way instead of after just the last multiplication makes each multiplication easier because it keeps `result` small.

After N multiplications, `result` is the answer we are seeking. However, doing N multiplications is impractical if N is a 100-digit number. In fact, if N is one billion it is impractical on all but the fastest machines.

A faster algorithm is based on the following observation. That is, if N is even, then $X^N = (X \cdot X)^{\lfloor N/2 \rfloor}$ and if N is odd, $X^N = X \cdot X^{N-1} = X \cdot (X \cdot X)^{\lfloor N/2 \rfloor}$. (Recall that $\lfloor X \rfloor$ is the largest integer that is smaller than or equal to X.) As before, to perform modular exponentiation, apply an `operator%` after every multiplication.

The recursive algorithm in Figure 7.14 represents a direct implementation of this strategy. Lines 7 and 8 handle the base case: X^0 is 1, by definition.[3] At line 10, we make a recursive call based on the identity stated in the previous paragraph. If N is even, then this computes the desired answer. If N is odd, we need to multiply by an extra X (and use `operator%`).

Exponentiation can be done in a logarithmic number of multiplications.

This algorithm is faster than the simple algorithm proposed earlier. If $M(N)$ is the number of multiplications that are used by `power`, then we have $M(N) \leq M\lfloor N/2 \rfloor + 2$. This is because if N is even, we perform one multiplication, plus those done recursively, and if N is odd, we perform two multiplications, plus those done recursively. Since $M(0) = 0$, we can show that $M(N) < 2\log N$. The logarithmic factor can be seen without direct calculation by application of the halving principle (see Section 5.5), which tells us the number of recursive invocations of `power`. Moreover, an average value of $M(N)$ is $(3/2)\log N$. This is because in each recursive step, N is equally likely to be even or odd. If N is a 100-digit number, then in the worst case, only about 665 multiplications (and typically only 500 on average) are needed.

```
1     /**
2      * Return x^n (mod p)
3      * Assumes that p is positive and power( 0, 0, p ) is 1
4      */
5     public static long power( long x, long n, long p )
6     {
7         if( n == 0 )
8             return 1;
9
10        long tmp = power( ( x * x ) % p, n / 2, p );
11        if( n % 2 != 0 )
12            tmp = ( tmp * x ) % p;
13
14        return tmp;
15    }
```

Figure 7.14 Modular exponentiation routine

3. For the purposes of this algorithm, $0^0 = 1$, N is nonnegative, and P is positive.