# Standard algorithms

Ref As Pseudocode

## Linear search

A **linear search** checks the elements of an array **one by one**, from the first, until it finds the item or reaches the end. It works on any array, sorted or not.

### ■ Stop as soon as it is found

A good search stops early. Use a `WHILE` loop with a **flag** that becomes `TRUE` when the item is found:

```
DECLARE Names : ARRAY[1:6] OF STRING
DECLARE SearchName : STRING
DECLARE Index : INTEGER
DECLARE Found : BOOLEAN
Names[1] ← "Ola"
Names[2] ← "Raj"
Names[3] ← "Kim"
Names[4] ← "Tom"
Names[5] ← "Eva"
Names[6] ← "Ivy"
SearchName ← "Tom"

Found ← FALSE
Index ← 1
WHILE Found = FALSE AND Index <= 6
   IF Names[Index] = SearchName THEN
      Found ← TRUE
   ELSE
      Index ← Index + 1
   ENDIF
ENDWHILE

IF Found THEN
   OUTPUT SearchName, " found at position ", Index
ELSE
   OUTPUT SearchName, " not found"
ENDIF
```

- The loop stops when the item is found **or** there are no more elements.
- `Index` is only increased when the item is **not** found, so after the loop it still points to the item.
- Change `SearchName` to `"Zoe"` and run it again to see the “not found” path.

### As a function

In Paper 2 a search is often written as a function that returns the position, or a special value such as -1 when the item is missing:

```
DECLARE Data : ARRAY[1:5] OF INTEGER

FUNCTION FindValue(Target : INTEGER) RETURNS INTEGER
   DECLARE i : INTEGER
   FOR i ← 1 TO 5
      IF Data[i] = Target THEN
         RETURN i
      ENDIF
   NEXT i
   RETURN -1
ENDFUNCTION

Data[1] ← 14
Data[2] ← 3
Data[3] ← 27
Data[4] ← 8
Data[5] ← 19
OUTPUT FindValue(27)
OUTPUT FindValue(5)
```

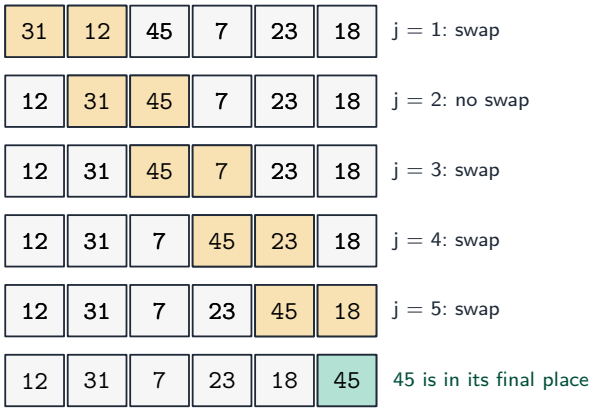
`RETURN i` leaves the function at once, so the `FOR` loop does not finish. `RETURN -1` is only reached when nothing matched.

**Common mistakes**

- Writing `ELSE OUTPUT "Not found"` **inside** the loop. That prints "Not found" for every element that does not match. Decide "not found" only **after** the loop.
- A loop condition that never stops: always include the `Index <= <upper bound>` test.

## Bubble sort

A **bubble sort** puts the elements of an array in order. It goes through the array comparing each **pair of neighbours**, and swaps them if they are in the wrong order. After one full pass pass , the largest value has "bubbled" to the end. Each pass moves one more value into its final place.




*One pass of a bubble sort: compare neighbours, swap if out of order*

### The improved version

The best version stops as soon as a pass makes **no swaps**, because then the array is already sorted. It also checks one fewer element on each pass.

```
DECLARE Numbers : ARRAY[1:6] OF INTEGER
DECLARE Top : INTEGER
DECLARE j : INTEGER
DECLARE Temp : INTEGER
DECLARE Swapped : BOOLEAN
Numbers[1] ← 31
Numbers[2] ← 12
Numbers[3] ← 45
Numbers[4] ← 7
Numbers[5] ← 23
Numbers[6] ← 18

Top ← 5
REPEAT
   Swapped ← FALSE
   FOR j ← 1 TO Top
      IF Numbers[j] > Numbers[j + 1] THEN
         Temp ← Numbers[j]
         Numbers[j] ← Numbers[j + 1]
         Numbers[j + 1] ← Temp
         Swapped ← TRUE
      ENDIF
   NEXT j
   Top ← Top - 1
UNTIL Swapped = FALSE OR Top = 0

FOR j ← 1 TO 6
   OUTPUT Numbers[j]
NEXT j
```

Read it in four parts:

1. **Compare neighbours**: `Numbers[j]` with `Numbers[j + 1]`. The loop goes only to `Top`, so `j + 1` never goes past the end.
2. **Swap** with a temporary variable. Without `Temp`, the first value would be lost.
3. **Remember** that a swap happened: `Swapped ← TRUE`.
4. **Stop** when a pass makes no swaps. `Top ← Top - 1` skips the values already in place.

To sort in **descending** order (largest first), change `>` to `<`.

### ■ Sorting records

To sort an array of records, compare one **field**, but swap the **whole records**:

```
IF Students[j].Mark > Students[j + 1].Mark THEN
   Temp ← Students[j]
   Students[j] ← Students[j + 1]
   Students[j + 1] ← Temp
ENDIF
```

Here `Temp` must be declared with the record type.

**Common mistakes**

- `FOR j ← 1 TO 6` in a 6-element array: then `Numbers[j + 1]` becomes `Numbers[7]`, which does not exist. The inner loop goes to one less than the size.
- Swapping without a temporary variable: `Numbers[j] ← Numbers[j + 1]` then `Numbers[j + 1] ← Numbers[j]` copies the same value into both.
- Setting `Swapped ← FALSE` **outside** the `REPEAT`. It must be reset at the start of every pass.

## Validating input

**Validation** is an automatic check that input is **sensible** before the program uses it. It cannot prove the data is correct, only that it is reasonable.

| Check | Question it asks | Example |
|---|---|---|
| range check | Is the value between a lower and upper limit? | a mark from 0 to 100 |
| length check | Does it have an allowed number of characters? | a password of at least 8 characters |
| type check | Is it the right kind of data? | the age is a number |
| presence check | Was anything entered at all? | the name is not empty |
| format check | Does it follow a pattern? | a code of 2 letters then 4 digits |



exercises.lol exercises.com.cn

### The validation loop

Ask again and again until the input passes. `REPEAT … UNTIL` fits, because you must ask at least once:

```
DECLARE Entry : STRING
DECLARE Mark : INTEGER
DECLARE Valid : BOOLEAN
REPEAT
   OUTPUT "Enter a mark from 0 to 100"
   INPUT Entry
   Valid ← FALSE
   IF IS_NUM(Entry) THEN
      Mark ← STR_TO_NUM(Entry)
      IF Mark >= 0 AND Mark <= 100 THEN
         Valid ← TRUE
      ENDIF
   ENDIF
   IF Valid = FALSE THEN
      OUTPUT "Invalid. Try again."
   ENDIF
UNTIL Valid
OUTPUT "Mark accepted: ", Mark
```

This does a **type check** (`IS_NUM`) first, then a **range check**. The type check must come first, because a range check on text such as `"abc"` makes no sense.



exercises.lol exercises.com.cn

## A validation function

When the rules get longer, put them in a function that returns `TRUE` or `FALSE`:

```
FUNCTION ValidPassword(Password : STRING) RETURNS BOOLEAN
   DECLARE i : INTEGER
   DECLARE HasDigit : BOOLEAN
   DECLARE Ch : CHAR
   IF LENGTH(Password) < 8 THEN
      RETURN FALSE
   ENDIF
   HasDigit ← FALSE
   FOR i ← 1 TO LENGTH(Password)
      Ch ← MID(Password, i, 1)
      IF Ch >= '0' AND Ch <= '9' THEN
         HasDigit ← TRUE
      ENDIF
   NEXT i
   RETURN HasDigit
ENDFUNCTION

OUTPUT ValidPassword("sunshine")
OUTPUT ValidPassword("sunshine7")
OUTPUT ValidPassword("ab1")
```

It prints `FALSE` (no digit), `TRUE` and `FALSE` (too short).

## Practice

**1.** Change the bubble sort above so that it sorts the numbers from largest to smallest.

**2.** Write a function `CountOf(Target : INTEGER) RETURNS INTEGER` that returns how many times `Target` appears in a global array `Data[1:8]`.

<details> <summary>Model answer</summary>

1. Change the comparison to `IF Numbers[j] < Numbers[j + 1] THEN`. Nothing else changes.

```
DECLARE Data : ARRAY[1:8] OF INTEGER

FUNCTION CountOf(Target : INTEGER) RETURNS INTEGER
   DECLARE i : INTEGER
   DECLARE Count : INTEGER
   Count ← 0
   FOR i ← 1 TO 8
      IF Data[i] = Target THEN
         Count ← Count + 1
      ENDIF
   NEXT i
   RETURN Count
ENDFUNCTION

Data[1] ← 4
Data[2] ← 7
Data[3] ← 4
Data[4] ← 1
Data[5] ← 9
Data[6] ← 4
Data[7] ← 2
Data[8] ← 7
OUTPUT CountOf(4)
OUTPUT CountOf(7)
```

It prints `3` and `2`. Here the loop must **not** stop early: every element has to be counted, so a `FOR` loop is right.

</details>

**Common mistakes**

- Validating with a single `IF`: the user gets one more chance and then the program goes on with bad data. Use a loop.
- Confusing validation with **verification** . Verification checks that data was **copied correctly** (double entry, visual check); validation checks that it is **sensible**.