2.8 Recursion and Advanced Algorithms: Test Pack

AQA 7517 · Calculators not allowed · about 30 minutes

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A programmer writes the following recursive AQA pseudocode function to calculate the factorial of a non-negative integer n (where n! = n x (n-1) x (n-2) x ... x 1, and 0! is defined as 1).

```
FUNCTION Factorial(n)
   IF n = 0 THEN
      RETURN 1
   ELSE
      RETURN n * Factorial(n - 1)
   ENDIF
ENDFUNCTION
```

(a) State the base case of the function Factorial(n). **(1)**

(b) Complete a trace table showing the value of n passed to each recursive call made when evaluating Factorial(5), and the value each call returns, from the first call (n = 5) to the last. **(5)**

(c) State the value returned by the original call Factorial(5). **(1)**

(d) The function is instead called with a negative argument, Factorial(-3). Explain, with reference to the base case and the call stack, what would happen, and why this is a general risk when writing recursive functions. **(3)**

(Total for Question 1 is 10 marks)

© Revision Library | revisionlibrary.co.uk

2 A binary search tree stores nodes with a value, a left child pointer and a right child pointer (NULL if there is no child). The tree used in this question has the following structure:

Root: 50
50's left child: 30, 50's right child: 70
30's left child: 20, 30's right child: 40
70's left child: 60, 70's right child: 80
(20, 40, 60 and 80 are all leaves, with no children)

The following recursive procedure is defined.

```
PROCEDURE InOrder(node)
   IF node <> NULL THEN
      CALL InOrder(node.left)
      OUTPUT node.value
      CALL InOrder(node.right)
   ENDIF
ENDPROCEDURE
```

(a) State the standard name given to the tree traversal implemented by this pseudocode, in terms of the order in which a node's left subtree, the node itself, and its right subtree are processed. **(1)**

(b) State the full sequence of values output when CALL InOrder(root) is made on the tree described above. **(1)**

(c) Complete a trace table showing, in order, the node value passed to each call InOrder(node) made on a node that actually holds a value (i.e. exclude any call made where node = NULL, which returns immediately), listing the order in which these calls are first entered. **(4)**

(d) The order of calls found in part (c) is identical to a different, standard tree traversal order. Identify this traversal, and justify your answer by referring to when a node's own value would be processed relative to its child subtrees in that traversal. **(2)**

(Total for Question 2 is 8 marks)

© Revision Library | revisionlibrary.co.uk

3 A games programmer called Rhys is writing a recursive backtracking function to find a path through a maze, represented as a 2D array grid, where grid[row][col] = 0 means an open cell and grid[row][col] = 1 means a wall. The array visited keeps track of cells already tried during the current search, and target holds the row and column of the exit cell. Two lines of the pseudocode, labelled ____(i)____ and ____(ii)____, have been removed.

```
FUNCTION SolveMaze(grid, row, col, target, visited)
   IF row < 0 OR row >= LEN(grid) OR col < 0 OR col >= LEN(grid[0]) THEN
      RETURN FALSE
   ENDIF
   IF grid[row][col] = 1 OR visited[row][col] = TRUE THEN
      RETURN FALSE
   ENDIF
   IF row = target[0] AND col = target[1] THEN
      RETURN TRUE
   ENDIF
   ____(i)____
   IF SolveMaze(grid, row + 1, col, target, visited) = TRUE THEN
      RETURN TRUE
   ENDIF
   IF SolveMaze(grid, row - 1, col, target, visited) = TRUE THEN
      RETURN TRUE
   ENDIF
   IF SolveMaze(grid, row, col + 1, target, visited) = TRUE THEN
      RETURN TRUE
   ENDIF
   IF SolveMaze(grid, row, col - 1, target, visited) = TRUE THEN
      RETURN TRUE
   ENDIF
   ____(ii)____
   RETURN FALSE
ENDFUNCTION
```

(a) State the missing pseudocode for the lines labelled ____(i)____ and ____(ii)____. **(2)**

(b) State the purpose of the visited array in this algorithm. **(1)**

(c) Explain why line ____(ii)____ (resetting visited[row][col] back to FALSE) is essential for the algorithm to correctly explore every possible path through the maze, rather than just the first path it happens to try. **(2)**

© Revision Library | revisionlibrary.co.uk

(d) State what the original call to SolveMaze returns if no path exists between the start cell and the target, and explain, with reference to the recursive calls, how the algorithm reaches this result. **(3)**

(Total for Question 3 is 8 marks)

© Revision Library | revisionlibrary.co.uk

Mark scheme · 2.8 Recursion and Advanced Algorithms: Test Pack

Question 1

(a) n = 0 (the function returns 1 immediately without a further recursive call) oe **B1**

(a) Answer: The base case is n = 0, where the function returns 1 without making any further recursive call.

(b) call with n = 1 returns 1 (1 * Factorial(0)); the n = 0 call, which returns 1 and is the base case, may also be shown as an extra first row (do not penalise) **B1**

(b) call with n = 2 returns 2 **B1**

(b) call with n = 3 returns 6 **B1**

(b) call with n = 4 returns 24 **B1**

(b) call with n = 5 returns 120 cao **B1**

(b) Answer:

n	returns
0	1 (base case, the last call made)
1	1
2	2
3	6
4	24
5	120

(c) 120 cao ft from (b) **B1**

(c) Answer: 120

(d) the base case n = 0 is never reached, because each recursive call passes n - 1, so the sequence of values (-3, -4, -5, ...) decreases without ever equalling 0 **B1**

(d) the function therefore keeps calling itself indefinitely (infinite recursion), with each call adding a new frame to the call stack, but no call ever returns **B1**

(d) eventually the call stack runs out of available memory, causing a stack overflow error and the program to crash; this shows why every recursive function needs a base case that is guaranteed to be reached for all inputs it might realistically be called with **B1**

(d) Answer: Because the base case only checks for n = 0, and each call passes n - 1, calling Factorial(-3) produces the sequence -3, -4, -5, ... which never equals 0. The base case is never reached, so the function recurses indefinitely, adding a new stack frame on every call without any of them returning, until the call stack runs out of memory and the program crashes with a stack overflow error. This illustrates the general risk that a recursive function's base case must be reached for every input it might realistically be called with.

Question 2

(a) in-order traversal (left subtree, then the node itself, then right subtree) oe **B1**

(a) Answer: In-order traversal (left subtree, then the node, then right subtree).

(b) 20, 30, 40, 50, 60, 70, 80 cao **B1**

(b) Answer: 20, 30, 40, 50, 60, 70, 80

(c) 50, 30, 20 correct as the first three calls entered (descending down the left spine) **B1**

(c) 40 correctly placed as the fourth call entered (after backtracking from 20, following node 30's right pointer) **B1**

(c) 70, 60 correctly placed as the fifth and sixth calls entered (after backtracking all the way to 50, then following its right pointer, then 70's left pointer) **B1**

(c) complete, correct full order: 50, 30, 20, 40, 70, 60, 80 cao **A1**

(c) Answer: 50, 30, 20, 40, 70, 60, 80

(d) pre-order traversal **B1**

(d) pre-order processes a node's own value before (immediately on entering, ahead of) recursing into its left and right subtrees, i.e. "node, then left, then right", which is exactly the order that calls are entered in part (c), regardless of when in-order's own OUTPUT statement actually runs **B1**

(d) Answer: Pre-order traversal. Pre-order visits (processes) a node before recursing into either of its subtrees, "node, left, right", and since every traversal, including in-order, always enters InOrder(node.left) as its very first action inside a call, the order in which calls are first entered on real nodes always matches the node-first order of a pre-order traversal, whichever traversal's own OUTPUT statement is actually being run.

© Revision Library | revisionlibrary.co.uk

Question 3

(a) (i): visited[row][col] <- TRUE **B1**

(a) (ii): visited[row][col] <- FALSE **B1**

(a) Answer:(i) visited[row][col] <- TRUE
(ii) visited[row][col] <- FALSE

(b) it records which cells are currently part of the path being explored, so the algorithm does not revisit a cell already being tried and get stuck moving back and forth between the same cells forever (oe) **B1**

(b) Answer: The visited array keeps track of which cells are already part of the path currently being explored, so the algorithm does not try to move back into a cell it is already using and end up looping between the same cells indefinitely.

(c) without this line, once a cell had been marked visited it would remain marked as such even after the recursive call exploring onward from it had returned FALSE (a dead end) **B1**

(c) so if a different, still-active call later tried to pass back through that same cell as part of an alternative route to the target, it would incorrectly be treated as unavailable, and a genuine solution path could be missed **B1**

(c) Answer: Without resetting visited[row][col] back to FALSE, a cell would stay marked as visited even after the branch of the search starting from it had failed (returned FALSE). If a different recursive call later needed to pass back through that same cell to reach the target by another route, it would wrongly be blocked, so the algorithm could fail to find a path that actually exists.

(d) FALSE is returned **B1**

(d) every recursive call from every reachable cell eventually returns FALSE, either immediately (hitting a wall, the maze boundary, or an already-visited cell) or because all four of its own recursive calls returned FALSE **B1**

(d) this propagates back up through every call in the chain (each one in turn finding none of its own four recursive calls returned TRUE), all the way back to the original call, which therefore also returns FALSE **B1**

(d) Answer: The original call returns FALSE. Every recursive call made during the search eventually returns FALSE, either straight away (because it hits a wall, the edge of the maze, or a cell already visited on this path) or because it tried all four directions and each of those calls also returned FALSE. Since no call anywhere in the search ever finds a path to the target, this FALSE result passes back up through every level of recursion until the original call itself returns FALSE.

© Revision Library | revisionlibrary.co.uk