Cambridge Assessment International Education

Cambridge International AS & A Level

COMPUTER SCIENCE **9618/22**

Paper 2 Problem Solving & Programming **October/November 2022**

MARK SCHEME

Maximum Mark: 75

Published

This mark scheme is published as an aid to teachers and candidates, to indicate the requirements of the examination. It shows the basis on which Examiners were instructed to award marks. It does not indicate the details of the discussions that took place at an Examiners' meeting before marking began, which would have considered the acceptability of alternative answers.

Mark schemes should be read in conjunction with the question paper and the Principal Examiner Report for Teachers.

Cambridge International will not enter into discussions about these mark schemes.

Cambridge International is publishing the mark schemes for the October/November 2022 series for most Cambridge IGCSE™, Cambridge International A and AS Level components and some Cambridge O Level components.

Generic Marking Principles

These general marking principles must be applied by all examiners when marking candidate answers. They should be applied alongside the specific content of the mark scheme or generic level descriptors for a question. Each question paper and mark scheme will also comply with these marking principles.

GENERIC MARKING PRINCIPLE 1:

Marks must be awarded in line with:

- the specific content of the mark scheme or the generic level descriptors for the question
- the specific skills defined in the mark scheme or in the generic level descriptors for the question
- the standard of response required by a candidate as exemplified by the standardisation scripts.

GENERIC MARKING PRINCIPLE 2:

Marks awarded are always **whole marks** (not half marks, or other fractions).

GENERIC MARKING PRINCIPLE 3:

Marks must be awarded **positively**:

- marks are awarded for correct/valid answers, as defined in the mark scheme. However, credit is given for valid answers which go beyond the scope of the syllabus and mark scheme, referring to your Team Leader as appropriate
- marks are awarded when candidates clearly demonstrate what they know and can do
- marks are not deducted for errors
- marks are not deducted for omissions
- answers should only be judged on the quality of spelling, punctuation and grammar when these features are specifically assessed by the question as indicated by the mark scheme. The meaning, however, should be unambiguous.

GENERIC MARKING PRINCIPLE 4:

Rules must be applied consistently, e.g. in situations where candidates have not followed instructions or in the application of generic level descriptors.

GENERIC MARKING PRINCIPLE 5:

Marks should be awarded using the full range of marks defined in the mark scheme for the question (however; the use of the full mark range may be limited according to the quality of the candidate responses seen).

GENERIC MARKING PRINCIPLE 6:

Marks awarded are based solely on the requirements as defined in the mark scheme. Marks should not be awarded with grade thresholds or grade descriptors in mind.

<table>
<tr><th>Question</th><th>Answer</th><th>Marks</th></tr>
<tr><td>1(a)(i)</td><td>One mark for each point (Max 2):

• When a task which is repeated / reused / performed in several places
• When a part of an algorithm performs a specific task
• Reduces complexity of program / program is simplified // subroutine already available
• Testing / debugging / maintenance is easier</td><td>2</td></tr>
<tr><td>1(a)(ii)</td><td>One mark for each part:

Term: Parameter(s)

Use: to pass values / arguments to the procedure</td><td>2</td></tr>
<tr><td>1(b)</td><td>One mark for test stage, one mark for each description point
(Max 3 for Description)

Test stage: Beta testing

Description:

1 Testing carried out by a small group of (potential) users
2 Users will check that the software works as required / works in the real world / does not contain errors
3 Users will feedback problems / suggestions for improvement
4 Problems / suggestions identified are addressed (before the program is sold)</td><td>4</td></tr>
<tr><td>1(c)</td><td>One mark per row:

<table>
<tr><th>Expression</th><th>Evaluation</th></tr>
<tr><td><code>MID(CharList, MONTH(FlagDay), 1)</code></td><td>'D'</td></tr>
<tr><td><code>INT(Count / LENGTH(CharList))</code></td><td>4</td></tr>
<tr><td><code>(Count >= 99) AND (DAY(FlagDay) > 23)</code></td><td>FALSE</td></tr>
</table>
</td><td>3</td></tr>
</table>

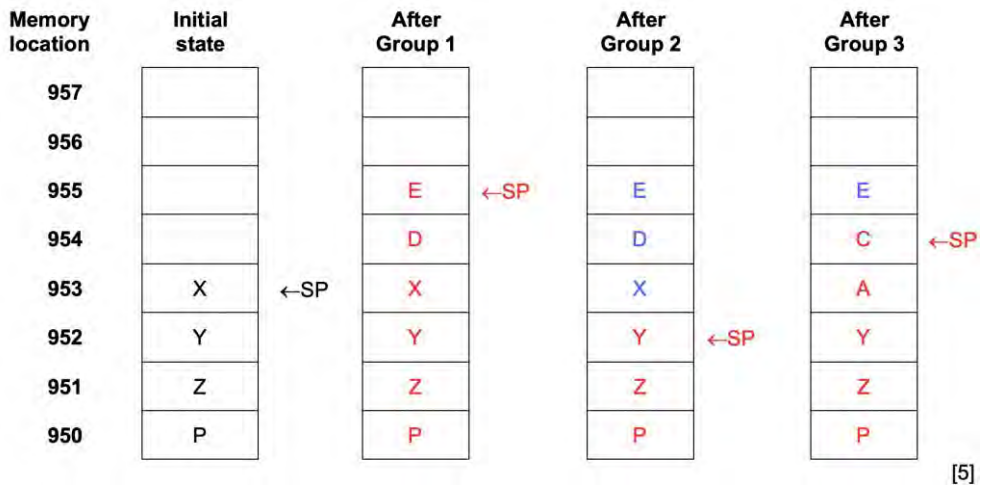
<table>
<tr><th>Question</th><th>Answer</th><th>Marks</th></tr>
<tr><td>2(a)(i)</td><td>One mark per step (or equivalent):

1 Open file in APPEND mode (and subsequent Close)
2 Prompt and Input a student name and mark
3 If mark greater than or equal to 20 jump to step 5
4 Write only the name to the file
5 Repeat from Step 2 for 35 times / the number of students</td><td>5</td></tr>
<tr><td>2(a)(ii)</td><td>Data in a file is saved after the computer is switched off / stored permanently // no need to re-enter the data when the program is re-run</td><td>1</td></tr>
<tr><td>2(a)(iii)</td><td>Example answer:

So that existing file data is not overwritten.</td><td>1</td></tr>
<tr><td>2(b)</td><td>One mark per row (row 2 to 5):

<table>
<tr><th>Input</th><th>Output</th><th>Next state</th></tr>
<tr><td></td><td></td><td>S1</td></tr>
<tr><td>Input-A</td><td>Output-X</td><td>S2</td></tr>
<tr><td>Input-A</td><td>(none)</td><td>S2</td></tr>
<tr><td>Input-B</td><td>Output-W</td><td>S3</td></tr>
<tr><td>Input-A</td><td>Output-W</td><td>S4</td></tr>
</table>
</td><td>4</td></tr>
</table>

Question	Answer	Marks
3(a)	One mark for name, Max two for features (**Max 3 in total**) Name: Queue Features: 1 Each queue element contains one data item 2 A Pointer to the front / start of the queue 3 A Pointer to the back / end of the queue 4 Data is added at back / end and removed from front / start // works on a FIFO basis 5 May be circular **ALTERNATIVE:** Name: Linked List Features: 1 Each node contains data and a pointer to the next node 2 A Pointer to the start of the list 3 Last node in the list has a null pointer 4 Data may be added / removed by manipulating pointers (not moving data) 5 Nodes are traversed in a specific sequence 6 Unused nodes are stored on a free list // a free-list pointer to the Free List	**3**
3(b)	One mark per point (**Max 5**): 1 Declare a (1D) array of data type `STRING` 2 The number of elements in that array corresponds to the size of the required stack 3 Declare an integer / variable for `StackPointer` 4. Declare an integer / variable for the size of the stack // for the max value of `StackPointer` 5 Use the `StackPointer` as an index to the array 6 Pointers and variables initialised to indicate empty stack 7 Store each item on the stack as one array element / Each stack item maps to one array element 8 Attempt to describe Push **and** Pop operations 9 Push **and** Pop routines need to check for full or empty conditions	**5**

Question	Answer	Marks
3(c)	One mark for each: 1 Data 'After Group 1' (as shown, including blank cells) 2 Data 'After Group 2' (as shown, including blank cells) 3 Data 'After Group 3' (as shown, including blank cells) 4 SP 'After Group 1' pointing to location 955 5 Final two SPs pointing to locations 952 and 954  [5]	**5**

Question	Answer	Marks

4(a)	One mark per point:
	
	1 Input `UserID` **and** use of `GetAverage()` **and** assignment
	2 Initialisation of `Total` to zero **and** `Index` to 4
	3 Conditional loop with `Index` from 4 to 6
	4 Assignment of `Last` from element `SameMonth[Index]` **in a loop**
	5 Structure: `IF...THEN...ELSE...ENDIF` **in a loop**
	6 Correct assignments **and** final call to `Update()` **after the loop**
	
	`INPUT UserID`
	`Average ← GetAverage(UserID)`
	`Total ← 0`
	`Index ← 4`
	
	`WHILE Index < 7 // REPEAT`
	`  Last ← SameMonth[Index]`
	`  IF Average > Last THEN`
	`  Total ← Total + Average`
	`  ELSE`
	`  Total ← Total + Last`
	`  ENDIF`
	`  Index ← Index + 1`
	`ENDWHILE // UNTIL Index = 7`
	
	`CALL Update(UserID, Total)`
	
	<u>Alternative solution using FOR loop</u>:
	
	One mark per point FOR loop solution:
	1 Input `UserID` **and** use of `GetAverage()` **and** assignment
	2 Initialisation of `Total` to zero
	3 loop `Index` from 4 to 6
	4 Assignment of `Last` from array `SameMonth` **in a loop**
	5 Comparison `IF...THEN...ELSE...ENDIF` **in a loop**
	6 Appropriate assignments **in a loop AND** final call to `Update()` **after the loop**
	
	`INPUT UserID`
	`Average ← GetAverage(UserID)`
	`Total ← 0`
	`FOR Index ← 4 TO 6`
	`  Last ← SameMonth[Index]`
	`  IF Average > Last THEN`
	`  Total ← Total + Average`
	`  ELSE`
	`  Total ← Total + Last`
	`  ENDIF`
	`NEXT Index`
	
	`CALL Update(UserID, Total)`
		6

Question	Answer	Marks
4(b)	Pre-condition (loop) / count-controlled (loop)	**1**

Question	Answer	Marks

5	One mark per `IF...THEN...ENDIF` clause:
	
	`1 IF A AND B AND C THEN`
	`  CALL Sub1()`
	`ENDIF`
	
	`2 IF (A AND B) AND NOT C THEN`
	`  CALL Sub2()`
	`ENDIF`
	
	`3 IF (NOT A) AND (NOT C) THEN`
	`  CALL Sub3()`
	`ENDIF`
	
	`4 IF (NOT A) AND C THEN`
	`  CALL Sub4()`
	`ENDIF`
		4

Question	Answer	Marks

6(a)	Example by repeated multiplication:
	
	Mark as follows (multiplication solution), (**Max 7**):
	
	1 Function heading and ending **including** parameter and return type
	2 Declaration and initialisation of local `Num`
	3 Any conditional loop
	4 Conditional loop until `ThisValue` found or `Try` out of range
	5 Multiply `Try` by `Num` **in a loop**
	6 Compare `Try` with `ThisValue` **and** set termination if the same **in a loop**
	7 Increment `Num` and repeat **in a loop**
	8 Attempt to Return `Num` if `ThisValue` is a factorial or -1 otherwise
	
	`FUNCTION FindBaseNumber(ThisValue : INTEGER) RETURNS INTEGER`
	`  DECLARE Num, Try : INTEGER`
	`  DECLARE Found : BOOLEAN`
	
	`  Num ← 0`
	`  Found ← FALSE`
	`  Try ← 1`
	
	`   WHILE Try <= ThisValue AND Found = FALSE`
	`      Num ← Num + 1`
	`      Try ← Try * Num`
	`      IF Try = ThisValue THEN //BaseNumber found`
	`         Found ← TRUE`
	`      ENDIF`
	`   ENDWHILE`
	
	`   IF Found = TRUE THEN`
	`      RETURN Num`
	`   ELSE`
	`      RETURN -1`
	`   ENDIF`
	
	`ENDFUNCTION`
		7

<table>
<tr><th>Question</th><th>Answer</th><th>Marks</th></tr>
<tr><td>6(a)</td><td>Alternative FOR LOOP solution.

Mark as follows:
1 Function heading and ending including parameter and return type
2 Declaration of local Integer value for <code>Num</code> and <code>Try</code>
3 Count-controlled Loop from 1 to <code>ThisValue</code>
4 Multiply <code>Try</code> by <code>Num</code> in a loop
5 Compare <code>Try</code> with <code>ThisValue</code> in a loop
6 ...Immediate return of <code>Num</code> if <code>they are the same</code> in a loop
7 Return –1 if <code>ThisValue</code> not found after loop

<pre>FUNCTION FindBaseNumber(ThisValue : INTEGER) RETURNS INTEGER
   DECLARE Num, Try : INTEGER

   Try ← 1

   FOR Num ← 1 TO ThisValue
      Try ← Try * Num
      IF Try = ThisValue THEN //BaseNumber found
         RETURN Num
      ENDIF
   NEXT Num

   RETURN -1

ENDFUNCTION</pre></td><td></td></tr>
<tr><td>6(b)</td><td>One mark per row.

Examples of invalid strings:

1 Non-numeric but not "End" // contains space or other non-numeric characters
2 Real number
3 Integer value out of range (i.e. <= 0)
4 Empty string
5 Correct word but wrong case e.g. "end" rather than "End"

<table><tr><th>Input</th><th>Reason for choice</th></tr><tr><td>"Aardvark"</td><td>Non-numeric (and not "End")</td></tr><tr><td>"27.3"</td><td>Numeric but not an integer</td></tr><tr><td>"-3" // "0"</td><td>A non-positive integer</td></tr><tr><td>""</td><td>An empty string</td></tr></table></td><td>4</td></tr>
</table>

Question	Answer	Marks

7(a)	One mark per point (**Max 6**):
	
	1 Procedure heading and ending **including** parameters
	2 Conditional loop containing incrementing `Index`...
	3 ...terminating when `ErrNum` found
	4 ...terminating when `ErrCode[Index] > ErrNum` (i.e. `ErrNum` not found)
	5 ... **OR** after element 500 tested
	6 Test if found and OUTPUT 'Found' message
	7 ...otherwise OUTPUT 'Not Found' message
	
	`PROCEDURE OutputError(LineNum, ErrNum : INTEGER)`
	`  DECLARE Index : INTEGER`
	
	`  Index ← 0`
	
	`  // Search until ErrNum found OR not present OR end of array`
	
	`  REPEAT`
	`  Index ← Index + 1`
	`  UNTIL ErrCode[Index] >= ErrNum OR Index = 500`
	
	`  IF ErrCode[Index] = ErrNum THEN`
	`  OUTPUT ErrText[Index], " on line ", LineNum //Found`
	`  ELSE`
	`  OUTPUT "Unknown error on line ", LineNum //Not found`
	`  ENDIF`
	
	`ENDPROCEURE`
		6

Question	Answer	Marks

7(b)	One mark per point (**Max 8**):
	
	1 Procedure heading and ending as shown
	2 Conditional loop correctly terminated
	3 An inner loop
	4 Correct range for inner loop
	5 Comparison (element J with J+1) **in a loop**
	6 Swap elements in **both** arrays **in a loop**
	7 'No-Swap' mechanism:
	• Conditional **outer** loop including flag reset
	• Flag set in **inner** loop to indicate swap
	8 Efficiency (this scenario): terminate inner loop when `ErrCode = 999`
	9 Reducing `Boundary` **in the <u>outer</u> loop**
	
	`PROCEDURE SortArrays()`
	`  DECLARE TempInt, J, Boundary : INTEGER`
	`  DECLARE TempStr : STRING`
	`  DECLARE NoSwaps : BOOLEAN`
	
	`  Boundary ← 499`
	
	`  REPEAT`
	`  NoSwaps ← TRUE`
	`  FOR J ← 1 TO Boundary`
	`  IF ErrCode[J]> ErrCode[J+1] THEN`
	`  //first swap ErrCode elements`
	`  TempInt ← ErrCode[J]`
	`  ErrCode[J] ← ErrCode[J+1]`
	`  ErrCode[J+1] ← TempInt`
	`  //now swap corresponding ErrText elements`
	`  TempStr ← ErrText[J]`
	`  ErrText[J] ← ErrText[J+1]`
	`  ErrText[J+1] ← TempStr`
	`  NoSwaps ← FALSE`
	`  ENDIF`
	`  NEXT J`
	`  Boundary ← Boundary - 1`
	`  UNTIL NoSwaps = TRUE`
	
	`ENDPROCEDURE`
		8

7(c)(i)	`ErrCode` should be an `INTEGER` // `ErrCode` should not be a `STRING`	**1**

Question	Answer	Marks
7(c)(ii)	Benefits include: 1 Array of records can store mixed data types / multiple data types under a single identifer 2 Tighter / closer association between `ErrCode` and `ErrText` // simpler code as fields may be referenced together // values cannot get out of step as with two arrays 3 Program easier to design / write / debug / test / maintain / understand One mark per point **Note: Max 2 marks**	**2**
7(c)(iii)	`DECLARE Error : ARRAY[1:500] OF ErrorRec`	**1**