RSA

**Key generation.** The following program generates a valid modulus and a couple of keys for RSA. The argument `secur` indicates the number of digits of the involved primes.

```python
# Generates a valid modulus and the private and public keys for RSA
#
def new_rsa(secur):
    # random primes
    p = random_prime(10^secur, proof=True, lbound=10^(secur-1))
    q = random_prime(10^secur, proof=True, lbound=10^(secur-1))
    N = p*q # product

    # public key = random coprime to phi(N)
    k1 = 0
    while ( gcd(k1,(p-1)*(q-1)) != 1):
        k1 = ZZ.random_element((p-1)*(q-1))

    # private key = inverse modulo phi(N)
    k2 = Integer(Mod(k1,(p-1)*(q-1))^(-1))

    print '(Modulus, public key, private key) =', (N,k1,k2)
```

**Encryption.** Working modulo $N$ we can only use $k$ ASCII characters at the same time with $256^k < N$. We include in our encryption map a previous subdivision into $l\text{block} = \lceil \log_{256} N \rceil$ blocks.

```python
# encrypt: text, modulus, public key
#
def rsa_encrypt(text, N, k1):
    textblocks = []
    # The length of the blocks depends on the size of N
    lblock = floor(log(N,256))

    i=0
    while i<len(text) :
        number = encoding(text[i:i+lblock])
        textblocks.append( Integer(Mod(number,N)^k1) )
        i += lblock

    print textblocks
```

The keyword `Integer` is not mandatory. It means that we want to consider the member of the result as integers (in principle they are classes in $\mathbb{Z}/N\mathbb{Z}$). This is technical point to avoid strange errors.
Decryption. The decryption map is similar. In fact simpler because we do not need to divide into blocks.

```python
# decrypt: list, modulus, private key
#
def rsa_decrypt(listblocks, N, k2):
    result = '',
    for i in range(len(listblocks)):
        result += decoding(Integer(Mod(listblocks[i], N)**k2))
    print result
```

Here, forgetting `Integer` may raise an error because the function `.digits(256)` requires an integer.

We assume that we are using the encoding scheme through the following functions:

```python
# text to number
def encoding(text):
    result = 0
    for c in text:
        result = 256*result + ord(c)
    return result

# number to text
def decoding(number):
    result = ''
    for i in number.digits(256):
        result = chr(i)+result
    return result
```

Examples. Recall that there is a random generator involved then the result varies from one execution to another.

The `new_rsa(6)` gave the output:

```
(Modulus, public key, private key) = (219266483983, 14540633643, 129377908227)
```

Now
```
rsa_encrypt( 'This is the message that I want to encrypt', 219266483983, 14540633643)
```

produces
```
[130485699147, 3134128073, 140924980646, 79706721856, 129273483413, 30693030229, 171723311418, 25554864482, 24138014026, 210176748125, 18986056854]
```

The decryption function
```
rsa_decrypt([130485699147, 31341128073, 140924980646, 79706721856, 129273483413, 30693030229, 171723311418, 25554864482, 24138014026, 210176748125, 18986056854], 219266483983, 129377908227)
```
recovers the original message: This is the message that I want to encrypt.

Using longer primes (bigger secure) we get a smaller number of blocks. We mention quickly the results corresponding to 30-digit primes. The indented paragraphs are the outputs.

```python
new_rsa(30)

(Modulus, public key, private key) =
(178822531229628350654373866803690746889961703402698680689,
  157265418534171569110276840013576630523458187664904640362369,
  158449061285455078635939913349462061699994698649250183999969)

rsa_encrypt( 'This is the message that I want to encrypt',
  178822531229628350654373866803690746889961703402698680689,
  157265418534171569110276840013576630523458187664904640362369)

[98323742423424547308239594268482588641480597776763416344456,
  172780601364428468575515034163911203354386676628954809107325]

rsa_decrypt(
  [98323742423424547308239594268482588641480597776763416344456,
  172780601364428468575515034163911203354386676628954809107325],
  178822531229628350654373866803690746889961703402698680689,
  158449061285455078635939913349462061699994698649250183999969)

This is the message that I want to encrypt
```