Past-paper walk-through

Algorithms ▪ 19.1

eXercise

Questions in this set

- 9618/21 J25 Q4 ▪ 8 marks
- 9618/32 N24 Q11 ▪ 12 marks
- 9618/33 N24 Q11 ▪ 12 marks
- 9618/41 N24 Q1 ▪ 22 marks
- 9618/43 N24 Q1 ▪ 22 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 4

9618/21 J25 ▪ Q4 ▪ 8 marks

Study the algorithm:

```
DECLARE Chars : ARRAY[1:4] OF CHAR DECLARE I, J : INTEGER DECLARE Key : CHAR I <- 2 Chars <- 'D' Chars <- 'T' Chars <- 'H' Chars <- 'R' WHILE I <= 4 //Outer loop Key <- Chars[I] J <- I - 1 WHILE J >= 0 AND Chars[J] > Key //Inner loop Chars[J + 1] <- Chars[J] J <- J - 1 ENDWHILE Chars[J + 1] <- Key I <- I + 1 ENDWHILE
```

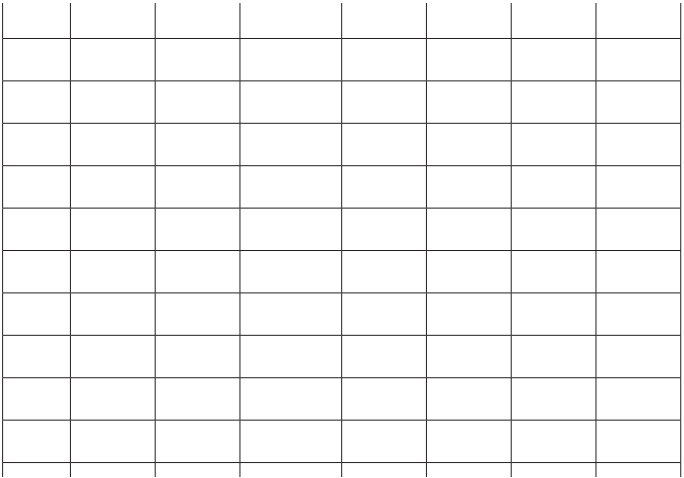
(a) The outer loop structure used in the algorithm is **not** the most appropriate one to use.

State the type of loop structure that would be the most appropriate to use and justify why it is the most appropriate. **[2]**

Question 4

I	Key	J	Chars[J]	Chars			
				[1]	[2]	[3]	[4]
2				'D'	'T'	'H'	'R'

Question 4



Question 4

[6]

Question 4

9618/21 J25 ▪ Q4 ▪ 8 marks

Study the algorithm:

```
DECLARE Chars : ARRAY[1:4] OF CHAR DECLARE I, J : INTEGER DECLARE Key : CHAR
I <- 2 Chars <- 'D' Chars <- 'T' Chars <- 'H' Chars <- 'R' WHILE I <= 4 //Outer loop Key
<- Chars[I] J <- I - 1 WHILE J >= 0 AND Chars[J] > Key //Inner loop Chars[J + 1] <-
Chars[J] J <- J - 1 ENDWHILE Chars[J + 1] <- Key I <- I + 1 ENDWHILE
```

(b) Complete the trace table by dry running the algorithm.

The first row has been completed.

Question 4

I	Key	J	Chars[J]	Chars			
2				'D'	'T'	'H'	'R'

Question 4

[6]

Solution 4

(a) Worked steps

- The most appropriate structure is a **count-controlled** loop — a `FOR` loop.
- Because **the number of iterations required is known** before the loop starts.

Two marks, and the second is the reason for the first. The rule that decides every question of this type:

- **Count-controlled (`FOR`)** — you know how many times, in advance.
- **Pre-condition (`WHILE`)** — you do not, and the loop might not run at all.
- **Post-condition (`REPEAT … UNTIL`)** — you do not, but the body must run at least once.

Solution 4

Here the array has a fixed size, so the outer pass count is fixed too; a conditional loop would work but has to maintain its own counter, which is exactly the work `FOR` does for you.

Mark scheme

- MP1
- Count-controlled
- MP2
- The number of iterations required is known

Solution 4

(b) Worked steps

A dry run is bookkeeping, and the discipline is what makes it reliable.

1. Take the algorithm **one line at a time**, in the order it executes.
2. Change **only** the variables that line changes; copy everything else down unchanged.
3. Write a new row **each time a value changes**, not each time a line is read.
4. When a loop repeats, go back to its first line — not to the top of the algorithm.

The columns here are `I`, `Key`, `J`, `Chars[J]` and the four array elements. Two things need care:

Solution 4

- **`Chars[J]` is a derived column.** It is not a variable — it is whatever the array element at the current `J` currently holds — so recompute it whenever `J` **or** that element changes.
- **The array columns keep their values** until something assigns to them. A blank means unchanged, so only write in an element's column on the row where it was actually written.

The algorithm is an **insertion sort**: `Key` holds the value being placed, and the inner loop shifts larger values one place right until the gap for `Key` is found. Knowing that makes the trace self-checking — after each outer pass, the first `I` elements should be in order.

Do not skip rows to save time. The mark scheme awards the columns semi-independently, so a complete trace with one arithmetic slip still scores; a sparse one cannot. **Mark scheme**

Solution 4

- I
- Key
- J
- Chars[J]
- Chars

1

2

3

Solution 4

4

- 2
- 'D'
- 'T'
- 'H'
- 'R'
- 'T'
- 1
- 'D'
- 3

Solution 4

- 'H'
- 2
- 'T'
- 'T'
- 1
- 'D'
- 'H'
- 4
- 'R'
- 3

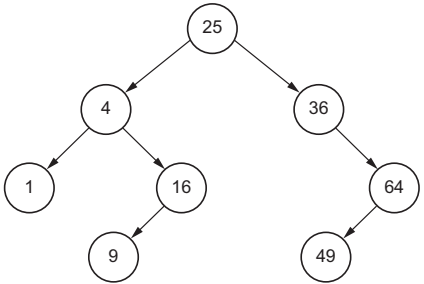
Solution 4

- 'T'
- 'T'
- 2
- 'H'
- ('R')
- 'R'

Question 11

9618/32 N24 ▪ Q11 ▪ 12 marks

11 This binary tree shows an ordered list of integers.



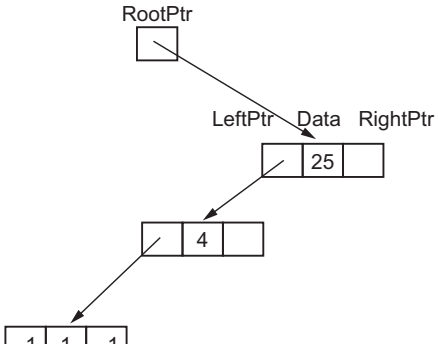
(a) A linked list of nodes is used to store the data. Each node consists of a left pointer, the data and a right pointer.

Question 11

and a right pointer.

−1 is used to represent a null pointer.

Complete this linked list to represent the given binary tree organisation.



Question 11

[4]

(b) A 2D array is used to store the nodes of the linked list in part **(a)**.

Complete the diagram using your answer for part **(a)**.

RootPtr
0

Index	LeftPtr	Data	RightPtr
0		25	

Question 11

	1		4	
	2		36	
	3		1	
	4		16	
	5		64	
FreePtr	**6**		9	
	7		49	
	8			

[4]

(c) The linked list in part **(a)** is implemented using a 1D array of records. Each record contains a left pointer, data and a right pointer.

The following pseudocode represents a function that searches for an element in the array of records `LinkList`. It returns the index of the record if the element is found, or it returns a

Question 11

Question 11

null pointer if the element is not found.

Complete the pseudocode for the function.

Question 11

```
NullPtr ← -1

..... ← RootPtr
WHILE NowPtr <> NullPtr
   IF LinkList[NowPtr].Data < Item THEN
      NowPtr ← LinkList[NowPtr].RightPtr
   ELSE
```

Question 11

```
            ELSE
               RETURN NowPtr
            ENDIF
         ENDIF
      ENDWHILE
      RETURN NullPtr
ENDFUNCTION
```

[4]

Solution 11

(a) Worked steps

Draw each node as **three boxes** — left pointer, data, right pointer — and connect them with arrows from the pointer boxes, not from the middle. Four marks:

1. **Five additional nodes with the correct data values.**
2. **Correct null pointers in all seven nodes**, and *no extra* nulls where an arrow already goes somewhere.
3. **Arrows from the correct left or right pointer box** to the correct child.
4. **All nodes in the right places**, with nothing written in the pointer boxes except pointers.

Solution 11

The tree is a **binary search tree**, which is what tells you where each value belongs: everything smaller than a node goes down its **left** branch, everything larger down its **right**. Place the values in the order the question lists them and each finds one position. Mark point 2 is where the marks usually go. A pointer box either holds an arrow **or** a null, never both — writing -1 in a box that also has an arrow leaving it contradicts itself, and leaving a leaf's boxes blank is not the same as marking them null. **Mark scheme**

- One mark per mark point (Max 4)
- MP1 Five additional nodes with correct data values

Solution 11

- MP2 Correct null pointers in all nodes (7) – no extra null pointers where the arrow points to the next node
- MP3 Correct arrows to represent pointers joining parent nodes to child nodes – must come from the correct left or right
- pointer not the middle
- MP4 All nodes in correct order and no additional data in left/right pointer boxes.
- 4
- 25
- 4
- -1
- -1 1

Solution 11

- LeftPtr Data RightPtr
- RootPtr
- -1
- -1 9
- -1
- 16
- -1
- -1 49
- -1
- 64

Solution 11

- -1 36
- 9618/32
- October/November 2024

Solution 11

(b) Worked steps

The same tree, stored as an array of records. Each row is one node: an index, a left pointer, the data, and a right pointer — with pointers being **row numbers**, not addresses.

Four marks: three for the first eight rows (three marks for all eight correct, two for six or seven, one for three to five), and one for the last row **left completely blank** together with a correct `FreePtr`.

Three rules do all the work:

- `RootPtr` **gives the row of the root**, so the tree is entered there, not at row 0 necessarily.
- **A null pointer is −1** (or whatever the question's convention is), meaning "no child".

Solution 11

- `FreePtr` **points at the first unused row**, which is how a new node would be added.

The last mark is the one to slow down for: the free row must be **entirely empty** — no leftover data and no pointers — and `FreePtr` must point at it. A row containing old values with `FreePtr` aimed at it says the space is free and also occupied.
Fill the table by walking the drawing from part (a) node by node. Reconstructing the tree mentally is where rows get the wrong children. **Mark scheme**

- First eight rows of table (Max 3)
- Three marks for all eight correct rows
- Two marks for six or seven correct rows

Solution 11

- One mark for three, four or five correct rows
- Last row of table and FreePtr (Max 1)
- One mark for correct FreePtr with last row of table completely blank
- RootPt
- r
- Index LeftPtr
- Data
- RightP
- tr
- 0

Solution 11

- 0
- 1
- 25
- 2
- 1
- 3
- 4
- 4
- 2
- -1

Solution 11

- 36
- 5
- 3
- -1
- 1
- -1
- 4
- 6
- 16
- -1

Solution 11

- 5
- 7
- 64
- -1
- FreePt
- r
- 6
- -1
- 9
- -1

Solution 11

- 8
- 7
- -1
- 49
- -1
- 8
- 4
- 9618/32
- October/November 2024

Solution 11

(c) Mark scheme

- One mark for any correct row (Max 4)
- FUNCTION SearchList(Item : INTEGER) RETURNS INTEGER
- NullPtr <- -1
- NowPtr <- RootPtr
- WHILE NowPtr <> NullPtr
- IF LinkList[NowPtr].Data < Item THEN
- NowPtr <- LinkList[NowPtr].RightPtr
- ELSE

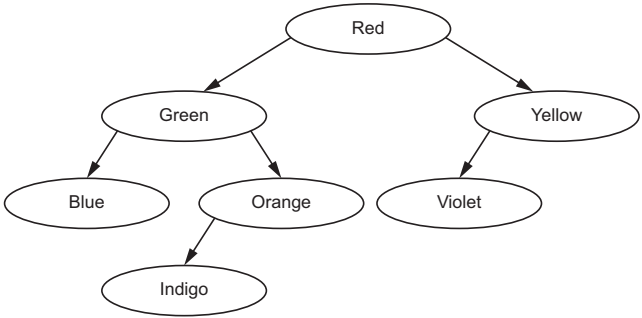
Solution 11

- IF LinkList[NowPtr].Data > Item THEN
- NowPtr <- LinkList[NowPtr].LeftPtr
- ELSE
- RETURN NowPtr
- ENDIF
- ENDIF
- ENDWHILE
- RETURN NullPtr
- ENDFUNCTION
- 4

Question 11

9618/33 N24 ▪ Q11 ▪ 12 marks

11 The following diagram shows an ordered binary tree.

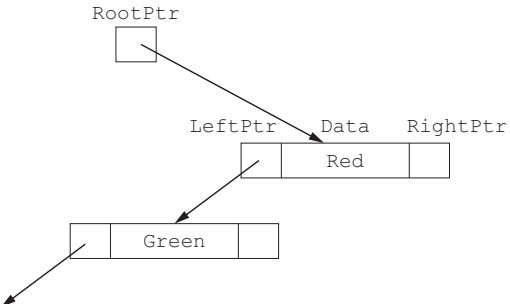


Question 11

(a) A linked list of nodes is used to store the data. Each node consists of a left pointer, the data and a right pointer.

–1 is used to represent a null pointer.

Complete this linked list to represent the given binary tree.



Question 11

[4]

(b) A user-defined record structure is used to store the nodes of the linked list in part (a).

Question 11

(b) A user-defined record structure is used to store the nodes of the linked list in part **(a)**.

Complete the diagram, using your answer for part **(a)**.

RootPtr

0

FreePtr

Index	LeftPtr	Data	RightPtr
0		Red	
1		Green	
2		Yellow	
3		Blue	
4		Orange	
5		Indigo	
6		Violet	
7			

Question 11

Question 11

[4]

(c) The linked list in part **(a)** is implemented using a 1D array of records. Each record contains a left pointer, data and a right pointer.

The following pseudocode represents a function that searches for an element in the array of records `BinTree`. It returns the index of the record if the element is found, or it returns a null pointer if the element is **not** found.

Complete the pseudocode for the function.

Question 11

```
WHILE NowPtr <> -1
```

Question 11

```
    NowPtr ← BinTree[NowPtr].LeftPtr
ELSE
    IF BinTree[NowPtr].Data < Item THEN
```

Question 11

```
            ELSE
                RETURN NowPtr
            ENDIF
        ENDIF
    ENDWHILE
    RETURN NowPtr
ENDFUNCTION
```

[4]

Solution 11

(a) Worked steps

Each node is **three boxes** — left pointer, data, right pointer — and the arrows come out of the pointer boxes, never out of the middle. Four marks:

1. The additional nodes with the **correct data values**.
2. **Correct null pointers everywhere**, with no extra null in a box that already has an arrow.
3. **Arrows from the correct left or right box** to the correct child.
4. All nodes in the right positions, with **nothing but pointers** in the pointer boxes.

Solution 11

It is a **binary search tree**: smaller values go left, larger values go right, applied recursively from the root. Insert the values in the order the question gives them and every one has exactly one place to go — which is also how to check a finished drawing, by reading it back and confirming each subtree stays on the correct side of its parent.

Mark scheme

- One mark per mark point (Max 4)
- MP1 Four additional nodes with correct data values
- MP2 Correct null pointers in all added nodes (6) with no extra null pointers where the arrow points to the next node
- MP3 Correct arrows to represent pointers joining parent nodes to child nodes

Solution 11

- MP4 All nodes in correct order and no extra data added to pointers.
- 4
- -1
- -1 Indigo
- -1
- Orange
- -1
- Yellow
- -1
- -1 Violet

Solution 11

- Red
- Green
- -1
- -1
- Blue
- RootPtr
- LeftPtr
- RightPtr
- Data
- 9618/33

Solution 11

- October/November 2024

Solution 11

(b) Worked steps

The array version stores one node per row: index, left pointer, data, right pointer, with pointers as **row numbers**.
Four marks: three for the first eight rows, scaled by how many are right, and one for the final row being **blank** with `FreePtr` pointing at it.

- `RootPtr` names the row where the tree starts.
- **−1 means "no child"**, and every leaf carries it in both pointer boxes.
- `FreePtr` names the first unused row.

Solution 11

Build the table straight from your drawing, one node at a time, rather than from the original list of values — the drawing already resolved where every child goes, and re-deriving it is where the pointers get crossed.
An in-order traversal of a correct table gives the values back in ascending order. That is a two-minute check worth doing: any node on the wrong side shows up immediately as a value out of sequence. **Mark scheme**

- One mark per mark point (Max 4)
- MP1 Correct Red and Green rows
- MP2 Correct Yellow and Blue rows
- MP3 Correct Orange, Indigo and Violet rows

Solution 11

- MP4 Correct FreePtr with blank row 7
- RootPt
- r
- Index LeftPtr
- Data
- RightPt
- r
- 0
- 0
- 1

Solution 11

- Red
- 2
- 1
- 3
- Green
- 4
- 2
- 6
- Yellow
- -1

Solution 11

- 3
- -1
- Blue
- -1
- 4
- 5
- Orange
- -1
- 5
- -1

Solution 11

- Indigo
- -1
- FreePt
- r
- 6
- -1
- Violet
- -1
- 7
- 7

Solution 11

- 4
- 9618/33
- October/November 2024

Solution 11

(c) Mark scheme

- One mark for any correct row (Max 4)
- FUNCTION SearchTree(Item : STRING) RETURNS INTEGER
- NowPtr <- RootPtr
- WHILE NowPtr <> -1
- IF BinTree[NowPtr].Data > Item THEN
- NowPtr <- BinTree[NowPtr].LeftPtr
- ELSE
- IF BinTree[NowPtr].Data < Item THEN

Solution 11

- NowPtr <- BinTree[NowPtr].RightPtr
- ELSE
- RETURN NowPtr
- ENDIF
- ENDIF
- ENDWHILE
- RETURN NowPtr
- ENDFUNCTION
- 4

Question 1

9618/41 N24 ▪ Q1 ▪ 22 marks

A program sorts string data using different sorting methods.

(a) The text file `Data.txt` stores string data items. Each data item is on a new line in the text file.

The function `ReadData()`:

- has a local array of strings that can store 45 items
- reads each line of data and stores it in the array
- returns the array.

Write program code for the function `ReadData()`.

Save your program as **Question1_N24**.

Copy and paste the program code into part **1(a)** in the evidence document. **[6]**

Question 1

9618/41 N24 ▪ Q1 ▪ 22 marks

A program sorts string data using different sorting methods.

(b)(i) Write program code for `FormatArray()`.

Save your program.

Copy and paste the program code into part **1(b)(i)** in the evidence document. **[2]**

(b)(ii) The main program:

- calls `ReadData()` and stores the returned array
- calls `FormatArray()` with the returned array and outputs the returned string.

Question 1

Write program code for the main program.
Save your program.
Copy and paste the program code into part **1(b)(ii)** in the evidence document. **[3]**

(b)(iii) Test your program.
Take a screenshot of the output.
Save your program.
Copy and paste the screenshot into part **1(b)(iii)** in the evidence document. **[1]**

Question 1

9618/41 N24 ▪ Q1 ▪ 22 marks

A program sorts string data using different sorting methods.

(c) The function `CompareStrings()`:

- takes two strings as parameters
- compares each string, one character at a time, to identify which string comes first alphabetically. If the first two characters are the same, the second character of each string is compared. This continues until the two characters are different.

The function:

- returns 1 if the first parameter comes before the second alphabetically
- returns 2 if the second parameter comes before the first alphabetically.

Question 1

Write program code for `CompareStrings()`.
Assume that all strings are in lower case.
Assume that a difference between two strings will always be identified before the end of one string is reached.
Do **not** use an in-built string comparison function.
The strings must be compared one character at a time.
Save your program.
Copy and paste the program code into part **1(c)** in the evidence document. **[4]**

Question 1

9618/41 N24 ▪ Q1 ▪ 22 marks

A program sorts string data using different sorting methods.

(d)(i) Write program code for `Bubble()`.

Save your program.

Copy and paste the program code into part **1(d)(i)** in the evidence document. **[3]**

(d)(ii) Write program code to amend the main program to:

- call `Bubble()` with the unsorted array as a parameter
- call `FormatArray()` with the sorted array and output the returned string.

Question 1

Save your program.
Copy and paste the program code into part **1(d)(ii)** in the evidence document. **[2]**
(d)(iii) Test your program.
Take a screenshot of the output.
Save your program.
Copy and paste the screenshot into part **1(d)(iii)** in the evidence document. **[1]**

Solution 1

(a) Worked steps

`ReadData()` is a **function**: it builds the array and hands it back.

```
FUNCTION ReadData() RETURNS ARRAY
    DECLARE Colours : ARRAY[0:44] OF STRING
    DECLARE Index : INTEGER
    TRY
        OPENFILE "Data.txt" FOR READ
        FOR Index <- 0 TO 44
            READFILE "Data.txt", Colours[Index]
        NEXT Index
        CLOSEFILE "Data.txt"
    EXCEPT
        OUTPUT "No file found"
    ENDTRY
    RETURN Colours
ENDFUNCTION
```

Solution 1

Six marks, one for each of:

1. Function declaration and close.
2. An **array declared** with room for the 45 items.
3. `Data.txt` **opened for read** and closed in a sensible place.
4. A loop over the whole file — 45 times, or until end of file.
5. **Each item stored** in its own array element.
6. The **populated array returned**.
7. **Exception handling**: a `TRY` around all the file access, a matching handler, and an output.

Solution 1

Two things carry the design. The array is declared **inside** the function and returned, so the caller receives it rather than depending on a global. And the exception handler must enclose **every** file operation — a file that disappears between the open and a later read raises the error there, not at the open. **Mark scheme**

- 1 mark each to max 6:
- ▪ Function declaration (and close where appropriate)
- ▪ Declaration/use of an array (with space/initialised with 45 spaces/strings)
- ▪ Opening the file Data.txt for read and closing in an appropriate place
- ▪ Looping through all file contents/Looping 45 times and reading each line ...
- ▪ ... storing all items from file into array
- ▪ Returning the populated array
- ▪ Exception handling with suitable try, catch and output
- 6 e.g.
- Python def ReadData():
- Colours = [] try:
- File = open("Data.txt")

Solution 1

(b)(i) Worked steps

`FormatArray()` turns the array into one printable line.

```
FUNCTION FormatArray(DataArray : ARRAY) RETURNS STRING
    DECLARE OutputText : STRING
    DECLARE Index : INTEGER
    OutputText <- ""
    FOR Index <- 0 TO 44
        OutputText <- OutputText & DataArray[Index] & " "
    NEXT Index
    RETURN OutputText
ENDFUNCTION
```

Solution 1

Two marks: the header with **at least one parameter**, and a loop that **concatenates each element with a space and returns the result**.
`OutputText` must start as the empty string and be **appended to** — `OutputText <- ...` inside the loop leaves only the last colour. That single-character difference between `<-` and `<- OutputText &` is the whole mark.

Solution 1

Take the array as a **parameter**. Reading a global would work here but throws away the mark for the parameter, and makes the function useless for any other array. **Mark scheme**

- 1 mark each
- ▪ Function header (and end where appropriate) taking (min) one parameter
- ▪ Looping through each parameter array element, concatenating with space and returning
- 2
- Python def FormatArray(DataArray):
- OutputText = "" for x in range(0, 45):
- OutputText = OutputText + DataArray[x] + " " return OutputText
- VB.NET
- Function FormatArray(DataArray)
- Dim OutputText As String = ""
- For X = 0 To 44
- OutputText = OutputText & DataArray(X) & " "
- Next

Solution 1

(b)(ii) Worked steps

The main program wires the two together.

```
Colours <- ReadData()
OUTPUT FormatArray(Colours)
```

Solution 1

Three marks: calling `ReadData()` and **storing** what it returns; passing that array to `FormatArray()`; and **outputting** what that returns.

Solution 1

Both are functions, so both return values that have to be used. `ReadData()` on its own line with nothing on the left discards the array it just built, and `FormatArray(Colours)` without an `OUTPUT` computes a string nobody sees. **Mark scheme**

- 1 mark each:
- ▪ Calling ReadData() and storing returned array ...
- ▪ ... calling FormatArray() with returned array
- ▪ Outputting return value from FormatArray()
- 3
- Python
- Colours = ReadData() #string array print(FormatArray(Colours))
- VB.NET
- Dim Colours(45) As String
- Colours = ReadData()
- Console.WriteLine(FormatArray(Colours))
- Java

Solution 1

(b)(iii)

Mark scheme

- 1 mark for output showing all colours in one string e.g.

Solution 1

(c) Worked steps

`CompareStrings()` decides which of two strings comes first, by comparing them **one character at a time from the left**.

```
FUNCTION CompareStrings(First : STRING, Second : STRING) RETURNS INTEGER
    DECLARE Count : INTEGER
    Count <- 1
    WHILE Count <= LENGTH(First) AND Count <= LENGTH(Second)
        IF MID(First, Count, 1) < MID(Second, Count, 1) THEN
            RETURN 1
        ENDIF
        IF MID(First, Count, 1) > MID(Second, Count, 1) THEN
            RETURN 2
        ENDIF
        Count <- Count + 1
    ENDWHILE
    RETURN 0
ENDFUNCTION
```

Solution 1

Four marks:

1. Header and close, taking **two parameters**, and **returning a value on every path**.
2. Looping through the characters of both strings.
3. Returning **1** when the first string's character is smaller.
4. Returning **2** when it is larger.

The comparison is decided by the **first character that differs**, and nothing after it matters — which is why both branches return immediately rather than setting a flag. Equal characters are the only case that continues the loop.

Solution 1

Mark 1 includes "returns a value in all cases", so there must be a return after the loop for two strings that never differ. A function that can fall off the end returns an undefined value in most languages, and here it would be silently wrong rather than an error. **Mark scheme**

- 1 mark each
- ▪ Function header (and close where appropriate) taking (min) two parameters and returns a value in all cases
- ▪ Looping through each character in each string parameter ...
- ▪ ... return 1 when first parameter second
- ▪ ... return 2 when first parameter second
- 4 e.g.
- Python def CompareStrings(First, Second):
- Count = 0 while True:
- if First[Count] < Second[Count]:
- return 1 elif First[Count] > Second[Count]:
- return 2 else:
- Count = Count + 1

Solution 1

(d)(i) Worked steps

A bubble sort over the array, using `CompareStrings()` from part (c) to decide the order.

```
FUNCTION Bubble(DataArray : ARRAY) RETURNS ARRAY
    DECLARE Outer, Inner, Result : INTEGER
    DECLARE Temp : STRING

    FOR Outer <- 0 TO 43
        FOR Inner <- 0 TO 43 - Outer
            Result <- CompareStrings(DataArray[Inner], DataArray[Inner + 1])
            IF Result = 2 THEN
                Temp <- DataArray[Inner]
                DataArray[Inner] <- DataArray[Inner + 1]
                DataArray[Inner + 1] <- Temp
            ENDIF
        NEXT Inner
    NEXT Outer
    RETURN DataArray
ENDFUNCTION
```

Solution 1

Three marks: the header taking the array and **returning the sorted array on every path**; comparing with `CompareStrings()` and swapping when needed; and a bubble sort that actually orders the data.

The mark for *using* `CompareStrings()` is the point of the question. Comparing the strings directly with < would sort them, but it throws away the module written in part (c) — and in a paper that has just asked you to write it, reuse is what is being examined.

`CompareStrings()` returns **2** when the first argument is the larger, so 2 is the swap condition. Swapping on 1 sorts the array into descending order — a plausible-looking result that is exactly wrong.

Solution 1

The inner bound shrinks by `Outer` because after each pass the largest remaining string is already in place; that is what makes it a bubble sort rather than 45 full passes. **Mark scheme**

- 1 mark each
- ▪ Bubble sort function header taking array parameter and returns sorted array in all cases
- ▪ Comparing strings using CompareStrings() and correctly swapping values when needed
- ▪ Correct bubble sort that sorts the data correctly
- 3
- Python def Bubble(DataArray):
- ArrayLength = len(DataArray) for x in range(ArrayLength - 1):
- for y in range(0, ArrayLength - x - 1):
- Result = CompareStrings(DataArray[y], DataArray[y + 1]) if Result == 2:
- DataArray[y], DataArray[y+1] = DataArray[y+1], DataArray[y] return DataArray
- 1(d)(i)
- VB.NET
- Function Bubble(DataArray)

Solution 1

(d)(ii) Worked steps

Call the sort, then format and print what it returns.

```
BubbleSorted <- Bubble(Colours)
OUTPUT FormatArray(BubbleSorted)
```

Solution 1

Two marks: calling `Bubble()` with the array and **storing or using** what it returns, and passing that to `FormatArray()` with the result output.
Both are functions, so both results have to be used. `Bubble(Colours)` on a line by itself discards the sorted array in any language that passes arrays by value, and the output then shows the original order — the failure this mark is checking for.

Solution 1

Keep the sorted array in its **own variable** rather than overwriting `Colours`. Later parts of the paper sort the same data by other methods, and each needs the unsorted original to start from. **Mark scheme**

- 1 mark each
- ▪ Calling Bubble() with array as parameter and using/storing return value
- ▪ Calling FormatArray() with return value from Bubble() and outputting return value
- 2
- Python
- BubbleSorted = Bubble(Colours) print(FormatArray(BubbleSorted))
- VB.NET
- Dim BubbleSorted(45) As String
- BubbleSorted = Bubble(Colours)
- Console.WriteLine(FormatArray(BubbleSorted))
- Java
- String[] BubbleSorted = new String[45];
- BubbleSorted = Bubble(Colours);

Solution 1

(d)(iii)

Mark scheme

- 1 mark for sorted data e.g.

Question 1

9618/43 N24 ▪ Q1 ▪ 22 marks

1 A program sorts string data using different sorting methods.

(a) The text file `Data.txt` stores string data items. Each data item is on a new line in the text file.

The function `ReadData()`:

- has a local array of strings that can store 45 items
- reads each line of data and stores it in the array
- returns the array.

Write program code for the function `ReadData()`.

Question 1

Save your program as **Question1_N24**.

Copy and paste the program code into part **1(a)** in the evidence document.

[6]

(b) The function `FormatArray()` takes an array of strings as a parameter. It concatenates the contents of the array into one string with a space between each array element. The function returns the concatenated string.

(i) Write program code for `FormatArray()`.

Save your program.

Copy and paste the program code into part **1(b)(i)** in the evidence document.

[2]

Question 1

(ii) The main program:

- calls `ReadData()` and stores the returned array
- calls `FormatArray()` with the returned array and outputs the returned string.

Write program code for the main program.

Save your program.

Copy and paste the program code into part **1(b)(ii)** in the evidence document.

[3]

(iii) Test your program.

Take a screenshot of the output.

Question 1

Save your program.

Copy and paste the screenshot into part **1(b)(iii)** in the evidence document.

[1]

(c) The function `CompareStrings()`:

- takes two strings as parameters
- compares each string, one character at a time, to identify which string comes first alphabetically. If the first two characters are the same, the second character of each string is compared. This continues until the two characters are different.

The function:

- returns `1` if the first parameter comes before the second alphabetically
- returns `2` if the second parameter comes before the first alphabetically.

Question 1

Write program code for `CompareStrings()`.

Assume that all strings are in lower case.

Assume that a difference between two strings will always be identified before the end of one string is reached.

Do **not** use an in-built string comparison function.

The strings must be compared one character at a time.

Save your program.

Copy and paste the program code into part **1(c)** in the evidence document.

[4]

Question 1

(d) The function `Bubble()` takes an array of strings as a parameter and sorts the data into ascending alphabetical order, using a bubble sort. The bubble sort uses `CompareStrings()` to compare each string.

The function returns the sorted list.

(i) Write program code for `Bubble()`.

Save your program. Copy and paste the program code into part **1(d)(i)** in the evidence document.

[3]

(ii) Write program code to amend the main program to:

Question 1

- call `Bubble()` with the unsorted array as a parameter
- call `FormatArray()` with the sorted array and output the returned string.

Save your program.

Copy and paste the program code into part **1(d)(ii)** in the evidence document.

[2]

(iii) Test your program.

Take a screenshot of the output.

Question 1

Save your program.

Copy and paste the screenshot into part **1(d)(iii)** in the evidence document.

[1]

Solution 1

(a) Worked steps

`ReadData()` is a **function**: it builds the array and returns it.

```
FUNCTION ReadData() RETURNS ARRAY
    DECLARE Colours : ARRAY[0:44] OF STRING
    DECLARE Index : INTEGER
    TRY
        OPENFILE "Data.txt" FOR READ
        FOR Index <- 0 TO 44
            READFILE "Data.txt", Colours[Index]
        NEXT Index
        CLOSEFILE "Data.txt"
    EXCEPT
        OUTPUT "No file found"
    ENDTRY
    RETURN Colours
ENDFUNCTION
```

Solution 1

Six marks: function declaration and close; an **array with room for the 45 items**; the file **opened for read and closed**; a loop over the whole file; **each item stored** in its own element; the **array returned**; and **exception handling** with all the file access inside the `TRY`.

Solution 1

The array is declared **inside** the function and returned, so the caller receives it rather than depending on a global. And the handler must enclose **every** file operation — a file that disappears between the open and a later read fails at the read. **Mark scheme**

- 1 mark each to max 6:
- ▪
- Function declaration (and close where appropriate)
- ▪
- Declaration/use of an array (with space/initialised with 45 spaces/strings)
- ▪
- Opening the file Data.txt for read and closing in an appropriate place
- ▪
- Looping through all file contents/Looping 45 times and reading each line ...
- ▪
- ... storing all items from file into array
- ▪

Solution 1

(b)(i) Worked steps

```
FUNCTION FormatArray(DataArray : ARRAY) RETURNS STRING
    DECLARE OutputText : STRING
    DECLARE Index : INTEGER
    OutputText <- ""
    FOR Index <- 0 TO 44
        OutputText <- OutputText & DataArray[Index] & " "
    NEXT Index
    RETURN OutputText
ENDFUNCTION
```

Solution 1

Two marks: a header taking **at least one parameter**, and a loop **concatenating each element with a space** and returning the result.

Solution 1

`OutputText` starts empty and is **appended to**. Writing `OutputText <- DataArray[Index] & " "` leaves only the last colour — a one-character difference that is the whole mark. **Mark scheme**

- 1 mark each
- ▪
- Function header (and end where appropriate) taking (min) one parameter
- ▪
- Looping through each parameter array element, concatenating with space and returning
- 2
- Python
- def FormatArray(DataArray):
- OutputText = ""
- for x in range(0, 45):
- OutputText = OutputText + DataArray[x] + " "
- return OutputText
- VB.NET

Solution 1

(b)(ii) Worked steps

```
Colours <- ReadData()
OUTPUT FormatArray(Colours)
```

Three marks: calling `ReadData()` and **storing** the array; passing it to `FormatArray()`; and **outputting** what that returns.

Solution 1

Both are functions, so both results must be used. `ReadData()` alone on a line discards the array; `FormatArray(Colours)` with no `OUTPUT` builds a string nobody sees. **Mark scheme**

- 1 mark each:
- ▪
- Calling ReadData() and storing returned array ...
- ▪
- ... calling FormatArray() with returned array
- ▪
- Outputting return value from FormatArray()
- 3
- Python
- Colours = ReadData() #string array
- print(FormatArray(Colours))
- VB.NET
- Dim Colours(45) As String

Solution 1

(b)(iii)

Worked steps

One mark, for the screenshot showing **all the colours in a single string**, space separated, on one line.
Run the program you have just written and paste the actual output. The mark is for evidence that the previous three parts work together, so a hand-typed list is not what is being asked for.

Mark scheme

- 1 mark for output showing all colours in one string
- e.g.
- 1
- 9618/43
- October/November 2024

Solution 1

(c) Worked steps

Compare the two strings **one character at a time from the left**, and stop at the first difference.

```
FUNCTION CompareStrings(First : STRING, Second : STRING) RETURNS INTEGER
    DECLARE Count : INTEGER
    Count <- 1
    WHILE Count <= LENGTH(First) AND Count <= LENGTH(Second)
        IF MID(First, Count, 1) < MID(Second, Count, 1) THEN
            RETURN 1
        ENDIF
        IF MID(First, Count, 1) > MID(Second, Count, 1) THEN
            RETURN 2
        ENDIF
        Count <- Count + 1
    ENDWHILE
    RETURN 0
ENDFUNCTION
```

Solution 1

Four marks: the header with **two parameters** returning a value **on every path**; looping through the characters; returning **1** when the first is smaller; returning **2** when it is larger.

Solution 1

The comparison is decided by the **first character that differs**, so both branches return immediately. Equal characters are the only case that continues the loop — and the return after the loop handles two strings that never differ, which is what "returns a value in all cases" requires. **Mark scheme**

- 1 mark each
- ▪
- Function header (and close where appropriate) taking (min) two parameters and returns a value in all cases
- ▪
- Looping through each character in each string parameter ...
- ▪
- ... return 1 when first parameter second
- ▪
- ... return 2 when first parameter second
- 4
- e.g.
- Python

Solution 1

(d)(i) Worked steps

A bubble sort that uses `CompareStrings()` to decide the order.

```
FUNCTION Bubble(DataArray : ARRAY) RETURNS ARRAY
    DECLARE Outer, Inner, Result : INTEGER
    DECLARE Temp : STRING
    FOR Outer <- 0 TO 43
        FOR Inner <- 0 TO 43 - Outer
            Result <- CompareStrings(DataArray[Inner], DataArray[Inner + 1])
            IF Result = 2 THEN
                Temp <- DataArray[Inner]
                DataArray[Inner] <- DataArray[Inner + 1]
                DataArray[Inner + 1] <- Temp
            ENDIF
        NEXT Inner
    NEXT Outer
    RETURN DataArray
ENDFUNCTION
```

Solution 1

Three marks: the header taking the array and **returning it sorted on all paths**; comparing with `CompareStrings()` and swapping when needed; and a bubble sort that orders the data correctly.

Solution 1

Using `CompareStrings()` is the point — comparing with < directly would work but throws away the module just written. And **2 is the swap condition**, because that is what the function returns when the first argument is the larger; swapping on 1 sorts into descending order. **Mark scheme**

- 1 mark each
- ▪
- Bubble sort function header taking array parameter and returns sorted array in all cases
- ▪
- Comparing strings using CompareStrings() and correctly swapping values when needed
- ▪
- Correct bubble sort that sorts the data correctly
- 3
- Python
- def Bubble(DataArray):
- ArrayLength = len(DataArray)
- for x in range(ArrayLength - 1):

Solution 1

(d)(ii) Worked steps

```
BubbleSorted <- Bubble(Colours)
OUTPUT FormatArray(BubbleSorted)
```

Two marks: calling `Bubble()` and **storing or using** its return value, and passing that to `FormatArray()` with the result output.

Solution 1

Keep the sorted array in its **own** variable rather than overwriting `Colours`: later parts sort the same data by other methods and each needs the unsorted original. **Mark scheme**

- 1 mark each
- ▪
- Calling Bubble() with array as parameter and using/storing return value
- ▪
- Calling FormatArray() with return value from Bubble() and outputting return value
- 2
- Python
- BubbleSorted = Bubble(Colours)
- print(FormatArray(BubbleSorted))
- VB.NET
- Dim BubbleSorted(45) As String
- BubbleSorted = Bubble(Colours)
- Console.WriteLine(FormatArray(BubbleSorted))

Solution 1

(d)(iii)

Worked steps

One mark, for the screenshot showing the **sorted** data.
Compare it against the output from part (b)(iii): the same 45 strings, now in order. That side-by-side is the evidence the mark is for.

Mark scheme

- 1 mark for sorted data
- e.g.
- 1
- 9618/43
- October/November 2024