROCEDURE
```

Solution 3.4

- copying backwards here would fill the rest of the array with one repeated value