Encoding Schemes and Finite Fields

Encoding and decoding. The most common encoding scheme is ASCII (American Standard Code for Information Interchange). It assign a one-byte number (actually a 7-bit number in its original form) to each character and to some control characters.

Given a string of characters $c_n c_{n-1} \dots c_1 c_0$ with ASCII codes $a_n a_{n-1} \dots a_1 a_0$ the natural encoding is represent this string by the number $\sum_{i=0}^{n} 256^i a_i$.

In Python the function `ord(c)` gives the ASCII code of `c` and `chr(n)` reverses this map. Then the follwing function performs the natural encoding.

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

For instance `ord('H')=72` and `ord('i')=105`. Then `encoding('Hi')` produces $18537 = 256 \cdot 72 + 105$. With more characters we obtian bigger numbers, for instance `encoding('Hello')` is `310939249775`.

The inverse bunction consist of getting the digits in base 256. This can be done with some special functions (see below) but with our knowledge the natural approach is

```
# number to text
def decoding(number):
    number = int(number)
    result = ''
    while number !=0:
        result = chr( (number % 256) ) + result
        number //= 256
    return result
```

For instance with `decoding(310939249775)` we recover `'Hello'`.

Example: 78 556 652 729 377 means `'Great!` after decoding.

The direct approach is using the sentence n.digits(b) to obtain in Sage the list of digits of `n` in base `b`. The list start from least significant digits. If you don't like this ordering, you can apply the .reverse Python list method. For instance

```
sage: 18537.digits(10)
[7, 3, 5, 8, 1]
sage: 18537.digits(256)
[105, 72]
sage:  L = 18537.digits(10)
sage: L.reverse()
sage: L
[1, 8, 5, 3, 7]
```

Then our encoding function reduces to

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

As we mentioned before it is unrealistic to assume that we can encode the message with a single number m when we are working modulo p because for a long message, p should have zillions of digits. The simplest and most common solution is to divide the message in block of fixed length.

```
long_text = 'En un lugar de la Mancha de cuyo nombre no quiero...'
for i in range(0,len(long_text),2):
    print elgamal_encrypt(210904,3,2^19-1,encoding(long_text[i:i+2]))
```

The previous program allows to employ any prime $p > 256^2$. In general , employing blocks of length k we need $p > 256^k$. For p having 100 decimal digits $k < 42$.

Finite fields. If we wish to define a finite field $\mathbb{F}_{p^k}$ as $\mathbb{F}_p[x]/\langle M\rangle$ then we should to write in Sage

```
F3pol.<X> = GF(3)[] # These are the polynomials F_3[X]
F9.<X> = GF(3^2, modulus=X^2 + X + 2 ) #These are the classes
```

Of course the names `F3pol` and `F9` are not mandatory.

Probably the following lines and their output give some hint about how it works.

```
print 'F3pol =', F3pol
print 'F9 =', F9
print 'Elements of F9 :',
for i in F9:
    print i,',',
print 'There are',len(F9),'elements'
```

```
F3pol = Univariate Polynomial Ring in X over Finite Field of size 3
F9 = Finite Field in X of size 3^2
Elements of F9 : 0 , 2*X , 2*X + 1 , X + 1 , 2 , X , X + 2 , 2*X + 2 , 1 ,
                There are 9 elements
```

Note that `X` is in `F3pol` the variable of the polynomial but in the `F9` becomes an element of the field. Sometimes it is needed to avoid this clash of notation. If we want to name the second `X` as `Y` then we may define define

```
F3pol.<X> = GF(3)[] # These are the polynomials F_3[X]
F9.<Y> = GF(3^2, name = 'Y', modulus=X^2 + X + 2 ) #Eq. classes
```

Anyway we prefer here to conserve the double meaning of `X`. Let us consider some The field `F3pol`. We can change the irreducible polynomial giving the modulus and even leave Sage to choose it internally. the method `.modulus()` allow to know it.

```
F3pol.<X> = GF(3)[]
F9tilde.<Y> = GF(3^2, name = 'Y', modulus=X^2 + 1 )
print F9tilde
F9.<X> = GF(3^2, modulus=X^2 + X + 2 )
print F9
F9bysage.<X> = GF(3^2)
print F9bysage
print F9bysage.modulus()
```

```
Finite Field in Y of size 3^2
Finite Field in X of size 3^2
Finite Field in X of size 3^2
x^2 + 2*x + 2
```

Some computations in $\mathbb{F}_9$

```
F3pol.<X> = GF(3)[]
F9.<X> = GF(3^2, modulus=X^2 + X + 2 )

print '1) (X+1)^9 =', (X+1)^9
print '2) 1/X =', 1/X
print '3) (X+2)/(X^100+X+1) =', (X+2)/(X^100+X+1)
```

The group of units of the finite field $\mathbb{F}_{p^k}$ is obviously $\mathbb{F}^*_{p^k} = \mathbb{F}_{p^k} - \{\overline{0}\}$, then it contains $p^k - 1$ elements. By Lagrange's theorem

$$a^{p^k-1} = 1 \qquad \forall a \in \mathbb{F}^*_{p^k}.$$

For instance

```
F.<X> = GF(3)[]
F81.<X> = GF(3^4)
for i in F81:
    print i^80
```

prints a list of a zero and 80 ones.

To compute a generator in Sage use `K.multiplicative_generator()` where `K` is the field. In the previous example `printF81.multiplicative_generator()` gives `X`.

Encoding and decoding using finite fields. Note that `X` must be in F_p^k with k greater than the maximum number of characters.

```
# text to element of F_p^k (p>256)
def encodingff(text):
    result = 0
    for c in text:
        result = X*result +ord(c)
    return result

# Element of F_p^k (p>256) to text
def decodingff(poly):
    result = ''
    for i in poly.polynomial().coeffs():
        result = chr(i) + result
    return result
```

For instance, working in $\mathbb{F}_{257^{20}}$ we can manage strings of at most 20 characters.

```
F.<X> = GF(257)[]
K.<X> = GF(257^20)
print encodingff('Hi!')
print decodingff(72*X^2 + 105*X + 33)
print decodingff( encodingff( 'This text is too long ' ) )
```

gives

```
72*X^2 + 105*X + 33
Hi!
```

and a bunch of strange symbols.

ElGamal cryptosystem works in the same way using finite fields

```
def elgamal_decrypt(pri_key,g,p,(m1,m2)):
    return Mod(m2,p)*Mod(m1,p)^(-pri_key)

def elgamal_encrypt(pub_key,g,p,message):
    k = floor( 1+1000000*random() )
    return (Mod(g,p)^k, message*Mod(pub_key^k,p))


# ElGamal in finite fields
F.<X> = GF(257)[]
K.<X> = GF(257^20, modulus=X^20 +X+70)

g = X + 4
pri_key = 123456789
pub_key = g^pri_key
p = X^20 +X+70
message = encodingff('This is a message')

print elgamal_encrypt(pub_key,g,p,message)
print decodingff( elgamal_decrypt(pri_key,g,p,
                  elgamal_encrypt(pub_key,g,p,message)   ) )
```

If you find difficult to figure out an irreducible write `K.<X>=GF(257^20)` and extract the modulus with `p=K(K.modulus())`. The first `K` is to specify that you want an element of the field, not a polynomial.