Past-paper walk-through

Arrays ▪ 8.2

eXercise

Questions in this set

- 0478/21 N25 Q11 ▪ 15 marks
- 0478/22 N25 Q2 ▪ 15 marks
- 0478/22 N25 Q5 ▪ 13 marks
- 0478/21 J25 Q3 ▪ 4 marks
- 0478/23 J25 Q3 ▪ 12 marks
- 0478/22 N24 Q12 ▪ 15 marks
- 0478/21 J24 Q10 ▪ 15 marks
- 0478/22 J24 Q11 ▪ 15 marks

Reading the mark scheme

- **B1** a mark that stands on its own – the point is either made or not.
- **M1** a *method* mark: the right approach, even if the number is wrong.
- **A1** an *answer* mark, and it usually needs its M mark first.
- **C1** a working mark on the way to the answer.
- // separates answers the examiner accepts equally.
- **ecf** = error carried forward: a later part is marked on your own earlier answer.
- **owtte** = "or words to that effect" – the wording need not match.

Question 11

0478/21 N25 ▪ Q11 ▪ 15 marks

The scores for each competitor in an archery competition are recorded.
The one-dimensional (1D) array `CompetitorName[]` is used to store the names of the competitors.
The maximum number of competitors is 30.
There are 10 rounds in the competition with a maximum score of 30 per round.
The two-dimensional (2D) array `CompetitorScore[]` stores the score in each of the 10 rounds for each competitor.
The first index of the array `CompetitorScore[]` is the same as the index of the competitor in `CompetitorName[]`
After the 10 rounds have been completed, the highest score and the lowest score for each competitor are discarded.

Question 11

The overall score for each competitor is the total of the remaining 8 scores (after discarding the highest and lowest scores).
A competitor with an overall score of 210 or more moves onto the next competition. A competitor with an overall score between 180 and 209 inclusive becomes a reserve competitor. A competitor with an overall score below 180 does not qualify.
Write a program that meets the following requirements:

- inputs the total score (out of 30) for each competitor in each round
- validates each input is between 0 and 30 inclusive
- calculates the overall score for each competitor discarding the highest and lowest scores
- outputs the name of each competitor and if they move onto the next competition, are a reserve or do not qualify

Question 11

- calculates and outputs the number of competitors who:
  - have qualified for the next competition
  - are reserve competitors
  - do not qualify.

You must use pseudocode or program code **and** add comments to explain how your code works.
You do **not** need to declare any arrays or variables; assume that this has already been done.
All inputs and outputs must contain suitable messages. **[15]**

Solution 11

Worked steps

A 15-mark question is marked on **bands**, not on a checklist: roughly **AO2 (9 marks)** for applying the right techniques and **AO3 (6 marks)** for a solution that actually works as a whole. So completeness and correctness matter more than elegance.

Before writing anything, list two things.

1. **The data structures, with the names the scenario gives them.** The mark scheme underlines those names, so `Scores[]` must be `Scores[]`, not `ScoreArray[]`. Add any variables you need — a counter, a running total, a best-so-far — and declare them all.
2. **The requirements, one line each**, taken from the bullet points in the question. Each bullet is a section of your program, and the surest way to lose marks is to answer three of the four.

Solution 11

Worked steps

Then write it in that order, with one clearly separated section per requirement. Most of these scenarios need the same four:

- **Input and validate** — read the data, reject what is out of range, and re-ask.
- **Process** — a loop over the array doing the calculation: a total, a count, a highest or lowest, a search.
- **Store** — write results back into the array the scenario names.
- **Output** — print what was asked for, with a label saying what each number is.

What separates a top band from a middle one:

- **Loops that use the array's actual size**, not a hard-coded 12 or 20 when the scenario says "up to".

Solution 11

Worked steps

- **A running maximum found properly**: initialise it to the first element, not to zero — a competition of negative adjustments would break the zero version.
- **Output that is labelled.** `OUTPUT Total` is a number on a screen; `OUTPUT "Total score: ", Total` is a report.
- **Every declared variable used**, and every requirement visibly answered.

Write in **pseudocode or a real language**, consistently — mixing the two costs marks under AO3. And leave the sections separated with a blank line and a comment naming the requirement each one meets: a marker awarding a band needs to see all four. **Mark scheme**

Solution 11

Worked steps

- Use the tables for A02 and A03 below to award a mark in a suitable band using a best fit approach
- Then add up the total.
- Marks are available for:
- ▪ AO2 (maximum 9 marks)
- ▪ AO3 (maximum 6 marks)
- Data Structures required names shown underlined must be used as given in the scenario Arrays or lists:
- CompetitorName, CompetitorScore
- Requirements (techniques)

Solution 11

Worked steps

- R1 inputs the score for each round validating the input is in the range 0 to 30 inclusive (input, output, nested iteration, validation)
- R2 calculates the score of each competitor discarding the highest and lowest scores of the 10 rounds (nested iteration, output and selection)
- R3 outputs the name of the competitor and whether they have qualified, are a reserve or not qualified, output the number of competitors who have qualified, are reserve and not qualified (totalling, output and selection)
- 15
- 11
- Example 15-mark answer in pseudocode.

Solution 11

Worked steps

- // programming techniques of iteration, selection, counting, totalling and output are used
- // Candidates did not need to declare any arrays or variables.
- DECLARE Score : INTEGER
- DECLARE Highest : INTEGER
- DECLARE Lowest : INTEGER
- DECLARE Qualified : INTEGER
- DECLARE Reserve : INTEGER
- DECLARE NotQualified : INTEGER
- Qualified <- 0

Solution 11

Worked steps

- Reserve <- 0
- NotQualified <- 0
- FOR X <- 1 TO 30 //looping for number of competitors
- FOR Round <- 1 TO 10 //looping for number of rounds
- OUTPUT CompetitorName[X]
- OUTPUT "Enter the score for the round: ", Round
- INPUT Score
- // validation of input for between 0 and 30 inclusive
- WHILE Score < 0 OR Score > 30 DO

Solution 11

Worked steps

- OUTPUT "The score must be between 0 and 30 inclusive. Please try again."
- INPUT Score
- ENDWHILE
- CompetitorScore[X, Round] <- Score // adding score to 2-D array
- NEXT Round
- NEXT X
- // or suitable method of determining how many competitors
- FOR X <- 1 TO 30
- Count <- 0

Solution 11

Worked steps

- Highest <- 0
- Score <- 0
- Lowest <- 999 // resetting variables for each competitor
- 11
- FOR Y <- 1 TO 10 // loop through rounds
- Score <- Score + CompetitorScore[X, Y]
- IF CompetitorScore[X, Y] > Highest // finds highest score
- THEN
- Highest <- CompetitorScore[X, Y]

Solution 11

Worked steps

- ENDIF
- IF CompetitorScore[X, Y] < Lowest // finds lowest score
- THEN
- Lowest <- CompetitorScore[X, Y]
- ENDIF
- NEXT Y
- Score <- Score – Highest – Lowest // calculating score to subtract highest and lowest
- IF Score >= 210 // competitor has qualified
- THEN

Solution 11

Worked steps

- OUTPUT "Competitor ", CompetitorName[X], " has qualified"
- Qualified <- Qualified + 1
- ELSE
- IF Score >= 180 AND Score <= 209 // competitor is a reserve competitor
- THEN
- OUTPUT "Competitor ", CompetitorName[X], " are a reserve"
- Reserve <- Reserve + 1
- ELSE
- OUTPUT "Competitor ", CompetitorName[X], " has not qualified" // competitor not qualified

Solution 11

Worked steps

- NotQualified <- NOTQualified + 1
- ENDIF
- NEXT X
- OUTPUT "The number of competitors qualified is ", Qualified // output of results
- OUTPUT "The number of competitors who are a reserve is ", Reserve
- OUTPUT "The number of competitors not qualified is ", NotQualified
- 11
- Marking Instructions in italics

Solution 11

Worked steps

- AO2: Apply knowledge and understanding of the principles and concepts of computer science to a given context, including the analysis and design of computational or programming problems
- 0
- 1–3
- 4–6
- 7–9
- No creditable response.
- At least one programming technique has been used.
- Any use of selection, iteration, counting, totalling, input and output.

Solution 11

Worked steps

- Some programming techniques used are appropriate to the problem.
- More than one technique seen applied to the scenario, check the list of techniques needed.
- The range of programming techniques used is appropriate to the problem.
- All criteria stated for the scenario have been covered by the use of appropriate programming techniques, check the list of techniques needed.
- Some data has been stored but not appropriately.
- Any use of variables or arrays or other language dependent data structures e.g. Python lists.

Solution 11

Worked steps

- Some of the data structures chosen are appropriate and store some of the data required.
- More than one data structure used to store data required by the scenario.
- The data structures chosen are appropriate and store all the data required.
- The data structures used store all the data required by the scenario.
- 11
- Marking Instructions in italics
- AO3: Provide solutions to problems by:
- ▪ evaluating computer systems
- ▪ making reasoned judgements

Solution 11

Worked steps

- ▪ presenting conclusions
- 0
- 1–2
- 3–4
- 5–6
- No creditable response.
- Program seen without relevant comments.
- Program seen with some relevant comment(s).
- The program has been fully commented.

Solution 11

Worked steps

- Some identifier names used are appropriate.
- Some of the data structures used have meaningful names.
- The majority of identifiers used are appropriately named.
- Most of the data structures used have meaningful names.
- Suitable identifiers with names meaningful to their purpose have been used throughout.
- All of the data structures used have meaningful names.
- The solution is illogical.
- The solution contains parts that may be illogical.
- The program is in a logical order.

Solution 11

Worked steps

- The solution is inaccurate in many places.
- Solution contains few lines of code with errors that attempt to perform a task given in the scenario.
- The solution contains parts that are inaccurate.
- Solution contains lines of code with some errors that logically perform tasks given in the scenario. Ignore minor syntax errors.
- The solution is accurate.
- Solution logically performs all the tasks given in the scenario. Ignore minor syntax errors.
- The solution attempts at least one of the requirements.

Solution 11

Worked steps

- Solution contains lines of code that attempt at least one task given in the scenario.
- The solution meets most of the requirements.
- Solution contains lines of code that attempts most tasks given in the scenario.
- The solution meets all the requirements given in the question.
- Solution performs all the tasks given in the scenario.

Question 2

0478/22 N25 ▪ Q2 ▪ 15 marks

This pseudocode algorithm is intended to input a first name from a user and search for it in a two-dimensional (2D) array. If the name is found, the algorithm outputs all the remaining details for that name. The algorithm will continue to allow other names to be entered for searching, until the user input stops it.

01 DECLARE Contacts : ARRAY[1:500, 1:4] OF REAL 02 DECLARE Row : INTEGER 03 DECLARE Column : INTEGER 04 DECLARE Continue : BOOLEAN 05 DECLARE Stop : BOOLEAN 06 DECLARE FirstName : STRING 07 Continue <- TRUE 08 Stop <- FALSE 09 WHILE Continue 10 Row <- 1 11 OUTPUT "Enter a first name " 12 OUTPUT FirstName 13 REPEAT 14 IF Contacts[Row, 1] = FirstName 15 THEN 16 FOR Column <- 1 TO 4 17 OUTPUT Contacts[Row, 1] 18 NEXT Column 19 Stop <- TRUE 20 ELSE 21 Row <- Row + 1 22 ENDIF 23 NEXT Stop OR Row > 500 24

Question 2

OUTPUT "Search for another name? (Y or N)" 25 INPUT Answer 26 IF Answer = 'N' OR Answer = 'n' 27 THEN 28 Continue <- TRUE 29 ENDIF 30 ENDWHILE

(a) Identify the line numbers of **five** errors in the pseudocode and suggest a correction for each error.

Error 1 line number **[5]**

Question 2

0478/22 N25 ▪ Q2 ▪ 15 marks

This pseudocode algorithm is intended to input a first name from a user and search for it in a two-dimensional (2D) array. If the name is found, the algorithm outputs all the remaining details for that name. The algorithm will continue to allow other names to be entered for searching, until the user input stops it.

01 DECLARE Contacts : ARRAY[1:500, 1:4] OF REAL 02 DECLARE Row : INTEGER 03 DECLARE Column : INTEGER 04 DECLARE Continue : BOOLEAN 05 DECLARE Stop : BOOLEAN 06 DECLARE FirstName : STRING 07 Continue <- TRUE 08 Stop <- FALSE 09 WHILE Continue 10 Row <- 1 11 OUTPUT "Enter a first name " 12 OUTPUT FirstName 13 REPEAT 14 IF Contacts[Row, 1] = FirstName 15 THEN 16 FOR Column <- 1 TO 4 17 OUTPUT Contacts[Row, 1] 18 NEXT Column 19 Stop <- TRUE 20 ELSE 21 Row <- Row + 1 22 ENDIF 23 NEXT Stop OR Row > 500 24 OUTPUT "Search for another name? (Y or

Question 2

N)" 25 INPUT Answer 26 IF Answer = 'N' OR Answer = 'n' 27 THEN 28 Continue <- TRUE 29 ENDIF 30 ENDWHILE

(b)(i) Complete the pseudocode for `PROCEDURE NewData`

PROCEDURE NewData(\hspace{15em})
FOR Row <- 1 TO \hspace{15em}
FOR Column <- 1 TO 4
INPUT \hspace{15em}
\hspace{25em}
\hspace{25em}
ENDPROCEDURE **[4]**

(b)(ii) Write the pseudocode to input the number of new contacts and to use procedure `NewData`. A variable declaration is **not** required. **[2]**

Question 2

0478/22 N25 ▪ Q2 ▪ 15 marks

This pseudocode algorithm is intended to input a first name from a user and search for it in a two-dimensional (2D) array. If the name is found, the algorithm outputs all the remaining details for that name. The algorithm will continue to allow other names to be entered for searching, until the user input stops it.

01 DECLARE Contacts : ARRAY[1:500, 1:4] OF REAL 02 DECLARE Row : INTEGER 03 DECLARE Column : INTEGER 04 DECLARE Continue : BOOLEAN 05 DECLARE Stop : BOOLEAN 06 DECLARE FirstName : STRING 07 Continue <- TRUE 08 Stop <- FALSE 09 WHILE Continue 10 Row <- 1 11 OUTPUT "Enter a first name " 12 OUTPUT FirstName 13 REPEAT 14 IF Contacts[Row, 1] = FirstName 15 THEN 16 FOR Column <- 1 TO 4 17 OUTPUT Contacts[Row, 1] 18 NEXT Column 19 Stop <- TRUE 20 ELSE 21 Row <- Row + 1 22 ENDIF 23 NEXT Stop OR Row > 500 24 OUTPUT "Search for another name? (Y or

Question 2

N)" 25 INPUT Answer 26 IF Answer = 'N' OR Answer = 'n' 27 THEN 28 Continue <- TRUE 29 ENDIF 30 ENDWHILE

(c)(i) Explain why the given algorithm may **not** always give the expected results. **[2]**

(c)(ii) Explain how you could improve the algorithm so that the search is more likely to find the required results. **[2]**

Solution 2

(a) Worked steps

Five errors, each needing **the line number and the correction** — so answer in that shape every time: "Line NN / <what it says> should be <what it should say>."
Read the listing for the five classes of fault that these questions are built from:

- **A declared data type that does not match what is stored.** An array of names declared `OF REAL` should be `OF STRING`; a loop counter declared `CHAR` should be `INTEGER`.
- **A loop bound that is wrong by one**, or that counts the wrong way for its `STEP`.
- **A comparison operator the wrong way round** — < where > was meant, or = where <> was.
- **A flag set to the wrong value**, so the loop it controls never ends or never runs.

Solution 2

- **An assignment to the wrong variable**, or one missing entirely.

Two habits find them faster than reading top to bottom. **Check every declaration against the data it is given** — that catches the type errors immediately. And **trace the loop once with the first and last values** — that catches the bounds and the flag.

Quote the line number. A correct fix with no line number does not match the mark point. **Mark scheme**

- One mark per mark point, max five
- ▪ Line 01 / DECLARE Contacts : ARRAY[1:500, 1:4] OF REAL should be DECLARE Contacts : ARRAY[1:500, 1:4] OF STRING

Solution 2

- ▪ Line 12 / OUTPUT FirstName should be INPUT FirstName
- ▪ Line 17 / OUTPUT Contacts[Row, 1] should be OUTPUT Contacts[Row, Column]
- ▪ Line 23 / NEXT Stop OR Row > 500 should be UNTIL Stop OR Row > 500
- ▪ Line 25 / 26 / Answer variable not declared should be DECLARE Answer : CHAR before line 25
- ▪ Line 28 / Continue <- TRUE should be Continue <- FALSE

Solution 2

(b)(i) Worked steps

Complete the procedure header and its body from the specification.
The header needs **an appropriate parameter** — the right name and the right data type — and the body must **use that parameter** rather than a global.
The marks in a "complete the pseudocode" question are per gap, so work out what each gap must be from the lines around it: a gap in a `PROCEDURE` line is a parameter list, a gap inside a loop is a statement, a gap in a condition is a comparison.
Take the parameter's type from what the caller will pass — a count of new contacts is an `INTEGER`, a name is a `STRING`. **Mark scheme**

- One mark per mark point, max four

Solution 2

- MP1
- Appropriate parameter in PROCEDURE line
- MP2
- Correct index outer loop termination value; must use Number as stated in question
- MP3
- Correct array reference in INPUT
- MP4
- Correct termination of both loops.
- Example:
- PROCEDURE NewData(Number : INTEGER)

Solution 2

- FOR Row <- 1 TO Number
- FOR Column <- 1 TO 4
- INPUT Contacts[Row, Column]
- NEXT Column
- NEXT Row
- ENDPROCEDURE

Solution 2

(b)(ii) Worked steps

Input the value, then pass it to the procedure.

```
OUTPUT "How many new contacts?"
INPUT NumberOfContacts
CALL NewData(NumberOfContacts)
```

Solution 2

Two marks: a **correct input statement** and a **correct call** with the variable as its argument.

Solution 2

`CALL` is required for a procedure in Cambridge pseudocode — a bare `NewData(...)` is how a *function* is used, inside an expression. And pass the **variable**, not a literal: the whole point of the input is that the number is not known in advance. **Mark scheme**

- One mark per mark point, max two
- MP1
- Correct input statement
- MP2
- Correct call statement for NewData
- Example:
- INPUT MyNumber
- CALL NewData(MyNumber)

Solution 2

(c)(i) Mark scheme

- One mark per mark point, max two
- MP1
- There may be more than one person with the given first name // The name may not exist/found in the array
- MP2
- The search will stop at the first occurrence and output the data for the wrong person
- MP3
- If the name is not present there could be no output / the loop will continue.

Solution 2

(c)(ii) Mark scheme

- One mark per mark point, max two
- MP1
- Introduce additional search criteria into the algorithm to make the search more specific to each person in the array
- MP2
- ... such as the last name to go with the first name // ... such as some form of ID number / date of birth / contact number / post code
- MP3

Solution 2

- The algorithm should be able to search the whole array to find all available matches
- MP4
- ... by removing the stop variable
- MP5
- Use selection to confirm the name has not been found in the whole array
- MP6
- If the name is not present, output an appropriate message.

Question 5

0478/22 N25 ▪ Q5 ▪ 13 marks

This flowchart represents an algorithm.

(a) The first data row of the trace table shows the values currently stored in `CodeStore[]`

Complete the trace table for the algorithm, using the input data:

7686, Face, Speed, 432U, Yp79

			CodeStore[]						
Code	**Store**	**Count**							**OUTPUT**
			aBcd	1532	Face	63q8			

Question 5

[6]

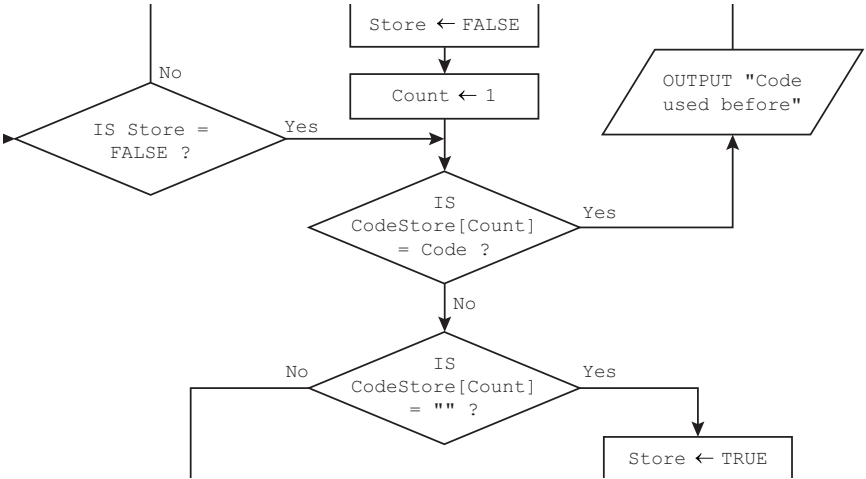
Question 5

			CodeStore[]						
Code	Store	Count	[1]	[2]	[3]	[4]	[5]	[6]	OUTPUT
			aBcd	1532	Face	63q8			

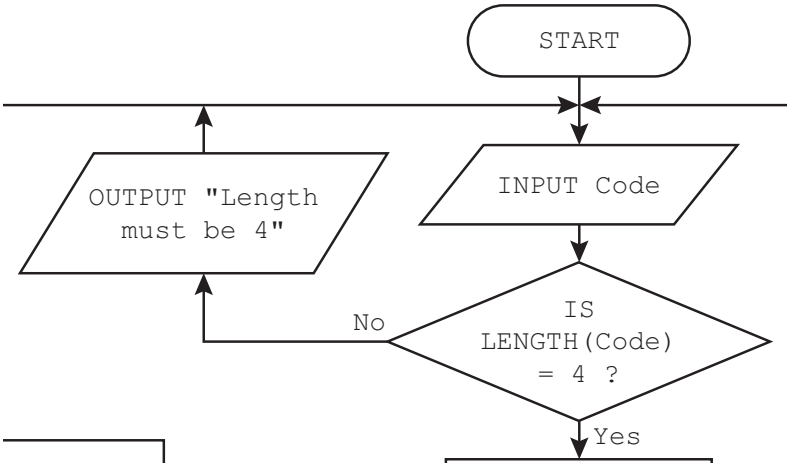
Question 5

[6]

Question 5


Store ← FALSE
Count ← 1
No
IS Store = FALSE ?
Yes
OUTPUT "Code used before"
IS CodeStore[Count] = Code ?
Yes
No
IS CodeStore[Count] = "" ?
No
Yes
Store ← TRUE

Question 5


START
INPUT Code
IS
LENGTH(Code)
= 4 ?
No
Yes
OUTPUT "Length
must be 4"

Question 5

0478/22 N25 ▪ Q5 ▪ 13 marks

This flowchart represents an algorithm.

(b) Describe the processes in the algorithm on page 6. **[3]**

Question 5

0478/22 N25 ▪ Q5 ▪ 13 marks

This flowchart represents an algorithm.

(c) The algorithm on page 6 requires the array `CodeStore[]` to be declared as a one-dimensional (1D) string array with six elements that are initially set to the null string `""`.

Write the pseudocode required to declare and initialise the array `CodeStore[]` You do need to declare any variables used. **[4]**

Solution 5

(a) Worked steps

A trace table is bookkeeping, and the discipline is what makes it reliable.

1. Take the algorithm **one line at a time**, in execution order.
2. Change **only** the variables that line changes; copy everything else down unchanged.
3. Start a **new row each time a value changes**, not each time a line is read.
4. When a loop repeats, go back to the **first line of the loop**, not to the top.
5. When an array element changes, write in **that element's column only**.

Solution 5

Six marks, awarded **per column** — so a complete trace with one arithmetic slip still scores, while a sparse one cannot. Fill every column on every row.
The first data row is given. Use it: it shows which columns exist and what the starting state is, and it settles whether the array is indexed from 0 or 1.
The column that catches people is a **derived one** such as `CodeStore[Index]` — that is not a variable but whatever the array holds at the current index, so recompute it whenever `Index` **or** that element changes. **Mark scheme**

- One mark per mark point, max six
- MP1
- Correct Code column

Solution 5

- MP2
- Correct Store column
- MP3
- Correct Count column
- MP4
- First four CodeStore columns left unchanged
- MP5
- Correct CodeStore[5] and CodeStore[6] columns
- MP6
- Correct OUTPUT column

Solution 5

- CodeStore[]
- Code
- Store Count

1

2

3

4

5

6

- OUTPUT aBcd 1532 Face 63q8

Solution 5

- 7686
- FALSE
- 1
- 2
- 3
- 4
- TRUE
- 5
- 7686
- 6

Solution 5

- Face
- FALSE
- 1
- 2
- 6
- 5(a)
- CodeStore[]
- Code
- Store
- Count

Solution 5

1

2

3 [4]

5

6

- OUTPUT
- 3
- Code used before
- Speed
- Length must be 4

Solution 5

- 432U
- FALSE
- 1
- 2
- 3
- 4
- 5
- TRUE
- 6
- 432U

Solution 5

- 7
- CodeStore full

Solution 5

(b) Mark scheme

- One mark per mark point, max three
- MP1
- Checks that the length of the code input is 4 characters.
- MP2
- Checks the code against the array contents to see if it has been used before (in the last 6 codes)
- MP3
- If the code passes the tests, it is stored in the array

Solution 5

- MP4
- Stores the code in the lowest available index
- MP5
- Checks that the code store isn't full before storing a new code // Outputs a code store is full message if appropriate
- // Algorithm continues until the code store is full.

Solution 5

(c) Worked steps

Declare the array and fill it, from the specification.

```
DECLARE CodeStore : ARRAY[1:100] OF STRING
FOR Index <- 1 TO 100
    CodeStore[Index] <- ""
NEXT Index
```

Solution 5

Four marks by mark point: the **declaration with the correct name and data type**, the **correct bounds**, a **loop over every element**, and **assigning the initial value**. Three details:

- **The name must be exactly as the scenario gives it.** A correct array under a different identifier does not match.
- **The bounds come from the question**, and 1-based and 0-based are different declarations — `[1:100]` holds 100 elements, and so does `[0:99]`.
- **Initialise with a value of the array's own type**: `""` for strings, `0` for integers. Assigning `0` to a string array is a type error even though the loop is right.

Solution 5

Mark scheme

- One mark per mark point, max four
- MP1
- Declaration of array using correct name and data type
- MP2
- Declaration of loop counter
- MP3
- Use of appropriate loop for six iterations (allow any type of loop)
- MP4
- Assignment of all array elements to null string (using loop counter as array index)
- Example:
- DECLARE CodeStore : ARRAY[1:6] OF STRING
- DECLARE Index : INTEGER
- FOR Index <- 1 TO 6
- CodeStore[Index] <- ""

Question 3

0478/21 J25 ▪ Q3 ▪ 4 marks

The following pseudocode algorithm uses a count-controlled loop to read in 10 names and store them in the array `Names[]`

FOR Count <- 1 TO 10 OUTPUT "Please enter a name" INPUT Name
Names[Count] <- Name NEXT Count

Rewrite the algorithm in pseudocode, using a condition-controlled loop instead of a count-controlled loop. **[4]**

Solution 3

Worked steps

Each mark is a **validation rule** stated precisely enough to be programmed — so give the check **and its limits**, not the name of a check.
The rules this data needs:

- **A range check** on the date of birth: it must fall **between 01/01/1900 and 01/01/2010**.
- **A presence check** on the name: something must have been entered — the field cannot be empty.
- **A length check** on the password: **exactly 12 characters**.

The five checks the syllabus expects, with what each rejects:

- **Range** — a number or date outside stated limits.

Solution 3

Worked steps

- **Presence** — an empty entry.
- **Length** — too few or too many characters.
- **Type** — the wrong data type, letters in a numeric field.
- **Format** — the right characters in the wrong pattern, such as a postcode or a date.

Include the **actual limits**. "A range check on the date" is half an answer; "between 01/01/1900 and 01/01/2010" is what makes it a rule someone could code.
Note what validation cannot do: it checks that data is **reasonable**, not that it is **correct**. A date of birth of 01/01/1950 passes every rule above and may still be the wrong person's — that is what **verification** is for, and the distinction is worth a sentence when the question asks for one. **Mark scheme**

Solution 3

Worked steps

- Date of birth must be between
- 01/01/1900 and 01/01/2010.
- Name has been entered.
- Password has exactly 12 characters.
- Student ID must contain 2 letters followed by 4-digit number.
- length check format check range check check digit presence check
- 3
- One mark per bullet point

Solution 3

Worked steps

- ▪ Initialising a count variable outside of loop
- ▪ Correct use of loop with condition for 10 times (WHILE (DO)...ENDWHILE, REPEAT...UNTIL) with close
- ▪ Incrementing the count variable inside the loop
- ▪ Rest of algorithm correct
- Examples
- Count $\leftarrow$ 1
- REPEAT
- OUTPUT "Please enter a name"
- INPUT Name

Solution 3

Worked steps

- Names[Count] $\leftarrow$ Name
- Count $\leftarrow$ Count $+ 1$
- UNTIL Count > 10
- Or
- Count $\leftarrow 1$
- WHILE Count $<= 10$ DO
- OUTPUT "Please enter a name"
- INPUT Name
- Names[Count] $\leftarrow$ Name

Solution 3

Worked steps

- Count $\leftarrow$ Count $+ 1$
- ENDWHILE

Question 3

0478/23 J25 ▪ Q3 ▪ 12 marks

The purpose of this pseudocode algorithm is to carry out a bubble sort to sort, in descending order, 1000 numbers stored in a one-dimensional (1D) array.

```
01 DECLARE Values : ARRAY[1:1000] OF REAL 02 DECLARE Index : CHAR 03
DECLARE Stop : BOOLEAN 04 DECLARE Hold : REAL 05 Stop <- FALSE 06
WHILE NOT Stop DO 07 Stop <- FALSE 08 FOR Index <- 1 TO 50 09 IF
Values[Index + 1] > Values[Index] 10 THEN 11 Hold <- Values[Index] 12
Values[Index] <- Values[Index + 1] 13 Values[Index + 1] <- Hold 14 Stop <- FALSE
15 ENDIF 16 NEXT Index 17 NEXT Stop
```

(a) Identify the line numbers of **four** errors in the pseudocode and suggest a correction for each error.

Error 1 line number

Correction

Question 3

Error 2 line number
Correction
Error 3 line number
Correction
Error 4 line number
Correction **[4]**

Question 3

0478/23 J25 ▪ Q3 ▪ 12 marks

The purpose of this pseudocode algorithm is to carry out a bubble sort to sort, in descending order, 1000 numbers stored in a one-dimensional (1D) array.

01 DECLARE Values : ARRAY[1:1000] OF REAL 02 DECLARE Index : CHAR 03 DECLARE Stop : BOOLEAN 04 DECLARE Hold : REAL 05 Stop <- FALSE 06 WHILE NOT Stop DO 07 Stop <- FALSE 08 FOR Index <- 1 TO 50 09 IF Values[Index + 1] > Values[Index] 10 THEN 11 Hold <- Values[Index] 12 Values[Index] <- Values[Index + 1] 13 Values[Index + 1] <- Hold 14 Stop <- FALSE 15 ENDIF 16 NEXT Index 17 NEXT Stop

(b)(i) Complete the pseudocode for `PROCEDURE Swap`

DECLARE Hold : REAL

ENDPROCEDURE **[4]**

(b)(ii) Write the pseudocode to transfer control to `PROCEDURE Swap` **[2]**

Question 3

0478/23 J25 ▪ Q3 ▪ 12 marks

The purpose of this pseudocode algorithm is to carry out a bubble sort to sort, in descending order, 1000 numbers stored in a one-dimensional (1D) array.

01 DECLARE Values : ARRAY[1:1000] OF REAL 02 DECLARE Index : CHAR 03 DECLARE Stop : BOOLEAN 04 DECLARE Hold : REAL 05 Stop <- FALSE 06 WHILE NOT Stop DO 07 Stop <- FALSE 08 FOR Index <- 1 TO 50 09 IF Values[Index + 1] > Values[Index] 10 THEN 11 Hold <- Values[Index] 12 Values[Index] <- Values[Index + 1] 13 Values[Index + 1] <- Hold 14 Stop <- FALSE 15 ENDIF 16 NEXT Index 17 NEXT Stop

(c) Explain the difference between global variables and the variable declared in **3(b)(i)**. **[2]**

Solution 3

(a) Worked steps

Four errors, each answered as **line number plus correction**: "Line NN / <as written> should be <as corrected>."
The faults these questions are built from, and how to find each:

- **A declaration whose type does not match its use.** `DECLARE Index : CHAR` used as a loop counter should be `INTEGER`. Check every declaration against the data it receives.
- **A flag initialised to the wrong value.** `Stop <- FALSE` where the loop runs `UNTIL Stop = FALSE` never ends. Trace the loop's first pass.
- **A loop bound or `STEP` that is wrong by one**, or that counts the wrong way.
- **A comparison the wrong way round**, or an assignment `<-` written where a comparison `=` was meant.

Solution 3

The two-minute method: read every **declaration** against its use, then **trace the loop once** with its first and last values. Between them those find almost every planted error, and they are much faster than reading the listing line by line.

Solution 3

Always give the **line number**. A correct correction without it does not match the mark point.

Mark scheme

- One mark per mark point
- ▪ Line 02 / DECLARE Index : CHAR should be DECLARE Index : INTEGER
- ▪ Line 07 / Stop <- FALSE should be Stop <- TRUE
- ▪ Line 08 / FOR Index <- 1 TO 50 should be FOR Index <- 1 TO 999 // FOR Index <- 1 TO 1000 - 1
- ▪ Line 17 / NEXT Stop should be ENDWHILE

Solution 3

(b)(i) Worked steps

The header needs **two integer parameters** — the two positions to swap — and the body needs to use them.

```
PROCEDURE Swap(First : INTEGER, Second : INTEGER)
    DECLARE Hold : STRING
    Hold <- Names[First]
    Names[First]  <- Names[Second]
    Names[Second] <- Hold
ENDPROCEDURE
```

Solution 3

Four marks, by mark point: the **two integer parameters** in the `PROCEDURE` line, and the three assignments of the swap.

Solution 3

The **temporary variable** is what makes a swap work, and its type is the type of the thing being swapped — a `STRING` here, because the array holds names, even though the parameters are integers. Assigning `Names[First] <- Names[Second]` first, without saving the old value, overwrites it and leaves both elements the same. **Mark scheme**

- One mark per mark point
- MP1
- Two integer parameters in PROCEDURE line
- MP2
- Correct index assigned to Hold <- Values[Index1]
- MP3
- Array element correctly assigned to new array element (Index2 to Index1)
- MP4
- Value in Hold assigned to correct array element (Index2).
- For example,
- PROCEDURE Swap(Index1 : INTEGER, Index2 : INTEGER)
- DECLARE Hold : REAL

Solution 3

(b)(ii) Worked steps

```
CALL Swap(Position1, Position2)
```

Two marks: the **correct call command** with the procedure's name, and the **arguments** in the right order.

Solution 3

`CALL` is required for a procedure in Cambridge pseudocode. And the arguments must match the header's parameters in **number, order and type** — two integers here, which is what the header in part (b)(i) declared. **Mark scheme**

- One mark per mark point
- MP1
- Correct call command and use of Swap for procedure name
- MP2
- Two parameters referring to the correct indices of array elements to be swapped, as shown in the original algorithm. Index + 1 and Index.
- For example,
- CALL Swap(Index + 1, Index)

Solution 3

(c) Mark scheme

- One mark per mark point
- MP1
- Global variables can be used throughout the program and its procedures // The memory used by the variables is not recovered until the program terminates.
- MP2
- The variable declared in part 3(b)(i) is a local variable and can only be used in that procedure // The memory used by a local variable is recovered at the end of the procedure.

Question 12

0478/22 N24 ▪ Q12 ▪ 15 marks

A one-dimensional (1D) array `Rooms[]` contains the names of up to 20 rooms in a house. A two-dimensional (2D) array `Dimensions[]` is used to store the length, width and area of each room.

The position of any room's data is the same in both arrays. For example, the data in index 5 of `Dimensions[]` belongs to the room in index 5 of `Rooms[]`

The variable `Number` stores the number of rooms for which data is to be input. There must be at least 3 rooms but no more than 20.

Write a program that meets the following requirements:

- allows the number of rooms for which data is required to be input, stored and validated

Question 12

- allows the name of the room and the length and width of the room, in metres, to be entered and stored
- allows the area of each room to be calculated as length multiplied by width and stored as square metres rounded to two decimal places
- calculates the average size of all the rooms by area, in square metres, rounded to two decimal places
- finds the largest room and smallest room by area
- outputs the names of all rooms with their dimensions and area
- outputs the names of the largest room and smallest room by area
- outputs the total area of the house and the average size of all the rooms by area.

Question 12

You must use pseudocode or program code **and** add comments to explain how your code works.
You do **not** need to declare any arrays or variables; you may assume that this has already been done.
All inputs and outputs must contain suitable messages. **[15]**

Solution 12

Worked steps

Banded marking: **AO2 up to 9 marks** for applying the right techniques, **AO3 up to 6** for the program working as a whole.

Two lists before any code.

1. **The data structures, named as the scenario names them**: `Rooms[]` for up to 20 room names and `Dimensions[]` for their measurements. The mark scheme underlines those identifiers. Declare every variable you add — an index, a total, a largest-so-far and its index.
2. **The requirements, one line each**, answered in order with one visibly separated section apiece.

The techniques these scenarios want:

Solution 12

Worked steps

- **Index alignment.** `Rooms[i]` and `Dimensions[i, …]` describe the same room, and every part of the program depends on keeping them in step. Never sort one without the other.
- **A loop over the 1D array**, reading each room's dimensions from the 2D one and **calculating its area** — length × width.
- **Accumulating a total** across all the rooms, initialised before the loop.
- **Finding the largest**, initialised to the **first** room's area rather than to zero, and remembering the **index** so the room can be named.
- **Handling "up to 20"**: loop to the number actually stored, not to 20. A count variable, or a check for an empty name, is what makes that possible.
- **Labelled output** — the room's name with its area, and the total.

Solution 12

Worked steps

What separates the bands: finding the largest area but not which room it belongs to; looping a fixed 20 times over a part-filled array; and unlabelled numeric output. One notation throughout, every variable declared, and a comment naming the requirement each section answers. **Mark scheme**

- ▪ AO2 (maximum 9 marks)
- ▪ AO3 (maximum 6 marks)
- Data Structures required with names as given in the scenario:
- Arrays or lists Rooms[], Dimensions[]
- Variables

Solution 12

Worked steps

- Number
- Requirements (techniques):
- R1 Input and store number of rooms, the names of the rooms and their dimensions, including validation of number of rooms (input with prompts,
- (nested) iteration, use of variables, 1D and 2D arrays, validation).
- R2 Calculate and store the area of each room, the total area of the house and the average room area rounded to two decimal places. Find the smallest and largest rooms (calculation, totalling, rounding, finding maximum and minimum values, iteration).
- R3 Output the results, including contents of the arrays and the calculated data (iteration, output).

Solution 12

Worked steps

- Example 15-mark answer in pseudocode
- // input and validation of number of rooms
- REPEAT
- OUTPUT "How many rooms are in your house (enter a number between 3 and 20 inclusive)?"
- INPUT Number
- IF Number $<$ 3 OR Number $>$ 20
- THEN
- OUTPUT "The number of rooms must be between 3 and
- 20 inclusive, please try again"

Solution 12

Worked steps

- ENDIF
- UNTIL Number >= 3 AND Number <= 20
- // input of room names and dimensions – could be more than one loop
- FOR InLoop ← 1 TO Number
- OUTPUT "Enter the name of room ", InLoop
- INPUT Rooms[InLoop]
- OUTPUT "Enter the length of the room in metres"
- INPUT Dimensions[InLoop, 1]
- OUTPUT "Enter the width of the room in metres"

Solution 12

Worked steps

- INPUT Dimensions[InLoop, 2]
- Dimensions[InLoop, 3] ← ROUND(Dimensions[InLoop, 1]
- * Dimensions[InLoop, 2], 2)
- NEXT InLoop
- // calculates total area and average area – rounded values have been stored
- TotArea ← 0
- FOR Total ← 1 TO Number
- TotArea ← TotArea + Dimensions[Total, 3]
- NEXT Total

Solution 12

Worked steps

- AvArea ← ROUND(TotArea / Number, 2)
- 15
- October/November
- 2024
- 12
- // finding largest and smallest rooms
- Larea ← 0
- Sarea ← 100000
- Lindex ← 1

Solution 12

Worked steps

- Sindex $\leftarrow$ 1
- FOR Count $\leftarrow$ 1 TO Number
- IF Dimensions[Count, 3] $>$ Larea
- THEN
- Larea $\leftarrow$ Dimensions[Count, 3]
- Lindex $\leftarrow$ Count
- ENDIF
- IF Dimensions[Count, 3] $<$ Sarea
- THEN

Solution 12

Worked steps

- Sarea ← Dimensions[Count, 3]
- Sindex ← Count
- ENDIF
- NEXT Count
- // outputting the results
- FOR OutLoop ← 1 TO Number
- OUTPUT "Room: ", Rooms[Outloop]
- OUTPUT "Length: ", Dimensions[OutLoop, 1], " metres"
- OUTPUT "Width: ", Dimensions[OutLoop, 2], " metres"

Solution 12

Worked steps

- OUTPUT "Area: ", Dimensions[OutLoop, 3], " square metres"
- Next OutLoop
- OUTPUT "The largest room is: ", Rooms[Lindex]
- OUTPUT "The smallest room is: ", Rooms[Sindex]
- OUTPUT "The total area of the house is: ", TotArea, " square metres"
- OUTPUT "The average area of the rooms is: ", AvArea, " square metres"
- October/November
- 2024
- Marking Instructions in italics

Solution 12

Worked steps

- AO2: Apply knowledge and understanding of the principles and concepts of computer science to a given context, including the analysis and design of computational or programming problems
- 0
- 1–3
- 4–6
- 7–9
- No creditable response.
- At least one programming technique has been used.
- Any use of selection, iteration, counting, totalling, input and output.

Solution 12

Worked steps

- Some programming techniques used are appropriate to the problem.
- More than one technique seen applied to the scenario, check list of techniques needed.
- The range of programming techniques used is appropriate to the problem.
- All criteria stated for the scenario have been covered by the use of appropriate programming techniques, check list of techniques needed.
- Some data has been stored but not appropriately.
- Any use of variables or arrays or other language dependent data structures e.g.
- Python lists.

Solution 12

Worked steps

- Some of the data structures chosen are appropriate and store some of the data required.
- More than one data structure used to store data required by the scenario.
- The data structures chosen are appropriate and store all the data required.
- The data structures used store all the data required by the scenario.
- October/November
- 2024
- Marking Instructions in italics
- AO3: Provide solutions to problems by:
- ▪ evaluating computer systems

Solution 12

Worked steps

- ▪ making reasoned judgements
- ▪ presenting conclusions
- 0
- 1–2
- 3–4
- 5–6
- No creditable response.
- Program seen without relevant comments.
- Program seen with some relevant comment(s).

Solution 12

Worked steps

- The program has been fully commented.
- Some identifier names used are appropriate.
- Some of the data structures used have meaningful names.
- The majority of identifiers used are appropriately named.
- Most of the data structures used have meaningful names.
- Suitable identifiers with names meaningful to their purpose have been used throughout.
- All of the data structures used have meaningful names.
- The solution is illogical. The solution contains parts that may be illogical.
- The program is in a logical order.

Solution 12

Worked steps

- The solution is inaccurate in many places.
- Solution contains few lines of code with errors that attempt to perform a task given in the scenario.
- The solution contains parts that are inaccurate.
- Solution contains lines of code with some errors that logically perform tasks given in the scenario. Ignore minor syntax errors.
- The solution is accurate.
- Solution logically performs all the tasks given in the scenario.
- Ignore minor syntax errors.
- The solution attempts at least one of the requirements.

Solution 12

Worked steps

- Solution contains lines of code that attempt at least one task given in the scenario.
- The solution attempts to meet most of the requirements.
- Solution contains lines of code that attempts most tasks given in the scenario.
- The solution meets all the requirements given in the question.
- Solution performs all the tasks given in the scenario.

Question 10

0478/21 J24 ▪ Q10 ▪ 15 marks

The one-dimensional (1D) array `Clubs[]` is used to store the names of 12 cricket clubs in a local sports league.
The two-dimensional (2D) array `Statistics[]` is used to store, for each cricket club, the number of:

- matches won
- matches drawn
- matches lost.

Question 10

The 1D array `Points[]` is used to store the total number of points each cricket club has been awarded.
The position of any cricket club's data is the same in all three arrays. For example, the data in index 2 of `Statistics[]` and index 2 of `Points[]` belongs to the cricket club in index 2 of `Clubs[]`
The variable `Matches` stores the number of matches played by each team. Each team plays the same number of matches.
Points are awarded for:

- a win – 12 points
- a draw – 5 points
- a loss – 0 points.

Write a program that meets the following requirements:

Question 10

- allows the number of matches played to be input and stored, with a maximum of 22 matches
- validates the number of matches played
- allows the names of the cricket clubs to be input and stored
- allows the number of matches won, drawn and lost to be input and stored for each team
- validates the number of matches won, drawn or lost against the number of matches played
- asks the user to re-enter the number of matches won, drawn or lost if the total does not match the number of matches played
- calculates and stores the total number of points for each club
- finds the cricket club or clubs with the highest number of points

Question 10

- outputs the name or names of the winning club or clubs, the number of wins and the total number of points awarded.

You must use pseudocode or program code **and** add comments to explain how your code works.
You do **not** need to declare any arrays or variables; you may assume that this has already been done.
All inputs and outputs must contain suitable messages. **[15]**

Solution 10

Worked steps

Banded marking, so aim for a **complete working solution** rather than a clever partial one. Roughly nine marks for applying the techniques and six for the program working as a whole.

Start with the two lists.

1. **Data structures, named exactly as the scenario names them**: `Clubs[]`, `Statistics[]`, `Points[]`, `TieIndex[]`. The mark scheme underlines those identifiers. Add the variables you need — an index, a total, a highest-so-far — and declare every one.
2. **The requirements, one line each**, from the question's bullets. Answer them in the order given, one visibly separated section per bullet.

Solution 10

Worked steps

The techniques these cricket-club scenarios always want:

- **A loop over a 2D array**, with the outer loop over the clubs and the inner over that club's statistics.
- **A calculation per club** — points from wins, draws and losses — stored back into `Points[]` at the **same index** as the club in `Clubs[]`. Keeping the arrays index-aligned is the whole design, and it is what `TieIndex[]` then depends on.
- **Finding the maximum**, initialised to the **first element** rather than to zero, and remembering the **index** rather than only the value — you need the index to look the club's name up.
- **Handling the tie**: a second pass that collects every index whose points equal the maximum into `TieIndex[]`, with a count of how many there are.

Solution 10

Worked steps

- **Labelled output** naming the club or clubs and their points.

Where marks are lost in the top band: finding the maximum but not its index, so the club cannot be named; overwriting `TieIndex[]` instead of appending to it; and stopping at the first tie instead of collecting them all.
Write consistently in **one** notation, and separate the sections with a comment naming the requirement each meets. **Mark scheme**

- Data structures required:
- The names underlined must match those given in the scenario:
- Arrays or lists Clubs[], Statistics[], Points[], TieIndex[]

Solution 10

Worked steps

- Variables
- Matches, Won, Drawn, Lost, Counter, Maximum, TieCheck, OutLoop, Max, Count
- Requirements (techniques):
- R1 Input and store number of matches, cricket club names, number of matches won, drawn and lost. Validation of number of matches played and matches won, lost and drawn against number of matches played (with prompts, iteration, storing data in variables, validation, 1D and 2D arrays).
- R2 Calculate and store the number of points for each cricket club. Finding the index of the array with the highest score / highest score (calculation, finding the maximum, iteration).

Solution 10

Worked steps

- R3 Identifying cricket clubs with highest number of points and whether there is a tie for the top position. Output with appropriate messages (iteration, selection and output).
- 15
- 10
- Example 15-mark answer in pseudocode
- // meaningful identifiers and appropriate data structures
- // similar variables grouped to save space as in HL languages
- DECLARE Clubs : ARRAY[1:12] OF STRING
- DECLARE Statistics : ARRAY[1:12, 1:3] OF INTEGER

Solution 10

Worked steps

- DECLARE Points : ARRAY[1 : 12] OF INTEGER
- DECLARE TieIndex : ARRAY[1 : 12] OF INTEGER
- DECLARE Matches, Won, Drawn, Lost : INTEGER
- DECLARE Counter, Maximum, TieCheck, OutLoop : INTEGER
- DECLARE Max, Count : INTEGER
- // input number of matches
- REPEAT
- OUTPUT "How many matches have been played (Maximum 22)? "
- INPUT Matches

Solution 10

Worked steps

- UNTIL Matches <= 22
- // input of data as a single loop – multiple loops for data entry
- // is also acceptable.
- FOR Counter <- 1 TO 12
- OUTPUT "Enter the name of the cricket club"
- INPUT Clubs[Counter]
- // input of match results with validation
- REPEAT
- OUTPUT "Enter the number of matches won, drawn and lost for ", Clubs[Counter]

Solution 10

Worked steps

- INPUT Won, Drawn, Lost
- IF Won + Drawn + Lost <> Matches
- THEN
- OUTPUT "Your inputs must total ", Matches, " please try again"
- ENDIF
- UNTIL Matches = Won + Drawn + Lost
- Statistics[Counter, 1] <- Won
- Statistics[Counter, 2] <- Drawn
- Statistics[Counter, 3] <- Lost

Solution 10

Worked steps

- // calculating and storing points
- 10
- Points[Counter] <- Statistics[Counter, 1] * 12 + Statistics[Counter, 2] * 5
- NEXT Counter
- // finding the highest points
- Max <- -100
- FOR Maximum <- 1 TO 12
- IF Points[Maximum] > Max
- THEN

Solution 10

Worked steps

- Max <- Points[Maximum]
- ENDIF
- NEXT Maximum
- // finding the winner(s)
- Count <- 0
- FOR TieCheck <- 1 TO 12
- IF Points[TieCheck] = Max
- THEN
- Count <- Count + 1

Solution 10

Worked steps

- TieIndex[Count] <- TieCheck
- ENDIF
- Next TieCheck
- // outputting the results
- FOR OutLoop <- 1 TO Count
- OUTPUT "Winning Club(s): ", Clubs[TieIndex[OutLoop]]
- OUTPUT "Number of wins: ", Statistics[TieIndex[OutLoop], 1]
- Next OutLoop
- OUTPUT "Winning Points: ", Max

Solution 10

Worked steps

- Marking Instructions in italics
- AO2: Apply knowledge and understanding of the principles and concepts of computer science to a given context, including the analysis and design of computational or programming problems
- 0
- 1–3
- 4–6
- 7–9
- No creditable response.
- At least one programming technique has been used.

Solution 10

Worked steps

- Any use of selection, iteration, counting, totalling, input and output.
- Some programming techniques used are appropriate to the problem.
- More than one technique seen applied to the scenario, check the list of techniques needed.
- The range of programming techniques used is appropriate to the problem.
- All criteria stated for the scenario have been covered by the use of appropriate programming techniques, check the list of techniques needed.
- Some data has been stored but not appropriately.
- Any use of variables or arrays or other language dependent data structures e.g. Python lists.

Solution 10

Worked steps

- Some of the data structures chosen are appropriate and store some of the data required.
- More than one data structure used to store data required by the scenario.
- The data structures chosen are appropriate and store all the data required.
- The data structures used store all the data required by the scenario.
- Marking Instructions in italics
- AO3: Provide solutions to problems by:
- ▪ evaluating computer systems
- ▪ making reasoned judgements
- ▪ presenting conclusions

Solution 10

Worked steps

- 0
- 1–2
- 3–4
- 5–6
- No creditable response.
- Program seen without relevant comments.
- Program seen with some relevant comment(s).
- The program has been fully commented.
- Some identifier names used are appropriate.

Solution 10

Worked steps

- Some of the data structures used have meaningful names.
- The majority of identifiers used are appropriately named.
- Most of the data structures used have meaningful names.
- Suitable identifiers with names meaningful to their purpose have been used throughout.
- All of the data structures used have meaningful names.
- The solution is illogical.
- The solution contains parts that may be illogical.
- The program is in a logical order.
- The solution is inaccurate in many places.

Solution 10

Worked steps

- Solution contains few lines of code with errors that attempt to perform a task given in the scenario.
- The solution contains parts that are inaccurate.
- Solution contains lines of code with some errors that logically perform tasks given in the scenario. Ignore minor syntax errors.
- The solution is accurate.
- Solution logically performs all the tasks given in the scenario. Ignore minor syntax errors.
- The solution attempts at least one of the requirements.
- Solution contains lines of code that attempt at least one task given in the scenario.

Solution 10

Worked steps

- The solution meets most of the requirements.
- Solution contains lines of code that perform most tasks given in the scenario.
- The solution meets all the requirements given in the question.
- Solution performs all the tasks given in the scenario.

Question 11

0478/22 J24 ▪ Q11 ▪ 15 marks

A one-player game uses the two-dimensional (2D) array `Grid[]` to store the location of a secret cell to be found by the player in 10 moves. Each row and column has 5 cells. At the start of the game:

- The program places an 'X' in a random cell (**not** in `Grid[1,1]`) and empties all the other cells in the grid.
- The player starts at the top left of the grid.
- The player has 10 moves.

During the game:

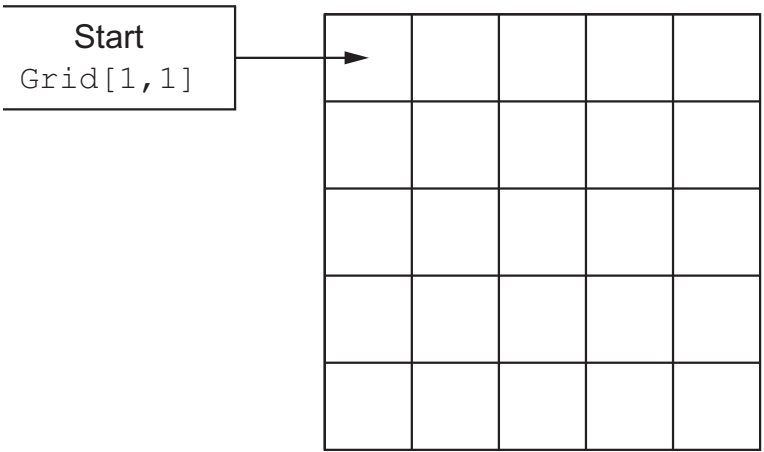
- The player can move left, right, up or down by one cell and the move must be within the grid.

Question 11

- "You Win" is displayed if the player moves to the cell with 'X' and has played 10 moves or less.
- "You Lose" is displayed if the player has made 10 moves without finding the 'X'.

Write a program that meets these requirements.
You must use pseudocode or program code **and** add comments to explain how your code works.
You do **not** need to declare any arrays or variables; you may assume that this has already been done.
All inputs and outputs must contain suitable messages. **[15]**

Question 11



Solution 11

Worked steps

Banded marking — about nine marks for the techniques and six for a solution that works end to end.

List the structures and the requirements first.

1. `Grid[]`, named exactly as the scenario names it, plus the variables the game needs: the secret row and column, the player's guesses, a counter of attempts, and a flag for whether the cell has been found.
2. **The requirements as one line each**, from the question's bullets — usually: set the game up, take a guess, judge it, repeat until found or out of turns, then report.

The techniques this kind of scenario wants:

Solution 11

Worked steps

- **Setting up**: clearing the grid with a **nested loop** over both dimensions, then generating a **random** row and column for the secret cell.
- **Input with validation**: a guess must be **inside the grid's bounds**, and an invalid one is rejected and re-asked rather than accepted.
- **A conditional loop** that ends on **either** condition — the cell found **or** the attempts used up. Both, joined by `OR`, or the game never ends when the player fails.
- **Comparing the guess with the secret cell** and giving feedback.
- **Labelled output** of the result and the number of attempts.

Where the marks go in the top band:

- The loop's **two exit conditions**, not one.

Solution 11

Worked steps

- Validation that **re-asks** rather than merely printing a message and continuing with a bad value.
- A **flag** that records the outcome, so the report after the loop can tell a win from running out of turns.
- **Nested loops for a 2D array** — one loop fills a row, not a grid.

Write in one notation throughout, declare every variable, and separate the sections with a comment naming the requirement each answers. **Mark scheme**

- Data Structures required with names as given in the scenario:
- Arrays or lists Grid

Solution 11

Worked steps

- Requirements (techniques)
- R1 Set up game – generate random cell, clear all other cells in array, set player start position and start player moves counter (iteration, use of arrays and library routines (round and random))
- R2 Input and check move – is it valid? (input, output, iteration and selection)
- R3 Decide outcome – has move found the X? If so, give appropriate output. If not increment counter and continue. If 10 moves exceeded, give appropriate output (use of arrays, iteration, selection and output).
- Example 15-mark answer in pseudocode
- // Set up game

Solution 11

Worked steps

- FOR Row <- 1 TO 5
- FOR Column <- 1 TO 5
- Grid[Row, Column] <- "// set grid cells to be empty
- NEXT Column
- NEXT Row
- 15
- 11
- REPEAT // not in cell 1,1
- XRow <- ROUND ((RANDOM() * 4) + 1, 0) // Random row position between 1 and 5 in GRID

Solution 11

Worked steps

- XColumn <- ROUND ((RANDOM() * 4) + 1, 0) // Random column position between 1 and 5 in
- GRID
- UNTIL XRow <> 1 and XColumn <> 1 // not in cell 1,1
- Grid [XRow, XColumn] <- 'X'
- MaxMove <- 10
- NumberMoves <- 0
- PlayerRow <- 1
- PlayerColumn <- 1
- Win <- FALSE

Solution 11

Worked steps

- // during game
- WHILE NumberMoves < MaxMove AND NOT Win
- MoveError <- FALSE
- OUTPUT "Please enter your move, L – Left, R – Right, U – Up or D - Down"
- INPUT UPPER(PlayerMove)
- REPEAT
- CASE OF PlayerMove
- 'L' : TempColumn <- PlayerColumn – 1
- 'R' : TempColumn <- PlayerColumn + 1

Solution 11

Worked steps

- 'U' : TempRow <- PlayerRow – 1
- 'D' : TempRow <- PlayerRow + 1
- OTHERWISE MoveError <- TRUE
- ENDCASE
- // check for out-of-range moves
- IF TempColumn < 1 or TempColumn > 5
- THEN
- MoveError <- TRUE
- ELSE

Solution 11

Worked steps

- PlayerColumn <- TempColumn
- ENDIF
- 11
- IF TempRow < 1 or TempRow > 5
- THEN
- MoveError <- TRUE
- ELSE
- PlayerRow <- TempRow
- ENDIF

Solution 11

Worked steps

- // check win if X Found
- IF Grid [PlayerRow, PlayerColumn] = 'X'
- THEN
- OUTPUT "You Win"
- Win <- TRUE
- ELSE
- IF NOT MoveError
- THEN
- NumberMoves <- NumberMoves + 1

Solution 11

Worked steps

- ENDIF
- ENDIF
- UNTIL NOT MoveError
- ENDWHILE
- IF NOT Win
- THEN
- OUTPUT "You Lose"
- ENDIF
- Marking Instructions in italics

Solution 11

Worked steps

- AO2: Apply knowledge and understanding of the principles and concepts of computer science to a given context, including the analysis and design of computational or programming problems
- 0
- 1–3
- 4–6
- 7–9
- No creditable response.
- At least one programming technique has been used.
- Any use of selection, iteration, counting, totalling, input and output.

Solution 11

Worked steps

- Some programming techniques used are appropriate to the problem.
- More than one technique seen applied to the scenario, check the list of techniques needed.
- The range of programming techniques used is appropriate to the problem.
- All criteria stated for the scenario have been covered by the use of appropriate programming techniques, check the list of techniques needed.
- Some data has been stored but not appropriately.
- Any use of variables or arrays or other language dependent data structures e.g. Python lists.

Solution 11

Worked steps

- Some of the data structures chosen are appropriate and store some of the data required.
- More than one data structure used to store data required by the scenario.
- The data structures chosen are appropriate and store all the data required.
- The data structures used store all the data required by the scenario.
- Marking Instructions in italics
- AO3: Provide solutions to problems by:
- ▪ evaluating computer systems
- ▪ making reasoned judgements
- ▪ presenting conclusions

Solution 11

Worked steps

- 0
- 1–2
- 3–4
- 5–6
- No creditable response
- Program seen without relevant comments.
- Program seen with some relevant comment(s).
- The program has been fully commented
- Some identifier names used are appropriate.

Solution 11

Worked steps

- Some of the data structures used have meaningful names.
- The majority of identifiers used are appropriately named.
- Most of the data structures used have meaningful names.
- Suitable identifiers with names meaningful to their purpose have been used throughout.
- All of the data structures used have meaningful names.
- The solution is illogical.
- The solution contains parts that may be illogical
- The program is in a logical order.
- The solution is inaccurate in many places.

Solution 11

Worked steps

- Solution contains few lines of code with errors that attempt to perform a task given in the scenario
- The solution contains parts that are inaccurate.
- Solution contains lines of code with some errors that logically perform tasks given in the scenario. Ignore minor syntax errors.
- The solution is accurate.
- Solution logically performs all the tasks given in the scenario. Ignore minor syntax errors.
- The solution attempts at least one of the requirements.
- Solution contains lines of code that attempt at least one task given in the scenario.

Solution 11

Worked steps

- The solution attempts to meet most of the requirements.
- Solution contains lines of code that attempt most tasks given in the scenario.
- The solution meets all the requirements given in the question.
- Solution performs all the tasks given in the scenario.