Recursion

Definition: Any time the body of a function contains a call to the function itself.

So, just as you are allowed to call function B from within function A, you are ALSO allowed to call function A from within function A!

Potential Problem: But if my function calls itself, how can we ever finish executing the original function?

What this means is that some calls to the function MUST NOT result in a recursive call. I think this can best be seen from an example.

```
// Pre-conditions: e is greater than or equal to 0.
// Post-conditions: returns b^e.
int Power(int base, int exponent) {

    if ( exponent == 0 )
        return 1;
    else
        return (base*Power(base, exponent-1));
}
```

To convince you that this works, let's look at an example:

Say we were trying to evaluate the expression

Power(5, 2)

Power(5,2) returns 5***Power(**5,2-1) = 5***Power(**5,1**)**

To evaluate this expression we must evaluate:

Power(5,1**)** returns 5***Power(**5, 0**)**

Finally, we have:

Power(5,0**) returns 1.**

Thus, Power(5,1**) returns 5*1, which is 5.**

Finally, Power(5,2) returns 5***Power(**5,2-1) = 5*5 = 25,
and we are done.

General Structure of Recursive Functions

What we can determine from the example above is that in general, when we have a problem, we want to break it down into chunks, where one of the chunks is a smaller version of the same problem.

(In the case of Power, we utilized the fact that $x^y = x * x^{y-1}$, and realized that x^{y-1} is in essence an easier version of the original problem.)

Eventually, this means that we break down our original problem enough that our sub-problem is quite easy to solve. At this point, rather than making another recursive call, directly return the answer, or complete the task at hand.

So, ideally, a general structure of a recursive function has a couple options:

1) Break down the problem further, into a smaller subproblem
2) OR, the problem is small enough on its own, solve it.

When we have two options, we often use an if statement. This is typically what is done with recursion. Here are the two general constructs of recursive functions:

Construct 1

```
if (terminating condition) {
        DO FINAL ACTION
}
else {
        Take one step closer to terminating condition
        Call function RECURSIVELY on smaller subproblem
}
```

Construct 2

```
if (!(terminating condition) ) {
        Take a step closer to terminating condition
        Call function RECURSIVELY on smaller subproblem
}
```

Typically, functions that return values take on the first construct, while void recursive functions use the second construct. Note that these are not the ONLY layouts of recursive programs, just common ones.

Example using construct 1

Let's write a function that takes in one positive integer parameter n, and returns the sum 1+2+...+n. (You probably did this is C numerous times, yet surprisingly it never gets old!)

Step 1: We need to solve this problem is such a way that part of the solution is a sub-problem of the exact same nature.

Let f(n) denote the value 1+2+3+...+n
Using our usual iterative logic, we have

f(n) = 1 + (2 + 3 + ... + n)

But, 2+3+... + n IS NOT a sub-problem of the form 1+2+...+n.

But, let's try this:

f(n) = 1 + 2 + ... + n = n + (1 + 2 + ... + (n-1))

So, here we have gotten the expression 1 + 2 + ... + (n-1) which IS a sub-problem we were looking for. Hence, we have

f(n) = 1 + 2 + ... + n = n + (1 + 2 + ... + (n-1)) = n + f(n-1)

If we look at the construct 1, the first thing we need to determine is a terminating condition. We know that f(0) = 0, so our terminating condition can be n=0.

Furthermore, our recursive call needs to be returning an expression for f(n) in terms of f(k), for some k<n. In this case, we just found that f(n) = n + f(n-1). So, now we can write our function:

```
// Pre-condition: n is a positive integer.
// Post-condition: Function returns the sum 1+2+...+n
int Triangle_Number(int n) {

    if ( n == 0 )
        return 0;
    else
        return (n + Triangle_Number(n-1));
}
```

Let's compare this to the ITERATIVE version you would have written in COP3223:

```
// Pre-condition: n is a positive integer.
// Post-condition: Function returns the sum 1+2+...+n
int Triangle_Number(int n) {
        int index, sum = 0;
        for (index=1; index <=n; index++)
                sum = sum + index;

        return sum;
}
```

Two Other Classic Examples that Use Construct #1

For positive integers n, n! (read, "n factorial"), is defined as follows:

n! = 1x2x3x4x…xn

In fact, this is identical to the triangle number definition except that we multiply the set of integers instead of adding them. Likewise, the recursive code looks very similar:

```
int fact(int n) {

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

Fibonacci numbers are defined recursively, thus, coding them is quite easy, based on the original mathematical definition:

$F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$, for all integers n > 2.

```
int fib(int n) {

  if (n < 3)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

This turns out to be inefficient, for reasons we will mention later.

Example using construct 2

We must write a function for an example of construct 2. A couple lectures ago, when I introduced loops, we wrote an algorithm that printed out a tip chart. We could have just as easily made that a function that takes in the lowest dollar value and highest dollar value on the chart. The method header of the function would look like:

```
// The function prints out a chart with the appropriate tips
// for meals ranging from first_val to last_val number of
// dollars, for every whole dollar amount. Both parameters
// must be positive integers.
void Tip_Chart (int first_val, int last_val)
```

Now, let's consider writing this function recursively.

First, we need to determine a terminating condition. That seems to be relatively simple: if the lowest dollar amount is greater than the highest dollar amount, then the chart is empty and we are done.

Next, we need to figure out how to break down our task into two steps: one that does part of the job, and secondly a recursive step.

1) **It is clear that the first value we have to print of the chart is associated with the lowest dollar value, since that must appear on the chart first. Printing out this line of the tip chart can be the part of the job we accomplish.**
2) **Now, are we left with a sub-problem of the same nature? Yes! We simply now need to print a tip chart from the lowest dollar value+1 to the highest dollar value.**

So, now, we can use these steps to come up with the function:

```
#define      TIP_RATE        0.15

// Pre-condition: Both parameters are integers with the first
//                one being less than or equal to the second one.
// Post-condition: A tip chart will be printed out, with a row for
//                 each dollar amount ranging in between the
//                 value of the two parameters.
void Tip_Chart (int first_val, int last_val)


      if ( !(first_val > last_val) ) {
            printf("On a meal of $%d", first_val);
            printf(" you should tip $%lf\n", first_val*TIP_RATE);
            Tip_Chart(first_val + 1, last_val)
      }
}
```

Using a Stack to Trace Recursive Code

A stack is a construct that can be used to store and retrieve items. It works just like a stack of plates in the cafeteria: The last plate placed on the top is the first one that must be removed. Essentially, it's a Last In, First Out(or LIFO) system.

Stacks can help us trace calls to recursive functions. Consider computing Power(8,3) using the recursive definition of Power given in lecture. We can put a line of code from our main algorithm as the first item on the stack:

Main algorithm: Unfinished: "total = Power(8,3)

Now, we need to compute that value, so the function call Power(8,3) is placed above this statement in the stack.

Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Now we repeat the process...

Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Again...

Power(3rd) b=8,e=1, Unfinished: "Power returns 8*Power(8,0)"
Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Finally,

Power(4th) b=8,e=0, returns 1
Power(3rd) b=8,e=1, Unfinished: "Power returns 8*Power(8,0)"
Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Now, we are ready to "collapse" the stack:

Power(3rd) b=8,e=1, "Power returns 8*1"
Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Power(2nd) b=8,e=2, "Power returns 8*8"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Power(1st) b=8,e=3, "Power returns 8*64"
Main algorithm: Unfinished: "total = Power(8,3)"

Main algorithm: "total = 512"

Practice Problem

Write a recursive function that takes in two non-negative integer parameters, and returns the product of these parameters.

```
// Precondition: Both parameters are non-negative integers.
// Postcondition: The product of the two parameters is retrned.
function Multipy returnsa Num (first, second isoftype in Num) {

    if (( second == 0 ) || ( first = 0 ))
        return 0;
    else
        returns (first +Multiply(first, second-1));
}
```

How can we adapt this solution to work for negative numbers?

```
// Precondition: None
// Postcondition: The product of the two parameters is retrned.
function Multipy returnsa Num (first, second isoftype in Num) {

    if ( second < 0 )
         return -Multiply(first,-second);
    else if ( first < 0 )
         return -Multiply(-first, second);
    else if (( second == 0 ) || ( first = 0 ))
        return 0;
    else
        return (first +Multiply(first, second-1));
}
```

Two more exercises for next time:

1) Write a recursive function that determines the n^{th} Lucas number. Here is the definition of the Lucas numbers:

$L_1 = 1$, $L_2 = 3$, $L_n = L_{n-1} + L_{n-2}$, for all integers $n > 2$.

```
// Pre-condition: n > 0
// Post-condition: Returns the n^th Lucas number, L_n.
int Lucas(int n);
```

2) Binomial coefficients, denoted C(n,k), can be computed as follows:

C(n,0) = C(n,n) = 1, for all non-negative integers n.
C(n,k) = C(n-1, k-1) + C(n-1, k), for all integers n > 0 with $0 < k < n$.

```
// Pre-condition: n ≥ 0, and 0 ≤ k ≤ n
// Post-condition: Returns C(n,k).
int bin_coeff(int n, int k);
```

Solutions to Practice Problems

To compute Lucas numbers

```
int Lucas(int n) {

   if (n == 1)
       return 1;
   else if (n == 2)
       return 3;
   else
       return Lucas(n-1)+Lucas(n-2);
}
```

To compute binomial coefficients

```
int bin_coeff(int n, int k) {

    if ((k == 0) || (n == k))
        return 1;
    else
        return bin_coeff(n-1, k-1) + bin_coeff(n-1, k);
}
```

Binary Search

One algorithm where recursion may seem more natural than iteration is with a binary search. Consider the following problem:

You are given a sorted array A, and a value to find in that array, val. You must determine whether or not val is in the array A.

One way we could look at this problem is by adding a couple pieces of information:

Rather than just being given A and val, consider also being given a low and high index value to the array as the bounds for the search. Thus, rather than searching for val in the whole array, your task is slightly more specific: you must decide whether or not val is in A, in between index low and index high.

Let's think about how we can break this problem down:

We are search for val in the array A in between indexes low and high.

1) We want to compare val to the "middle" value in the array.

Why would we want to do this?
What is the middle value?

Generally speaking, we want to minimize the worst case behavior of the algorithm. If we compare val to the 10th value out of 19 total values, then no matter what our answer is (val is smaller than this value, equal to it, or greater than it),

We make sure that after we make the comparison, the maximum number of values we have to search is 9. We really can't do any better than that. Basically, after one comparison, either I nail the value I am searching for, OR I have guaranteed to reduce my search space to 1/2 of what it was.

To determine the middle value with which to do the comparison, simply average the low and high indexes which are the bounds of your search.

Since we are writing this function recursively, we need to specify the terminating condition(s):

1) When the number is found!
2) When the search range is nothing (when low > high).

Now, we are ready to write the function:

```
int binSearch(int *values, int low, int high, int searchval)

    int mid;
    if (low <= high) {

        mid = (low+high)/2;
        if (searchval < values[mid])
            return binSearch(values, low, mid-1, searchval);
        else if (searchval > values[mid])
            return binSearch(values, mid+1, high, searchval);
       else
            return 1;
    }

    return 0;
}
```

Digital Root Problem (Programming Exercise #7)

To take the digital root of a number, you add up all of its digits, and then repeat the process until you get a single digit number. Consider the following example:

1729
Step 1: Add up 1+7+2+9 = 19, not a single digit
Step 2: Add up 1+9 = 10, not a single digit
Step 3: Add up 1+0 = 1, this is the digital root of 1729.

What task do we have to be able to accomplish?

We must be able to add up the digits of a number. Let's write a function (I'll make mine recursive) to do this.

When adding up the digits of a number, we can do one of the following:

1) Add the last digit to the sum of the rest of the number.
2) Add the first digit to the sum of the rest of the number.

Both are recursive characterizations of the problem. To decide between which one, we need to make the following key observation:

1) It is easy to isolate the units digit of an integer. (x%10)
2) It is more difficult to isolate the most significant digit of an integer.

Furthermore, it is also easy to create an integer with the same digits as the original number without the last digit. How can we do this?

Once we can do these two things, the recursive function to sum up the digits in a non-negative integer follows:

```
// Precondition : n >= 0
// Postcondition : The sum of the digits of n is returned.
int DigitSum(int n) {

    if (n > 0)
        return n%10 + DigitSum(n/10);
    return 0;
}
```

Let's trace through an example:

DigitSum(8345) returns 5 + DigitSum(834)
DigitSum(834) returns 4 + DigitSum(83)
DigitSum(83) returns 3 + DigitSum(8)
DigitSum(8) returns 8 + DigitSum(0)
DigitSum(0) returns 0, so now
DigitSum(8) returns 8, and
DigitSum(83) returns 11, and
DigitSum(834) returns 15, and finally
DigitSum(8345) returns 20.

Now we return to the problem calculating the digital root.

Basically we see the following:

If the sum of the digits is greater than 10, go find the digital root of that new number.

Otherwise, we have the digital root - return it!

Here is a C function that calculates a digital root:

```
int DigitalRoot(int n) {

    int sumd = DigitSum(n);
    if (sumd > 9)
        return DigitalRoot(sumd);
    return sumd;
}
```

Introduction to Towers of Hanoi

The story goes as follows: Some guy has this daunting task of moving this set of golden disks from one pole to another pole. There are three poles total and he can only move a single disk at a time. Furthermore, he can not place a larger disk on top of a smaller disk. Our guy, (some monk or something), has 64 disks to transfer. After playing this game for a while, you realize that he's got quite a task. In fact, he will have to make 2^{64} - 1 moves total, at least. (I have no idea what this number is, but it's pretty big...)

Although this won't directly help you code, it is instructive to determine the smallest number of moves possible to move these disks. First we notice the following:

It takes one move to move a tower of one disk.

For larger towers, one way we can solve the problem is as follows:

1) Move the subtower of n-1 disks from pole 1 to pole 3.
2) Move the bottom disk to pole 2.
3) Move the subtower of n-1 disks from pole 3 to pole 2.

We can now use this method of solution to write a method that will print out all the moves necessary to transfer the contents of one pole to another. Here is the prototype for our method:

void towers(int n, int start, int end);

n is the number of disks being moved, start is the number of the pole the disks start on, and end is the number of the pole that the disks will get moved to. The poles are numbered 1 to 3.

Here is the method:

```
void towers(int n, int start, int end) {

    int mid;
    if (n > 0) {
        mid = 6 - start - end;
        towers(n-1, start, mid);
        printf("Move disk %d from tower ", n);
        printf("%d to tower %d.", start, end);
        towers(n-1,mid,end);
    }
}
```

Recursive Problem to Solve

Write a recursive method to determine the number of 1s in the binary representation of a positive integer n. You should attempt to do this after recitation tomorrow, where you will discuss binary numbers. Here is the prototype:

```
// Precondition: n > 0.
int numOnes(int n);
```