# Modular exponentiation algorithm

V. Barbera

**Abstract**

This paper presents an extension of the left-to-right binary method to perform modular exponentiation b^e (mod m) by representing the exponent in base 2^d.

## Modular exponentiation

Modular exponentiation[1] is the remainder $r = b^e \pmod m$ when an integer $b$ is raised to the power $e$ and divided by a positive integer $m$ .

One method to perform modular exponentiation is left-to-right binary method[1]:

the exponent $e$ must be converted to binary representation

$$e = \sum_{i=0}^{n-1} e_i 2^i \quad \text{with} \quad n = \lfloor \log_2 e \rfloor + 1$$

then

$$b^e = b^{\sum_{i=0}^{n-1} e_i 2^i} = \prod_{i=0}^{n-1} b^{e_i 2^i}$$

the remainder is:

$$r = \prod_{i=0}^{n-1} b^{e_i 2^i} \pmod m$$

Algorithm of conversion to binary representation:

**Inputs** An positive integer $e$

**Outputs** The vector $E$ with the digits $e_i$ of binary representation of $e$

1. $i \leftarrow 0$
2. While $e > 0$ do
    1. $E[i] \leftarrow e \bmod 2$
    2. $e \leftarrow \lfloor e/2 \rfloor$
    3. $i \leftarrow i + 1$
3. Output $E$

Algorithm left-to-right modular exponentiation:

**Inputs** An integer $b$ , an integer $e$ , a vector $E$ (with $E[i]=e_i$ the digits of binary representation of $e$ ) and a positive integer $m$

**Outputs** $r=b^e \pmod m$

1. $r \leftarrow 1$
2. If $e>0$ then
    1. For $i \leftarrow n-1=\lfloor \log_2 e \rfloor$ to 0 do
        1. $r \leftarrow r^2 \bmod m$
        2. if $E[i] \neq 0$ then $r \leftarrow (r \cdot b) \bmod m$
3. Output $r$

If we use a base $2^d$ representation for $e$

$$e=\sum_{i=0}^{n-1} e_i (2^d)^i \quad \text{with} \quad n=\lfloor \log_{2^d} e \rfloor +1$$

then

$$b^e = 2^{\sum_{i=0}^{n-1} e_i (2^d)^i} = \prod_{i=0}^{n-1} b^{e_i (2^d)^i}$$

the remainder is:

$$r=\prod_{i=0}^{n-1} b^{e_i (2^d)^i} \pmod m = \prod_{i=0}^{n-1} (b^{e_i})^{2^{i \cdot d}} \pmod m$$

Algorithm of conversion to base $2^d$ representation:

**Inputs** An integer $e$ , a positive integer $d$

**Outputs** The vector $E$ with the digits of base $2^d$ representation of $e$

1. $i \leftarrow 0$
2. If e>= $2^d$ then
    1. While $e>0$ do
        1. $E[i] \leftarrow e \bmod 2^d$
        2. $e \leftarrow \lfloor e/2^d \rfloor$
        3. $i \leftarrow i+1$

    Else
        1. $E[i] \leftarrow e$
3. Output $E$

Modified algorithm left-to-right modular exponentiation:

**Inputs** An integer $b$ , a positive integer $d$ , a vector $E$ (with $E[i]=e_i$ the digits of representation of $e$ in base $2^d$ ) and a positive integer $m$

**Outputs** $r=b^e \pmod m$

1. $X[0] \leftarrow 1$
2. For $j \leftarrow 1$ to $2^d-1$ do
    1. $X[j] \leftarrow X[j-1] \cdot b \bmod m$
3. $i \leftarrow length(E)$
4. $r \leftarrow X[E[i-1]]$
5. While $i>1$ do
    1. $i \leftarrow i-1$
    2. For $j \leftarrow 1$ to $d$ do
        1. $r \leftarrow r^2 \bmod m$
    3. $r \leftarrow (r \cdot X[E[i-1]]) \bmod m$
6. Output $r$

Indeed at the start after step 4 $n=length(E)$

$$r=b^{e_{n-1}} \bmod m$$

after step 5.2

$$r=(b^{e_{n-1}})^{2^d} \bmod m$$

after step 5.3

$$r=b^{e_{n-2}} \cdot (b^{e_{n-1}})^{2^d} \bmod m$$

after step 5.2

$$r=(b^{e_{n-2}})^{2^d} \cdot (b^{e_{n-1}})^{b^{2 \cdot d}} \bmod m$$

after step 5.3

$$r=b^{e_{n-3}} \cdot (b^{e_{n-2}})^{2^d} \cdot (b^{e_{n-1}})^{2^{2 \cdot d}} \bmod m$$

…

after step 5

$$r=\prod_{i=0}^{n-1} (b^{e_i})^{2^{i \cdot d}} \pmod m$$

Note that if $d=1$ the algorithm is the same as the previous one.

Example of implementation in C++ with GMP library

```
#include <iostream>
#include <cmath>
#include <vector>
#include <cstdlib>
#include <gmp.h>
void mod_pow(std::vector<unsigned> &E, unsigned d, mpz_t &b, mpz_t &m, mpz_t &r){
   if(mpz_cmp_ui(b,2) == 0)
   {
      // get r = 2^e (mod m) with E vector containing the digits of base 2^d representation of exponent e
      long long len_E = E.size();
      mpz_set_ui(r, 1);
      mpz_mul_2exp(r, r, E[len_E - 1]);
      if (len_E > 1){
         while (len_E > 1){
            len_E--;
            for (unsigned j = 1; j < d; j++) {
               mpz_mul(r, r, r);
               mpz_fdiv_r(r, r, m);
            }
            mpz_mul(r, r, r);
            mpz_mul_2exp(r, r, E[len_E - 1]);
            mpz_fdiv_r(r, r, m);
         }
      }
      else
         mpz_fdiv_r(r, r, m);
   }
   else{
      // get r = b^e (mod m) with E vector containing the digits of base 2^d representation of exponent e
      unsigned dim_vec = 1 << d;
      mpz_t Vec[dim_vec];
      for (unsigned i = 0; i < dim_vec; i++)
         mpz_init(Vec[i]);
      mpz_set_ui(Vec[0], 1);
      for (unsigned i = 1; i < dim_vec; i++)
      {
         mpz_mul(Vec[i], Vec[i - 1], b);
         mpz_fdiv_r(Vec[i], Vec[i], m);
      }
      long long len_E = E.size();
      mpz_set_ui(r, 1);
      mpz_mul(r, r, Vec[E[len_E - 1]]);
      if (len_E > 1){
         while (len_E > 1){
            len_E--;
            for (unsigned j = 1; j < d; j++) {
               mpz_mul(r, r, r);
               mpz_fdiv_r(r, r, m);
            }
            mpz_mul(r, r, r);
            mpz_mul(r, r, Vec[E[len_E - 1]]);
```

```
            mpz_fdiv_r(r, r, m);
          }
        }
        else
          mpz_fdiv_r(r, r, m);
        for (unsigned i = 0; i < dim_vec; i++)
          mpz_clear(Vec[i]);
    }
}
int main(){
    unsigned long long exponent = 3007;
    unsigned long long base = 2;
    unsigned long long modulus = 101;
    mpz_t r, b, m;
    mpz_init_set_ui(m, modulus);
    mpz_init_set_ui(b, base);
    mpz_init(r);
    const unsigned d = 6;
    std::cout <<  mpz_get_str(NULL, 10, b) << "^" << exponent << " (mod " << mpz_get_str(NULL, 10, m) << ") = ";
    //conversion to base 2^d of exponent
    std::vector<unsigned> E;
    if (exponent >= (1ull << d)) {
        unsigned radix_m1 = (1 << d) - 1;
        while (exponent > 0) {
            E.push_back(exponent & radix_m1);
            exponent >>= d;
        }
    }
    else
        E.push_back(exponent);
    mod_pow(E, d, b, m, r);
    std::cout << mpz_get_str(NULL, 10, r);
    mpz_clear(b);
    mpz_clear(m);
    mpz_clear(r);
    return 0;
}
```

## References

[1] https://en.wikipedia.org/wiki/Modular_exponentiation