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);
```