Chapter 25 Recursion: Answers to coursebook questions and tasks

Syllabus sections covered: 4.1.4

Task 25.01 recursive factorial function

<table>
<tr><td>Python</td><td><pre>def Factorial(n) :
   if n == 0 :
      Result = 1
   else :
      Result = n * Factorial(n-1)
   return(Result)</pre></td></tr>
<tr><td>VB.NET</td><td><pre>Function Factorial(n) As Integer
    Dim Result As Integer
    If n = 0 Then
        Result = 1
    Else
        Result = n * Factorial(n - 1)
    End If
    Return (Result)
End Function</pre></td></tr>
<tr><td>Pascal</td><td><pre>function Factorial(n: integer) : integer;
begin
   if n = 0
      then
         Result := 1
      else
         Result := n * Factorial(n-1);
end;</pre></td></tr>
</table>

Question 25.01

The third rule is not satisfied. With each iteration, the base case is further away. This means the recursive calls go on forever (or until the heap is full // memory is used up).

Task 25.02

Call Number	procedure call	n=0	n=1	OUTPUT
1	X(19)	False	False	
2	X(9)	False	False	
3	X(4)	False	False	
4	X(2)	False	False	
5	X(1)	False	True	1
(4)	X(2)			0
(3)	X(4)			0
(2)	X(9)			1
(1)	X(19)			1

It converts a positive integer into binary

Task 25.03

<table>
<tr><td>Python</td><td><pre>def Factorial(n) :
   global CallNumber
   CallNumber += 1
   print(CallNumber,'   ', n)
   if n == 0 :
      Result = 1
   else :
      Result = n * Factorial(n-1)
      print('Return value: ', Result)
   return(Result)

CallNumber = 0
print(Factorial(3))</pre></td></tr>
<tr><td>VB.NET</td><td><pre>Module Module1
    Dim CallNumber
    Function Factorial(n) As Integer
        Dim Result As Integer
        CallNumber += 1
        Console.WriteLine(CallNumber & "   " & n)
        If n = 0 Then
            Result = 1
        Else
            Result = n * Factorial(n - 1)
        End If
        Console.WriteLine("Return value: " & Result)
        Return (Result)
    End Function

    Sub Main()
        CallNumber = 0
        Console.WriteLine(Factorial(3))
        Console.ReadLine()
    End Sub

End Module</pre></td></tr>
<tr><td>Pascal</td><td><pre>program Project2;

{$APPTYPE CONSOLE}

uses
  SysUtils;
 var CallNumber : integer;

  function Factorial(n: integer) : integer;
  begin
     CallNumber := CallNumber + 1;
     writeln(CallNumber: 5, n:5);
     if n = 0
        then
           Result := 1
        else
           Result := n * Factorial(n-1);
     Writeln('Return value: ', Result);
  end;</pre></td></tr>
</table>

	<pre>begin
	  CallNumber := 0;
	  Writeln(Factorial(3));
	  Readln;
	end.</pre>

Exam style questions

1 a Iteration: a number of program statements are executed repeatedly.

Recursion: a subroutine calls itself, re-entering the routine with different local variables.

b Advantages of recursive subroutines are that recursive solution is often much shorter than non-recursive ones and that when the solution to a problem is essentially recursive (such as factorial) the programmer can write a program that mirrors the solution. The disadvantage of recursive subroutines is that if the recursion continues for too long, the stack of return addresses (stack frames) may become full.

2 a A subroutine is recursively defined if in its definition there is a call to itself

b

Call number	Procedure call	Exponent = 0	Result
1	Power(2,4)	FALSE	
2	Power(2,3)	FALSE	
3	Power(2,2)	FALSE	
4	Power(2,1)	FALSE	
5	Power(2,0)	TRUE	1
(4)	Power(2,1)		2 * 1 = 2
(3)	Power(2,2)		2 * 2 = 4
(2)	Power(2,3)		2 * 4 = 8
(1)	Power(2,4)		2 * 8 = 16

Returns 16

c When the procedure is called the return address and the values of the local variables and the partial result of the calculation are stored on the stack. When the base case of Exponent = 0 is reached, the result is stored on the stack. The calls unwind and the return address, the values of the local variables and the result of the previous call are taken off the stack. This unwinding continues until the place in the program that called Power(2,4) is returned to.

d

```
FUNCTION Power(Base : INTEGER, Exponent : INTEGER)
RETURNS INTEGER
   Result ← 1
   WHILE Exponent > 0
      Result ← Result * Base
      Exponent ← Exponent - 1
   ENDWHILE
   RETURN Result
ENDFUNCTION
```

e i Fewer overheads when calling function with a large exponent.

Python

```python
def Fibonacci(n) :
   global CallNumber
   CallNumber +=  1
   ReturnCall = CallNumber  # local variable to remember call number
   print('This is call: ', CallNumber,'  ','n =', n, end='  ')
   if n == 0 or n== 1 :
      print('TRUE', end='    ')
      Result = n
   else :
      print('FALSE')
      Result = Fibonacci(n-1) + Fibonacci(n-2)
      print('Returning from call: ', ReturnCall, '   n =', n, end='   ')
   print('Return value: ', Result)
   return(Result)

CallNumber = 0
print('Fibonacci(4) = ', Fibonacci(4))
```

VB.NET

```vbnet
Module Module1
   Dim CallNumber
   Function Fibonacci(n) As Integer
      Dim ReturnCall, Result As Integer
      CallNumber += 1
      ReturnCall = CallNumber  'local variable to remember call number
      Console.Write("This is call: " & CallNumber & "  n = " & n & "   ")
      If n = 0 Or n = 1 Then
         Console.Write("TRUE    ")
         Result = n
      Else
         Console.WriteLine("FALSE")
         Result = Fibonacci(n - 1) + Fibonacci(n - 2)
         Console.Write("Returning from call: " & ReturnCall & " n = " & n & "
")
      End If
      Console.WriteLine("Return value: " & Result)
      Return (Result)
   End Function

   Sub Main()
      CallNumber = 0
      Console.WriteLine("Fibonacci(4) = " & Fibonacci(4))
      Console.ReadLine()
   End Sub

End Module
```

Pascal

```pascal
program Project2;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var  CallNumber : integer;
Function Fibonacci(n: integer) : Integer;
    var ReturnCall : Integer; //local to remember call number
    begin
        CallNumber := CallNumber + 1;
        ReturnCall := CallNumber;
        Write('This is call: ', CallNumber, '  n = ', n, '   ');
        If (n = 0) Or (n = 1)
        Then
           begin
```

```
                Write('TRUE    ');
                Result := n;
             end
          Else
             begin
                WriteLn('FALSE');
                Result := Fibonacci(n - 1) + Fibonacci(n - 2);
                Write('Returning from call: ', ReturnCall, '   n = ', n, '   ');
             end;
          WriteLn('Return value: ', Result)
      end;

var FibonacciResult : integer;
begin
  CallNumber := 0;
  FibonacciResult :=  Fibonacci(4);
  WriteLn('Fibonacci(4) = ', FibonacciResult);
  ReadLn
end.
```

ii It is an elegant solution and reflects the mathematical definition.

3 a i Line 04 is the base case

ii Line 06 is the general case

b

Call number	Procedure call	n	(n = 0) OR (n = 1)	Result
1	Fibonacci(4)	4	FALSE	
2	Fibonacci(3)	3	FALSE	
3	Fibonacci(2)	2	FALSE	
4	Fibonacci(1)	1	TRUE	1
5	Fibonacci(0)	0	TRUE	0
(3)		2		1
6	Fibonacci(1)	1	TRUE	1
(2)		3		2
7	Fibonacci(2)	2	FALSE	
8	Fibonacci(1)	1	TRUE	1
9	Fibonacci(0)	0	TRUE	0
(7)		2		1
(1)		4		3

Returns 3

Note: this is best demonstrated using a program with a programmed-in trace.