Fast Exponentiation

Problem: Given integers a, n, and m with $n \ge 0$ and $0 \le a < m$, compute $a^n \pmod m$.

A simple algorithm is:

```
y = a;
for ( i = 2, 3, ..., n )
    y = ya (mod m);
return y;
```

This simple algorithm uses $n-1$ modular multiplications.

It is completely impractical if n has, say, several hundred digits.

Much of public-key cryptography depends our ability to compute $a^n \pmod m$ fairly quickly for integers n of this size.

If n is a power of 2, say $n = 2^k$, there is a much faster way: simply square a, k times. For example, we can compute $a^{128} = a^{2^7} \pmod m$ using only 7 modular multiplications, like this:

$$a^2 = (a)^2 \pmod m$$
$$a^{2^2} = (a^2)^2 \pmod m$$
$$a^{2^3} = (a^{2^2})^2 \pmod m$$
$$a^{2^4} = (a^{2^3})^2 \pmod m$$
$$a^{2^5} = (a^{2^4})^2 \pmod m$$
$$a^{2^6} = (a^{2^5})^2 \pmod m$$
$$a^{2^7} = (a^{2^6})^2 \pmod m$$

Say n is not a power of 2, e.g., $n = 205 = (11001101)_2 = 2^7+2^6+2^3+2^2+2^0$. Given the computations above, only 4 more modular multiplications produce $a^{205} \pmod m$:

$$a^{205} = a^{2^7} \cdot a^{2^6} \cdot a^{2^3} \cdot a^{2^2} \cdot a \pmod m.$$

(We actually reduce mod m after each multiplication.)

In general, if $n = (\beta_k\beta_{k-1} \ldots \beta_0)_2$, where $\beta_k \ne 0$ unless $k = 0$, then $2^k \le n < 2^{k+1}$, and $k = \lfloor \lg(n) \rfloor$.

We can compute $a^n \pmod m$ using k modular multiplications to compute a^{2^i} ($i \le k$) followed by 0 to k additional modular multiplications to compute $\prod_{i=0,\beta_i=1}^{k} a^{2^i}$.

The total number of modular multiplications is k to $2k$, or $\lfloor \lg(n) \rfloor$ to $2\lfloor \lg(n) \rfloor$.

We don't really need an array to store all the a^{2^i} ($i \le k$).

Here is our first algorithm:

Input: Integers a, n, and m, with $n \ge 0$ and $0 \le a < m$.

Output: $a^n \pmod m$

Algorithm: Let $n = (\beta_k\beta_{k-1} \ldots \beta_0)_2$, where $\beta_k \ne 0$ unless $k = 0$. Then $k = \lfloor \lg(n) \rfloor$ and $n = \sum_{i=0}^{k} \beta_i 2^i$. Note $\beta_i = (n >> i) \,\&1$ in C notation. For notational purposes, let $n_i = (\beta_i\beta_{i-1} \ldots \beta_0)_2$ for $i = 0, 1, \ldots, k$.

```
Integer fastExp( Integer a, Integer n, Integer m )
    x = a;                                  // x = a^(2^0)
    y = (β0 == 1) ? a : 1;                  // y = a^(n_0)
    for ( i = 1, 2, ..., k )
        x = x^2 (mod m);                    // x = a^(2^(i-1))  →  x = a^(2^i)
        if ( βi == 1 )
            y = (y==1) ? x : yx (mod m);    // y = a^(n_(i-1))  →  y = a^(n_i)
    return y;
```

Here is a slight reworking of the algorithm that eliminates explicit reference to the bits β_i. It uses a function $odd(n)$ that returns true exactly when n is odd.

```
Integer fastExp( Integer a, Integer n, Integer m )
    x = a;                                  // x = a^(2^0)
    y = (odd(n)) ? a : 1;                   // y = a^(n_0)
    n' = ⌊n/2⌋;
    while ( n' > 0 )
        x = x^2 (mod m);                    // x = a^(2^(i-1))  →  x = a^(2^i)
        if (odd(n'))
            y = (y==1) ? x : yx (mod m);    // y = a^(n_(i-1))  →  y = a^(n_i)
        n' = ⌊n'/2⌋;
    return y;
```

Instead of computing $a^{n_0}, a^{n_1}, ..., a^{n_k}$, where $n_i = (\beta_i\beta_{k-1}\cdots\beta_0)_2$, the variation below computes $a^{m_k}, a^{m_{k-1}}, ..., a^{m_0}$, where $m_i = (\beta_k\beta_{k-1}\cdots\beta_i)_2$. It uses one less variable. Note $m_k = 1$, $m_i = 2m_{i+1}+\beta_i$ for $i = k-1,...,1,0$, and $m_0 = n$.

```
Integer fastExp2( Integer a, Integer n, Integer m )
    if ( n == 0 )
        return 1;
    y = a;                                  // y = a^(m_0)
    for ( i = k-1, k-2, ..., 0 )
        if (β_i == 0)
            y = y^2 (mod m);          }
        else                          }     // y = a^(m_(i+1)) → y = a^(m_i)
            y = y^2 a (mod m);        }
    return y;
```

Each algorithm performs between $\lfloor \lg(n) \rfloor$ and $2\lfloor \lg(n) \rfloor$ modular multiplications. The exact number is $\lfloor \lg(n) \rfloor + |\{\beta_i : 0 \le i < k, \beta_i = 1\}|$. For a random n in $[2^k, 2^{k+1})$, we would expect half the β_i to be 1, so the expected number of modular multiplications would be $3/2\lfloor \lg(n) \rfloor$.

If n has several hundred digits, $\lg(n)$ is somewhere around 1000. We can compute a^n (mod m) using about 1500 modular multiplications (expected case) and 2000 modular multiplications (worst case).

What is the running time of fast exponentiation?

1) Using the "standard" method of multiplying integers, we can multiply two q-bit integers in $\Theta(q^2)$ time. (The same applies to modular multiplication.)

   The integers multiplied in fast exponentiation are less than m, so they have at most $\lfloor \lg(m) \rfloor + 1$ bits — essentially at most $\lg(m)$ bits.

   This gives a running time for fast exponentiation of $O(\lg(n)(\lg(m))^2)$, or $O(\lg(m)^3)$ if we assume $n \le m$.

2) Later in this course, we will derive a practical faster algorithm for multiplying two integers. This algorithm multiplies two q-bit integers in $\Theta(q^{\lg(3)})$ time, or approximately $\Theta(q^{1.59})$ time.

   If we employ this algorithm, the running time of fast exponentiation becomes $O(\lg(n)(\lg(m))^{1.59})$, or $O(\lg(m)^{2.59})$ if we assume $n \le m$.

3) Still faster algorithms for multiplying two integers are known, but (as far as I am aware) they are not practical. In principle, at least, the running time of fast exponentiation can be reduced still further.