2.2 Problem Solving and Programming

OCR H446 · Calculators not allowed · about 135 minutes · Write-in edition

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A game awards a bonus based on a player's grade, using the following CASE statement:

```
CASE OF grade
    "A" : bonus <- 50
    "B" : bonus <- 30
    "C" : bonus <- 10
    OTHERWISE : bonus <- 0
```

ENDCASE

(a) State the value assigned to bonus when grade = "B". **(1)**

(b) Rewrite the CASE statement above as an equivalent structure using nested IF...THEN...ELSE statements. **(4)**

(c) Explain one advantage of using a CASE statement rather than nested IF statements in this situation. **(2)**

(d) A trainee programmer writes this version of the CASE statement: **(2)**

```
CASE OF grade
    "A" : bonus = 50
    "B" : bonus <- 30
    OTHERWISE : bonus <- 0
```

ENDCASE

Identify the error in this code and state how to fix it.

(Total for Question 1 is 9 marks)

2 A teacher stores the test scores of 3 students, each having sat 4 tests, in a 2D array scores[1:3, 1:4]:

Test1 Test2 Test3 Test4
Student1: 55 62 70 48
Student2: 80 75 90 85
Student3: 40 55 60 65

(a) Write a pseudocode DECLARE statement for the 2D array scores, capable of holding this data. **(2)**

(b) Write an algorithm to calculate and output the average score for Student2 (row 2) only. **(4)**

(c) This nested loop sums every score in the array: **(4)**

```
total <- 0
FOR row <- 1 TO 3
    FOR col <- 1 TO 4
        total <- total + scores[row, col]
    NEXT col
NEXT row
OUTPUT total
```

Complete the trace table below, showing the cumulative value of total after each row is fully processed, and state the final value output.

After row 1: total = ___
After row 2: total = ___
After row 3: total = ___

(d) State the value stored in scores[2, 3]. **(1)**

(Total for Question 2 is 11 marks)

3 A stock system uses product codes in the format "AB-1234": exactly 7 characters, with the first two characters being letters, the third character a hyphen, and the last four characters digits.

(a) Write a function ValidateCode(code : STRING) RETURNS BOOLEAN that checks a product code matches this format, using string handling functions such as LEN, MID and ASC. **(6)**

(b) State the value returned by MID("AB-1234", 4, 4). **(1)**

(c) State the value returned by UCASE(LEFT("ab-1234", 2)). **(1)**

(d) Explain why passing the validation in part (a) does not guarantee that a product code entered is accurate. **(2)**

(Total for Question 3 is 10 marks)

4 A programmer writes the following subroutines and main program:

```
PROCEDURE DoubleValue(BYREF x : INTEGER)
    x <- x * 2
ENDPROCEDURE

FUNCTION AddFive(y : INTEGER) RETURNS INTEGER
    y <- y + 5
    RETURN y
ENDFUNCTION

num1 <- 10
num2 <- 10
```

DoubleValue(num1)

```
num2 <- AddFive(num2)
OUTPUT num1
OUTPUT num2
```

(a) State the values output for num1 and num2. **(2)**

(b) Explain the difference between passing a parameter by reference and by value, using num1 and num2 as examples. **(4)**

(c) Rewrite DoubleValue as a function, DoubleValueFunc, that takes its parameter by value and returns the doubled value, without modifying the original argument. **(3)**

(d) State one reason a programmer might choose to write a subroutine as a procedure rather than a function. **(1)**

(Total for Question 4 is 10 marks)

5 A sorted array holds 8 integers with indices 1 to 8:

list = [3, 8, 15, 19, 24, 31, 42, 50]

The function below performs a binary search for a target value:

```
FUNCTION BinarySearch(list : ARRAY[1:8] OF INTEGER, target : INTEGER) RETURNS
    lower <- 1
    upper <- 8
    WHILE lower <= upper
        mid <- (lower + upper) DIV 2
        IF list[mid] = target THEN
            RETURN mid
        ELSE IF list[mid] < target THEN
            lower <- mid + 1
        ELSE
            upper <- mid - 1
        ENDIF
    ENDWHILE
    RETURN -1
ENDFUNCTION
```

BinarySearch(list, 24) is called.

(a) Complete a trace table showing the values of lower, upper, mid and list[mid] for each iteration of the WHILE loop, until the target 24 is found. **(5)**

(b) State the number of comparisons with list[mid] needed to locate the value 24. **(1)**

(c) Explain why binary search cannot reliably be used on the unsorted array [8, 3, 19, 15, 24, 31, 42, 50]. **(2)**

(d) State the time complexity of binary search in Big O notation, and explain why it is more efficient than linear search, O(n), for very large data sets. **(3)**

(e) Write pseudocode for a function LinearSearch(list : ARRAY[1:8] OF INTEGER, target : INTEGER) RETURNS INTEGER that returns the index of target in list, or -1 if it is not present. **(3)**

(Total for Question 5 is 14 marks)

6 The array [8, 3, 6, 1] is to be sorted into ascending order using a bubble sort.

(a) Complete the trace table below, showing the state of the array after each full pass of a standard bubble sort (comparing and swapping adjacent elements left to right), and the number of swaps made in that pass. **(4)**

Start: [8, 3, 6, 1]
After pass 1: [___] , swaps = ___
After pass 2: [___] , swaps = ___
After pass 3: [___] , swaps = ___
After pass 4: [___] , swaps = ___

(b) State how a bubble sort can be made more efficient by detecting that the array is already sorted, and explain how this is recognised. **(2)**

(c) Write a procedure BubbleSort(BYREF arr : ARRAY[1:n] OF INTEGER, n : INTEGER) that sorts arr into ascending order, including the early-exit optimisation from part (b). **(6)**

(d) State the worst-case time complexity of bubble sort in Big O notation. **(1)**

(Total for Question 6 is 13 marks)

7 A stack is implemented using an array stack[1:5] with an integer pointer top, initially 0 (empty stack).

(a) State the names of the two operations used to add data to, and remove data from, a stack. **(2)**

(b) The following sequence of operations is carried out on the empty stack: **(5)**
Push(5), Push(9), Pop(), Push(3), Push(7), Pop()
Complete a trace table showing the value of top and the contents of the stack after each operation.

(c) Explain why a stack, rather than a queue, is an appropriate data structure for checking that brackets in an expression such as ((a+b)*(c-d)) are balanced. **(3)**

(d) State one real-world application of a queue in a computer system. **(1)**

(Total for Question 7 is 11 marks)

8 The following recursive function is defined:

```
FUNCTION Mystery(n : INTEGER) RETURNS INTEGER
    IF n = 1 THEN
        RETURN 1
    ELSE
        RETURN n * Mystery(n - 1)
    ENDIF
ENDFUNCTION
```

(a) State what Mystery(n) calculates. **(1)**

(b) Complete a trace of the recursive calls made, and the value returned at each stage, for the call Mystery(4). **(5)**

(c) Explain, using the term "base case", why calling Mystery(0) would cause a runtime error, and state the name commonly given to this type of error. **(3)**

(d) Rewrite Mystery as an iterative (non-recursive) function, MysteryIterative, using a FOR loop, that calculates the same result for positive integer values of n. **(4)**

(Total for Question 8 is 13 marks)

9 A computer system uses 8-bit binary representation for integers.

(a) Convert the denary number 179 to 8-bit binary. **(2)**

(b) Convert the binary number 10110011 to hexadecimal. **(2)**

(c) A system stores signed integers using 8-bit two's complement. State the denary value represented by 11101001. **(3)**

(d) Explain why overflow occurs when adding the 8-bit two's complement numbers 01111111 and 00000001, and state the effect this has on the result. **(3)**

(e) State the number of distinct values that can be represented using 8-bit two's complement. **(1)**

(Total for Question 9 is 11 marks)

10 A program is required to input 10 integer temperature readings (in degrees Celsius) and output: the number of days above 20 degrees Celsius, and whether the average of all 10 readings is a whole number.

(a) Write pseudocode for a complete program that performs this task, including reading 10 temperatures, counting the number above 20 degrees Celsius, calculating the average, and outputting whether the average is a whole number. **(8)**

(b) The program validates each temperature reading using the rule -50 ≤ temperature ≤ 50. Identify one appropriate test value for each of: normal data, boundary data, and erroneous data, for this validation rule. **(3)**

(c) Explain why boundary test data is important when testing this validation rule. **(2)**

(Total for Question 10 is 13 marks)

Mark scheme · 2.2 Problem Solving and Programming

Question 1

(a) bonus = 30, cao **B1**

(a) Answer: 30

(b) IF grade = "A" THEN bonus <- 50 **M1**

(b) nested ELSE IF grade = "B" THEN bonus <- 30 **M1**

(b) nested ELSE IF grade = "C" THEN bonus <- 10 **M1**

(b) final ELSE bonus <- 0, with all IF structures correctly closed with ENDIF; any correct logic, ignore minor syntax **A1**

(b) Answer:

```
IF grade = "A" THEN
    bonus <- 50
ELSE
    IF grade = "B" THEN
        bonus <- 30
    ELSE
        IF grade = "C" THEN
            bonus <- 10
        ELSE
            bonus <- 0
        ENDIF
    ENDIF
ENDIF
```

(c) identifies benefit, e.g. CASE is more readable/easier to follow when testing many discrete values of the same variable, oe **B1**

(c) expands the point, e.g. avoids deeply nested IF statements, which are harder to trace, maintain and debug, oe **B1**

(c) Answer: CASE is clearer than deeply nested IF statements when checking several discrete values of one variable, reducing the chance of logic errors.

(d) identifies that the "A" branch uses "=" instead of the assignment operator **B1**

(d) states the correction: bonus <- 50 **B1**

(d) Answer: The line "A" : bonus = 50 uses "=" (comparison) instead of "<-" (assignment); it should read "A" : bonus <- 50

Question 2

(a) correct 2D array syntax with bounds [1:3, 1:4], oe equivalent bounds **B1**

(a) correct data type OF INTEGER **B1**

(a) Answer: DECLARE scores : ARRAY[1:3, 1:4] OF INTEGER

(b) total <- 0 initialised before the loop **M1**

(b) FOR col <- 1 TO 4 loop **M1**

(b) total <- total + scores[2, col] inside the loop, correctly indexing row 2 **M1**

(b) average <- total / 4 and OUTPUT average, correct value 82.5; any correct logic, ignore minor syntax **A1**

(b) Answer:

```
total <- 0
FOR col <- 1 TO 4
    total <- total + scores[2, col]
NEXT col
average <- total / 4
OUTPUT average
```

Output = 82.5

(c) after row 1: total = 235 **B1**

(c) after row 2: total = 565 **B1**

(c) after row 3: total = 785 **B1**

(c) final OUTPUT value = 785, cao **B1**

(c) Answer: 785

(d) 90, cao **B1**

(d) Answer: 90

Question 3

(a) checks LEN(code) = 7 and sets valid to false if not **M1**

(a) checks position 3 equals "-" using MID(code, 3, 1) **M1**

(a) loop/checks over positions 1-2, confirming each is a letter (e.g. using ASC ranges for "A" to "Z") **M1**

(a) loop/checks over positions 4-7, confirming each is a digit (e.g. using ASC ranges for "0" to "9") **M1**

(a) correct use of MID to extract each character being tested **M1**

(a) returns a single BOOLEAN value combining all checks correctly; any correct logic, ignore minor syntax **A1**

(a) Answer:

```
FUNCTION ValidateCode(code : STRING) RETURNS BOOLEAN
    valid <- true
    IF LEN(code) <> 7 THEN
        valid <- false
    ELSE
        IF MID(code, 3, 1) <> "-" THEN
            valid <- false
        ENDIF
        FOR i <- 1 TO 2
            letter <- MID(code, i, 1)
            IF (ASC(letter) < ASC("A")) OR (ASC(lette
                valid <- false
            ENDIF
        NEXT i
        FOR i <- 4 TO 7
            digit <- MID(code, i, 1)
            IF (ASC(digit) < ASC("0")) OR (ASC(digit)
                valid <- false
            ENDIF
        NEXT i
    ENDIF
    RETURN valid
ENDFUNCTION
```

(b) "1234", cao **B1**

(b) Answer: "1234"

(c) "AB", cao **B1**

(c) Answer: "AB"

(d) validation only checks that data conforms to a set of format rules (length, character types, hyphen position), oe **B1**

(d) it does not check the data represents a real/correct product, e.g. "ZZ-9999" could pass validation without being an actual issued code, oe **B1**

(d) Answer: Validation checks the code's format, not whether it corresponds to a real product; a well-formatted but invented code would still pass.

Question 4

(a) num1 = 20 **B1**

(a) num2 = 15 **B1**

(a) Answer: num1 = 20, num2 = 15

(b) passing by reference passes the memory address/reference of the actual variable into the subroutine, oe **B1**

(b) so changes made to the parameter (x <- x * 2) directly change the original variable, num1, giving 20, oe **B1**

(b) passing by value copies the value of the argument into a new, separate local variable, oe **B1**

(b) so changes to y inside AddFive do not directly change num2; num2 only changes because the returned value is explicitly assigned back to it, oe **B1**

(b) Answer: By reference links the parameter directly to the caller's variable, so DoubleValue permanently changes num1. By value copies the data, so AddFive cannot change num2 directly; num2 is only updated because its new value is returned and reassigned.

(c) correct FUNCTION header with a by-value parameter (no BYREF) and RETURNS INTEGER **M1**

(c) correct calculation, e.g. RETURN x * 2 or result <- x * 2 **M1**

(c) correct RETURN statement, and the original argument is not modified since the parameter is passed by value; any correct logic, ignore minor syntax **A1**

(c) Answer:

```
FUNCTION DoubleValueFunc(x : INTEGER) RETURNS INTEGER
    RETURN x * 2
ENDFUNCTION
```

(d) a procedure is appropriate when the subroutine performs an action (e.g. output, or modifying data via reference parameters) and does not need to return a single value, oe **B1**

(d) Answer: Procedures are used when no value needs to be returned, e.g. the subroutine only needs to perform an action or modify parameters passed by reference.

Question 5

(a) iteration 1: lower=1, upper=8, mid=4, list[4]=19; 19<24 so lower becomes 5 **B1**

(a) iteration 2: lower=5, upper=8, mid=6, list[6]=31; 31>24 so upper becomes 5 **B1**

(a) iteration 3: lower=5, upper=5, mid=5, list[5]=24; match found **B1**

(a) correct comparison/decision applied at each step (equal / less than / greater than target) **B1**

(a) final index returned = 5, cao **B1**

(a) Answer: Index 5 (list[5] = 24)

(b) 3, cao **B1**

(b) Answer: 3

(c) binary search relies on the data being sorted so that comparing the middle value with the target tells you which half to discard, oe **B1**

(c) on unsorted data this less-than/greater-than comparison is not reliable, so a whole half of the array could be discarded even though it contains the target, oe **B1**

(c) Answer: Binary search assumes ordering to eliminate half the search space each time; on unsorted data that assumption fails and the target could be discarded incorrectly.

(d) O(log n) **B1**

(d) explains that binary search halves the search space at each step, whereas linear search may check every item, oe **B1**

(d) as n grows very large, log n grows far more slowly than n, so binary search needs far fewer comparisons in the worst case, oe **B1**

(d) Answer: O(log n); it is faster than O(n) linear search for large n because each comparison eliminates half the remaining data rather than checking one item at a time.

(e) loop through each index of the array, e.g. FOR i <- 1 TO 8 **M1**

(e) compares list[i] to target and returns i as soon as a match is found **M1**

(e) returns -1 after the loop completes with no match found; any correct logic, ignore minor syntax **A1**

(e) Answer:

```
FUNCTION LinearSearch(list : ARRAY[1:8] OF INTEGER, t
    FOR i <- 1 TO 8
        IF list[i] = target THEN
            RETURN i
        ENDIF
    NEXT i
    RETURN -1
ENDFUNCTION
```

Question 6

(a) pass 1: [3, 6, 1, 8], swaps = 3 **B1**

(a) pass 2: [3, 1, 6, 8], swaps = 1 **B1**

(a) pass 3: [1, 3, 6, 8], swaps = 1 **B1**

(a) pass 4: [1, 3, 6, 8], swaps = 0 (array already sorted) **B1**

(a) Answer: Sorted array: [1, 3, 6, 8]

(b) use a boolean flag (e.g. swapped) that is reset to false at the start of each pass, oe **B1**

(b) if no swaps occur during a full pass, the flag stays false, meaning the array is fully sorted, so the algorithm can stop early instead of running further unnecessary passes, oe **B1**

(b) Answer: Use a flag that records whether any swap occurred in a pass; if a whole pass completes with no swaps, the array is sorted and the algorithm can terminate early.

(c) outer loop controlled by a swapped flag, e.g. WHILE swapped = true **M1**

(c) swapped is reset to false at the start of each pass **M1**

(c) inner FOR loop comparing each pair of adjacent elements, e.g. FOR j <- 1 TO n - 1 **M1**

(c) correct swap of arr[j] and arr[j+1] using a temporary variable when arr[j] > arr[j+1] **M1**

(c) swapped is set to true whenever a swap occurs **M1**

(c) algorithm correctly terminates once a full pass makes no swaps; any correct logic, ignore minor syntax **A1**

(c) Answer:

```
PROCEDURE BubbleSort(BYREF arr : ARRAY[1:n] OF INTEGE
    DECLARE i, j, temp : INTEGER
    DECLARE swapped : BOOLEAN
    swapped <- true
    WHILE swapped = true
        swapped <- false
        FOR j <- 1 TO n - 1
            IF arr[j] > arr[j + 1] THEN
                temp <- arr[j]
                arr[j] <- arr[j + 1]
                arr[j + 1] <- temp
                swapped <- true
            ENDIF
        NEXT j
    ENDWHILE
ENDPROCEDURE
```

(d) $O(n^2)$ **B1**

(d) Answer: $O(n^2)$

Question 7

(a) Push (add data) **B1**

(a) Pop (remove data) **B1**

(a) Answer: Push and Pop

(b) Push(5): top=1, stack=[5] **B1**

(b) Push(9): top=2, stack=[5,9] **B1**

(b) Pop() returns 9: top=1, stack=[5] **B1**

(b) Push(3) then Push(7): top=3, stack=[5,3,7] **B1**

(b) Pop() returns 7: top=2, stack=[5,3] (final state) **B1**

(b) Answer: Final state: top = 2, stack contents [5, 3]

(c) a stack is LIFO (last in, first out), oe **B1**

(c) each opening bracket is pushed onto the stack, and each closing bracket should match (pop) the most recently pushed, still-unmatched opening bracket, oe **B1**

(c) if the stack is empty at the end (and never needed to pop from empty) the brackets are balanced; a queue's FIFO order would not correctly match the most recent unmatched bracket, oe **B1**

(c) Answer: A stack's LIFO behaviour naturally matches each closing bracket with the most recently opened, still-unclosed bracket; a FIFO queue would match brackets in the wrong order.

(d) any valid example, e.g. a print queue, a task/process scheduling queue, a keyboard input buffer **B1**

(d) Answer: Example: a print queue, where documents are printed in the order they were sent.

Question 8

(a) n factorial (n!), for positive integer n **B1**

(a) Answer: n factorial

(b) Mystery(4) calls Mystery(3), which calls Mystery(2), which calls Mystery(1), correctly shown in sequence **B1**

(b) Mystery(1) is the base case (n=1) and returns 1 **B1**

(b) Mystery(2) returns 2 * 1 = 2 **B1**

(b) Mystery(3) returns 3 * 2 = 6 **B1**

(b) Mystery(4) returns 4 * 6 = 24, cao **B1**

(b) Answer: Mystery(4) = 24

(c) the base case is n = 1, but starting from n = 0 and repeatedly subtracting 1 gives 0, -1, -2, ... which never exactly equals 1, oe **B1**

(c) so the base case is never reached and the function calls itself indefinitely (infinite recursion), oe **B1**

(c) each call uses memory on the call stack; because the recursion never stops, this is called a stack overflow **B1**

(c) Answer: Because n decreases past 1 without ever equalling it, the base case (n=1) is never reached, causing infinite recursion and a stack overflow.

(d) result initialised to 1 before the loop **M1**

(d) FOR loop iterating from 1 to n (e.g. FOR i <- 1 TO n) **M1**

(d) result <- result * i inside the loop **M1**

(d) correct RETURN result after the loop; any correct logic, ignore minor syntax **A1**

(d) Answer:

```
FUNCTION MysteryIterative(n : INTEGER) RETURNS INTEGE
    result <- 1
    FOR i <- 1 TO n
        result <- result * i
    NEXT i
    RETURN result
ENDFUNCTION
```

Question 9

(a) correct place values used (128,64,32,16,8,4,2,1) with correct bits identified **M1**

(a) 10110011, cao **A1**

(a) Answer: 10110011

(b) splits into two 4-bit nibbles: 1011 and 0011 **M1**

(b) B3, cao **A1**

(b) Answer: B3

(c) recognises the MSB is 1, so the number is negative **M1**

(c) correctly inverts the bits and adds 1 (00010110 + 1 = 00010111), or uses correct negative place-value weighting **M1**

(c) -23, cao **A1**

(c) Answer: -23

(d) 01111111 represents the largest positive value storable in 8-bit two's complement (127), oe **B1**

(d) binary addition gives 10000000, but with two's complement this bit pattern represents -128, oe **B1**

(d) the result of adding two positive numbers has incorrectly become negative because there are not enough bits to store the true result (128), which is overflow, oe **B1**

(d) Answer: 01111111 + 00000001 = 10000000, which as two's complement represents -128, not the correct positive result (128); this incorrect sign change is overflow, caused by the result exceeding the range that 8 bits can represent.

(e) 256 (2^8) **B1**

(e) Answer: 256

Question 10

(a) total and count20 initialised to 0 before the loop **M1**

(a) FOR loop running exactly 10 times **M1**

(a) INPUT temp inside the loop **M1**

(a) running total correctly updated (total <- total + temp) **M1**

(a) correct selection test (temp > 20) and count20 incremented when true **M1**

(a) count20 correctly output after the loop **M1**

(a) average correctly calculated as total / 10 **M1**

(a) correct test for whether the average is a whole number (e.g. total MOD 10 = 0) with an appropriate message output in both cases; any correct logic, ignore minor syntax **A1**

(a) Answer:

```
total <- 0
count20 <- 0
FOR i <- 1 TO 10
    INPUT temp
    total <- total + temp
    IF temp > 20 THEN
        count20 <- count20 + 1
    ENDIF
NEXT i
OUTPUT "Days above 20 degrees: ", count20
average <- total / 10
IF total MOD 10 = 0 THEN
    OUTPUT "The average is a whole number: ", average
ELSE
    OUTPUT "The average is not a whole number: ", ave
ENDIF
```

(b) normal data: any value clearly within range, e.g. 15 (should be accepted) **B1**

(b) boundary data: a value exactly on a limit, e.g. 50 or -50 (should be accepted) **B1**

(b) erroneous data: a value outside the range or wrong type, e.g. 75 or "hot" (should be rejected) **B1**

(b) Answer: Normal: 15. Boundary: 50 (or -50). Erroneous: 75 (or a non-numeric value such as "hot").

(c) boundary values test the exact edges of the accepted range, which is where off-by-one errors (e.g. using < instead of ≤) are most likely to occur, oe **B1**

(c) testing values exactly on the limit confirms whether the validation correctly accepts (or rejects) them as intended by the specification, oe **B1**

(c) Answer: Boundary values check that the comparison operators (≤, ≥) are correct at the exact edges of the valid range, where logic errors such as excluding a valid limit value are most likely.