

Exercise walk-through

10.2.2

Arrays - combined exam practice

A-Level Computer Science

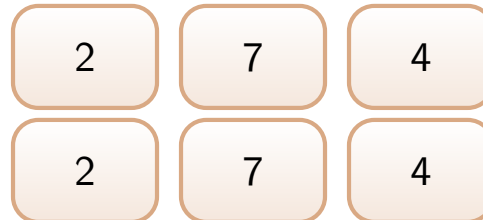
Arrays

An **array** holds many values of the **same** type under one name, reached by an **index**. A **record** is the opposite: different types, always belonging together.

1-D

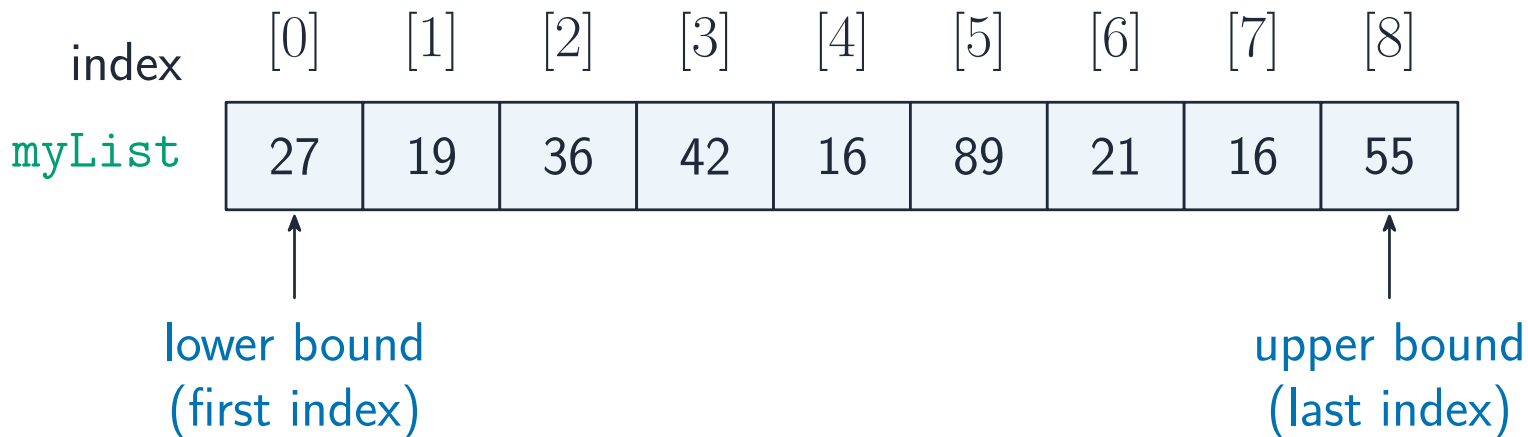


2-D – [row, column]



1-D arrays

One index. Marks [3] is the third element – but check the **lower bound**: Cambridge pseudocode usually starts at 1, most languages at 0.



Array vocabulary, and the standard sweep

index	the position of an element within the array
bounds	the lowest and highest valid indices – going outside them is an error
dimension	how many indices are needed: 1-D for a single sequence, 2-D for a table
a 2-D array	the first index is the row, the second the column; nested loops visit every cell
to traverse	visit every element in turn
the sweep pattern	to find a sum, count, maximum or minimum, set a running variable before the loop and update it inside

The running variable must be initialised **outside** the loop. Inside, it resets every pass – the single commonest bug in this topic.

2-D arrays: a table

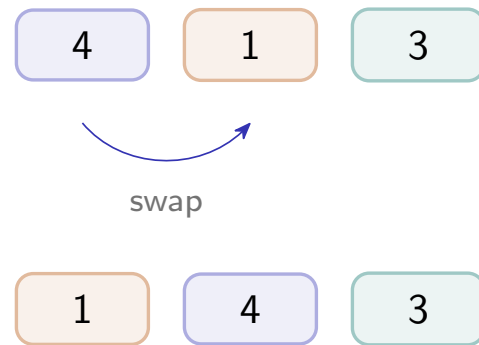
Two indices, **row first**: `Grid[2,5]` is row 2, column 5.

		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)

Searching and sorting

Linear search walks from the start until it finds the target – works on any list, $O(n)$. **Bubble sort** swaps adjacent pairs that are out of order, so the largest “bubbles” to the end each pass – $O(n^2)$.



Exercise 2.1 (a)

2.1 `Data[1:20]` holds 4, 9 in its first two elements. `Count` = 2; the remaining elements are unused. The next input is 9.

(a) In a strictly increasing sequence, should this input be stored? Explain. [2]

Exercise 2.1 (a)

2.1 `Data[1:20]` holds 4, 9 in its first two elements. `Count` = 2; the remaining elements are unused. The next input is 9.

- (a) In a strictly increasing sequence, should this input be stored? Explain. [2]
- (a) no; it equals the last stored value 9 and is not greater, so the sequence stops.

Exercise 2.1 (b)

2.1 `Data[1:20]` holds 4, 9 in its first two elements. `Count = 2`; the remaining elements are unused. The next input is 9.

(b) State the next free index **before** this input, and explain what must happen when `Count = 20`. [2]

Exercise 2.1 (b)

2.1 `Data[1:20]` holds 4, 9 in its first two elements. `Count = 2`; the remaining elements are unused. The next input is 9.

(b) State the next free index **before** this input, and explain what must happen when `Count = 20`. [2]

(b) index 3; at `Count = 20`, stop before another input, preventing an out-of-bounds write at index 21.

Exercise 2.2 (a)

2.2 Use the worked example's array `Valid[0:10]`.

(a) State the results of looking up scores 7 and 8.

[2]

Exercise 2.2 (a)

2.2 Use the worked example's array `Valid[0:10]`.

(a) State the results of looking up scores 7 and 8.

[2]

(a) `Valid[7] = TRUE; Valid[8] = FALSE.`

Exercise 2.2 (b)

2.2 Use the worked example's array `Valid[0:10]`.

(b) Explain why testing `Valid[11]` is an error rather than a way to return `FALSE`. [1]

Exercise 2.2 (b)

2.2 Use the worked example's array `Valid[0:10]`.

(b) Explain why testing `Valid[11]` is an error rather than a way to return `FALSE`. [1]

(b) index 11 is outside the declared range `0:10` and must not be accessed.

Exercise 2.3

2.3 A boundary is 50. State the lowest and highest marks within two marks of it. Explain why joining the two comparisons with **OR** would be wrong. [3]

Exercise 2.3

2.3 A boundary is 50. State the lowest and highest marks within two marks of it. Explain why joining the two comparisons with **OR** would be wrong. [3]

2.3

- minimum 48; maximum 52.
- both comparisons must hold for the same mark. With **OR**, even 40 passes because $40 \leq 52$; values outside the interval would be accepted.

Exercise 2.4

2.4 Global array `Scores[1:10]` contains integer marks. Write pseudocode to output their mean and the number of marks at least 50. Declare any local variables used. [5]

Exercise 2.4

2.4 Global array `Scores[1:10]` contains integer marks. Write pseudocode to output their mean and the number of marks at least 50. Declare any local variables used. [5]

2.4

- declarations and initial values; loop over ten values; total values; conditional count; output mean and count.

```
DECLARE Index, Total, Count : INTEGER
Total ← 0
Count ← 0
FOR Index ← 1 TO 10
    Total ← Total + Scores[Index]
    IF Scores[Index] >= 50 THEN
        Count ← Count + 1
    ENDIF
NEXT Index
OUTPUT "Mean: ", Total / 10, ", passes: ", Count
```

Exercise 2.5 (a)

2.5 Data is declared as `ARRAY[0:3] OF INTEGER` and contains 4, -1, 7, 7 in index order.

(a) State its lower bound, upper bound and number of elements. [2]

Exercise 2.5 (a)

2.5 Data is declared as `ARRAY[0:3] OF INTEGER` and contains 4, -1, 7, 7 in index order.

(a) State its lower bound, upper bound and number of elements. [2]

(a) lower bound 0, upper bound 3; $3 - 0 + 1 = 4$ elements.

Exercise 2.5 (b)

2.5 Data is declared as `ARRAY[0:3] OF INTEGER` and contains 4, -1, 7, 7 in index order.

(b) Trace this algorithm. Give **Total** and **Count** after **each** iteration, then the final output. [4]

```
Total ← 0
Count ← 0
FOR Index ← 0 TO 3
    IF Data[Index] > 0 THEN
        Total ← Total + Data[Index]
        Count ← Count + 1
    ENDIF
NEXT Index
OUTPUT Total, ", ", Count
```

Exercise 2.5 (b)

2.5 Data is declared as `ARRAY[0:3] OF INTEGER` and contains 4, -1, 7, 7 in index order.

(b) Trace this algorithm. Give **Total** and **Count** after **each** iteration, then the final output. [4]

```
Total ← 0
Count ← 0
FOR Index ← 0 TO 3
    IF Data[Index] > 0 THEN
        Total ← Total + Data[Index]
        Count ← Count + 1
    ENDIF
NEXT Index
OUTPUT Total, ", ", Count
```

(b) correct total column, correct count column, unchanged values at index 1, final output 18, 3.

Full solution 2.5 (b)

(b) correct total column, correct count column, unchanged values at index 1, final output 18, 3.

Index	Total	Count
0	4	1
1	4	1
2	11	2
3	18	3

Exercise 2.6

2.6 Complete the code to input up to ten real values into global array `Values[1:10]`. Input `-999` ends the sequence and must not be stored. It may be the first input. Do not request another input when the array is full. [5]

```
DECLARE Count : INTEGER
DECLARE Value : REAL
Count ← .....
REPEAT
    INPUT Value
    IF Value <> ..... THEN
        Count ← .....
        Values[Count] ← .....
    ENDIF
UNTIL Value = -999 OR Count = .....
```

Explain why **both** stopping conditions are needed. [2]

Exercise 2.6

2.6 Complete the code to input up to ten real values into global array `Values[1:10]`. Input `-999` ends the sequence and must not be stored. It may be the first input. Do not request another input when the array is full. [5]

```

DECLARE Count : INTEGER
DECLARE Value : REAL
Count ← .....
REPEAT
    INPUT Value
    IF Value <> ..... THEN
        Count ← .....
        Values[Count] ← .....
    ENDIF
UNTIL Value = -999 OR Count = .....

```

Explain why **both** stopping conditions are needed. [2]

2.6

- 0; -999; Count + 1; Value; 10.
- the sentinel condition stops when input is finished; the capacity condition stops after ten stored values, preventing another input or an out-of-bounds write. The IF keeps -999 out of the array.

Exercise 2.7

2.7 A weather station takes one real temperature reading every **ten minutes** for one week. Calculate how many readings are taken and declare a 1-D array `WeekReading`, indexed from 1, with exactly enough elements. State the upper bound if the same number of elements were instead indexed from 0. [3]

Exercise 2.7

2.7 A weather station takes one real temperature reading every **ten minutes** for one week. Calculate how many readings are taken and declare a 1-D array **WeekReading**, indexed from 1, with exactly enough elements. State the upper bound if the same number of elements were instead indexed from 0. [3]

2.7

- $6 \times 24 \times 7 = \mathbf{1008}$ readings.
- `DECLARE WeekReading : ARRAY[1:1008] OF REAL.`
- if indexed from zero, the upper bound is 1007, because $1007 - 0 + 1 = 1008$.

Exercise 3.1

3.1 Global real array `Number[1:20]` is initially unused. Write procedure `Store()` to input and store a strictly increasing sequence. Always store the first input. Store later inputs only while each is greater than the last stored value. Stop at the first value that is not greater, or when 20 values have been stored. Do not request another input once full. Output the number stored with a message. Declare local variables. [6]

Exercise 3.1

3.1 Global real array `Number[1:20]` is initially unused. Write procedure `Store()` to input and store a strictly increasing sequence. Always store the first input. Store later inputs only while each is greater than the last stored value. Stop at the first value that is not greater, or when 20 values have been stored. Do not request another input once full. Output the number stored with a message. Declare local variables. [6]

3.1

- declare locals; store first input; capacity-controlled loop; comparison and stop on non-increasing input; update count and store correctly; final count/message.

```
PROCEDURE Store()  
    DECLARE Count : INTEGER  
    DECLARE Value, Last : REAL  
    DECLARE Stopped : BOOLEAN  
    INPUT Last  
    Number[1] ← Last
```

[Full solution on next slide](#)

Full solution 3.1 (1/2)

3.1

- declare locals; store first input; capacity-controlled loop; comparison and stop on non-increasing input; update count and store correctly; final count/message.

```
PROCEDURE Store()  
  DECLARE Count : INTEGER  
  DECLARE Value, Last : REAL  
  DECLARE Stopped : BOOLEAN  
  INPUT Last  
  Number[1] ← Last  
  Count ← 1
```

Full solution 3.1 (2/2)

```
Stopped ← FALSE
WHILE Count < 20 AND Stopped = FALSE
    INPUT Value
    IF Value > Last THEN
        Count ← Count + 1
        Number[Count] ← Value
        Last ← Value
    ELSE
        Stopped ← TRUE
    ENDIF
ENDWHILE
OUTPUT Count, " values stored"
ENDPROCEDURE
```

- Equal input ends the sequence. The rejected value is not stored. The loop requests no 21st input.

Exercise 3.2 (a)

3.2 A quiz has three questions worth 2 points each and two questions worth 5 points each. Global Boolean array **Possible** will indicate which total scores can be achieved; the index represents the total score.

(a) Declare **Possible** with lower bound 0 and the smallest suitable upper bound. Initialise every element to **FALSE**. Declare the loop variable. [4]

Exercise 3.2 (a)

3.2 A quiz has three questions worth 2 points each and two questions worth 5 points each. Global Boolean array **Possible** will indicate which total scores can be achieved; the index represents the total score.

(a) Declare **Possible** with lower bound 0 and the smallest suitable upper bound. Initialise every element to **FALSE**. Declare the loop variable. [4]

(a) maximum score $3 \times 2 + 2 \times 5 = 16$, so the array has 17 elements.

```
DECLARE Possible : ARRAY[0:16] OF BOOLEAN
DECLARE Index : INTEGER
FOR Index ← 0 TO 16
    Possible[Index] ← FALSE
NEXT Index
```

- array declaration; local declaration; correct loop bounds; assign **FALSE**.

Exercise 3.2 (b)

3.2 A quiz has three questions worth 2 points each and two questions worth 5 points each. Global Boolean array **Possible** will indicate which total scores can be achieved; the index represents the total score.

(b) Use nested loops to set every achievable score to **TRUE**, including zero. [4]

Exercise 3.2 (b)

3.2 A quiz has three questions worth 2 points each and two questions worth 5 points each. Global Boolean array **Possible** will indicate which total scores can be achieved; the index represents the total score.

(b) Use nested loops to set every achievable score to **TRUE**, including zero. [4]

(b) declare loop variables; all four easy-score values; all three hard-score values; mark the sum, including zero.

```
DECLARE Easy, Hard : INTEGER
FOR Easy ← 0 TO 6 STEP 2
    FOR Hard ← 0 TO 10 STEP 5
        Possible[Easy + Hard] ← TRUE
    NEXT Hard
NEXT Easy
```

Exercise 3.2 (c)

3.2 A quiz has three questions worth 2 points each and two questions worth 5 points each. Global Boolean array **Possible** will indicate which total scores can be achieved; the index represents the total score.

(c) Describe, in words, how a function checks whether an integer score is achievable. Include scores outside the array bounds. [3]

Exercise 3.2 (c)

3.2 A quiz has three questions worth 2 points each and two questions worth 5 points each. Global Boolean array **Possible** will indicate which total scores can be achieved; the index represents the total score.

(c) Describe, in words, how a function checks whether an integer score is achievable. Include scores outside the array bounds. [3]

(c) check whether the score lies from 0 to 16; if outside, return **FALSE** before indexing; otherwise return the Boolean stored at that score's index. Valid scores are 0, 2, 4, 5, 6, 7, 9, 10, 11, 12, 14, 16.

Exercise 3.3 (a)

3.3 A college uses global integer array `Result[1:100, 1:2]`. Column 1 stores marks; column 2 stores candidate IDs. Only the first `Used` rows contain data, where $0 \leq \text{Used} \leq 100$. Global integer array `Boundary[1:5]` contains five grade boundaries.

(a) Write function `NeedsCheck(Mark : INTEGER) RETURNS BOOLEAN`. Return `TRUE` if `Mark` is within two marks of **any** boundary, including the endpoints; otherwise return `FALSE`. [5]

Exercise 3.3 (a)

3.3 A college uses global integer array `Result[1:100, 1:2]`. Column 1 stores marks; column 2 stores candidate IDs. Only the first `Used` rows contain data, where $0 \leq \text{Used} \leq 100$. Global integer array `Boundary[1:5]` contains five grade boundaries.

(a) Write function `NeedsCheck(Mark : INTEGER) RETURNS BOOLEAN`. Return `TRUE` if `Mark` is within two marks of **any** boundary, including the endpoints; otherwise return `FALSE`. [5]

(a) parameter/return type and local declaration; examine the five boundaries; inclusive lower comparison; inclusive upper comparison with `AND`; correct success/failure returns.

```
FUNCTION NeedsCheck(Mark : INTEGER) RETURNS BOOLEAN
    DECLARE Index : INTEGER
    DECLARE Near : BOOLEAN
    FOR Index ← 1 TO 5
        Near ← Mark >= Boundary[Index] - 2
        IF Near AND Mark <= Boundary[Index] + 2 THEN
```

Full solution 3.3 (a)

(a) parameter/return type and local declaration; examine the five boundaries; inclusive lower comparison; inclusive upper comparison with AND; correct success/failure returns.

```
FUNCTION NeedsCheck(Mark : INTEGER) RETURNS BOOLEAN
  DECLARE Index : INTEGER
  DECLARE Near : BOOLEAN
  FOR Index ← 1 TO 5
    Near ← Mark >= Boundary[Index] - 2
    IF Near AND Mark <= Boundary[Index] + 2 THEN
      RETURN TRUE
    ENDIF
  NEXT Index
  RETURN FALSE
ENDFUNCTION
```

- Both comparisons must be true for the same boundary. Return FALSE only after all five boundaries fail.

Exercise 3.3 (b)

3.3 A college uses global integer array `Result[1:100, 1:2]`. Column 1 stores marks; column 2 stores candidate IDs. Only the first `Used` rows contain data, where $0 \leq \text{Used} \leq 100$. Global integer array `Boundary[1:5]` contains five grade boundaries.

(b) Global integer array `ReviewID[1:100]` and integer `ReviewCount` are available. Write procedure `CheckAll(Used : INTEGER)` to reset the count, call `NeedsCheck()` for each used row, and copy matching IDs consecutively into `ReviewID`. Update and output the count with a message. Declare locals; leave unused elements unchanged. [7]

Exercise 3.3 (b)

3.3 A college uses global integer array `Result[1:100, 1:2]`. Column 1 stores marks; column 2 stores candidate IDs. Only the first `Used` rows contain data, where $0 \leq \text{Used} \leq 100$. Global integer array `Boundary[1:5]` contains five grade boundaries.

(b) Global integer array `ReviewID[1:100]` and integer `ReviewCount` are available. Write procedure `CheckAll(Used : INTEGER)` to reset the count, call `NeedsCheck()` for each used row, and copy matching IDs consecutively into `ReviewID`. Update and output the count with a message. Declare locals; leave unused elements unchanged. [7]

(b) header and local declaration; reset global count; iterate only used rows; call with the mark column; increment only on a match; store the corresponding ID at the new count; final count/message.

```
PROCEDURE CheckAll(Used : INTEGER)
    DECLARE Row : INTEGER
    ReviewCount ← 0
    FOR Row ← 1 TO Used
```

Full solution 3.3 (b)

(b) header and local declaration; reset global count; iterate only used rows; call with the mark column; increment only on a match; store the corresponding ID at the new count; final count/message.

```
PROCEDURE CheckAll(Used : INTEGER)
  DECLARE Row : INTEGER
  ReviewCount ← 0
  FOR Row ← 1 TO Used
    IF NeedsCheck(Result[Row, 1]) THEN
      ReviewCount ← ReviewCount + 1
      ReviewID[ReviewCount] ← Result[Row, 2]
    ENDIF
  NEXT Row
  OUTPUT ReviewCount, " candidates need checking"
ENDPROCEDURE
```

- A candidate near more than one boundary is still copied once. `ReviewCount` is both the number selected and the last used index in `ReviewID`.

Exercise 3.3 (c)

3.3 A college uses global integer array `Result[1:100, 1:2]`. Column 1 stores marks; column 2 stores candidate IDs. Only the first `Used` rows contain data, where $0 \leq \text{Used} \leq 100$. Global integer array `Boundary[1:5]` contains five grade boundaries.

(c) Explain the result when `Used = 0`, including the count and whether any array element is accessed or changed. [2]

Exercise 3.3 (c)

3.3 A college uses global integer array `Result[1:100, 1:2]`. Column 1 stores marks; column 2 stores candidate IDs. Only the first `Used` rows contain data, where $0 \leq \text{Used} \leq 100$. Global integer array `Boundary[1:5]` contains five grade boundaries.

(c) Explain the result when `Used = 0`, including the count and whether any array element is accessed or changed. [2]

(c) `ReviewCount` becomes zero and the message reports zero candidates; no array elements are accessed or changed because the loop has no iterations. Any old values in `ReviewID` are unused.