Lecture 16 — Recursive Functions

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 21, 2016

A function that includes a call to itself is said to be recursive.

At first glance, this might seem like a really strange idea.
Why would a function call itself?

Recall that we said earlier that a common engineering strategy is to break a big problem down into some smaller problems.

Recursion is useful if we are breaking down a problem so that the subproblems are smaller versions of the same problem.

You may have learned in math class about the factorial.
Commonly written $n!$ in mathematical notation for a number n.

The factorial of a non-negative integer n is defined as the product of all positive integers less than or equal to n.

Examples:

$1! = 1$
$2! = 2 \times 1 = 2$
$3! = 3 \times 2 \times 1 = 6$
$4! = 4 \times 3 \times 2 \times 1 = 24$
$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Recursion Example: Factorial

Observe that 5! could be rewritten as $5 \times 4!$.

The problem of 5! can be broken down into two subproblems:

1. 4!
2. $5\times$ the result of (1).

The subproblem 4! is a smaller version of the problem we're working on (5!), so this problem is a good candidate for recursion.

Suppose now you are going to implement the factorial function.

```
int factorial ( int n )
{
    return n * factorial( n - 1 );
}
```

There is a problem with this implementation.

The Factorial Problem

What happens when the expression n - 1 becomes zero or negative?

This loop will continue forever...
- Except in practice this will be stopped by an error.

Running this program produces an error called a Stack Overflow.

In a later lecture we will discuss what a stack is and how it works.
- For now, a simplified view of the problem.

Stack Overflow

When `main()` calls `factorial()`, the computer needs to keep track of where it was in `main()` at the time that it went to the other function.

It puts that information in a designated memory area called the stack.

Each time `factorial()` calls itself, more information is added to the stack to keep track of where it was in `factorial()`.

If we do this too many times, the stack gets "full" (exceeds available memory for it) and this results in a stack overflow error.

```
int factorial ( int n )
{
    return n * factorial( n - 1 );
}
```

The above implementation lacks a stopping condition.

The function will keep calling itself until a stack overflow occurs.

Even if program execution did not terminate abnormally as a result of a stack overflow, there's another problem, mathematically.

$5! = 5 \times 4 \times 3 \times 2 \times 1$

When we reach 1, we no longer multiply by the next smallest integer.

Our stopping case is therefore when `n` is 1.

The way the code is written, when `n` is 1 the evaluation goes on and the expression to the right of `return` is `1 * factorial( 0 )`.

```
int factorial ( int n )
{
    if ( n == 1 )
    {
        return 1;
    }
    return n * factorial( n - 1 );
}
```

The revised implementation has the stopping condition of `n` equals 1.

The function will keep calling itself until `n` is 1.
When that happens, 1 is returned.

To make proper use of recursion, we need:

1. One or more cases in which the function calls itself; and
2. One or more cases in which the function does not call itself.

Point (2) is called the stopping case or base case.

As we saw, without a properly defined stopping case, recursion will result in a stack overflow.

Recursion versus Iteration

Recursion can be a difficult or confusing topic. Is it strictly necessary?

Any task that can be accomplished with recursion can be done without using recursion, such as using a loop (iteratively).

As a small note on efficiency, a recursively written function may run slower than an iteratively written one. Recursion has two advantages:

1. Write less code; and
2. The computer tracks state on the stack; iteratively, we must keep track of the state ourselves.

Here's the factorial function implemented iteratively:

```
int factorial ( int n )
{
    int product = 1;
    for ( int i = n; i > 1; i-- )
    {
        product *= i;
    }
    return product;
}
```

The iterative implementation uses a `for` loop, but it could have been written using a `while` loop.

When designing a recursive function, there are three important criteria to consider:

1. There is no infinite recursion;
   (A chain of recursive calls eventually reaches a stopping case)
2. Each stopping case returns the correct value for that case; and
3. The final returned value is correct if the recursive call(s) returns the correct value(s).

This is an example of *mathematical induction*.
If you have not yet learned about it, ignore this for now; you will see it next term in ECE 103 (Discrete Mathematics).

Consider another mathematical problem, exponentiation: x^y.

If we wrote a function signature for exponentiation:
`int pow ( int x, int y )`

Is this problem a good candidate for recursive solution?

Mathematically, $a^b = a \times a^{b-1}$.

So, yes, this is the kind of problem that lends itself well to recursion.

Now, let's write an implementation for `pow()`.

```
// Precondition: y >= 0
// Postcondition: returns x to the power of y
int pow ( int x, int y )
{
    if ( y == 0 )
    {
        return 1;
    }
    return pow( x, y - 1 ) * x;
}
```

Applying the Test

Let's examine the design criteria and see if this is going to work.

1. There is no infinite recursion.
The second argument to `pow(x, y)` is decreased by 1 in each call, so eventually we must get to `pow(x, 0)`, a stopping case.
(As long as the precondition is not violated.)

Thus, there is no infinite recursion; criterion 1 is satisfied.

2. Each stopping case returns the correct value for that case.

Yes. $x^0 = 1$ is mathematically correct.

3. The final returned value is correct if the recursive call returns the correct value.

`pow( x, y - 1 ) * x` follows the rule that $a^b = a^{b-1} \times a$.

Criteria 2 and 3 are satisfied.

Having checked those things, we can now be satisfied that the implementation of `pow()` is correct.

Here's the exponential function implemented iteratively:

```
int pow ( int x, int y )
{
    int result = 1;
    for ( int i = 0; i < y; i++ )
    {
        result *= x;
    }
    return result;
}
```

Like the factorial function, the iterative implementation uses a `for` loop, but it could have been written using a `while` loop.