标准算法

Ref As Pseudocode

线性查找

线性查找（linear search）从第一个元素开始，**逐个**检查数组中的元素，直到找到目标或到达末尾为止。无论数组是否排过序，它都适用。

■ 一找到就停止

好的查找会提前停止。用一个 `WHILE` 循环，加上一个找到目标时变为 `TRUE` 的**标志**：

```
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
```

- 找到目标**或者**没有剩余元素时，循环就停止。
- 只有在**没有**找到时才增加 `Index`，所以循环结束后它仍然指向目标。
- 把 `SearchName` 改成 `"Zoe"` 再运行一次，看看”没找到”的情况。

■ 写成函数

在 Paper 2 中，查找常常写成一个函数：返回目标的位置，找不到时返回 -1 这样的特殊值：

```
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)
```

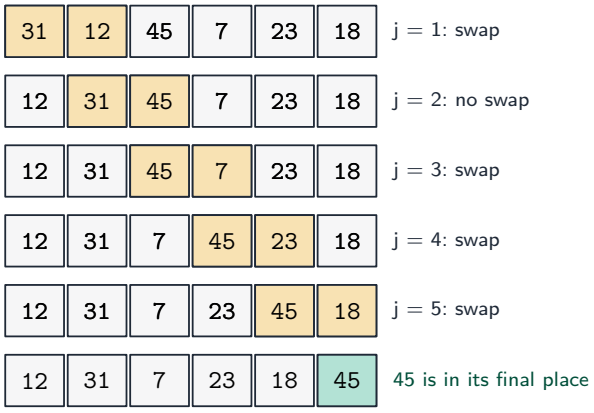
`RETURN i` 会立刻离开函数，所以 `FOR` 循环不会走完。只有在什么都没匹配上时，才会执行到 `RETURN -1`。

常见错误

- 在循环**里面**写 `ELSE OUTPUT "Not found"`。这样每遇到一个不匹配的元素就输出一次"Not found"。“ 没找到 ” 只能在循环**结束之后**判断。
- 循环条件永远不会停止：一定要加上 `Index <= <` 上界 `>` 的判断。

冒泡排序

冒泡排序（bubble sort）把数组元素按顺序排好。它从头到尾比较每一对**相邻的**元素，如果顺序不对就交换。完整地走完一**遍**（pass）后，最大的值就像气泡一样 “ 冒 ” 到了末尾。每走一遍，就有一个值被放到它最终的位置。



冒泡排序的一遍：比较相邻元素，顺序不对就交换

■ 改进版

最好的写法是：一旦某一遍**没有发生交换**，就立即停止，因为这时数组已经排好了。它还会每一遍少检查一个元素。

```
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
```

分四部分来读：

1. **比较相邻元素**：`Numbers[j]` 与 `Numbers[j + 1]`。循环只走到 `Top`，所以 `j + 1` 永远不会越界。
2. 借助一个临时变量来**交换**。没有 `Temp`，第一个值就会丢失。
3. **记住**发生过交换：`Swapped ← TRUE`。
4. 某一遍没有交换时就**停止**。`Top ← Top - 1` 跳过已经到位的值。

要按**降序**（从大到小）排序，把 `>` 改成 `<` 即可。

■ 对记录排序

对记录数组排序时，比较的是某一个**字段**，交换的却是**整条记录**：

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

这里 Temp 必须声明为该记录类型。

常见错误

- 对 6 个元素的数组写 `FOR j ← 1 TO 6`：这样 `Numbers[j + 1]` 会变成不存在的 `Numbers[7]`。内层循环要比数组大小少一。
- 不用临时变量就交换：先写 `Numbers[j] ← Numbers[j + 1]` 再写 `Numbers[j + 1] ← Numbers[j]`，两个位置会变成同一个值。
- 把 `Swapped ← FALSE` 写在 `REPEAT` **外面**。每一遍开始时都必须重新设置它。

输入验证

验证（validation）是在程序使用输入之前，自动检查这个输入是否**合理**。它不能证明数据是正确的，只能说明数据是合理的。

检查	它要问的问题	示例
范围检查（range check）	值是否在上下限之间?	分数在 0 到 100 之间
长度检查（length check）	字符个数是否在允许的范围内?	密码至少 8 个字符
类型检查（type check）	数据的种类对不对?	年龄是一个数字
存在检查（presence check）	是否输入了内容?	姓名不为空
格式检查（format check）	是否符合规定的模式?	编号是 2 个字母加 4 个数字

■ 验证循环

一遍又一遍地询问，直到输入通过检查。REPEAT … UNTIL 很合适，因为至少要问一次：

```
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
```

这段程序先做**类型检查**（IS_NUM），再做**范围检查**。类型检查必须放在前面，因为对 "abc" 这样的文字做范围检查毫无意义。

■ 验证函数

规则变长时，就把它们放进一个返回 TRUE 或 FALSE 的函数：

```
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")
```

输出依次是 FALSE（没有数字）、TRUE 和 FALSE（太短）。

■ 练习

1. 修改上面的冒泡排序，让它把数字从大到小排列。

2. 写一个函数 `CountOf(Target : INTEGER) RETURNS INTEGER`，返回 `Target` 在全局数组 `Data[1:8]` 中出现的次数。

<details> <summary> 参考答案 </summary>

1. 把比较改成 `IF Numbers[j] < Numbers[j + 1] THEN`，其他都不用改。

```
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)
```

输出 3 和 2。这里循环**不能**提前停止：每个元素都要数到，所以用 FOR 循环是对的。
</details>

常见错误

- 只用一个 IF 来验证：用户只多得到一次机会，之后程序就带着错误数据继续运行。要用循环。
- 把验证和**核实**（verification）混为一谈。核实检查数据是否被**正确地复制**（两次输入、目视检查）；验证检查数据是否**合理**。