Topic: 4.1.4 Recursion

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). It is when a function or procedure contains a call to itself or when you define a subroutine in terms of itself. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.

When we write a method for solving a particular problem, one of the basic design techniques is to break the task into smaller subtasks. For example, the problem of adding (or multiplying) **N** consecutive integers can be reduced to a problem of adding (or multiplying) **N – 1** consecutive integers:

```
1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]
1 * 2 * 3 *... * n = n * [1 * 2 * 3 * .. * (n-1)]
```

For example, if N = 9 then you would have to add 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9. OR you could simply do 9 + (1 + 2 + …… + 8).

Therefore, if we introduce a method sumR(n) or multiplyR(n) that adds (or multiplies) integers from 1 to n, then the above arithmetics can be rewritten as:

```
sumR(n)   = n + sumR(n-1)
timesR(n) = n * timesR(n-1)
```

```
public int sum(int n)
{
   int res = 0;
   for(int i = 1; i = n; i++)
      res = res + i;

   return res;
}
```

Loop

```
public int sumR(int n)
{
  if(n == 1)
     return 1;
  else
     return n + sumR(n-1);
}
```

Recursion

An important point to keep in mind for recursion is that it is called to do "part" of the work. If it is called to do "all" the work, the result would be an infinite recursion (equivalent to an infinite loop). The recursive function should run until all the work is done.

03-111-222-ZAK | OlevelComputer AlevelComputer | @zakonweb | zak@zakonweb.com | www.zakonweb.com

Topic: 4.1.4 Recursion

Let's take an example of the ***factorial*** function of a positive integer n, written **n!**, is the product of all the positive integers from 1 up to and including **n**. Thus, we have:

1! = 1
2! = 1 * 2 = 2
3! = 1 * 2 * 3 = 6
4! = 1 * 2 * 3 * 4 = 24
Etc.
Note that for example, 4! = 1 * 2 * 3 * 4 = (1 * 2 * 3) * 4 = 3! * 4, and in general n! = (n - 1)! * n.

Here is a function to calculate the factorial of a number (in Pascal):

```
01 FUNCTION Factorial(n)
02 IF n = 1 THEN
03 RETURN 1
04 ELSE
05 RETURN n * Factorial(n – 1)
06 END IF
07 END FUNCTION
```

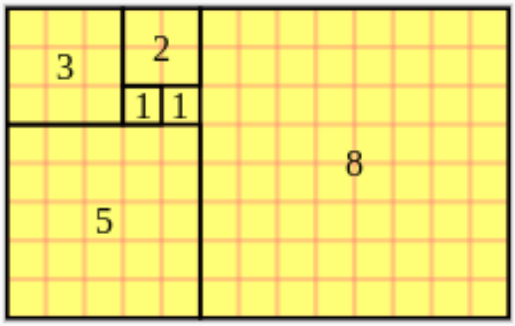
Note that the stopping condition in the above code is IF n = 1 and the recursion happens on line 5.

Suppose we want to trace the execution of x = Factorial(3)

Line Number	Function Call	Returned Value	Condition	State of Function Call
01	Factorial(3)			
02			False	
05 →		3 * Factorial(2)		New Call: n=2 n=3: Postponed
01	Factorial(2)			
02			False	
05 →		2 * Factorial(1)		New Call: n=1 n=2: Postponed
01	Factorial(1)			
02			True	
03		1		
07				n=1: Completed
→ 05		2 * 1 = 2		n=2: *Resumed*
07				n=2: Completed
→ 05		3 * 2 = 6		n=3: *Resumed*
07				n=3: Completed

Topic: 4.1.4 Recursion

Fibonacci number example (pronounced Fibonachy)



Fibonacci numbers are found in nature and have fascinated mathematicians for centuries

Another good example is a method to calculate the Fibonacci numbers. By definition, the first two Fibonacci numbers are 0 and 1, and each subsequent number is the sum of the previous two:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$$

For example, take the sixth number, 5. This is the sum of the fifth number, 3, and the fourth number 2.

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recursive statement

The Fibonacci sequence can $F_n = F_{n-1} + F_{n-2},$ be defined in a programming language too:

```
public int fibonacci(int n)
{
   if (n <= 0) return 0;
   else if (n == 1) return 1
   else return fibonacci(n-1) + fibonacci(n-2);
}
```


03-111-222-ZAK | OlevelComputer AlevelComputer | @zakonweb | zak@zakonweb.com | www.zakonweb.com

Topic: 4.1.4 Recursion

Iteration vs Recursion

Iteration	Recursion
01 INPUT n 02 a=1 03 FOR i=1 TO n 04 a=a*n 05 NEXT i 06 OUTPUT a 07 END	01 FUNCTION Factorial(n) 02 IF n=1 THEN 03 RETURN 1 04 ELSE 05 RETURN n*Factorial(n-1) 06 END IF 07 END
There is only one set of variable values that are used within the iteration.	A recursive subroutine is called repeatedly and each call has its own set of variables that are distinct from the variables in the other function calls.
Iteration requires less memory than recursion – because it only uses one set of variable values.	
Iteration variables may need to be initialised and the value of the tracking variable may need to be tested for each iteration.	
	There needs to be a stopping condition that causes the subroutine to stop without calling itself.
	Recursive subroutines are often more elegant and easier to understand than the equivalent iterations.

Topic: 4.1.4 Recursion

When Do You Use Recursion?

It is a proven fact that you never **need** recursion; any recursive program can be changed into a non-recursive one, for example by the use of stacks. (In the same way that you can prove that any program could be written in absolute binary.)

The best answer to this question could simply be, if you find it useful to use recursion in the problem. A rule of thumb is to use recursion when you're processing recursively defined data structures. If you try to evaluate an arithmetic expression, for example, parenthesis may be used to enclose a "subexpression" which must be evaluated first, and is an expression in its own right. The only reasonable way to do this is to write a recursive expression routine; loops alone do not suffice.

Recursion can produce simpler, more natural solutions to a problem.

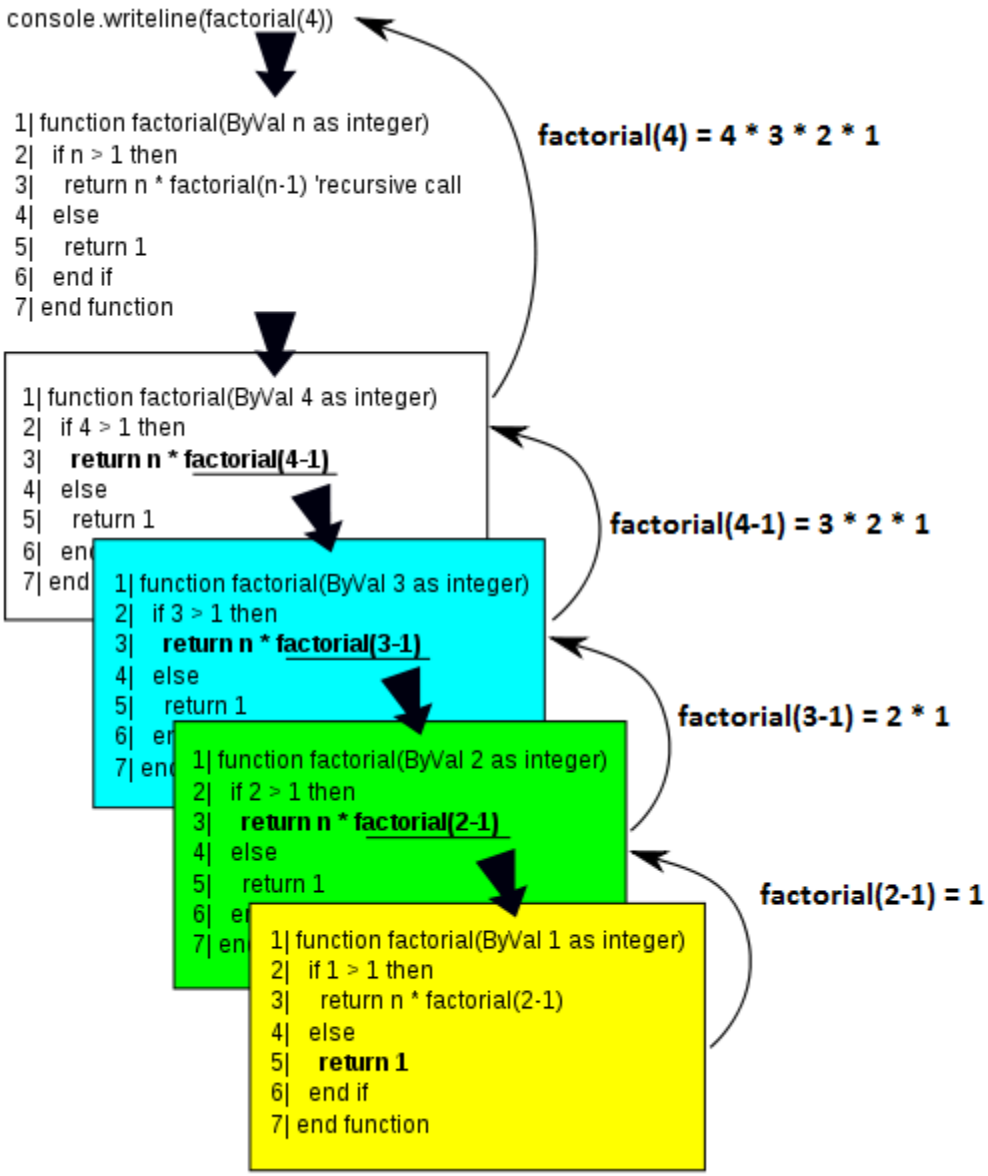
On the other hand, it takes up a large amount of computer resources storing the return addresses and states.

Recursion and Stacks

- Most compilers implement recursion using stacks.
- When a method is called in the main section of the code, the complier pushes the arguments to the method and the return address on the stack, then transfers the control to the method.
- When the method returns, it pops these values off the stack.
- The arguments disappear and the control return to the return address and the main section of the code continues to execute

Topic: 4.1.4 Recursion

Suppose you want to build a truth table and trace the execution for factorial(4). How do we do it using stacks?



Topic: 4.1.4 Recursion

Let's build a trace table and see what happens. This trace table will be different from the ones that you have built before as we are going to have to use a stack.

Function call	n	Return Line
1	4	

All is going well so far until we get to line 3. Now what do we do? We'll soon have two values of n, one for Function call 1 and one for Function call 2. Using the trace table as a stack (with the bottom of the stack at the top and the top of the stack at the bottom) we'll save all the data about the function call including its value of n and make note of what line we need to return to when we have finished with factorial(3).

Function call	n	Return Line
1	4	3
2	3	

We now have a similar situation to before, let's store the return address and go to factorial(2).

Function call	n	Return Line
1	4	3
2	3	3
3	2	

We now have a similar situation to before, let's store the return address and go to factorial(1)

Function call	n	Return Line	Return Value
1	4	3	
2	3	3	
3	2	3	
4	1		1

Topic: 4.1.4 Recursion

Now we have another problem, we have found an end to the factorial(1). What line do we go to next? As we are treating our trace table as a stack we'll just pop the previous value off the top and look at the last function call we stored away, that is function call 3, factorial(2), and we even know what line to return to, line 3:

```
return 2 * factorial(1)
```

We know that factorial(1) = 1 from the previous returned value. Therefore factorial(2) returns 2 * 1 = 2

Function call	n	Return Line	Return Value
1	4	3	
2	3	3	
3	2	3	2
~~4~~	~~1~~		~~1~~

Again we'll pop the last function call from the stack leaving us with function call 2, factorial(3) and line 3.

```
return 3 * factorial(2)
```

We know that factorial(2) = 2 from the previous returned value. Therefore factorial(3) returns 3 * 2 = 6

Function call	n	Return Line	Return Value
1	4	3	
2	3	3	6
~~3~~	~~2~~	~~3~~	~~2~~
~~4~~	~~1~~		~~1~~

Again we'll pop the last function call from the stack leaving us with function call 1, factorial(4) and line 3.

```
return 4 * factorial(3)
```

We know that factorial(3) = 6 from the previous returned value. Therefore factorial(4) returns 4 * 6 = 24

Topic: 4.1.4 Recursion

Function call	n	Return Line	Return Value
1	4	3	24
~~2~~	~~3~~	~~3~~	~~6~~
~~3~~	~~2~~	~~3~~	~~2~~
~~4~~	~~1~~		~~1~~

We reach the end of function call 1. But where do we go now? There is nothing left on the stack and we have finished the code. Therefore the result is 24.