From <https://www.promptworks.com/blog/public-keys-in-perl-6>

Public Keys in Perl 6

When I first learned that Perl 6 had built in arbitrary precision integers, primality testing and support for modular arithmetic, I immediately thought, hey, let's implement the RSA public key algorithm.

Nowadays, many of us take for granted that we can go to our bank's website with a web browser or ssh to a remote server remotely, or join a VPN when we're out of the office and digital eavesdroppers will have a hard time watching our banking or seeing the commands we type.

But in order to do this there is some very cool math, namely math which allows us to communicate privately over a public channel. One of the first algorithms to give us this ability was the RSA encryption algorithm, developed in the 1970s by Rivest, Shamir, and Adelman.

Here's the algorithm, in short:

1. Generate two large prime numbers, p and q.
2. Multiply them together to get pq.
3. Multiply p-1 and q-1 together. This is called the Euler Phi function, written $\Phi(pq)$.
4. Pick another number e (that has no factors in common with the above).
5. Figure out the number d, such that d times e is 1 mod $\Phi(pq)$.
   This is the inverse of e mod $\Phi(pq)$, also written e^{-1}.
6. That's it! Your private key is [d , pq]. Your public key is [e, pq].

The public key can be advertised publicly, and anyone can use it to encrypt a message. Only a holder of the private key can decrypt the message.

To encrypt a message (an integer m), raise it to the power of e, mod pq.
To decrypt it, raise the result to the power of d mod pq.

Why this works is very interesting and elegant. It revolves around Fermat's Little Theorem. Why decryption is hard depends on the difficulty of factoring pq into p and q.

How can this algorithm be implemented in Perl 6? This is a really fun exercise in a language which provides us with primitives such as:

- `is-prime` checks to see if a number is prime.
- `expmod` raises a number to a power and take the remainder.
  Note this even works for negative exponents, allowing us to compute e^{-1}.
- `gcd` finds the greatest common divisor of two numbers.
- `..` constructs a range.
- `pick` picks a random element from a range.

Here it is, note that it really does generate primes that are 110 digits long!

```
sub random-prime(:$digits) {
  repeat { $_ = (10**$digits .. (10**($digits+1))).pick } until .is-prime;
  return $_;
}

my $p = random-prime(:110digits);
my $q = random-prime(:110digits);
my $pq = $p * $q;
my $phi = ($p-1) * ($q-1);
my $e;
repeat {
  $e = (1..$pq).pick;
} until $e gcd $phi == 1;

my $d = expmod($e, -1, $phi);
my $public  = [ $e , $pq ];
my $private = [ $d, $pq ];

```

Wasn't that easy? Now here are the encryption and decryption functions:

```

sub encrypt(:$message, :$key) {
  return expmod($message,$key[0],$key[1])
}

sub decrypt(:$message, :$key) {
  return expmod($message,$key[0],$key[1])
}

```

A testing framework is also built in to Perl 6, so add these few lines to test it on some random “message"s:

```
use Test;

for 1..100 {
  my $message = (1..$pq).pick;
  my $encrypted = encrypt(message => $message, key => $public);
  my $decrypted = decrypt(message => $encrypted, key => $private);
  ok $decrypted==$message, "decrypted message";
}
```

Here's the output (run through prove):

```
prove --exec=perl6 ./rsa.p6
./rsa.p6 .. ok
All tests successful.
Files=1, Tests=100,  1 wallclock secs ( 0.03 usr  0.01 sys +  0.48 cusr \
   0.04 csys =  0.56 CPU)
Result: PASS
```

Yes—it took about half a second.

If you want to encrypt words instead of numbers, you'll want to convert phrases into a sequence of numbers that are less than 110 digits long. A straightforward way to do that is to use the UTF-8 representation of the string. This is a sequence of integers that we can concatenate into a sequence of hex bytes, convert to decimal, then break up into 100-digit numbers. There are some nice built-ins to help us with this task. Namely:

- `Blob` is a type that is a sequence of unsigned ints.
- `encode` turn texts into a Blob (using UTF-8 by default).
- `decode` goes the other way.
- `:16(...)` and `base(16)` convert to and from base 16 (or other bases).
- `fmt` formats every element of an array (like `sprintf`) and joins them together with a separator.
- `comb` breaks up a string into pieces.

By the way, we'll also convert our 100 digit numbers to base 36 (which uses 0-9 and A-Z) to make them a little shorter.

```
loop {
  my $which = prompt "p)lain or c)ipher text?" or last;
  my $text  = prompt "Input: " or last;
  if ($which eq 'p') {
    say $text.encode      # encode into a UTF-8 Blob
     .List                # turn into a list of unsigned ints
     .fmt("%02x","")      # turn into hex digits
     .map( {:16($_)} )    # turn into a big number
     .Str                 # turn into a string
     .comb(100)           # break up into 100-digit pieces
     .map({ encrypt(message => $m, key => $public).base(36) })
     .join(' ');         # join encrypted base 36 numbers
  } else {
    my @bytes =
    $text.split(' ')  # create an array of base 36 numbers
      .map({ decrypt(message => :36($_), key => $private) })
      .join              # combine into a string
      .Int               # turn into an int
      .base(16)          # make it hex
      .comb(2)           # divide into pairs of digits
      .map({:16($_)});   # each pair is an unsigned int
    say "Output: " ~
        Blob.new(@bytes).decode
  }
}
```

And a sample run:

```
p)lain or c)ipher text?p
Input: My voice is my passport.
5J9767KKLO5ND7M0DMQZ5CM9UZFK7BTTH68930YCZ3LA68FRC3MXRP6BPI54KPN82CSNTRZVG1141\
  IWO525J42M3Q7EHPDF9LFKUKP77CDG47TMIKJ8Y6NWXH4LNX1DSNGHCF30DQXW3Q9
p)lain or c)ipher text?c
Input: 5J9767KKLO5ND7M0DMQZ5CM9UZFK7BTTH68930YCZ3LA68FRC3MXRP6BPI54KPN82CSNTR\
  ZVG1141IWO525J42M3Q7EHPDF9LFKUKP77CDG47TMIKJ8Y6NWXH4LNX1DSNGHCF30DQXW3Q9
Output: My voice is my passport.
```

Since we’re using UTF-8, unicode works fine too:

```
p)lain or c)ipher text?p
Input: ☺
3XSKWJCB3F6AN5SH2UZH2B6MMVLYPO0MM6EU0BN6NORFHJRO382XUIG756SVDN3SL7EL07ERUIMGK\
  5J4N6SW3OJIBXM24HK13UJ4JNVLM1KPWQRT0PBBLLLPL6729OY8GI5K7IUGKLF9MDZ
p)lain or c)ipher text?c
Input: 3XSKWJCB3F6AN5SH2UZH2B6MMVLYPO0MM6EU0BN6NORFHJRO382XUIG756SVDN3SL7EL07\
  ERUIMGK5J4N6SW3OJIBXM24HK13UJ4JNVLM1KPWQRT0PBBLLLPL6729OY8GI5K7IUGKLF9MDZ
Output: ☺
```