Exercise walk-through

Algorithms ▪ 19.1

eXercise

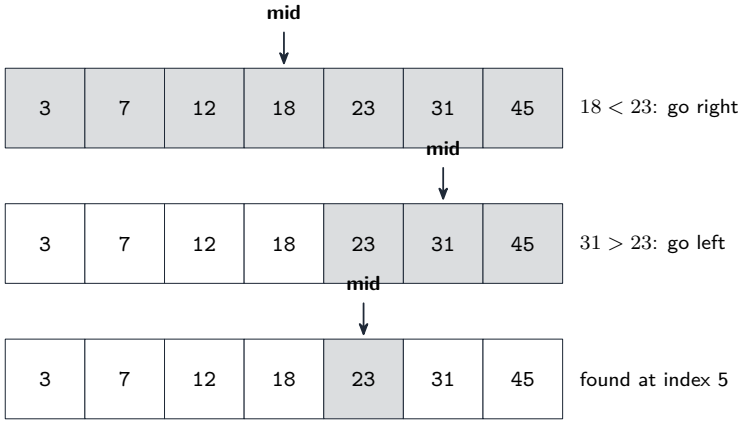
You must be able to

- trace a **linear** and a **binary** search
- trace a **bubble sort** and an **insertion sort**
- compare algorithms by **time** and **memory** (Big-O)

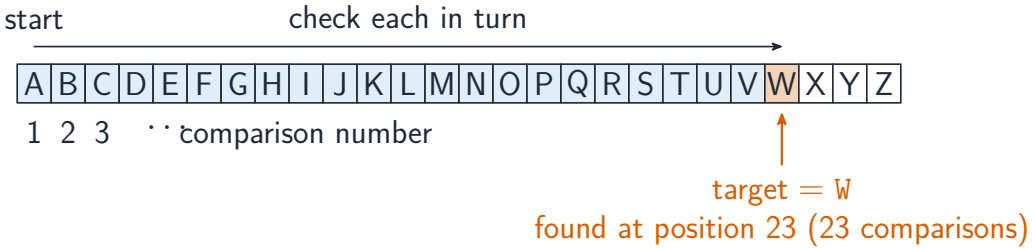
Method — Searching

A **linear search** checks each element in turn — works on **any** list, worst case **O(n)**. A **binary search** needs the data **sorted**: check the middle, then keep the half that could contain the target, halving the range each step — worst case **O(log n)**:

Method — Searching



Method — Searching



Linear search compared with binary search

Method — Bubble sort

Repeatedly walk the array, **swapping** adjacent out-of-order pairs, so the largest "bubbles" to the end each pass:

```
FOR pass ← 1 TO n - 1
    FOR i ← 1 TO n - pass
        IF A[i] > A[i+1] THEN swap A[i], A[i+1]
    NEXT i
NEXT pass
```

Method — Insertion sort

Build a sorted prefix; insert each new element by **shifting larger ones right**:

```
FOR i ← 2 TO n
    key ← A[i]
    j ← i - 1
    WHILE j >= 1 AND A[j] > key DO
        A[j+1] ← A[j]; j ← j - 1
    ENDWHILE
    A[j+1] ← key
NEXT i
```

Method — Insertion sort

Both are **O(n^2)** worst case; both are **O(n)** on already-sorted data (with an early exit / no shifts). Compare algorithms by **time taken** and **memory used**.

Practice 2.1 ▪ 1 mark

State the **precondition** that a binary search needs.

Solution 2.1

Method: Searching

Worked steps

The data must be **sorted** (in order) B1.

Practice 2.2 ▪ 2 marks

State the **worst-case** time complexity of a **linear** search and a **binary** search.

Solution 2.2

Method: Searching

Worked steps

Linear search: **O(n)**; binary search: **O(log n)** B1 B1.

Practice 2.3 ▪ 1 mark

In a **bubble sort**, describe what happens during **one pass**.

Solution 2.3

Method: Bubble sort

Worked steps

Adjacent pairs are compared along the array and **swapped if out of order**, moving the largest remaining value to the end B1.

Practice 2.4 ▪ 1 mark

State **one** criterion used to compare two algorithms.

Solution 2.4

Method: Insertion sort

Worked steps

Any one: **time taken** (number of operations); **memory used** B1.

Exam-style 3.1 ▪ 3 marks

Trace a **binary search** for 23 in `[3, 7, 12, 18, 23, 31, 45]` (indices 1–7). State the **index** examined at each step.

Solution 3.1

Method: Searching

Worked steps

mid = index **4** (value 18, <23 → go right); mid = index **6** (value 31, >23 → go left); mid = index **5** (value 23 → found) B1 B1 B1.

Exam-style 3.2 ▪ 2 marks

The array `[5, 2, 8, 1]` is sorted with a **bubble sort**. Show the array **after the first complete pass**.

Solution 3.2

Method: Bubble sort

Worked steps

First pass: 5↔2→`[2,5,8,1]`; `5,8` ok; 8↔1→`[2,5,1,8]`. Answer: **`[2, 5, 1, 8]`** M1 A1 *correct comparisons/swaps, final array.*

Exam-style 3.3 ▪ 3 marks

State **two** advantages of a binary search over a linear search, and **one** disadvantage.

Solution 3.3

Method: Searching

Worked steps

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) B1 B1 B1.

Exam-style 3.4 ▪ 1 mark

State why an **insertion sort** is efficient on a list that is **already nearly sorted**.

Solution 3.4

Method: Insertion sort

Worked steps

Most elements are already in place, so the inner `WHILE` does **few or no shifts** — close to O(n) B1.

Exam-style 3.5 ▪ 4 marks

The algorithm below is intended to find the position of the first negative value in an array Data of 10 integers, or to output a message if there is none.

```
Found ← FALSE
Index ← 1
FOR Index ← 1 TO 10
    IF Data[Index] < 0
      THEN
        Found ← TRUE
        Position ← Index
    ENDIF
NEXT Index
IF Found = TRUE
  THEN OUTPUT Position
  ELSE OUTPUT "None found"
ENDIF
```

Exam-style 3.5 ▪ 4 marks

The array holds: `8, 3, -6, 11, -2, 7, 4, 9, 1, 5`.

(a) **Complete** the trace table by dry running the algorithm for the first **five** iterations.

Index	`Data[Index]`	**Found**	**Position**

Exam-style 3.5 ▪ 4 marks

(b) **State** the value the algorithm outputs, and **explain** why it is **not** the position of the *first* negative value.
(c) The outer loop structure is **not** the most appropriate one for this task. **Explain** why, and **name** the loop you would use instead.
(d) **Write** corrected pseudocode using that loop, so the algorithm stops as soon as it finds the first negative value.
(e) **Calculate** how many comparisons your corrected version makes on this data, and how many the original makes. **Describe** what this shows about the two loop choices when the target is near the start.

Solution 3.5

Method: Searching

Worked steps

(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` $= 5$ B1 B1 B1 B1 *for the 'Data' column; for 'Found'; for 'Position' set to 3; for 'Position' overwritten with 5.*

(b) It outputs **5** B1. The `FOR` loop always runs all ten times B1, so every later negative value **overwrites** `Position`, leaving the position of the **last** one rather than the first B1.

(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 B1 B1. A **condition-controlled** loop is appropriate: a `WHILE` loop B1.

(d)

Solution 3.5

```
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
```

Solution 3.5

B1 B1 B1 B1 *'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 B1; the original always makes **10** B1. 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, whatever the data B1.