WIKIPEDIA

Modular exponentiation

Modular exponentiation is a type of exponentiation performed over a modulus. It is useful in computer science, especially in the field of public-key cryptography.

The operation of modular exponentiation calculates the remainder when an integer b (the base) raised to the eth power (the exponent), b^e, is divided by a positive integer m (the modulus). In symbols, given base b, exponent e, and modulus m, the modular exponentiation c is: $c = b^e \bmod m$. From the definition of c, it follows that $0 \leq c < m$.

For example, given $b = 5$, $e = 3$ and $m = 13$, the solution $c = 8$ is the remainder of dividing $5^3 = 125$ by 13.

Modular exponentiation can be performed with a *negative* exponent e by finding the modular multiplicative inverse d of b modulo m using the extended Euclidean algorithm. That is:

$$c = b^e \bmod m = d^{-e} \bmod m, \text{ where } e < 0 \text{ and } b \cdot d \equiv 1 \pmod{m}.$$

Modular exponentiation similar to the one described above is considered easy to compute, even when the integers involved are enormous. On the other hand, computing the modular discrete logarithm – that is, the task of finding the exponent e when given b, c, and m – is believed to be difficult. This one-way function behavior makes modular exponentiation a candidate for use in cryptographic algorithms.

Contents

External links

Direct method

The most direct method of calculating a modular exponent is to calculate b^e directly, then to take this number modulo m. Consider trying to compute c, given $b = 4$, $e = 13$, and $m = 497$:

$$c \equiv 4^{13} \pmod{497}$$

One could use a calculator to compute 4^{13}; this comes out to 67,108,864. Taking this value modulo 497, the answer c is determined to be 445.

Note that b is only one digit in length and that e is only two digits in length, but the value b^e is 8 digits in length.

In strong cryptography, b is often at least 1024 bits.[1] Consider $b = 5 \times 10^{76}$ and $e = 17$, both of which are perfectly reasonable values. In this example, b is 77 digits in length and e is 2 digits in length, but the value b^e is 1,304 decimal digits in length. Such calculations are possible on modern computers, but the sheer magnitude of such numbers causes the speed of calculations to slow considerably. As b and e increase even further to provide better security, the value b^e becomes unwieldy.

The time required to perform the exponentiation depends on the operating environment and the processor. The method described above requires $O(e)$ multiplications to complete.

Memory-efficient method

Keeping the numbers smaller requires additional modular reduction operations, but the reduced size makes each operation faster, saving time (as well as memory) overall.

This algorithm makes use of the identity

$$(a \cdot b) \bmod m = [(a \bmod m) \cdot (b \bmod m)] \bmod m$$

The modified algorithm is:

1. Set $c = 1$, $e' = 0$.
2. Increase e' by 1.
3. Set $c = (\mathrm{b} \cdot \mathrm{c}) \bmod m$.
4. If $e' < e$, go to step 2. Else, c contains the correct solution to $c \equiv b^{\mathrm{e}} \pmod{m}$.

Note that in every pass through step 3, the equation $c \equiv b^{e'} \pmod{m}$ holds true. When step 3 has been executed e times, then, c contains the answer that was sought. In summary, this algorithm basically counts up e' by ones until e' reaches e, doing a multiply by b and a modulo operation each time it adds one (to ensure the results stay small).

The example $b = 4$, $e = 13$, and $m = 497$ is presented again. The algorithm passes through step 3 thirteen times:

- $e' = 1$. $c = (1 \cdot 4) \bmod 497 = 4 \bmod 497 = \mathbf{4}$.
- $e' = 2$. $c = (4 \cdot 4) \bmod 497 = 16 \bmod 497 = \mathbf{16}$.
- $e' = 3$. $c = (16 \cdot 4) \bmod 497 = 64 \bmod 497 = \mathbf{64}$.
- $e' = 4$. $c = (64 \cdot 4) \bmod 497 = 256 \bmod 497 = \mathbf{256}$.
- $e' = 5$. $c = (256 \cdot 4) \bmod 497 = 1024 \bmod 497 = \mathbf{30}$.
- $e' = 6$. $c = (30 \cdot 4) \bmod 497 = 120 \bmod 497 = \mathbf{120}$.
- $e' = 7$. $c = (120 \cdot 4) \bmod 497 = 480 \bmod 497 = \mathbf{480}$.
- $e' = 8$. $c = (480 \cdot 4) \bmod 497 = 1920 \bmod 497 = \mathbf{429}$.
- $e' = 9$. $c = (429 \cdot 4) \bmod 497 = 1716 \bmod 497 = \mathbf{225}$.
- $e' = 10$. $c = (225 \cdot 4) \bmod 497 = 900 \bmod 497 = \mathbf{403}$.
- $e' = 11$. $c = (403 \cdot 4) \bmod 497 = 1612 \bmod 497 = \mathbf{121}$.
- $e' = 12$. $c = (121 \cdot 4) \bmod 497 = 484 \bmod 497 = \mathbf{484}$.
- $e' = 13$. $c = (484 \cdot 4) \bmod 497 = 1936 \bmod 497 = \mathbf{445}$.

The final answer for c is therefore 445, as in the first method.

Like the first method, this requires $O(e)$ multiplications to complete. However, since the numbers used in these calculations are much smaller than the numbers used in the first algorithm's calculations, the computation time decreases by a factor of at least $O(e)$ in this method.

In pseudocode, this method can be performed the following way:

```
function modular_pow(base, exponent, modulus) is
    if modulus = 1 then
        return 0
    c := 1
    for e_prime = 0 to exponent-1 do
        c := (c * base) mod modulus
    return c
```

Right-to-left binary method

A third method drastically reduces the number of operations to perform modular exponentiation, while keeping the same memory footprint as in the previous method. It is a combination of the previous method and a more general principle called exponentiation by squaring (also known as *binary exponentiation*).

First, it is required that the exponent e be converted to binary notation. That is, e can be written as:

$$e = \sum_{i=0}^{n-1} a_i 2^i$$

In such notation, the *length* of e is n bits. a_i can take the value 0 or 1 for any i such that $0 \leq i < n$. By definition, $a_{n-1} = 1$.

The value b^e can then be written as:

$$b^e = b^{\left(\sum_{i=0}^{n-1} a_i 2^i\right)} = \prod_{i=0}^{n-1} b^{a_i 2^i}$$

The solution c is therefore:

$$c \equiv \prod_{i=0}^{n-1} b^{a_i 2^i} \pmod{m}$$

Pseudocode

The following is an example in pseudocode based on Applied Cryptography by Bruce Schneier.[2] The inputs *base*, *exponent*, and *modulus* correspond to b, e, and m in the equations given above.

```
function modular_pow(base, exponent, modulus) is
    if modulus = 1 then
        return 0
    Assert :: (modulus - 1) * (modulus - 1) does not overflow base
    result := 1
    base := base mod modulus
    while exponent > 0 do
        if (exponent mod 2 == 1) then
            result := (result * base) mod modulus
        exponent := exponent >> 1
        base := (base * base) mod modulus
    return result
```

Note that upon entering the loop for the first time, the code variable *base* is equivalent to b. However, the repeated squaring in the third line of code ensures that at the completion of every loop, the variable *base* is equivalent to $b^{2^i} \bmod m$, where i is the number of times the loop has been iterated. (This makes i the next working bit of the binary exponent *exponent*, where the least-significant bit is $exponent_0$).

The first line of code simply carries out the multiplication in $\prod_{i=0}^{n-1} b^{a_i 2^i} \pmod{m}$. If a is zero, no code executes since this effectively multiplies the running total by one. If a instead is one, the variable *base* (containing the value $b^{2^i} \bmod m$ of the original base) is simply multiplied in.

In this example, the base b is raised to the exponent $e = 13$. The exponent is 1101 in binary. There are four binary digits, so the loop executes four times, with values $a_0 = 1$, $a_1 = 0$, $a_2 = 1$, and $a_3 = 1$.

First, initialize the result R to 1 and preserve the value of b in the variable x:

$R \leftarrow 1\ (= b^0)$ and $x \leftarrow b$.
Step 1) bit 1 is 1, so set $R \leftarrow R \cdot x\ (= b^1)$;
set $x \leftarrow x^2\ (= b^2)$.

Step 2) bit 2 is 0, so do not reset R;
set $x \leftarrow x^2 \ (= b^4)$.
Step 3) bit 3 is 1, so set $R \leftarrow R \cdot x \ (= b^5)$;
set $x \leftarrow x^2 \ (= b^8)$.
Step 4) bit 4 is 1, so set $R \leftarrow R \cdot x \ (= b^{13})$;
This is the last step so we don't need to square x.

We are done: R is now b^{13}.

Here is the above calculation, where we compute $b = 4$ to the power $e = 13$, performed modulo 497.

Initialize:

$R \leftarrow 1 \ (= b^0)$ and $x \leftarrow b = 4$.
Step 1) bit 1 is 1, so set $R \leftarrow R \cdot 4 \equiv 4 \pmod{497}$;
set $x \leftarrow x^2 \ (= b^2) \equiv 4^2 \equiv 16 \pmod{497}$.
Step 2) bit 2 is 0, so do not reset R;
set $x \leftarrow x^2 \ (= b^4) \equiv 16^2 \equiv 256 \pmod{497}$.
Step 3) bit 3 is 1, so set $R \leftarrow R \cdot x \ (= b^5) \equiv 4 \cdot 256 \equiv 30 \pmod{497}$;
set $x \leftarrow x^2 \ (= b^8) \equiv 256^2 \equiv 429 \pmod{497}$.
Step 4) bit 4 is 1, so set $R \leftarrow R \cdot x \ (= b^{13}) \equiv 30 \cdot 429 \equiv 445 \pmod{497}$;

We are done: R is now $4^{13} \equiv 445 \pmod{497}$, the same result obtained in the previous algorithms.

The running time of this algorithm is O(log *exponent*). When working with large values of *exponent*, this offers a substantial speed benefit over the previous two algorithms, whose time is O(*exponent*). For example, if the exponent was $2^{20} = 1048576$, this algorithm would have 20 steps instead of 1048576 steps.

Implementation in Lua

```
function modPow(b, e, m)
  if m == 1 then
    return 0
  else
    local r = 1
    b = b % m
    while e > 0 do
      if e % 2 == 1 then
        r = (r*b) % m
      end
      e = e >> 1     --use 'e = math.floor(e / 2)' on Lua 5.2 or older
      b = (b^2) % m
    end
    return r
  end
end
```

Left-to-right binary method

We can also use the bits of the exponent in left to right order. In practice, we would usually want the result modulo some modulus m. In that case, we would reduce each multiplication result (mod m) before proceeding. For simplicity, the modulus calculation is omitted here. This example shows how to compute b^{13} using left to right binary exponentiation. The exponent is 1101 in binary; there are 4 bits, so there are 4 iterations.

Initialize the result to 1: $r \leftarrow 1\,(= b^0)$.

Step 1) $r \leftarrow r^2\,(= b^0)$; bit 1 = 1, so compute $r \leftarrow r \cdot b\,(= b^1)$;
Step 2) $r \leftarrow r^2\,(= b^2)$; bit 2 = 1, so compute $r \leftarrow r \cdot b\,(= b^3)$;
Step 3) $r \leftarrow r^2\,(= b^6)$; bit 3 = 0, so we are done with this step;
Step 4) $r \leftarrow r^2\,(= b^{12})$; bit 4 = 1, so compute $r \leftarrow r \cdot b\,(= b^{13})$.

Minimum multiplications

In *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, page 463, Donald Knuth notes that contrary to some assertions, this method does *not* always give the minimum possible number of multiplications. The smallest counterexample is for a power of 15, when the binary method needs six multiplications. Instead, form x^3 in two multiplications, then x^6 by squaring x^3, then x^{12} by squaring x^6, and finally x^{15} by multiplying x^{12} and x^3, thereby achieving the desired result with only five multiplications. However, many pages follow describing how such sequences might be contrived in general.

Generalizations

Matrices

The m-th term of any constant-recursive sequence (such as Fibonacci numbers or Perrin numbers) where each term is a linear function of k previous terms can be computed efficiently modulo n by computing $A^m \bmod n$, where A is the corresponding $k \times k$ companion matrix. The above methods adapt easily to this application. This can be used for primality testing of large numbers n, for example.

Pseudocode

A recursive algorithm for `ModExp(A, b, c)` = $A^b \bmod c$, where A is a square matrix.

```
function Matrix_ModExp(Matrix A, int b, int c) is
    if b == 0 then
        return I  // The identity matrix
    if (b mod 2 == 1) then
        return (A * Matrix_ModExp(A, b - 1, c)) mod c
    Matrix D := Matrix_ModExp(A, b / 2, c)
    return (D * D) mod c
```

Finite cyclic groups

Diffie–Hellman key exchange uses exponentiation in finite cyclic groups. The above methods for modular matrix exponentiation clearly extend to this context. The modular matrix multiplication $C \equiv AB \pmod{n}$ is simply replaced everywhere by the group multiplication $c = ab$.

Reversible and quantum modular exponentiation

In quantum computing, modular exponentiation appears as the bottleneck of Shor's algorithm, where it must be computed by a circuit consisting of reversible gates, which can be further broken down into quantum gates appropriate for a specific physical device. Furthermore, in Shor's algorithm it is possible to know the base and the modulus of exponentiation at every call, which enables various circuit optimizations.[3]

Software implementations

Because modular exponentiation is an important operation in computer science, and there are efficient algorithms (see above) that are much faster than simply exponentiating and then taking the remainder, many programming languages and arbitrary-precision integer libraries have a dedicated function to perform modular exponentiation:

- Python's built-in `pow()` (exponentiation) function [1] (https://docs.python.org/library/functions.html#pow) takes an optional third argument, the modulus
- .NET Framework's `BigInteger` class has a `ModPow()` (http://msdn.microsoft.com/en-us/library/system.numerics.biginteger.modpow%28v=vs.100%29.aspx#pow) method to perform modular exponentiation
- Java's `java.math.BigInteger` class has a `modPow()` (https://docs.oracle.com/javase/10/docs/api/java/math/BigInteger.html#modPow(java.math.BigInteger,java.math.BigInteger)) method to perform modular exponentiation
- MATLAB's `powermod` function from Symbolic Math Toolbox
- Wolfram Language has the PowerMod (https://reference.wolfram.com/language/ref/PowerMod.html) function
- Perl's `Math::BigInt` module has a `bmodpow()` method [2] (http://perldoc.perl.org/Math/BigInt.html#bmodpow%28%29) to perform modular exponentiation
- Raku has a built-in routine `expmod`.
- Go's `big.Int` type contains an `Exp()` (exponentiation) method [3] (https://golang.org/pkg/big/#Int.Exp) whose third parameter, if non-nil, is the modulus
- PHP's BC Math library has a `bcpowmod()` function [4] (http://www.php.net/manual/en/function.bcpowmod.php) to perform modular exponentiation
- The GNU Multiple Precision Arithmetic Library (GMP) library contains a `mpz_powm()` function [5] (http://gmplib.org/manual/Integer-Exponentiation.html) to perform modular exponentiation
- Custom Function `@PowerMod()` (http://www.briandunning.com/cf/1482) for FileMaker Pro (with 1024-bit RSA encryption example)
- Ruby's `openssl` package has the `OpenSSL::BN#mod_exp` method [6] (http://ruby-doc.org/stdlib-trunk/libdoc/openssl/rdoc/OpenSSL/BN.html#method-i-mod_exp) to perform modular

exponentiation.

- The HP Prime Calculator has the CAS.powmod() function [7] (http://h20628.www2.hp.com/km-ext/kmcsdirect/emr_na-c04120022-1.pdf) to perform modular exponentiation. For a^b mod c, a can be no larger than 1 EE 12. This is the maximum precision of most HP calculators including the Prime.

See also

- Montgomery reduction, for calculating the remainder when the modulus is very large.
- Kochanski multiplication, serializable method for calculating the remainder when the modulus is very large
- Barrett reduction, algorithm for calculating the remainder when the modulus is very large.

References

1. "Weak Diffie-Hellman and the Logjam Attack" (https://weakdh.org/). *weakdh.org*. Retrieved 2019-05-03.
2. Schneier 1996, p. 244.
3. I. L. Markov, M. Saeedi (2012). "Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation". *Quantum Information and Computation*. **12** (5–6): 0361–0394. arXiv:1202.6614 (https://arxiv.org/abs/1202.6614). Bibcode:2012arXiv1202.6614M (https://ui.adsabs.harvard.edu/abs/2012arXiv1202.6614M).

External links

- Schneier, Bruce (1996). *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition* (https://archive.org/details/Applied_Cryptography_2nd_ed._B._Schneier) (2nd ed.). Wiley. ISBN 978-0-471-11709-4.
- Paul Garrett, Fast Modular Exponentiation Java Applet (http://www.math.umn.edu/~garrett/crypto/a01/FastPow.html)
- Gordon, Daniel M. (1998). "A Survey of Fast Exponentiation Methods" (https://www.dmgordon.org/papers/jalg.pdf) (PDF). *Journal of Algorithms*. Elsevier BV. **27** (1): 129–146. doi:10.1006/jagm.1997.0913 (https://doi.org/10.1006%2Fjagm.1997.0913). ISSN 0196-6774 (https://www.worldcat.org/issn/0196-6774).