A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

11.3.3 Efficient pseudocode, structure charts and breaking a task into modules

Name: _____ Class: _____ Date: _____ **Total: 30 marks**

KEY WORDS procedure 过程 • function 函数 • return value 返回值 • efficient pseudocode 伪代码 • arguments 实参

Objective

Build the skills to answer the **structured programming** 结构化编程 questions that are not about one subroutine at a time. Sheets 11.3.1 and 11.3.2 covered writing a **procedure** 过程 or **function** 函数 and passing its parameters; this sheet is about writing **efficient pseudocode** 伪代码, deciding where the boundaries between **modules** 模块 go, reading a **structure chart** 结构图, and testing the pieces separately before joining them.

You must be able to:

- explain where in the construction of an algorithm it is appropriate to use a procedure, and where a function
- write efficient pseudocode: no work repeated that could be done once, and no test made that the answer is already known
- break a described task into modules, and give each one a header and an **interface**
- read a structure chart: what a box is, what a downward arrow is, and what the small side arrows carry
- explain why modules are tested on their own before being combined

1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

■ What "efficient" means in this syllabus

slower

```
FOR i = 1 TO n
 x = slow() // every pass
 use(x, i)
NEXT i
```

faster

```
x = slow() // once
FOR i = 1 TO n
 use(x, i)
NEXT i
```

Efficiency here is not about big-O. It is four concrete habits the mark scheme rewards:

1. **Move unchanging work out of the loop.** `LENGTH(InString)` does not change while the loop runs, so compute it once into a local and use that.
2. **Stop as soon as the answer is known.** Add the flag to the loop condition – `WHILE Index <= Num AND IsPal = TRUE` – or `RETURN` immediately.
3. **Do not test what you already know.** After `IF Mark >= 70`, a later test only needs `Mark >= 60`, never `Mark < 70 AND Mark >= 60`.
4. **Use arithmetic instead of a search.** A band of ten is `(Percent DIV 10) + 1`, not a loop or an eleven-branch `CASE`.

Worked example. *Rewrite this so the same work is not repeated.*

```
FOR Index <- 1 TO LENGTH(InString)          // recomputed every pass
    IF MID(InString, Index, 1) <> " " THEN
        OutString <- OutString & MID(InString, Index, 1)
    ENDIF
NEXT Index
```

```
DECLARE Len : INTEGER
Len <- LENGTH(InString)                     // once
FOR Index <- 1 TO Len
    ThisChar <- MID(InString, Index, 1)     // once per pass, not twice
    IF ThisChar <> " " THEN
        OutString <- OutString & ThisChar
    ENDIF
NEXT Index
```

- Two changes, two marks: the length is found once, and each character is extracted once instead of twice.

■ Where the module boundaries go

```
FUNCTION CountAbove(BYVAL Limit : INTEGER) RETURNS INTEGER      header: name, parameters, return type
    DECLARE Index, Count : INTEGER                              local variables declared with a type
    Count ← 0                                                   total initialised before the loop
    FOR Index ← 1 TO 50                                         loop over every element
        IF Score[Index] > Limit THEN                            the condition, with the right boundary
            Count ← Count + 1                                   the update, inside the IF
        ENDIF                                                   every construct closed
    NEXT Index
    RETURN Count                                                the return value, after the loop
ENDFUNCTION
```

Given a task in prose, the boundaries fall where the **jobs** fall. Ask of each job:

- Is it needed **in more than one place**, or does it simply have to be *done*? Make it a **procedure**.
- Does it produce **one value** the rest of the algorithm then uses? Make it a **function**.
- Does it change something the caller must see, but produce no single value? A procedure with a `BYREF` parameter.

Worked example. *A program reads a file of marks, validates each one, works out the average and prints a report.* Three modules:

Module	Kind	Header
read and validate one mark	function	`FUNCTION GetMark() RETURNS INTEGER`
work out the average of a total and a count	function	`FUNCTION Average(Total : INTEGER, Count : INTEGER) RETURNS REAL`
print the report	procedure	`PROCEDURE PrintReport(Av : REAL, Count : INTEGER)`

- Name each module for the **one** job it does. A module you cannot name in a few words is doing two jobs.
- A procedure may have **none**, one or more parameters. `PROCEDURE ShowMenu()` needs none at all, because it works on nothing the caller supplies. Where there are parameters, decide for each one separately whether it is passed by value or by **reference**.

■ Reading a structure chart

A structure chart shows the same decomposition as a picture:

- a **box** is a module,
- a **downward line** from one box to another means the upper module **calls** the lower one,

- the small arrows beside that line are the **parameters**: an arrow pointing **down** into the lower module is data passed in, and an arrow pointing **up** is data returned.

So a lower box with only a downward arrow is doing something with data it is given; one with an upward arrow is handing a value back, which usually means it is a function. Reading the chart tells you each module's **interface** without seeing any code, and the **terminology associated** with procedures and functions is what the question will use to ask about it: the **header** names the module and its **parameters**, the caller supplies the matching **arguments**, and an upward arrow is the **return value**.

■ Testing the pieces, then the whole

- **Module (unit) testing** – each module is tested on its own, with values chosen to exercise it, before the program is assembled.
- **Integration testing** – modules that have already been tested individually are combined into a single sub-program, which is then tested as a whole.
- This is the payoff of writing self-contained modules with local variables and parameters: a module with no globals can be tested by itself, because everything it touches arrives through its interface.

2 Practice

Now use the methods above.

2.1 Rewrite this loop so that no work is repeated unnecessarily, and state the change you made. [3]

```
FOR Index <- 1 TO LENGTH(Names)
    OUTPUT UCASE(MID(Names, Index, 1))
NEXT Index
```

2.2 State **two** things that make pseudocode inefficient. [2]

2.3 A program must: input a password; check whether it is valid; and display a welcome screen. State, for each of the three jobs, whether it is better written as a **procedure** or a **function**, with a reason. [3]

2.4 In a structure chart, state what a **box** represents, and state what a small arrow pointing **up** beside a call line represents. [2]

2.5 State what is meant by **integration testing**. [1]

3 Exam-style questions

3.1 A module counts how many values in a global array `Readings` of 500 integers are above a given limit.

```
FUNCTION HowMany(Limit : INTEGER) RETURNS INTEGER
    DECLARE Index : INTEGER
    Count <- 0
    FOR Index <- 1 TO 500
        IF Readings[Index] > Limit AND Readings[Index] <> -1 THEN
            IF Readings[Index] > Limit THEN
                Count <- Count + 1
            ENDIF
        ENDIF
    NEXT Index
    RETURN Count
ENDFUNCTION
```

(a) State **two** ways in which this pseudocode is inefficient. [2]

(b) Rewrite the module efficiently. [3]

3.2 A program for a library must: find a member by their card number; work out the fine owed on a late book; and print a receipt.

Break the task into **three** modules. For each one, state whether it is a procedure or a function and write its header. [6]

3.3 A programmer writes each module so that it uses only its parameters and its own local variables.

(a) Explain why this makes a module easier to **test**. [2]

(b) State **two** other benefits of building a program from modules. [2]

3.4 A function is written to return the position of the first space in a string.

```
FUNCTION FirstSpace(InString : STRING) RETURNS INTEGER
    DECLARE Index, Found : INTEGER
    Found <- 0
    FOR Index <- 1 TO LENGTH(InString)
        IF MID(InString, Index, 1) = " " THEN
            IF Found = 0 THEN
                Found <- Index
            ENDIF
        ENDIF
    NEXT Index
    RETURN Found
ENDFUNCTION
```

State **two** inefficiencies, and rewrite the loop so the function stops as soon as it has the answer. [4]

Solutions

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

2.1

- `LENGTH(Names)` is recomputed on every pass, so compute it once into a local variable

```
DECLARE Len : INTEGER
Len <- LENGTH(Names)
FOR Index <- 1 TO Len
    OUTPUT UCASE(MID(Names, Index, 1))
NEXT Index
```

- the rewritten loop, with the declaration

2.2

- any two: repeating a calculation inside a loop whose answer does not change; carrying on looping after the answer is known; testing a condition that an earlier test has already settled; using a loop or a long `CASE` where arithmetic would do

2.3

- input the password: a **procedure** – it performs an action and returns nothing the caller must have
- check whether it is valid: a **function** – it produces one value (`TRUE`/`FALSE`) that the rest of the algorithm uses in an `IF`
- display the welcome screen: a **procedure** – it does something, and there is no value to return

2.4

- a box represents a **module** (a procedure or function)
- an arrow pointing **up** beside a call line represents data **returned** from the lower module to the one that called it

2.5

- modules that have already been tested individually are combined into a single sub-program, which is then tested as a whole

3.1

(a) the inner `IF` repeats a test the outer `IF` has already made

- `Readings[Index]` is read from the array three times in one pass, when once into a local would do; and `Count` is used without being declared

(b)


```
FUNCTION HowMany(Limit : INTEGER) RETURNS INTEGER
    DECLARE Index, Count, ThisOne : INTEGER
    Count <- 0
    FOR Index <- 1 TO 500
        ThisOne <- Readings[Index]
        IF ThisOne > Limit AND ThisOne <> -1 THEN
            Count <- Count + 1
        ENDIF
    NEXT Index
    RETURN Count
ENDFUNCTION
```

- one test instead of two; the element read once into a local; `Count` declared and the return kept outside the loop

3.2

- find a member: a **function** – it produces one value the rest of the program uses `FUNCTION FindMember(CardNumber : STRING) RETURNS INTEGER`
- work out the fine: a **function** – one value, used in an expression `FUNCTION Fine(DaysLate : INTEGER) RETURNS REAL`
- print a receipt: a **procedure** – it performs an action and returns nothing `PROCEDURE PrintReceipt(MemberID : INTEGER, Amount : REAL)`
- sensible names and types are accepted; a module that both finds **and** prints would lose a mark, because it does two jobs

3.3

(a) everything the module uses arrives through its **interface**, so it can be called on its own with chosen test values

- nothing outside it can change its variables, so the same inputs always give the same result and a fault is traceable to that module

(b) any two: the same module can be reused elsewhere in the program or in another program; several programmers can work on different modules at once; a change is made in one place only; the program is easier to read because each module has one job

3.4

- the inner `IF Found = 0` is only needed because the loop keeps running after the answer is known
- `LENGTH(InString)` is recomputed on every pass

```
FUNCTION FirstSpace(InString : STRING) RETURNS INTEGER
    DECLARE Index, Len : INTEGER
    Len <- LENGTH(InString)
    FOR Index <- 1 TO Len
        IF MID(InString, Index, 1) = " " THEN
            RETURN Index
        ENDIF
    NEXT Index
    RETURN 0
ENDFUNCTION
```

- returning immediately removes the flag and the inner test
- a final `RETURN 0` for a string with no space, so a value comes back in both cases