Solutions to Practice Problem

```
int numOnes(int n) {

  // one digit base case
  if (n < 2)
    return n;
  else
    return n%2 + numOnes(n/2);
}
```

Printing a String in Reverse Order

The code should explain itself:

```
void printReverse(char word[], int length) {

  if (length > 0) {
    printf("%c", word[length-1]);
    printReverse(word, length-1);
  }
}
```

Roughly speaking, if we want to print a string, say "HELLO", in reverse order, we must first print the O. Once we do that, the remaining task is to print "HELL" backwards.

In the if statement, we first print the last letter of the string, and follow that up by reversing the portion of the original string from the first letter to the second to last letter.

Decimal to Binary Conversion

Consider writing a function that takes in a number in decimal, and prints out the equivalent value in binary. We can utilize what we learned about base conversion. The key is as follows:

If we are converting 78 from base 10 to base 2, we calculate 78%2 = 0. This is the LAST digit we want to print, since it's the units digit of our answer.

Preceding that zero, we must take the decimal number 78/2 = 39, and convert THAT to binary. But, this is a recursive task!!!

The code follows:

```
void dectobin(int n) {

     if (n < 2)
         printf("%d", n);

     else {
         dectobin(n/2);
         printf("%d", n%2);
     }
}
```

By taking a base-case of n < 2, we ensure that a 0 gets printed out if it is the original number passed to the function.

What changes would have to be made so that this function prints out n in any arbitrary base (less than 10)?

Fast Exponentiation

The first recursive version of exponentiation shown works fine, but is very slow for very large exponents. It turns out that one prevalent method for encryption of data (such as credit card numbers) involves modular exponentiation, with very big exponents. Using the original recursive algorithm with current computation speeds, it would take thousands of years just to do a single calculation. Luckily, with one very simply observation and tweak, the algorithm can take a second or two with these large numbers.

The key idea is that IF the exponent is even, we can exploit the following mathematical formula:

$$b^e = (b^{e/2}) \times (b^{e/2}).$$

The key here is that we calculate $b^{e/2}$ only ONCE and can reuse the value that we get to do the multiplication.

But, even in this situation, the problem is that the sheer size of $b^{e/2}$ for very large e would make that one multiplication very slow.

But, consider the situation, were instead of calculating b^e, we were calculating b^e % n, for some relatively large value of n, maybe 20-100 digits. In this situation, the answer and any intermediate answer that is necessary, never exceeds n^2, which is relatively few digits.

In this case, reusing the value of $b^{e/2}$ % n accrues a HUGE benefit.

Note: When we test the following function (with mod) in C, it's important to choose a base that is smaller than 2^{15} to avoid overflow errors. The exponent may be any positive allowable int.

Fast Exponentiation Code (without mod)

The following code is not practical (since overflows would happen very quickly), but is shown to illustrate the key idea behind fast exponentiation:

```
int powerB(int base, int exp) {

    if (exp == 0)
        return 1;

    else if (exp == 1)
        return base;

    else if (exp%2 == 0)
        return powerB(base*base, exp/2);

    else
        return base*powerB(base, exp-1);
}
```

In the following lecture we will analyze the huge gain in efficiency made by this algorithm, in terms of the value of the exponent.

Fast Modular Exponentiation

The key changes here are adding a parameter, n, representing the number by which we mod and then adding that mod to each relevant calculation. Note that no intermediate result will exceed n^2. Thus, to ensure this code does not overflow, we must make sure that $n^2 < 2^{31}$.

```
int modPow(int base, int exp, int n) {

    base = base%n;

    if (exp == 0)
        return 1;

    else if (exp == 1)
        return base;

    else if (exp%2 == 0)
        return modPow(base*base%n, exp/2, n);

    else
        return base*modPow(base, exp-1, n)%n;
}
```

In order to test this function and show that is saves time over the regular version, an iterative version is used because the recursive version of modPow (slowModPow in recursion.c) would overflow the call stack with a relatively small value of exp.

This is clearly shown in the file recursion.c.