

## 10.2.1 Bubble sort, array bounds and the standard array routines

Name: \_\_\_\_\_ Class: \_\_\_\_\_ Date: \_\_\_\_\_

Total: 30 marks

**KEY WORDS** index • bubble sort • lower bound • upper bound  
• linear search

### Objective

Extend sheet 10.2 with **bubble sort**, insertion and deletion. Revisit **bounds** and array choices before moving data inside an array.

**You must be able to:**

- use the technical terms associated with arrays, including **index**, **lower bound** and **upper bound**
- 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        |

For integer array `Data[1:n]`, where  $n \geq 1$ :

```
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
      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 outside this array's declared bounds – 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.

**Past-paper practice:** finish sheets 10.2 and 10.2.1 before attempting set 10.2.1.

## 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`

(a) 

|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

 ... 

|  |
|--|
|  |
|--|

 (c) 

|  |  |
|--|--|
|  |  |
|  |  |
|  |  |
|  |  |

(b) 

|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

 ... 

|  |
|--|
|  |
|--|

|  |  |
|--|--|
|  |  |
|  |  |
|  |  |
|  |  |

**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]

|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

 ... 

|  |
|--|
|  |
|--|

  
5   6   7            14

**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

(a) 

|  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

 (b) 

|   |   |   |
|---|---|---|
| × |   | ○ |
|   | ○ |   |
|   |   | × |

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

|   |   |   |   |
|---|---|---|---|
| 7 | 3 | 9 | 2 |
|---|---|---|---|

---

---

**2.5** Before a linear search of `Data[0:99]`, a programmer sets `FoundAt <- -1` rather than 0. Explain why. [2]

FoundAt 

|    |
|----|
| -1 |
|----|

|  |
|--|
|  |
|--|

|  |
|--|
|  |
|--|

|  |
|--|
|  |
|--|

 ... 

|  |
|--|
|  |
|--|

---

---

### 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]

Data 

|   |   |   |   |
|---|---|---|---|
| 5 | 1 | 4 | 2 |
|---|---|---|---|

 ... 

|   |
|---|
| 8 |
|---|

  
1 2 3 4 n

**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]

Data 

|   |   |   |   |
|---|---|---|---|
| 5 | 1 | 4 | 2 |
|---|---|---|---|

  
later 

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 4 | 5 |
|---|---|---|---|

↓ passes

(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]

|   |   |   |   |   |
|---|---|---|---|---|
| 6 | 2 | 8 | 1 | 5 |
|---|---|---|---|---|

---

---

---

---

---

**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]

