Syllabus Content:

4.1.4 Recursion

- show understanding of the essential features of recursion
- show understanding of how recursion is expressed in a programming language
- trace recursive algorithms
- write recursive algorithms
- show understanding of when the use of recursion is beneficial
- show awareness of what a compiler has to do to implement recursion in a programming language

Recursion

A very efficient way of programming is to make the same function work over and over again in order to complete a task.

One way of doing this is to use 'Recursion'.

Recursion is where a function or sub-routine calls itself as part of the overall process. Some kind of limit is built in to the function so that recursion ends when a certain condition is met.

A classic computer programming problem that make clever use of recursion is to find the factorial of a number. i.e. 4 factorial is 4! = 4 x 3 x 2 x 1

Example

Find factorial 3!. The mathematical symbol for factorial is exclamation mark '!'

A function to find the factorial of a number is shown below

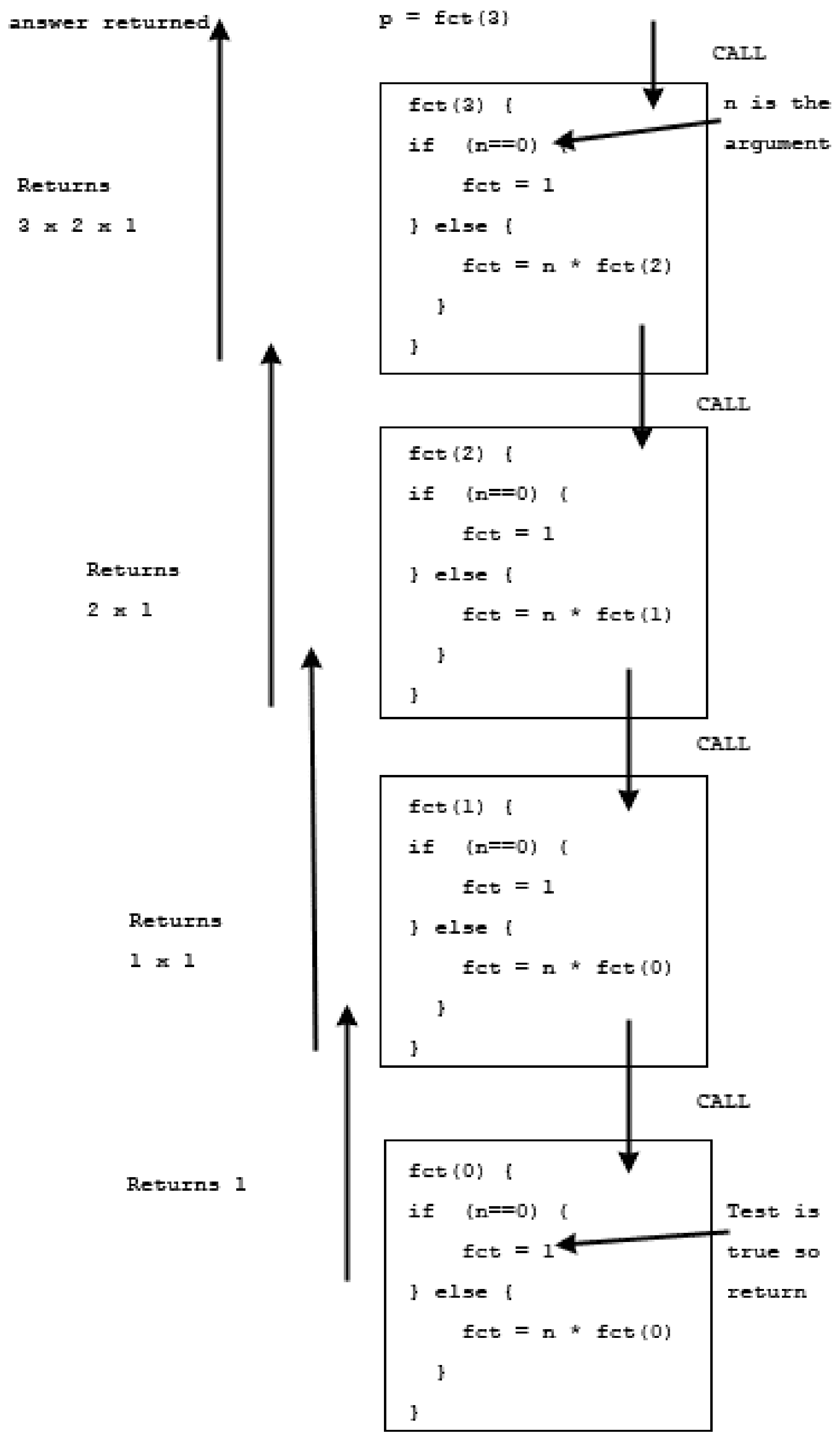
```
fct(n) {
if  (n==0) {
    fct = 1
} else {
    fct = n * fct(n-1)
  }
}
```

Notice that the function 'fct' above is calling itself within the code. Also notice that each time the function is called, a test is being carried out (if n==0) to see if the recursion should end.

To start it off the following statement is written

```
p = fct(3)
```

Step by step, this is what happens. Recursion winds and then unwinds.



RECURSION EXAMPLE

The factorial function is called with argument 3

- 3 is passed as an argument to the factorial function 'fct'
- the test (if n==0) is false and so the else statement is executed
- the statement fct = n * fct(n-1) becomes
- fct = 3 * fct(2)
- In order to resolve this another call is made to factorial with argument 2 this time
- **RECURSION** happens i.e. the function is calling itself as fct(2)
- 2 is passed as an argument to the factorial function 'fct'
- the test (if n==0) is false and so the else statement is executed
- the statement fct = n * fct(2-1) becomes
- fct = 2 * fctl(1)
- In order to resolve this another call is made to factorial with argument 1 this time
- **RECURSION** happens i.e. the function is calling itself as fct(1)
- 1 is passed as an argument to the factorial function 'fct'
- the test (if n==0) is false and so the else statement is executed
- the statement fct = n * factorial(1-1) becomes
- fct = 1 * fct(0)
- In order to resolve this another call is made to factorial with argument 0 this time
- **RECURSION** happens i.e. the function is calling itself as fctl(0)
- the test (if n==0) is TRUE and so the value 1 is returned to the calling function
- now each function call returns a value to the previous one, until the first function called returns a value to 'p'.

Programming a recursive subroutine

We can program the function factorial iteratively using a loop:

```
FUNCTION Fictorial : INTEGER:  RETURS  INTEGER
      Result ←  1
      FOR i ←     1 to n
            Result ←      Result  *  i
      NEXT
      RETURN Result
END FUNCTION
```

Or

```
FUNCTION Fictorial ( n : INTEGER)  RETURS  INTEGER
      IF n = 0
            THEN
                  Result ←       1
            ELSE
                  Result ←       n * Fictorial (n-1)

      END IF
RETURN Result
END FUNCTION
```

Programming a recursive subroutine in VB

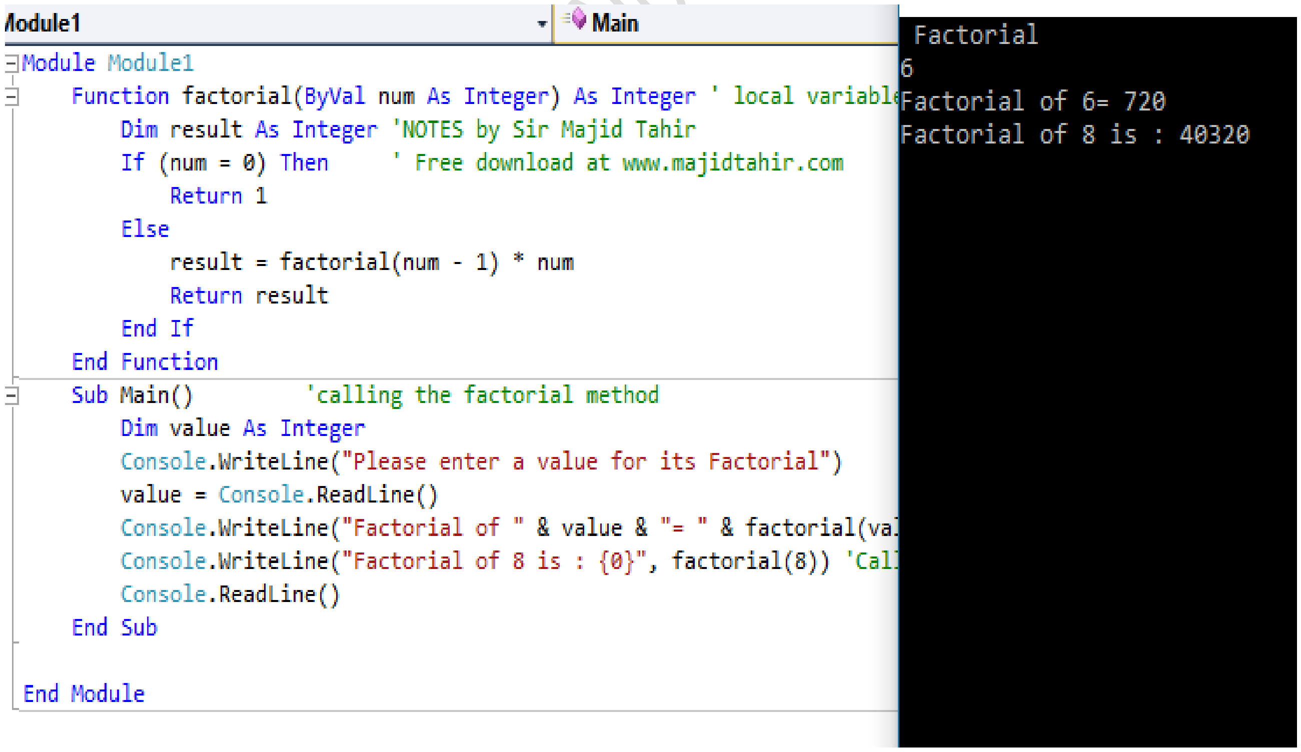
Following is an example that calculates factorial for a given number using a recursive function:

```
Module Module1
    Function factorial(ByVal num As Integer) As Integer ' local variable declaration
        Dim result As Integer
        If (num = 0) Then
            Return 1
        Else
            result = factorial(num - 1) * num
            Return result
        End If
    End Function
    Sub Main()         'calling the factorial method

        Dim value As Integer
        Console.WriteLine("Please enter a value for its Factorial")
        value = Console.ReadLine()
        Console.WriteLine("Factorial of " & value & "= " & factorial(value))
        Console.WriteLine("Factorial of 8 is : {0}", factorial(8)) 'Calling Factorial
function with direct value
        Console.ReadLine()
    End Sub

End Module
```

When the above code is compiled and executed, it produces the following result:



```
Module Module1
    Function factorial(ByVal num As Integer) As Integer ' local variabl
        Dim result As Integer 'NOTES by Sir Majid Tahir
        If (num = 0) Then     ' Free download at www.majidtahir.com
            Return 1
        Else
            result = factorial(num - 1) * num
            Return result
        End If
    End Function
    Sub Main()         'calling the factorial method
        Dim value As Integer
        Console.WriteLine("Please enter a value for its Factorial")
        value = Console.ReadLine()
        Console.WriteLine("Factorial of " & value & "= " & factorial(va
        Console.WriteLine("Factorial of 8 is : {0}", factorial(8)) 'Cal
        Console.ReadLine()
    End Sub

End Module
```

```
 Factorial
6
Factorial of 6= 720
Factorial of 8 is : 40320
```

Advantage of recursion

- Very efficient use of code

Disadvantage of recursion

- A faulty recursive function would never end and would rapidly run out of memory or result in a stack overflow thus causing the computer to freeze.
- Can be difficult to debug as it can fail many levels deep in the recursion
- Makes heavy use of the stack, which is a very limited resource compared to normal memory.

Recursion When to Use: You would process the list starting at the head or tail and then **recursively** traverse the list **using** the pointers. A tree is another case where **recursion** is often used Recursions are used when you satisfy of these conditions:

- You have a problem which is naturally recursive.
- The task must be indefinitely repetitive.
- At every round the same decision set must be applicable
- You can guarantee that you won't overflow the stack.

Function of compiler to implement recursion:

To understand this you just need to understand how a compiler interpret a function.
The compiler does not need to know whether the function is recursive or not. It just make CPU jump to the address of function entry and keep on executing instructions.

And that's why we can use that function even if its definition is not finished. The compiler just need to know a start address, or a symbol, and then it would know where to jump. The body of the function could be generated later.

However, you might want to know the **Tail Recursion**, that is a special case commonly in functional programming languages. The "tail recursion" means the recursive function call is the last statement in function definition.

When calling a function, the compiler need to push context and parameters into stack, and then recover the context and get return values from it. Thus, if your function calls itself and then itself ..., the stack would be too deep until the memory runs out. But if the function call is the last statement in function definition, then there would be no necessary to save context in the stack, and we can just overwrite it. Thus, the stack would not overflow even if the function calls itself infinitely.

References:

Computer Science AS & A Level Coursebook by Sylvia Langfield & Dave Duddell
http://teach-ict.com/as_as_computing/ocr/H447/F453/3_3_6/defining_syntax/miniweb/pg22.htm
https://stackoverflow.com/questions/40796473/how-do-compilers-understand-recursion