Intro to modular arithmetic and RSA encryption

by Richard K. Herz, www.ReactorLab.net, github.com/RichardHerz

Encryption of messages is essential to the protection of private information. The RSA public-key encryption algorithm allows this to be done. It involves the generation and use of a public key and a private key. The public key can encrypt a message but cannot decrypt it. Decryption requires use of the private key. This allows anyone to send an encrypted message securely to the holder of the private key. To understand RSA, we first need to introduce modular arithmetic.

Basic equations in modular arithmetic - key to encryption

All variables represent integers. The integers a and b are congruent modulo n.

$a \equiv b \pmod{n}$

where "mod" in parenthesis means that the modulus operation applies to all parts of the expression. This can also be expressed as

$a = b + kn$

That is, a is divisible k times by n, with remainder b.

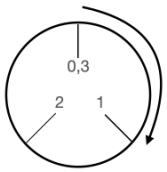
Those of us who use a 12 hour clock dial use modular arithmetic everyday. For example, the hour hand points to 6 on the dial at 6 am and 6 pm, where 6 pm is 18:00 in 24-hour time.

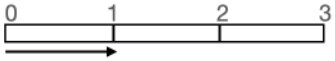
$18 \equiv 6 \pmod{12}$

$18 = 6 + 1(12)$

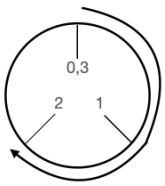
Cryptography and hashing use very large integers. For illustration here, we use small integers. For some operations below, we will use a dial or linear scale with three divisions to simplify things. The relationships shown work for numbers of all magnitudes.

$1 \equiv 1 \pmod{3}$

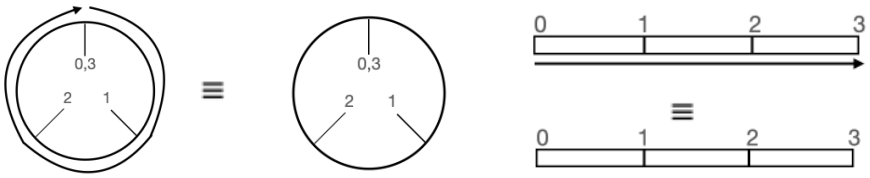




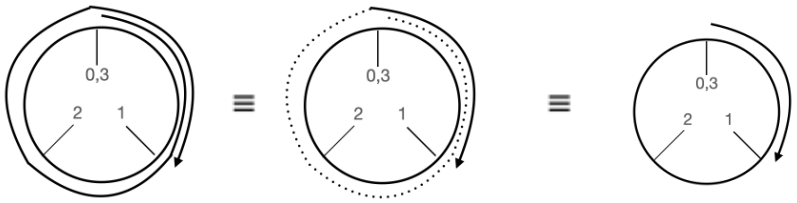
$2 \equiv 2 \pmod{3}$

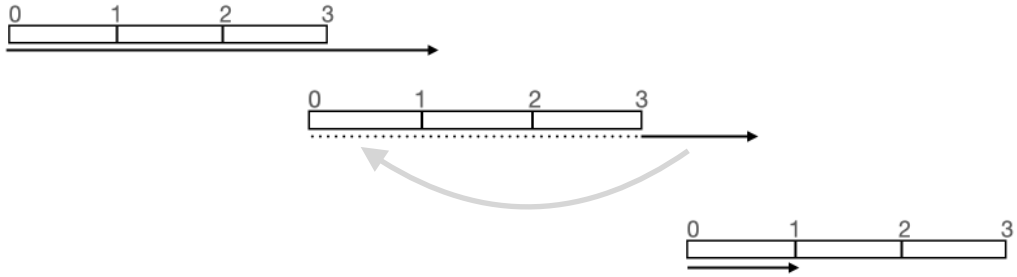


$3 \equiv 0 \pmod 3$



$4 \equiv 1 \pmod 3$





One way to think of this is that numbers greater than some multiple k of the modulus n "wraps around" so that the maximum value in the system is n. In the examples above $k = 1$ but k may have any integer value.

When we discuss hash algorithms, you will see that this "wrap around" feature is what keeps hash values a constant length, regardless of the length of the message being hashed.

One of the important relations in RSA cryptography is Fermat's Little Theorem.

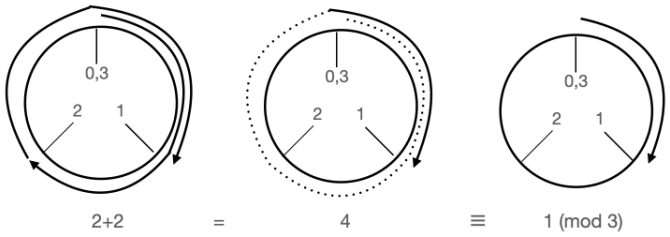
In the first relationship in the theorem, m is an integer and p is a prime number (prime integer)

$m^{(p-1)} \equiv 1 \pmod p$

As an example,

$2^{(3-1)} \equiv 1 \pmod 3$

$2^{(3-1)} = 2^2 = (2 + 2) \equiv 1 \pmod 3$

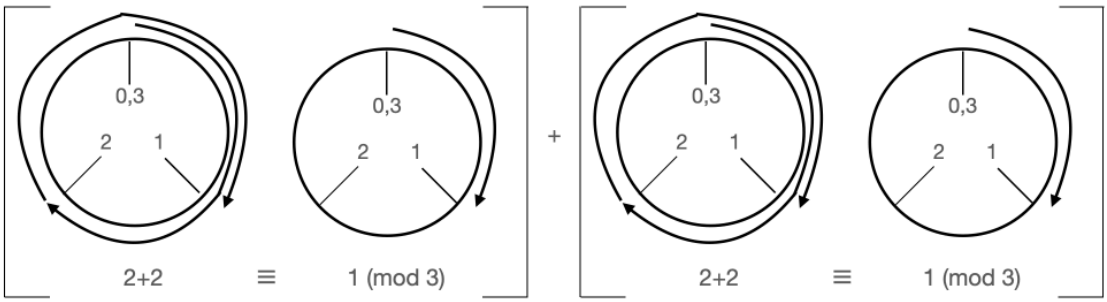


In the second relationship,

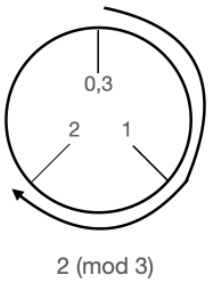
$m^p \equiv m \pmod{p}$

$2^3 \equiv 2 \pmod{3}$

$2^3 = 2(2^2) = (2 + 2) + (2 + 2) = 8 \equiv 2 \pmod{3}$



≡



Fermat's Little Theorem is a special case of the Euler-Fermat Theorem.

In that theorem, if the following apply,

$m^{(p-1)} \equiv 1 \pmod{p}$ and $m^{(q-1)} \equiv 1 \pmod{q}$

where p and q are prime numbers,

then

$m^{(p-1)(q-1)} \equiv 1 \pmod{pq}$

This can be expressed as

$m^{\phi(n)} \equiv 1 \pmod{n}$

where $n = pq$

and ϕ is Euler's phi function

$$\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$$

RSA public-key encryption

Encryption of messages is essential to the protection of private information. The RSA public-key encryption algorithm allows this to be done. RSA was published in 1977 by Rivest, Shamir and Adleman, and is in wide use today. It involves the generation and use of a public key and a private key.

The public key can encrypt a message but cannot decrypt it. Decryption requires use of the private key. This allows anyone to send an encrypted message securely to the holder of the private key.

The primes p and q are the basis numbers of the RSA public-key encryption method. Their choice results in the generation of the public and private keys. They are kept secret.

The public key consists of two integers, n and e. The integer e is chosen to be coprime with $n = pq$. That is, $e < n$ and e will not evenly divide n.

The private key consists of two integers, n and d. The integer d is the "modular inverse" of e, such that

$$ed \equiv 1 \pmod{\phi(n)}$$

This can also be expressed as

$$ed = 1 + k\phi(n)$$

A message can be encrypted by the public key but once encrypted, the encoded message cannot be decrypted by the public key in order to recover the original message.

The encoded message can only be decrypted by the private key.

A plain-text message is first converted to an integer by converting the message to a string of the ASCII or Unicode values of the characters in the message.

Message $m < n$ is encrypted to the encoded message c by this equation:

$$m^e \equiv c \pmod{n}$$

The holder of the private key can decrypt c in order to recover m by this equation:

$c^d \equiv m \pmod{n}$

This can be seen by the following development:

$c^d = (m^e)^d \pmod{n}$

$= (m^e)^d = m^{(ed)} = m^{(1+k\phi(n))} = m(m^{k\phi(n)}) \pmod{n}$

$= m(m^{k\phi(n)}) = m(m^{\phi(n)})^k \pmod{n}$

From the result in the section above, $m^{\phi(n)} \equiv 1 \pmod{n}$. Continuing,

$= m(m^{\phi(n)})^k = m(1)^k \pmod{n}$

$= m \pmod{n}$

$= m,$ since $m < n$

An example with small values: $p = 3, q = 11, n = 33, \phi(n) = 20, e = 3, d = 7, m = 4, c = 31$.

In actual RSA encryption, the values are large such that the computation of m^e cannot be done directly on a computer before the modulo operation, as can be done with small values such as 2^3. This is because the value of m^e that would be obtained with the large values of m and e that are used would be so large that it could not be contained by a computer's method of storing numbers.

The solution of this problem is to compute the results using the modular exponentiation algorithm. In that algorithm, the largest value that must be stored during computation is m^2. This algorithm is listed on the next page.

Continued next page…

Modular Exponentiation Algorithm

The modular exponentiation algorithm used in RSA encryption is shown below as a MATLAB function. The value of array element `key(1)` is n. The value of `key(2)` is e for encryption and d for decryption. The modulus operation $m \pmod{n}$, for example, is coded as `mod(m, n)`.

```
function r = fModExp(m,key)
    %
    % result r = m^key(2) mod key(1), where ^ is exponentiation
    % input m is one integer, output r is one integer
    % input key(1) and key(2) are each one integer
    % use modular exponentiation algorithm
    % since can't exponentiate directly with large numbers
    % here, only need to be able to square m and keep all significant figs

    % convert key(2) to binary char array
    % so we know which square terms in array p we need
    % in modular exponentiation
    b = dec2bin(key(2));
    blen = length(b);

    % get results in array p of successive squares of m mod key(1)
    p(1) = m;
    for i = 2:blen
      p(i) = mod( p(i-1)^2 , key(1) );
    end

    % compute result using the p(i) required by key(2)
    % use only the powers-of-two of bit values = 1 in b = key(2)
    % increasing index in p is higher power-of-two
    % increasing index in b is lower power-of-two
    % flip order of b to match index of powers-of-two in p
    b = flip(b);

    % use only the powers-of-two of bit values in flipped b == '1'
    C = 1; % initialize product
    for i = 1:blen
      if (b(i) == '1')
         C = mod(C, key(1)) * mod(p(i), key(1));
         C = mod(C, key(1));
      end
    end
    r = C; % return r
end
```