Fast Modular 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.

```
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;
}
```

If the exponent passed to the algorithm is odd, the next recursive call will contain an even exponent. Any call to an even exponent divides it by 2. Thus, for every two recursive calls, we divide the exponent by two. This, given the exponent, the number of steps the algorithm takes is O(log exp). Thus, even if exp = 10^{30}, this would take at most about 200 recursive calls total, which is much, much better than calculating this using a for loop that runs 10^{30} times.

This idea of "repeated squaring" or "dealing with even exponents by dividing by 2", can be replicated in many places.

One place is matrix exponentiation. If we have a matrix we wish to raise to a high power (usually in these cases we might want the entries mod some value), then we can utilize this same exact concept!

Your multiplication would have to be a function and the recursive code would look a great deal like what is shown on the previous page, except for * would be replaced by the multiply function.

If you are clever, you can build matrices whose entries are answers to particular questions. Consider the following:

Let's say I wanted to add up $(1 + a + a^2 + a^n)$ mod p. I could calculate the following:

$$\begin{bmatrix} a & 1 \\ 0 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Notice that for n = 1, the result is $\begin{bmatrix} a+1 \\ 1 \end{bmatrix}$.
For n = 2, the result is $\begin{bmatrix} a^2+a+1 \\ 1 \end{bmatrix}$. We can prove via induction that the result, in general is

$$\begin{bmatrix} a & 1 \\ 0 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sum_{k=0}^{n} a^k \\ 1 \end{bmatrix}$$

This, if we want to find that desired sum, we simply set up the fast modular matrix exponentiation described above, multiplying the result with the column matrix 1, 1.

In general, a very high term of any linear recurrence relation mod a value can be calculated using this technique. Basically, you set up your matrix to store the coefficients of the recurrence relation and the last column vector will store the "base cases" of the recurrence, so that in successive multiplications, the resulting column vector will store the last few values of the recurrence relation needed to build the next value of the recurrence relation.