

10.2.1 Bubble sort, array bounds and the standard array routines

Name: _____ Class: _____ Date: _____ **Total: 30 marks**

KEY WORDS index 索引 • bubble sort 冒泡排序 • lower bound 下界 • upper bound 上界
• linear search 线性查找

Objective

Build the skills to answer the **array** 数组 questions that sheet 10.2 does not reach. That sheet declares 1-D and 2-D arrays and processes them with a **linear search** 线性查找 and a running maximum; this one covers the algorithm the syllabus names alongside it and that sheet never shows – the **bubble sort** 冒泡排序 – together with counting elements from the **bounds** 边界, choosing 1-D or 2-D for a task, and the standard routines that move data about inside an array.

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 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

■ 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 $150 \times 2 =$ **300** elements, and `ARRAY[0:4, 0:5]` is $5 \times 6 =$ **30**.

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

■ 1-D or 2-D?

		column index			
		[r, 1]	[r, 2]	[r, 3]	[r, 4]
row index	[1, c]	12	31	17	48
	[2, c]	19	67	99	29
	[3, c]	42	22	95	61

Grid[2,3]
(row 2, column 3)

- 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.

■ The bubble sort, one pass at a time

5	2	8	1	start
2	5	8	1	compare 5, 2 → swap
2	5	8	1	compare 5, 8 → already in order
2	5	1	8	compare 8, 1 → swap (8 reaches the end)

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

```

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

```

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.

■ 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.

■ 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**.
- **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

Now use the methods above.

2.1 State how many elements each of these arrays holds. [3]

- (a) DECLARE Data : ARRAY[1:120] OF REAL
- (b) DECLARE Flags : ARRAY[0:8] OF BOOLEAN
- (c) DECLARE Table : ARRAY[1:150, 1:2] OF STRING

2.2 An array is declared as `ARRAY[5:14] OF INTEGER`. State its **lower bound**, its **upper bound**, and the number of elements. [2]

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

- (a) the marks of 30 students in one test
- (b) a noughts-and-crosses board

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

2.5 Before a linear search, a programmer sets `FoundAt <- -1` rather than 0. Explain why. [2]

3 Exam-style questions

3.1 A 1-D array `Data[1:n]` holds `n` integers. Write pseudocode to sort it into ascending order using a **bubble sort**. [5]

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

(a) Rewrite it so that it stops as soon as a pass makes no swap, and so that each pass examines one fewer element. [3]

(b) Explain what **each** of those two changes saves. [2]

3.3 Describe a bubble sort as a series of steps. Do **not** use pseudocode. [4]

3.4 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. [5]

Solutions

Each answer shows the steps, then the **result**.

2.1

- (a) $120 - 1 + 1 = \mathbf{120}$
- (b) $8 - 0 + 1 = \mathbf{9}$ – not 8
- (c) $150 \times 2 = \mathbf{300}$

2.2

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

2.3

- (a) **1-D** – the marks are a single sequence of values, one per student
- (b) **2-D** – the board has two natural dimensions, three rows by three columns

2.4

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

2.5

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

3.1

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

```
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
```

3.2

- (a) a **Swapped** flag, set **FALSE** at the start of each pass and **TRUE** inside the IF
- the outer loop becomes `REPEAT ... UNTIL Swapped = FALSE`
- a **Limit** that decreases by one after each pass, with the inner loop running to **Limit - 1**

```
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
    ENDIF
  NEXT Index
  Limit <- Limit - 1
  UNTIL Swapped = FALSE
```

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

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

3.3 any four, in a sensible order:

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

3.4

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

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

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