4(a)

Example solution - conditional:

```
PROCEDURE Store()
   DECLARE ThisNum, LastNum : REAL
   DECLARE Index : INTEGER // index to global array

   INPUT ThisNum
   LastNum ← ThisNum
   Number[1] ← ThisNum

   INPUT ThisNum
   Index ← 2

   WHILE Index < 21 AND ThisNum > LastNum
      Number[Index] ← ThisNum
      LastNum ← ThisNum
      Index ← Index + 1
      IF Index < 21 THEN
         INPUT ThisNum
      ENDIF
   ENDWHILE

   OUTPUT Index - 1, " values were stored in the array"
ENDPROCEDURE
```

Alternative loop:

```
   // Loop without using LastNum
   WHILE Index < 21 AND ThisNumNum > Number[Index - 1]
      Number[Index] ← ThisNum
      Index ← Index + 1
      IF Index < 21 THEN      //skip input if array full
         INPUT ThisNum
      ENDIF
   ENDWHILE
```

Mark as follows:

1 Declare local variables `Index` as integer, `ThisNum` and `LastNum` as real
2 Input value and assign to first element
3 **Attempt** at conditional loop including **both** tests
4 **Completely correct** conditional loop including **both** tests
5 Assign input value to current element if greater than previous element **and** input next value **in a loop**
6 Final output giving attempted count of elements stored plus message **after a loop**

4(a)

Alternative FOR loop example solution:

```
PROCEDURE Store()
   DECLARE ThisNum, LastNum : REAL
   DECLARE Index : INTEGER // index to global array
   DECLARE Count : INTEGER
   INPUT ThisNum
   LastNum ← ThisNum
   Number[1] ← ThisNum
   Count ← 1
   INPUT ThisNum
   FOR Index ← 2 TO 20
      IF ThisNum > LastNum THEN
         Number[Index] ← ThisNum
         LastNum ← ThisNum
         Count ← Count + 1
         IF Index < 20 THEN
            INPUT ThisNum
         ENDIF
      ELSE
         BREAK
      ENDIF
   NEXT Index
   OUTPUT Count, " values were stored in the array"
ENDPROCEDURE
```

Alternative loop:

```
   Count ← 1
   INPUT ThisNum
   FOR Index ← 2 TO 20
      IF ThisNum > Number[Index - 1] THEN
         Number[Index] ← ThisNum
         Count ← Count + 1
         IF Index < 20 THEN
            INPUT ThisNum
         ENDIF
      ELSE
         BREAK
      ENDIF
   NEXT Index
```

Mark as follows:

1 Declare local variables `Index` as integer, `ThisNum` and `LastNum` as real
2 Input value and assign to first element
3 Count-controlled loop
4 Count-controlled loop **with BREAK** if current value greater than previous value
5 ... Assign input value to current element **following correct comparison in a loop**
6 Final output giving count of elements stored plus message following a reasonable attempt **after loop**

4(b)(i)	One mark per point: 1 The amount of values **stored** (in a text file) is **not fixed** / **not limited** (other than by secondary storage available) // The amount values **stored** is **not limited** to the size of the array 2 The data is stored **permanently** / **non-volatile** // Data can be restored / reused without **re-entering** values
4(b)(ii)	Each (real) value would have to be **converted** to a string

6 Example solution:

```
FUNCTION Compare(String1, String2 : STRING, __
          CaseMatters : BOOLEAN, Position : CHAR) RETURNS
BOOLEAN
   DECLARE S1Length : INTEGER

   S1Length ← LENGTH(String1)
   IF LENGTH(String2) < S1Length THEN
      RETURN FALSE // String2 is shorter than String1
   ENDIF


   IF CaseMatters = FALSE THEN
      String1 ← TO_UPPER(String1)
      String2 ← TO_UPPER(String2)
   ENDIF

   CASE Position
      'e'  : String2 ← RIGHT(String2,  S1Length)
      's'  : String2 ← LEFT(String2,  S1Length)
   ENDCASE

   IF String1 = String2 THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   ENDIF

ENDFUNCTION
```

Mark as follows:

MP1 Function heading **and** parameters **and** ending **and** return type
MP2 Calculate length of `String1`
MP3 **if** `String2` is shorter than `String1`, return `FALSE`
MP4 Compare (modified) strings
MP5 Test `CaseMatters` - if `FALSE` **and** convert two string(s) to same case
MP6 Test for the value of 's'/'e' **and** form modified string(s) **in one** or both cases
MP7 Return appropriate `BOOLEAN` value in all cases

8(a)

Example solution:

```
PROCEDURE CountLoans(ThisStudentID : STRING)
   DECLARE Index, LCount, RCount : INTEGER

   LCount ← 0
   RCount ← 0

   FOR Index ← 1 TO 7000
      IF Loan[Index].StudentID = ThisStudentID THEN
         IF Loan[Index].OnLoan = FALSE THEN
            RCount ← RCount + 1
         ELSE
            LCount ← LCount + 1
         ENDIF
      ENDIF
   NEXT Index

   OUTPUT "Student has ", LCount, " books on loan"
   OUTPUT "and has returned ", RCount, " books"

ENDPROCEDURE
```

Mark as follows:

MP1 Declaration of Index **and** two count variables
MP2 Loop for 7000 iterations
MP3 `Index` references an **indexed data** item with correct **.dot notation**
MP4 Attempt to compare the `StudentID` field = **parameter in a loop**
MP5 Test of the `OnLoan` field **and** correct logic for both outcomes
MP6 Both count variables initialised **and** increment appropriate count **in a loop**
MP7 Output includes both counts with a <u>suitable</u> message

8(b)

Example solution:

```
FUNCTION NewLoan(ThisStudentID, ThisBookID : STRING)
                                                    RETURNS
BOOLEAN
   DECLARE Index : INTEGER
   DECLARE IsStored : BOOLEAN
   CONSTANT Blank = ""

   Index ← 1
   IsStored ← FALSE

   WHILE Index <= 7000 AND NOT IsStored
      IF Loan[Index].StudentID = Blank THEN
         Loan[Index].StudentID ← ThisStudentID
         Loan[Index].BookID ← ThisBookID
         Loan[Index].OnLoan ← TRUE
         IsStored ← TRUE
      ENDIF
      Index ← Index + 1
   ENDWHILE

   RETURN IsStored
ENDFUNCTION
```

Mark as follows:

MP1 Loop through 7000 elements in `Loan` array – correct loop structure
MP2 or until the first unused element is found // Correct `RETURN` inside a `FOR` loop
MP3 Correct use of the indexed dot notation
MP4 Test if record is unused (`.StudentID = ""`) **in a loop**
MP5 Two item assignment statements correct
MP6 `Loan[Index].OnLoan` assigned TRUE
MP7 Correct logic to return `BOOLEAN` in both cases (position found / no position available)

Alternative solution: loop only (no 'found' mechanism)

```
   FOR Index ← 1 TO 7000
      IF Loan[Index].StudentID = Blank THEN
         Loan[Index].StudentID ← ThisStudentID
         Loan[Index].BookID ← ThisBookID
         Loan[Index].OnLoan ← TRUE
         RETURN TRUE
      ENDIF
    NEXT
    RETURN FALSE
```

8(c)	**MP1** Save to/Open a **(text) file** **MP2** **Single line** – form a string with separator characters and write to the file // **Multiple lines** - write each data item to a separate line in the file
8(d)	Additional date(s) solution **MP1** Store the **return date** of each loan // the **start date <u>and</u>** the **loan period** **MP2** Algorithm will do a **calculation** involving the MP1 data item(s) OR The student email address solution **MP3** Store the student <u>email address</u> **MP4** In order **to send** the email // the student is alerted to return the book **Max 2**

7(a)

Example solution:

```
FUNCTION CustomerOrder(Number, Points : INTEGER) RETURNS
                                                    INTEGER

   DECLARE NewPoints, NumberFree: INTEGER

   NewPoints ← Points + Number
   NumberFree ← 0
   WHILE NewPoints >= 11 AND Number > 0
      NumberFree ← NumberFree + 1
      NewPoints ← NewPoints -11
       Number ← Number - 1
   END WHILE

   OUTPUT "Number of free coffees is: ", NumberFree
   RETURN NewPoints
ENDFUNCTION
```

MP1 Function header **and** ending **and** parameters **and** return type
MP2 Number of coffees ordered added to points

MP3 Correct calculation of one free coffee

MP4 Attempted calculation of multiple free coffees **and** reduced `NewPoints`

MP5 Output number of free drinks with suitable message

MP6 Return of `NewPoints`

7(b)(i)

Example solution:

```
PROCEDURE AddNewCustomers(NumToAdd : INTEGER)

   DECLARE CustomerID : INTEGER
   DECLARE Count : INTEGER
   DECLARE Line : STRING
   DECLARE NewLine : STRING
   OPENFILE "Loyalty.txt" FOR READ

   WHILE NOT EOF("Loyalty.txt")
      READFILE "Loyalty.txt", Line
   ENDWHILE
   CLOSEFILE "Loyalty.txt"

   OPENFILE "Loyalty.txt" FOR APPEND

   CustomerID ← STR_TO_NUM((LEFT(Line, 6))

   FOR Count ← 1 TO NumToAdd
      CustomerID ← CustomerID + 1
      OUTPUT CustomerID
      NewLine ← NUM_TO_STR(CustomerID) & ",0"
      WRITEFILE "Loyalty.txt", NewLine
   NEXT Count

   CLOSEFILE "Loyalty.txt"
ENDPROCEDURE
```

Mark as follows:

MP1 All variables used are declared using the correct type including a string **and** an integer

MP2 Open file in read mode **and** close (before opening in append mode)

MP3 Conditional loop with `EOF("Loyalty.txt")`

MP4 Read `Line` from file // Count number of records in the file

MP5 Open the file in append mode **and** subsequently close

MP6 Extract `CustomerID` from `Line` **and** convert to integer // use count from MP4 to generate last `CustomerID` stored

MP7 A (count controlled) loop for the number of customers to add

MP8 Increment `CustomerID` **and** output the new `CustomerID` in loop

MP9 Create string for new `CustomerID` **and** write to file in loop

Max 8

7(b)(ii)	**MP1** Check if the text file `loyalty.txt` exists / is empty **MP2** If file does not exist it must be created (using write mode) **MP3** If the file has to be created / is empty then set the first `CustomerID` to `100001` **MP4** Write `"100001,0"` to `loyalty.txt` (using write mode) **MP5** For all but the first customer (to be added to the empty file) **use the existing code / module** **Max 4**

3	Example solution*:*
	
	`FUNCTION RollDice(NoOfTimes : INTEGER) RETURNS REAL`
	`  DECLARE RollValue : INTEGER`
	`  DECLARE Average : REAL`
	`  DECLARE Count : INTEGER`
	`  DECLARE Sum : INTEGER`
	
	`  Sum ← 0`
	
	`  FOR Count ← 1 TO NoOfTimes`
	`  RollValue ← INT(RAND(6)) + 1`
	`  OUTPUT RollValue`
	`  Sum ← Sum + RollValue`
	`  NEXT Count`
	
	`  Average ← Sum / NoOfTimes`
	`  RETURN Average`
	`ENDFUNCTION`
	
	MP1 Declare all variables used **and** initialise `Sum` to `0`
	MP2 Loop for 'number of times' parameter
	
	MP3 Use `RAND()` function with Integer parameter **in a loop**
	MP4 Use `INT()` function **in a loop**
	MP5 Generate a random integer between 1 and 6 **in a loop**
	
	MP6 Output each value generated i**n a loop**
	MP7 Calculate `Sum` **and** if `Average` used check declared as `REAL` **and** return `Average` **and** check function header returns `REAL`

5

Example solution:

```
FUNCTION Parity(BitString : STRING) RETURNS STRING
   DECLARE Index, Count: INTEGER
   Count ← 0

   FOR Index ← 1 TO LENGTH(BitString)
     IF MID(BitString, Index, 1) = '1' THEN
       Count ← Count + 1
     ENDIF
   NEXT Index

   IF Count MOD 2 = 1 THEN
      BitString ← BitString & '1'
   ELSE
      BitString ← BitString & '0'
   ENDIF

   RETURN BitString
ENDFUNCTION
```

MP1 Function heading **and** ending **and** parameter (`STRING`) **and** return type `STRING`
MP2 Loop for length of `BitString`
MP3 Extract current character **in a loop**
MP4 Compare current character to `'1'` or `'0'` **in a loop**
MP5 Increment count **and** was Initialised earlier **in a loop**

MP6 Check if even/odd number of 1s
MP7 Append `'1'` or `'0'` as appropriate
MP8 Return amended `BitString`

3(a)

One mark per emboldened part:

1 Open the file **(Result.txt) in <u>read</u> mode**
2 Loop until **EOF(Result.txt) // EOF // end of file**
3 **<u>Read a / the / next line</u> (from the file)** and store in `ThisLine`
4 Assign `LineY` to `LineX`
5 Assign `LineZ` to `LineY`
6 Assign `ThisLine` to `LineZ`
7 After the loop, **close the file (Result.txt)**
8 Output `LineX`, `LineY`, `LineZ`

3(b)	Example answer: So that the last three lines of the file are output in the correct **order** A mark for mentioning one of: • (Ensuring) the lines are **output in correct order** • Ordering the last <u>three</u> lines • Ordering the lines so they are in the same order as (they occur) <u>in the file</u> **Max 1 marks**
3(c)	Two loop solution One mark per point: 1 Loop until given line number (-1) / `<parameter> (-1)` (lines have been read) 2 ... if end of file is reached then return FALSE 3 Loop for three lines // Read three lines // Repeat of 4 and 5 for three lines 4 Read a line and output it 5 ... if end of file is reached then return FALSE 6 After outputting the (required) lines return TRUE 7 Ordering of lines no longer needed **Max 4 marks** Alternative solution **One loop solution mark scheme that reads to given line number + 2** One mark per point 1 Loop until given line number + 2 / `<parameter> + 2` (lines have been read) 2 **OR** end of file is reached 3 Return FALSE if EOF reached 4 Read a line from the file **in the loop** 5 Continue to order the last three lines read i**n the loop** // Steps 4 to 6 stay the same 6 (If FALSE has **not** been returned) output the required three lines (in the correct order) 7 … **and** Return TRUE **Max 4 marks**

4	Example solution:
	
	`PROCEDURE Timer(Mins, Secs : INTEGER)`
	`  DECLARE WarningTick, EndTick : INTEGER`
	
	`  EndTick ← Tick + 1000 * ((Mins * 60) + Secs)`
	`  WarningTick ← EndTick - (30 * 1000)`
	
	`  REPEAT`
	`  //do nothing`
	`  UNTIL Tick = WarningTick`
	`  OUTPUT "30 seconds to go"`
	
	`  REPEAT`
	`  //do nothing`
	`  UNTIL Tick = EndTick`
	
	`  OUTPUT "The time is up!"`
	
	`ENDPROCEDURE`
	
	Mark as follows:
	MP1 Procedure heading **and** parameters **and** ending
	MP2 **'**Attempt' to calculate 'total time'/`EndTick` // 'elapsed time' // `WarningTick`
	MP3 Correct calculation of `EndTick` **and** `WarningTick`
	
	MP4 **(Design mark)**
	• Two separate loops – checking warning time then the final time, **OR** …
	• Single loop checking the final time with an IF statement to check for warning time, **OR** …
	• Single loop with two IF statements checking the warning time and final time
	
	MP5 Completely correct MP4
	MP6 Output both messages (must be meaningful and follow successful MP4)

6(a)

Example solution:

```
PROCEDURE Special()
   DECLARE Index : INTEGER
   DECLARE Filling1, Filling2 : STRING

   REPEAT
      Index ← INT(RAND(35)) + 1
   UNTIL Filling[Index] <> ""

   Filling1 ← Filling[Index]

   REPEAT
      Index ← INT(RAND(35)) + 1
   UNTIL Filling[Index] <> "" AND Filling1 <>
                                          Filling[Index]

   Filling2 ← Filling[Index]

   REPEAT
      Index ← INT(RAND(10) + 1)
   UNTIL Bread[Index] <> ""

   OUTPUT "The daily special is ", Filling1, " and ", __
          Filling2, " on ", Bread[Index], " bread."

ENDPROCEDURE
```

Mark as follows:

MP1 Loop for Filling 1, avoiding unused elements
MP2 Loop for Filling 2 avoiding unused elements
MP3 Check Filling 2 is different from Filling 1 – could correctly compare either the indices or the array contents

MP4 Loop for Bread, avoiding unused elements

MP5 Using `RAND(10)` / `RAND(35)`
MP6 Completely correct use of `RAND()` – including `INT()` and +1 in all cases

MP7 Correct output - **once only** – following a reasonable attempt at selection of filings and bread

6(b)

Answers include:

MP1 For each filling, create a list of <u>acceptable</u> / <u>incompatible</u> fillings/indexes
MP2 When selecting the second filling, (as well as checking for an unused element) <u>check</u> that the filling / index is / is not on the list

ALTERNATIVE:
MP1 Create a list of 'good' combinations
MP2 <u>Randomly select</u> from this list

8(a)

Example solution

```
PROCEDURE Count(ThisPlayer : INTEGER, ThisRole : STRING)
   DECLARE Index, Num, Total : INTEGER

   Num ← 0
   Total ← 0

   FOR Index ← 1 TO 45
      IF Character[Index].Player = ThisPlayer AND __
         Character[Index].Role = ThisRole THEN
         Num ← Num + 1
         Total ← Total + Character[Index].SkillLevel
   NEXT Index

   IF Num > 0 THEN
      OUTPUT "Player ", ThisPlayer, __
             " has ", Num, " characters with the role of
             ", ThisRole, " and the total skill level is
             ", Total
   ELSE
          OUTPUT "No characters with that role are
                 assigned to this player"
   ENDIF

ENDPROCEDURE
```

MP1 Initialisation of local integers for `Num` and `Total`
MP2 Loop through 45 elements

MP3 Attempt to check Player and Role fields **in a loop**
MP4 Correctly compare `Player` field with parameter **in a loop**
MP5 Correctly compare `Role` field with parameter **in a loop**

MP6 ... if player and role found, increment `Num` **and** sum Skill Total **in a loop**
MP7 Test for any matches **after the loop**
MP8 **Both** possible OUTPUT statements correctly formed following an attempt at MP6 but outputting one only

Max 7 marks

8(b)	Example solution:
	
	`PROCEDURE Restore()`
	`  DECLARE Index : INTEGER`
	`  DECLARE Line  : STRING`
	
	`  OPENFILE "SaveFile.txt" FOR READ`
	`  FOR Index ← 1 TO 45`
	`  READFILE "SaveFile.txt", Line`
	`  Character[Index].Player ← STR_TO_NUM(Extract(Line, 1))`
	`  Character[Index].Role ← Extract(Line, 2)`
	`  Character[Index].Name ← Extract(Line, 3)`
	`  Character[Index].Skill ← STR_TO_NUM(Extract(Line, 4))`
	`  NEXT Index`
	
	`  CLOSEFILE "SaveFile.txt"`
	
	`ENDPROCEDURE`
	
	Mark as follows:
	MP1 Open the file in read mode and subsequently close
	MP2 Loop through 45 elements
	MP3 Read a line from the file **in a loop**
	MP4 Attempt to use `Extract()` **in a loop**
	MP5 Correct use of `Extract()` for all fields **in a loop**
	MP6 Use of `STR_TO_NUM()` on Player and Skill **in a loop**
	MP7 Completely correct extraction and assignment of all fields **in a loop**

8(c)	**MP1** The `Character` array / character data can be saved before the program is closed **MP2** Allowing the **game to continue** using the same data / from the point it was saved

7(a)(i)	To make the solution easier to design / implement / solve

7(a)(ii)	One mark per item **and** justification Item: mobile phone number Justification: to send the text message Item: name Justification: to personalise the text message Item: exercise interest Justification: to determine whether this member would be interested
7(a)(iii)	Examples include: • Add a member to a list of those interested in the new class • Remove the member from future SMS messages • Read/process Message • Identify who from One mark for each. **Max 2 marks**
7(b)(i)	Means that `Update` calls (one of) either `Sub-A, Sub-B` or `Sub-C` One mark for each point: • reference to selection / decision / if • naming all four modules correctly

7(b)(ii)	`PROCEDURE Sub-A (Name : STRING, BYREF P2 : BOOLEAN)`
	
	`FUNCTION Sub-B (P1 : REAL) RETURNS REAL`
	
	One mark per underlined part in each case

<table>
<tr><td>7(a)</td><td>Examples include:

Module: <code>IdentifyMember()</code>
Use: Identifies a club member who has expressed an interest in a given class

Module: <code>GetMemberPhoneNumber()</code>
Use: Gets the mobile phone number of a member

Module: <code>CreateMessage()</code>
Use: Generates a text message to a member

Module: <code>SendMessage()</code>
Use: Sends a text message to a member in the waiting list

One mark for name and use

Note: max 3 marks</td></tr>
<tr><td>7(b)(i)</td><td>
<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>none</td><td>S3</td></tr>
<tr><td>Input-A</td><td>Output-W</td><td>S3</td></tr>
<tr><td>Input-B</td><td>none</td><td>S2</td></tr>
<tr><td>Input-B</td><td>none</td><td>S5</td></tr>
<tr><td>Input-A</td><td>none</td><td>S2</td></tr>
<tr><td>Input-A</td><td>Output-X</td><td>S4</td></tr>
</table>

One mark per row 3 to 7</td></tr>
<tr><td>7(b)(ii)</td><td>Input-B, Input-A</td></tr>
</table>