Computer Science 1MD3

Lab 2 – Recursion and Its Implementation In C

Recursion allows us to make very simple and natural definitions of functions that would otherwise have a very complicated and explicit formulas. It is the purpose of this lab to demonstrate, through example the procedure of converting from explicit to recursive solutions.

A CLASSIC EXAMPLE

We are all familiar with factorial which is explicitly defined as:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

with corresponding C code:

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

However, this solution may not be intuitively obvious; it would be desirable to have a simpler definition. Through some analysis, we can discover that:

$$\begin{aligned} n! &= n \times \left[(n-1) \times (n-2) \times \ldots \times 2 \times 1\right] \\ &= n \times (n-1)! \end{aligned}$$

Now, realizing that $0! = 1$ we can implement a C function:

```
int fact(int n) {

      if (n==0) return 1;
      return n * fact(n-1);

}
```

Now lets try tracing this function to see how it works.

$$
\begin{aligned}
&= \texttt{fact}(4) \\
&= 4 \times \texttt{fact}(3) \\
&= 4 \times \left[3 \times \texttt{fact}(2)\right] \\
&= 4 \times \left[3 \times \left[2 \times \texttt{fact}(1)\right]\right] \\
&= 4 \times \left[3 \times \left[2 \times \left[1 \times \texttt{fact}(0)\right]\right]\right] \\
&= 4 \times \left[3 \times \left[2 \times \left[1 \times \left[1\right]\right]\right]\right] \\
&= 4 \times 3 \times 2 \times 1 \times 1 \\
&= 24
\end{aligned}
$$

Please note that every time the function is called a new stack frame is built. When `n=0`, `1` is returned and the stack is resolved from top to bottom. So, the computer actually calculates `1 × 2 × 3 × 4` not `4 × 3 × 2 × 1`.

Your stack frame (informally) may look like this:

Top

fact(0) n=0 return 1	fact(1) n=1 return 1*fact(1)	fact(2) n=2 return 2*fact(1)	fact(3) n=3 return 3*fact(2)	fact(4) n=4 return 4*fact(3)

Bottom

QUALITIES OF A RESCURSIVE FUNCTION

Looking back to our factorial function we can determine that a recursive function requires two things: a terminating condition (`if n==0 return 1`) and a recursive step (`return n*fact(n-1)`). Without a terminating condition the recursive procedure would act like an infinite loop. This is similar to forgetting to increment a while loop.

So how do we properly choose a terminating condition?

The terminating condition is usually the base case, or the smallest value we could ever possibly calculate. In our factorial function, since we cannot take the factorial of a negative number, our base case is `fact(0)=1`.

The recursive step is harder to determine. It is wise to investigate the algorithm you are modeling and see if one part of the equation is expressible using your current value and the function at the step before.

RECURSIVE POWER

In mathematics, there are a lot of recursive definitions that we take for granted, or don't even realize are recursive. However using the fundamental operation addition, it is possible to describe recursive definitions for many rudimentary operations.

For instance, subtraction is inverse-addition, multiplication is recursive addition, and modulus may be recursively defined from subtraction. Using modulus, it is possible to recursively define ceiling and floor division.

From a practical standpoint these functions would not be worth deriving for computational purposes, since they are already defined in the compiler. One function not yet mentioned, exponentiation, would however be nice to define. Exponentiation, or power, is something that we often use, and it would be nice if we could come up with a simple recursive definition for it.

We must first come up with a terminating step, or the most basic calculation we can do with power. Well assuming we are only working with the integers, the base case would be `x` to the power of `0` (x^0), which will always be `1`. So we should use:

```
if (exponent==0) return 1;
```

as our base case.

Investigating the nature of exponentiation, we can see that $x^y=x*x^{y-1}$.

Our recursive step will follow from this, giving us the function:

```
int pow(int x, int y){

      if (y==0) return 1;
      return x * pow(x, y-1);

}
```

LESS OBVIOUS EXAMPLES

Lets consider the function

$$f(n) = n^2 \text{ for } \forall x \geq 0$$

Without using multiplication, is it possible to define a recursive formula for this?

The base case is easy, the smallest value of `x` we can consider is `0` so we have `f(0)=0`, or the C code, `if (n==0) return 0.`

To find the recursive step lets try investigating this formula for `f(n+1)` and see if it expressible through `f(n)`. We do know that $(n+1)^2 = n^2 + 2n + 1$ so we can say $f(n+1) = f(n) + n + n + 1$. But we would like a function of `n`, not `n+1`. So, if $n' = n - 1$, we have :

$$\begin{aligned} f(n') &= f(n-1) + (n-1) + (n-1) + 1 \\ &= f(n-1) + n + n - 1 \end{aligned}$$

So we have the mathematical recursive procedure for squaring $f(n) = n^2 = f(n-1) + n + n + 1$. The corresponding C function would be:

```
int f(int n) {
      if (n==0) return 0;
      return f(n-1)+n+n-1;
}
```

RECURSIVE DOT PRODUCT

A dot product is a vector operation defined as follows

$$\langle v_1, v_2, \ldots, v_n \rangle \cdot \langle w_1, w_2, \ldots, w_n \rangle = v_1w_1 + v_2w_2 + \ldots + v_nw_n$$

For example,

$$\langle 2, 3, 4 \rangle \cdot \langle 3, 4, 5 \rangle = 2 \times 3 + 3 \times 4 + 4 \times 5$$
$$= 38$$

Examining this definition we can conclude that the smallest thing we can calculate is the dot product of two vectors with one component, $\langle a \rangle \cdot \langle b \rangle = a \times b$ so our base case will follow from this.

As for the recursive step, after much pondering, we discover that

$$\langle v_1, v_2, \ldots, v_n \rangle \cdot \langle w_1, w_2, \ldots, w_n \rangle = \langle v_1, v_2, \ldots, v_{n-1} \rangle \cdot \langle w_1, w_2, \ldots, w_{n-1} \rangle + v_nw_n$$

that is you can take the last element out of each vector and add the product to the dot product of the new reduced vectors.

Now lets implement a C function which accepts vectors as arrays. Our header will look like this

```
int dot_product (int *A, int *B, int n);
```

where n is the length of the respective vectors.

Recalling the conditions of the base case we have

```
if (n==1) return A[0]*B[0];
```

as our base case.

Our recursive step follows from our discovery

```
return (A[n]*B[n]) + dot_product(A, B, n-1);
```

Our completed code will look like this

```
int dot_product(int *A, int *B, int n){

      if (n == 1) return (A[0] * B[0]);
      return (A[n] * B[n]) + dot_product(A, B, n-1);

}
```

Self-Test Problems

1. The recursive definition of all odd positive integers would be `f(n) =f(n-1)+2`, where `f(0)=1`. This said give a recursive definition for
   a. the set of positive integer powers of 3
   b. the set of all integers not divisible by 5

2. Write a recursive function to return the greatest common divisor (gcd) of two integers `m` and `n`.

3. Only using addition write a recursive formula to do multiplication

   ```
   int mult (int x, int y);
   ```