Recursion

by Tue Doan

Recursion is a method of solving problems based on dividing and conquering mentality. The basic idea is that you take the original problem and divide it into smaller (more easily solved) instances of itself, solve those smaller instances (usually by using the same algorithm again) and then reassemble them into the final solution.

The canonical example is a routine to generate the Factorial of n. The Factorial of n is calculated by multiplying all of the numbers between 1 and n. An iterative solution in C# looks like this:

```
int Fact(int n)

{
  int fact = 1;
  for( int i = 2; i <= n; i++)
  {
    fact = fact * i;
  }
  return fact;
}
```

There's nothing surprising about the iterative solution and it should make sense to anyone familiar with C#.

The recursive solution is found by recognizing that the nth Factorial is n * Fact(n-1). Or to put it another way, if you know what a particular Factorial number is you can calculate the next one. Here is the recursive solution in C:

```
int FactRec(int n)
{
  if( n < 2 )
  {
    return 1;
  }
  return n * FactRec( n - 1 );
}
```

The first part of this function is known as a **Base Case** (or sometimes Guard Clause) and is what prevents the algorithm from running forever. It just returns the value 1 whenever the function is called with a value of 1 or less. The second part is more interesting and is known as the **Recursive Step**. Here we call the same method with a slightly modified parameter (we decrement it by 1) and then multiply the result with our copy of n.

When first encountered this can be kind of confusing so it's instructive to examine how it works when run. Imagine that we call FactRec(5). We enter the routine, are not picked up by the base case and so we end up like this:

```
// In FactRec(5)
return 5 * FactRec( 5 - 1 );
// which is
return 5 * FactRec(4);
```

If we re-enter the method with the parameter 4 we are again not stopped by the guard clause and so we end up at:

```
// In FactRec(4)
return 4 * FactRec(3);
```

If we substitute this return value into the return value above we get

```
// In FactRec(5)
return 5 * (4 * FactRec(3));
```

This should give you a clue as to how the final solution is arrived at so we'll fast track and show each step on the way down:

```
return 5 * (4 * FactRec(3));
return 5 * (4 * (3 * FactRec(2)));
return 5 * (4 * (3 * (2 * FactRec(1))));
return 5 * (4 * (3 * (2 * (1))));
```

That final substitution happens when the base case is triggered. At this point we have a simple algrebraic formula to solve, which equates directly to the definition of Factorials in the first place.

It's instructive to note that every call into the method results in either a base case being triggered or a call to the same method where the parameters are closer to a base case (often called a recursive call). If this is not the case then the method will run forever.