Solutions

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

2.1

- the data must be **sorted** (in order)

2.2

- linear search: **O(n)**
- binary search: **O(log n)**

2.3

- adjacent pairs are compared along the array and **swapped if out of order**, moving the largest remaining value to the end

2.4

- any one: **time taken** (number of operations); **memory used**

3.1

- mid = index **4** (value 18, $< 23 \rightarrow$ go right)
- mid = index **6** (value 31, $> 23 \rightarrow$ go left)
- mid = index **5** (value 23 $\rightarrow$ found)

3.2

- first pass: `5`↔`2` → `[2,5,8,1]`; `5,8` ok; `8`↔`1` → `[2,5,1,8]`
- **`[2, 5, 1, 8]`**

3.3

- advantages (any two): **far fewer comparisons** on large lists (O(log n) vs O(n)); much faster when searching repeatedly
- disadvantage: the data must be **sorted first** (a one-off cost)

3.4

- most elements are already in place, so the inner `WHILE` does **few or no shifts** — close to O(n)

3.5

(a) rows for `Index` 1 to 5: `Data[Index]` $= 8, 3, -6, 11, -2$

- `Found` stays `FALSE` until `Index` = 3, then `TRUE`
- `Position` is unset until `Index` = 3, when it becomes 3, then becomes 5 at `Index` = 5

(b) it outputs **5**

- the `FOR` loop always runs all ten times
- so every later negative value **overwrites** `Position`, leaving the position of the **last** one rather than the first

(c) a `FOR` loop is a **count-controlled** loop and cannot stop early, but here the number of iterations is not known in advance — it depends on the data

- a **condition-controlled** loop is appropriate: a `WHILE` loop

(d)

```
Found ← FALSE
Index ← 1
WHILE Index <= 10 AND Found = FALSE
    IF Data[Index] < 0
      THEN
        Found ← TRUE
        Position ← Index
      ELSE
        Index ← Index + 1
    ENDIF
ENDWHILE
IF Found = TRUE
  THEN OUTPUT Position
  ELSE OUTPUT "None found"
ENDIF
```

- `WHILE` with both conditions; the flag set and the index advanced correctly; the loop cannot run past element 10; output unchanged and correct

(e) the corrected version stops at the third element, so it makes **3** comparisons

- the original always makes **10**
- a condition-controlled loop can leave as soon as the answer is known, so on data where the target appears early it does a fraction of the work; a count-controlled loop pays the full cost every time