Recursion

CIE AS & A Level
Computer Science Revision Notes 9618

Syllabus Content:

19.2 Recursion
show understanding of recursion

Notes and guidance

Essential features of recursion.

How recursion is expressed in a programming Language.

Write and trace recursive algorithms

Show awareness of what a compiler has to do to translate recursive program code.

Notes and guidance

Use of stacks and unwinding

Have you ever seen a set of Russian dolls? At first, you see just one figurine, usually painted wood, that looks something like this:

When we open the doll, we find another in it little smaller than the first one, then another one in it and so on. We started with one big Russian doll, and we saw smaller and smaller Russian dolls, until we saw one that was so small that it could not contain another.

What do Russian dolls have to do with algorithms? Just as one Russian doll has within it a smaller Russian doll, which has an even smaller Russian doll within it, all the way down to a tiny Russian doll that is too small to contain

another, we'll see how to design an algorithm to solve a problem by solving a smaller instance of the same problem, unless the problem is so small that we can just solve it directly. We call this technique **recursion**.

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.

Example

A *recursive* procedure is one that calls itself. In general, this is not the most effective way to write Visual Basic code.
The following procedure uses recursion to calculate the factorial of its original argument.

```
VB

Function Factorial(n As Integer) As Integer
    If n <= 1 Then
        Return 1
    End If
    Return Factorial(n - 1) * n
End Function
```
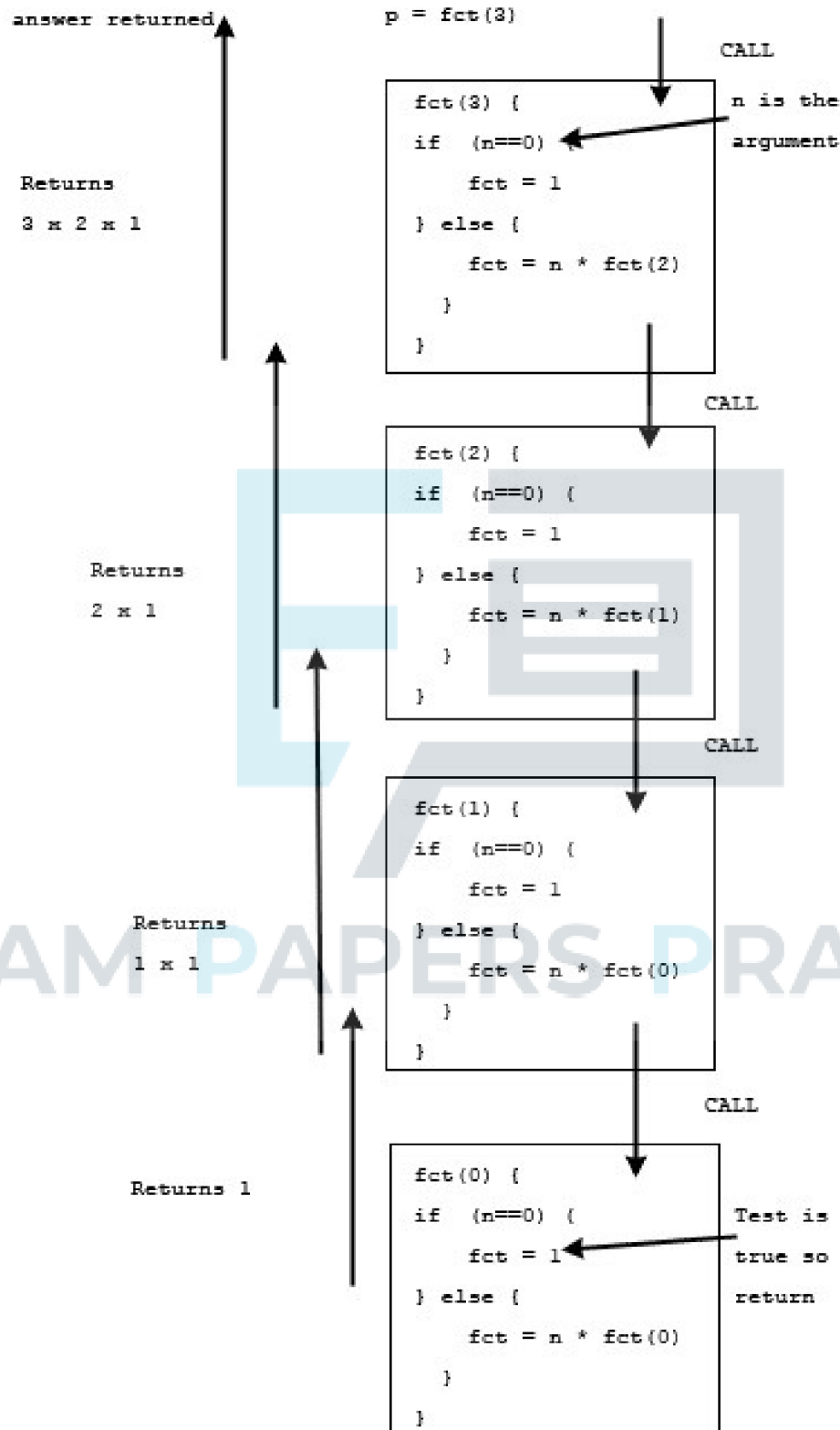
Step by step, this is what happens. Recursion winds and then unwinds.

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
Lets see how **Recursion winds and unwinds** in program given below:

answer returned

p = fct(3)

CALL

```
fct(3) {
if  (n==0)
     fct = 1
} else {
     fct = n * fct(2)
  }
}
```

n is the
argument

Returns

3 x 2 x 1

CALL

```
fct(2) {
if  (n==0) {
     fct = 1
} else {
     fct = n * fct(1)
  }
}
```

Returns

2 x 1

CALL

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

Returns

1 x 1

CALL

Returns 1

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

Test is
true so
return

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:

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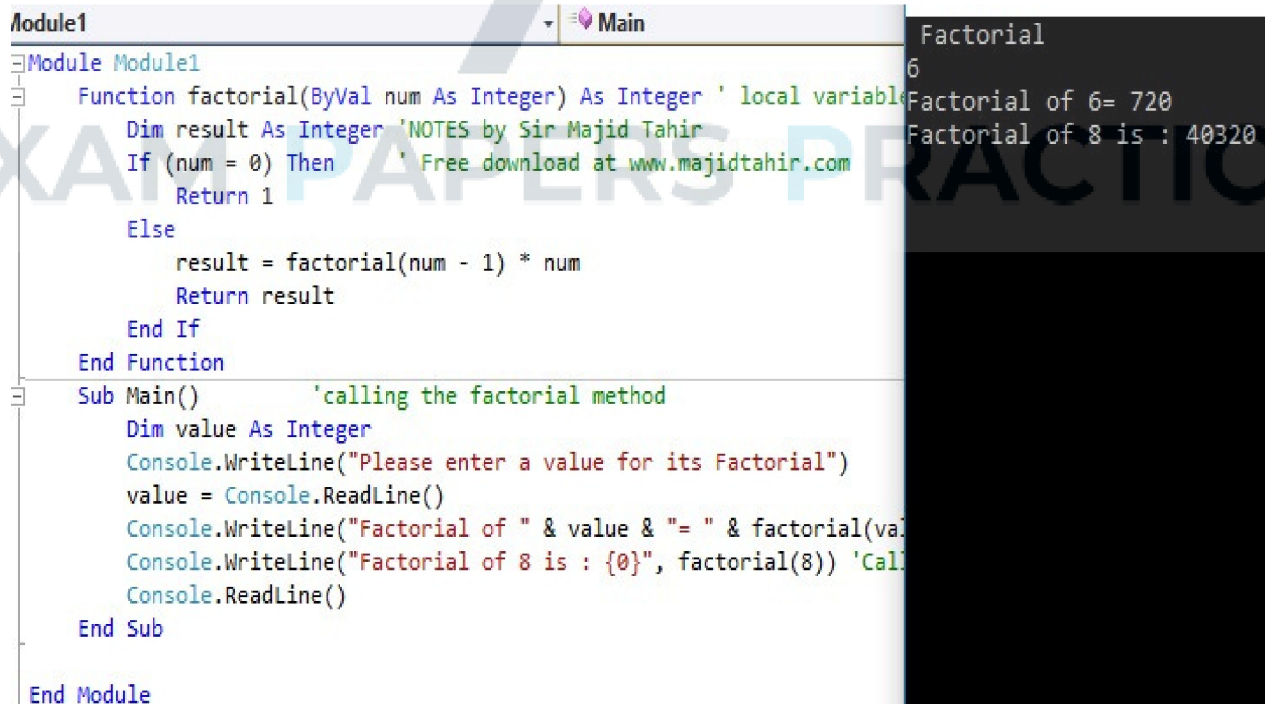
When the above code is compiled and executed, it produces the following result:

For more help, please visit our website www.exampaperspractice.co.uk

Advantage of recursion

Very efficient use of code if problem is naturally recursive.

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.

What is tail recursion?

A recursive function is tail recursive when a recursive call is the last thing executed by the function.
For example the following C++ function print() is tail recursive.

| C | Java | Python3 | C# |

```csharp
// An example of tail recursive function
static void print(int n)
{
    if (n < 0)
        return;

    Console.Write(" " + n);

    // The last executed statement
        // is recursive call
    print(n - 1);
}

// This code is contributed by divyeshrabadiya07
```

Why do we care?

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by the compiler.

Compilers usually execute recursive procedures by using a **stack**. This stack consists of all the pertinent information, including the parameter values, for each recursive call.

When a procedure is called, its information is **pushed** onto a stack, and when the function terminates the information is **popped** out of the stack.

Thus for the non-tail-recursive functions, the **stack depth** (maximum amount of stack space used at any time during compilation) is more.

The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.