Past-paper walk-through

Recursion ▪ 19.2

eXercise

Questions in this set

- 9618/33 N25 Q11 ▪ 3 marks
- 9618/43 N25 Q3 ▪ 20 marks
- 9618/33 J25 Q13 ▪ 4 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

9618/33 N25 ▪ Q11 ▪ 3 marks

Describe when the use of recursion would be beneficial **and** give an example. **[3]**

Solution 11

- One mark per mark point (Max 3)
- MP1
- Recursion is beneficial for problems that can be broken down into smaller, repetitive problems
- MP2
- ... especially for problems that have many possible branches / are too complex for an iterative approach
- MP3
- An example e.g. solving mathematical series, sorting a pile of documents, etc.

Question 3

9618/43 N25 ▪ Q3 ▪ 20 marks

A program is written to perform different individual processes using arrays.

(a)(i) A recursive function `RecursiveCount()` takes three parameters:

- `ArrayCopy`, an array of integers
- `NumberElements`, the number of elements in the array of integers
- `DataToFind`, an integer data to find in the array.

The function counts and returns the number of times `DataToFind` is in `ArrayCopy`
The recursive algorithm:

- returns 0 if there are no elements in `ArrayCopy`
- compares the first element in `ArrayCopy` with `DataToFind`
  - if the element matches, the function returns 1 added to the return value from a recursive call (passing the array without the first element)

Question 3

- if the element does not match, the function returns the return value from a recursive call (passing the array without the first element).

Write program code for `RecursiveCount()`
Save your program as **Question3_N25**.
Copy and paste the program code into part **3(a)(i)** in the evidence document. **[6]**
(a)(ii) The main program stores the following data in a 1D array of integers in the order given:
0 5 1 2 5 9 9 6 5 0
The main program calls `RecursiveCount()` with the parameters:

- 0 as the data to find
- 10 as the number of elements
- the array of 10 integers.

Question 3

The return value is output.
Write program code for the main program.
Save your program.
Copy and paste the program code into part **3(a)(ii)** in the evidence document. **[3]**

(a)(iii) Test your program.
Take a screenshot of the output(s). **[1]**

Question 3

9618/43 N25 ▪ Q3 ▪ 20 marks

A program is written to perform different individual processes using arrays.

(b)(i) Write program code to amend the main program to store the string `"x=0;y=1;x=x+y;y++;"` in a local variable. **[1]**

(b)(ii) The function `SplitData()` splits a string parameter into **four** individual lines of code.

The function creates an array of the strings where each line is stored in a new array element without the semi-colon.

The function returns the array of strings.

Do **not** use an inbuilt function to split the string.

Write program code for `SplitData()` **[6]**

Question 3

(b)(iii) The main program needs to call `SplitData()` with the string from part 3(b)(i). The main program then needs to output each element from the returned array on a new line.

Write program code to amend the main program. **[2]**

(b)(iv) Test your program.

Take a screenshot of the output(s). **[1]**

Solution 3

(a)(i) Worked steps

Every recursive function needs two things: a **base case** that stops, and a **recursive case** that moves towards it.

```
FUNCTION RecursiveCount(DataToFind : INTEGER, ArrayCopy : ARRAY,
                        NumberElements : INTEGER) RETURNS INTEGER
    IF NumberElements = 0 THEN
        RETURN 0
    ENDIF

    IF ArrayCopy[0] = DataToFind THEN
        RETURN 1 + RecursiveCount(DataToFind, Tail(ArrayCopy), NumberElements - 1)
    ELSE
        RETURN RecursiveCount(DataToFind, Tail(ArrayCopy), NumberElements - 1)
    ENDIF
ENDFUNCTION
```

Solution 3

`Tail(ArrayCopy)` is the array **without its first element** — a slice in Python, a copy loop in Java or VB.NET.
Six marks:

1. Function header and end taking **three parameters**.
2. Checking for **zero elements** and returning 0 — the base case.
3. Otherwise **comparing the first element** to `DataToFind`.
4. On a match, returning the recursive call **plus 1**.
5. On no match, returning the recursive call **without adding 1**.
6. **Every** recursive call passing the array without its first element, `NumberElements - 1`, and the same `DataToFind`.

Solution 3

The base case must come **first**. Testing the first element of an empty array is an error in every language, and it is what an infinite recursion looks like just before it crashes.

Solution 3

Mark 6 is the one that decides whether it terminates. Every call must be strictly closer to the base case — one element shorter, one smaller count — and passing the array unchanged is the classic way to recurse forever.

Mark scheme

- 1 mark each
- ▪ Function header (and end) taking three parameters
- ▪ Checking number of elements for 0 and returning 0 (e.g. ArrayCopy = [] // len(ArrayCopy) == 0)
- ▪ (otherwise) comparing first array element to parameter DataToFind ...
- ▪ ... if match return recursive call adding 1
- ▪ ... if no match return recursive call without adding 1
- ▪ ... all recursive calls have array without first element, NumberElements-1 and DataToFind
- Example program code
- Java public static Integer RecursiveCount(Integer DataToFind, Integer[] ArrayCopy, Integer
- NumberElements){ if(NumberElements > 0){
- Integer[] NewArray = new Integer[NumberElements-1];
- for(Integer x = 1; x < NumberElements; x++){

Solution 3

(a)(ii) Worked steps

```
MyArray <- [0, 5, 1, 2, 5, 9, 9, 6, 5, 0]
OUTPUT RecursiveCount(0, MyArray, 10)
```

Solution 3

Three marks: **storing the data** in an array in the given order; **calling `RecursiveCount()` with the correct parameters**; and **outputting** the return value. The arguments go in the order the header declares them — the value to find, then the array, then the element count. And the count is **10**, the number of elements, not the last index.

Solution 3

With this data the answer is 3: the value 0 appears at the first, ninth and tenth positions. Checking the returned number against the data by eye is a two-second test of the whole function. **Mark scheme**

- 1 mark each
- ▪ Storing the correct data in an array
- ▪ Calling RecursiveCount() with the correct parameters
- ▪ Outputting the return value
- Example program code
- Java
- Integer[] MyArray = new Integer[]{0, 5, 1, 2, 5, 9, 9, 6, 5, 0};
- System.out.println(RecursiveCount(0, MyArray, 10));
- VB.NET
- Dim MyArray() As Integer = {0, 5, 1, 2, 5, 9, 9, 6, 5, 0}
- Console.WriteLine(RecursiveCount(0, MyArray, 10))
- Python
- MyArray = [0,5,1,2,5,9,9,6,5,0] print(RecursiveCount(0, MyArray, 10))

Solution 3

(a)(iii)

Mark scheme

- 1 mark for screenshot showing correct output of 2
- Example

Solution 3

(b)(i) Worked steps

```
Code <- "x=0;y=1;x=x+y;y++;"
```

Solution 3

One mark, for storing the string in a variable. Copy it **exactly**, including every semicolon and the one at the end — part (b)(ii) splits on those, and a missing final semicolon loses the last line. **Mark scheme**

- 1 mark for storing the string in a variable
- Example program code
- Java
- String Code = "x=0;y=1;x=x+y;y++;";
- VB.NET
- Dim Code As String = "x=0;y=1;x=x+y;y++;"
- Python
- Code = "x=0;y=1;x=x+y;y++;"

Solution 3

(b)(ii) Worked steps

Walk the string one character at a time, building up each line and storing it whenever a semicolon is reached.

```
FUNCTION SplitData(DataString : STRING) RETURNS ARRAY
    DECLARE SplitDataArray : ARRAY[0:9] OF STRING
    DECLARE Count, Index : INTEGER
    DECLARE TempString, ThisChar : STRING

    Count <- 0
    TempString <- ""
    FOR Index <- 1 TO LENGTH(DataString)
        ThisChar <- MID(DataString, Index, 1)
        IF ThisChar = ';' THEN
            SplitDataArray[Count] <- TempString
            Count <- Count + 1
            TempString <- ""
        ELSE
            TempString <- TempString & ThisChar
        ENDIF
    NEXT Index
```

Solution 3

Six marks: the header taking **one string parameter**; a loop through the characters; **comparing each to `';'`**; concatenating the character to a working string until one is found; **storing that string in the array** when it is; and **returning the array without the semicolons**.

Two things make it work:

- **Reset `TempString` to empty after storing.** Forget that and each line accumulates all the previous ones.
- **The semicolon is not appended.** It goes down the `IF` branch, so it never reaches `TempString` — which is what "without semicolons" in the last mark point means.

Solution 3

Mark scheme

- 1 mark each
- ▪ Function header, taking one string parameter
- ▪ Loop e.g. four times/through each character in parameter
- ▪ Comparing character from parameter to ';' ...
- ▪ ... concatenate current character to a string until ';' is found
- ▪ ... when ';' found storing string in array
- ▪ Returning string array without semicolons
- Example program code
- Java public static String[] SplitData(String DataString){
- String[] SplitDataArray = new String[10];
- Integer Count = 0;
- String TempString = "";
- String Character = "";
- Integer LastElement = 0;

Solution 3

(b)(iii) Worked steps

```
SplitDataArray <- SplitData(Code)
FOR Index <- 0 TO 3
    OUTPUT SplitDataArray[Index]
NEXT Index
```

Solution 3

Two marks: calling `SplitData()` with the string and **storing or using** the returned array, and **outputting each element on a new line**.

Solution 3

Output them in a loop rather than four separate statements — the loop is what makes the marker's "each element on a new line" visibly true, and it does not have to change if the string gains a fifth statement. **Mark scheme**

- 1 mark each
- ▪ Calling SplitData() with array as argument and storing/using return value
- ▪ Outputting each element of the returned array on a new line
- Example program code
- Java
- String Code = "x=0;y=1;x=x+y;y++;";
- String[] SplitDataArray = new String[10];
- SplitDataArray = SplitData(Code);
- for(Integer x = 0; x < 4; x++){
- System.out.println(SplitDataArray[x]);
- }
- VB.NET

Solution 3

(b)(iv)

Mark scheme

- 1 mark for output x=0 y=1 x=x+y y++
- Example

Question 13

9618/33 J25 ▪ Q13 ▪ 4 marks

The recursive procedure `Delete()` is defined as follows:
PROCEDURE Delete(Index, Target) IF Numbers[Index] $>$ 0 THEN IF Numbers[Index] $>=$ Target THEN Numbers[Index] <- Numbers[Index + 1] ENDIF Index <- Index + 1 CALL Delete(Index, Target) ENDIF ENDPROCEDURE
An array `Numbers` is used to store a sorted data set of non-zero positive integers. Unused cells contain zero.
The contents of the array at the start of the algorithm are:

Numbers									
2	3	7	11	15	17	19	23	0	0

Question 13

Complete the trace table for the algorithm for the procedure call:
CALL Delete(1, 15)

		Numbers									
Index	**Target**										
		2	3	7	11	15	17	19	23	0	0

Question 13

[4]

Question 13

		Numbers									
Index	**Target**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**
		2	3	7	11	15	17	19	23	0	0

[4]

Question 13

Numbers									
[1]	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**
2	3	7	11	15	17	19	23	0	0

Solution 13

Worked steps

A recursive trace is the same bookkeeping as any other, with **one addition**: each call gets its own copy of the parameters, so the table has a row every time the procedure is entered as well as every time a value changes.

The method:

1. Write the **initial call's** `Index` and `Target`.
2. Work through the procedure body one line at a time, filling only the columns that change.
3. When it **calls itself**, start a new row with the **new argument values** and carry on there — the caller is suspended, not finished.

Solution 13

Worked steps

4. When a call **returns**, resume the caller at the line after its recursive call. Its own `Index` and `Target` are unchanged; they were never touched by the call.
5. Stop when the **base case** is reached and every suspended call has resumed.

Four marks, and they are grouped by column: the `Index` and `Target` columns together; the `Numbers[5]` column; `Numbers[6]` and `Numbers[7]`; and `Numbers[8]` **with no incorrect data written into columns 1, 2, 3, 4, 9 or 10**.

That last mark is the one to be careful about, and it is worth reading twice. Elements outside the shifted region are **never written**, so their columns must stay **blank**. Writing the unchanged value into them is treated as incorrect data — a blank means "unchanged", and filling it in claims an assignment that never happened.

The procedure shifts each element down one place from the deletion point onward, which is why consecutive elements change in sequence: `Numbers[5]` takes the old

Solution 13

Worked steps

`Numbers[6]`, then `Numbers[6]` takes `Numbers[7]`, and so on. Recursion here is doing the job of a loop, one element per call.
Keep the calls visually nested — indent each new call one step — so it stays obvious which values belong to which invocation when they start returning. **Mark scheme**

- One mark for each marking point
- MP1
- Correct Index and Target columns
- MP2
- Correct Numbers[5] column

Solution 13

Worked steps

- MP3
- Correct Numbers[6] and Numbers[7] columns
- MP4
- Correct Numbers[8] column and no incorrect data added to any other of columns [1, 2, 3, 4, 9, 10]
- Numbers
- Index
- Target

1

2

Solution 13

Worked steps

3

4

5

6

7

8

9

10

- 1

Solution 13

Worked steps

- 15
- 2
- 3
- 7
- 11
- 15
- 17
- 19
- 23

Solution 13

Worked steps

- 0
- 0
- 2
- 3
- 4
- 5
- 17
- 6
- 19

Solution 13

Worked steps

- 7
- 23
- 8
- 0
- 9