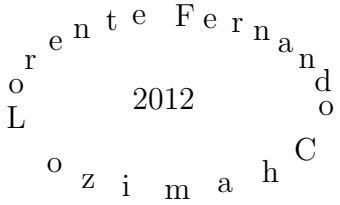
Basic Lectures on Cryptography

(with `Sage` programs)

Fernando Chamizo

Graduate course 2010–2011



Contents

Preface

English is the Esperanto of the real world and under the excuse of the internationalization of our graduate school (perhaps motivated by the European Higher Education Area), I was told in a corridor of the Math Department that my course on Cryptography had to be taught in English. I asked to the person in charge and he confirmed that it was mandatory (later I learned that the meaning of "mandatory" depends on how obedient you are). To my shame my English is very poor (as you have probably already noticed). These basic lecture notes served as a backup for me and provided support for foreign students that probably did not follow my mumbling. Sorry Shakespeare for the zillions of mistakes that these notes surely contain.

Very soon I realized that my best chance to hide my handicap with language was to turn part of the lectures toward the computers because they usually understand me better than flesh and bone beings. But do not be mistaken, there was a theoretical part of each topic, simply I did not typed it.

A glance through the table of contents shows that besides the lectures themselves there is some extra material. I have included the home assignments and quizzes employed for the students assessment. I have also separated some programs of the course. Most of them are also included in a way or another in the first part but here it is easier to see them and copy and paste to your favorite text editor. Finally, there is an appendix containing some practical points in programing and computer science focused on the software and the operating system (Ubuntu Linux) employed in this and other courses.

Madrid, July 2012

Fernando Chamizo

Part I

Lectures

Elementary Number Theory

Definition: Given $a, b \in \mathbb{Z}$ not simultaneously zero their *greatest common divisor*, denoted $\gcd(a, b)$ or (a, b) is the largest integer dividing both a and b. If $\gcd(a, b) = 1$ then a and b are said to be *relatively prime* or *coprime*.

Proposition: *Given a and b as before, there exist $x, y \in \mathbb{Z}$ such that*

$$ax + by = \gcd(a, b).$$

Sketch of the proof: Use Euclidean algorithm. i.e., note that $a = bc + r \Rightarrow \gcd(a, b) = \gcd(b, r)$ and iterate this fact. □

IMPORTANT REMARK: The computation of $\gcd(a, b)$ and the computation of x and y require a quantity of operations comparable to the number of digits. This is very quick for a computer with the numbers employed in actual cryptography (hundreds of digits).

Sage commands: `gcd(a,b)` or `gcd([list])`. The extended version `xgcd(a,b)` gives the gcd and x and y in the proposition.

```
sage: gcd(18,42)
6
sage: gcd([18,42,14])
2
sage: xgcd(18,42)
(6, -2, 1)
```

Definition: We say that a *is congruent to* b *modulo* $m \in \mathbb{Z}^+$, denoted $a \equiv b \pmod{m}$ or $a \equiv b\ (m)$, if $m \mid a - b$.

Recall: the symbol $\mid$ means "divides". Its negation is $\nmid$.

For a fixed m the congruence defines an equivalence relation. The classes are denoted by $\mathbb{Z}/m\mathbb{Z}$. This set inherits the $+$ and $\times$ operations from $\mathbb{Z}$. Note that $\overline{a} \in \mathbb{Z}/m\mathbb{Z}$ represents the set of all number differing from a in a multiple of m. In some cases we replace the heavy notation $\overline{a}$ by a.

Proposition: *An element* $\overline{a} \in \mathbb{Z}/m\mathbb{Z}$ *has a multiplicative inverse if and only if* $\gcd(a, m) = 1$.

Sketch of the proof: Note that $1 = ax + my$ means $\overline{a} \cdot \overline{x} = \overline{1}$ in $\mathbb{Z}/m\mathbb{Z}$. □

Sage commands: `n.inverse_mod(m)`. In some sense `Mod(a,n)` means $\overline{a}$.

```
sage: 10.inverse_mod(13)
4
sage: Mod(4*10,13)
1
sage: Mod(4,13)*Mod(10,13)
1
```

Definition: The elements of $\mathbb{Z}/m\mathbb{Z}$ having multiplicative inverse are called *units*. The Euler ϕ-function is the function that assigns to each m the number of units in $\mathbb{Z}/m\mathbb{Z}$.

Fact: It is not difficult to prove that if $m = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ is the prime factorization of m then

$$\phi(m) = \prod_{i=1}^{k} p_i^{\alpha_i - 1}(p_i - 1) = m \prod_{p|m} \left(1 - \frac{1}{p}\right).$$

Sage commands: `euler_phi(m)`. This function involves the factorization of m hence it is in general very hard for the computer when m has hundreds of digits.

```
sage: euler_phi(39)
24
sage: time euler_phi(10^120+1)
CPU times: user 1494.33 s, sys: 1.20 s, total: 1495.53 s
Wall time: 1504.81 s
```

Proposition (Chinese Remainder Theorem): *Given* m_1 *and* m_2 *relatively prime, there exists* x *such that*

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \end{cases}$$

Indeed x *is unique if we impose* $0 \leq x < m_1 m_2$.

Sketch of the proof: Consider $x = a_1 M_2 m_2 + a_2 M_1 m_1$ where M_1 is the inverse of m_1 modulo m_2 and M_2 is the inverse of m_2 modulo m_1. □

Sage commands: `crt(a_1,a_2,m_1,m_2)`.

```
sage: crt(4,6,7,11)
39
sage: Mod(39,7)
4
sage: Mod(39,11)
6
```

A little of Algebra (see actual definitions in your favorite book):
An abelian group is a set G with an operation that behaves as the addition in $\mathbb{Z}$, i.e, satisfies commutativity, associativity, existence of identity element and existence of inverse element.

The units of $\mathbb{Z}/m\mathbb{Z}$ form a group with respect to multiplication, the group of units. The entire set $\mathbb{Z}/m\mathbb{Z}$ is an abelian group when endowed with addition.

One of the most basic theorems in group theory is *Lagrange Theorem*, saying that if $H \subset G$ are finite groups (with the same operation) then $\#H$ divides $\#G$.

In a finite abelian group the powers of an element (repeated operation of an element with itself) form a group. Its cardinality is called the *order* of the element. In other words the order is the minimal $k \in \mathbb{Z}^+$ such that it takes k self-operations over a to reach the identity element.

Proposition (Euler-Fermat congruence): *Let a and m be relatively prime. then*

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

Proof: According to Lagrange Theorem applied to the units of $\mathbb{Z}/m\mathbb{Z}$ we have $a^k \equiv 1 \pmod{m}$ for some k dividing the cardinality of the group which is $\phi(m)$. □

For p prime the units of $\mathbb{Z}/p\mathbb{Z}$ are all the classes except $\overline{0}$ then the previous result reads

$$a^{p-1} \equiv 1 \pmod{p} \qquad \text{for every } p \nmid a.$$

This is called *Fermat's little theorem.*

Definition: It can be proved (it is elementary but not simple) that there are some elements $\overline{a} \in \mathbb{Z}/p\mathbb{Z}$ such that $\{\overline{a}^1, \overline{a}^2, \ldots, \overline{a}^{p-1}\} = \mathbb{Z}/p\mathbb{Z} - \{\overline{0}\}$. An element $\overline{a}$ or a with this property is called a *primitive root* modulo p.

Sage commands: The order of a unit in $\overline{a} \in \mathbb{Z}/m\mathbb{Z}$ can be computed with `Mod(a,m).multiplicative_order()`. The command `primitive_root(p)` generates a primitive root modulo p.

```
sage: Mod(2,5).multiplicative_order()
4
sage: Mod(2^4,5)
1sage: print [Mod(2^i,5) for i in range(12)]
[1, 2, 4, 3, 1, 2, 4, 3, 1, 2, 4, 3]
sage: primitive_root(5)
2
sage: primitive_root(103)
5
```

A computational short cut to calculate powers modulo m is

1. Use Euler-Fermat congruence if possible.
2. Employ the repeated squaring method.

The latter method consists of writing the exponent in base-2 system to reduce all the calculations to squaring over and over.

For instance to compute $x \equiv 2011^{196} \pmod{127}$ note firstly $2010 \equiv 106 \equiv -21$, then $x \equiv 21^{196}$. Euler-Fermat congruence says $21^{126} \equiv 1$ then $x \equiv 21^{70}$. Writing $70 = 2^6 + 2^2 + 2$ we have $x \equiv 21^{2^6} \cdot 21^{2^2} \cdot 21^2$ that can be computed by iterated squaring of 21 (recall to reduce modulo 127 after each squaring).

Sage commands: Computing $a^n \pmod{m}$ is easy for the computers even when large numbers (of hundred of digits) are involved in the calculation. The most direct command is `power_mod(a,n,m)`.

```
sage: power_mod(5^40+2,7^30,10^20+1)
65622873844338379843
sage: # Also
sage: Mod(5^40+2,10^20+1)^(7^30)
65622873844338379843
sage: # Avoid this way of doing the calculation!!!
sage: Mod((5^40+2)^(7^30),10^20+1)
```

Affine ciphers

We assign to each letter `A-Z` a number `0-25`. After this coding we work in $\mathbb{Z}/26\mathbb{Z}$. In this context an affine cipher is a map $f : \mathbb{Z}/26\mathbb{Z} \longrightarrow \mathbb{Z}/26\mathbb{Z}$ given by $f(x) = ax + b$ with a and 26 relatively prime. The numbers a and b are our secret keys. In the following program correspond to the variables `key1` and `key2`.

```
alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

message = 'MYSECRET'

# ENCRYPT MESSAGES
key1 = 3
key2 = 0

encrypted = ''
for c in message:
    loc = alph.find(c)
    encrypted += alph[Mod(key1*loc+key2,26)]
print message
print encrypted
```

If you are not a skilled Pythonist or Sage you will appreciate the following comments:

```
# Start with and empty encrypted message
# we'll add characters with a loop
encrypted = ''
# This is the loop c is each character in the message
for c in message:
    # Search the position of c in alph.
    # This is  the code corresponding to c
    loc = alph.find(c)
    # Computes (key1*loc+key2 modulo 26 and
    # append the corresponding character to encrypted
    encrypted += alph[Mod(key1*loc+key2,26)]
#Print the original and the encrypted messages
print message
print encrypted
```

Reusing is part of the Python philosophy and we can recycle our program more easily using functions. In the computer science jargon this is a kind of encapsulation. We call the function using the message and the keys and we do not care about the internal definition of the alphabet or the access to it.

```
1 def encrypt(message, key1,key2):
2     alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
3     encrypted = ''
4     for c in message:
5         loc = alph.find(c)
6         encrypted += alph[Mod(key1*loc+key2,26)]
7     print message,'->',encrypted
```

Now we encrypt with

```
# ENCRYPT MESSAGES
encrypt('MYSECRET', 3,0)
```

that gives KUCMGZMF.

The inverse function of $f(x) = ax + b$ is $g(x) = a^{-1}(x - b) = a^{-1}x - a^{-1}b$. We have to use it to decrypt messages

```
encrypt('KUCMGZMF', 3.inverse_mod(26),0)
```

gives MYSECRET.

In general

```
encrypt(..., key1.inverse_mod(26),-key1.inverse_mod(26)*key2)
```

inverts

```
encrypt(..., key1,key2)
```

Note that we really need $a =$ key1 and 26 to be relatively prime because we have to compute the multiplicative inverse of $\overline{a}$ in $\mathbb{Z}/26\mathbb{Z}$ to apply the inverse map.

We can add new characters to our alphabet (the usual ASCII code employs 256 characters, one byte) modifying the modulo.

Here we have an example:

```
1  def encrypt27(message, key1,key2):
2      alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ?'
3      encrypted = ''
4      for c in message:
5          loc = alph.find(c)
6          encrypted += alph[Mod(key1*loc+key2,27)]
7      print message,'->',encrypted
8
9  # ENCRYPT MESSAGES
10 encrypt27('HOWAREYOU?', 2,0)
11 encrypt27('OBRAHIVBNZ', 2.inverse_mod(27),0)
```

If we encrypt with

```
encrypt27('HOWAREYOU?', 2,0)
```

we decrypt with

```
encrypt27('OBRAHIVBNZ', 2.inverse_mod(27),0)
```

Hill ciphers

Hill cipher and affine cipher are alike. Both of them employ a function $f(x) = ax + b$ to encrypt. the difference is the dimension. Affine cipher are one-dimensional, take one character each time. Hill ciphers act on blocks of k characters. It forces to consider $x = \vec{x}$ and $b = \vec{b}$ with $\vec{x}$ and $\vec{b}$ two k-dimensional vectors and $a = A$ a $k \times k$ matrix.

In the following program we take

$$A = \begin{pmatrix} \texttt{key11} & \texttt{key12} \\ \texttt{key21} & \texttt{key22} \end{pmatrix} = \begin{pmatrix} 9 & 5 \\ 7 & 4 \end{pmatrix} \qquad \text{and} \qquad \vec{b} = \begin{pmatrix} \texttt{key1} \\ \texttt{key2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The blocks are of dimension 2. The variable `digraph` runs over the block of two consecutive characters.

```
def encrypt_hill(message, key11, key12, key21, key22, key1, key2):
    alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

    encrypted = ''
    for i in range(0,len(message),2):
        digraph = message[i:i+2]
        encrypted += alph[Mod(key11*alph.find(digraph[0])
                    +key12*alph.find(digraph[1]) + key1,26)]
        encrypted += alph[Mod(key21*alph.find(digraph[0])
                    +key22*alph.find(digraph[1]) + key2,26)]
    print message,'->',encrypted
```

Encrypting `WHATEVER`

```
# encrypt using the matrix [9, 5; 7, 4] and b=0
encrypt_hill('WHATEVER', 9,5,7,4, 0,0)
```

we get `ZARYLIRS`.

To decrypt this message we have to employ the inverse function $A^{-1}(\vec{x}-\vec{b}) = A^{-1}\vec{x} - A^{-1}\vec{b}$. The inverse should be computed $\pmod{26}$. In this case it does not matter because A^{-1} is an integral matrix.

$$A^{-1} = \begin{pmatrix} 9 & 5 \\ 7 & 4 \end{pmatrix}^{-1} = \begin{pmatrix} 4 & -5 \\ -7 & 9 \end{pmatrix} \qquad \text{and} \qquad A^{-1}\vec{b} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Consequently

```
# decrypt using its inverse [4, -5; -7, 9] and b=0
encrypt_hill('ZARYLIRS', 4,-5,-7,9, 0,0)
```

gives `WHATEVER`.

Let us practice with a non-zero vector $\vec{b}$, take for instance

$$A = \begin{pmatrix} \texttt{key11} & \texttt{key12} \\ \texttt{key21} & \texttt{key22} \end{pmatrix} = \begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix} \qquad \text{and} \qquad \vec{b} = \begin{pmatrix} \texttt{key1} \\ \texttt{key2} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

```
# encrypt using the matrix A=[3, 5; 7, 2] and b=[1,1]
encrypt_hill('SECRET', 3,5,7,2, 1,1)
```

We obtain `XFOXEP`.

To decrypt we have to compute

$$A^{-1} = \begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix}^{-1} = -\frac{1}{29}\begin{pmatrix} 2 & -5 \\ -7 & 3 \end{pmatrix} \equiv \begin{pmatrix} 8 & 19 \\ 11 & 25 \end{pmatrix} \qquad \text{and} \qquad A^{-1}\vec{b} \equiv \begin{pmatrix} 25 \\ 16 \end{pmatrix}.$$

Then we recover `SECRET` with

```
# decrypt using  A^-1=[8, 19; 11, 25] and -A^-1b=[25,16]   (26)
encrypt_hill('XFOXEP', 8,19,11,25, 25,16)
```

Theoretical Cryptography

Basic notions

A plaintext message is the original message in a *readable*[1] form.

An encoding scheme converts letters into numbers (usually in binary form). We assume that this scheme is well-known (public domain) and does not involve a real encryption.

EXAMPLE: In the previous simple encryption algorithms the encoding scheme is A–Z $\longrightarrow$ 0–25.

The standard encoding scheme in practice is ASCII (the acronym for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).

Symbols, control characters, letters	$\longrightarrow$	numbers 0–255 (or 0–127)
A–Z	$\mapsto$	65–90
a–z	$\mapsto$	97–122

UTF-8, a fully compatible extension of ASCII broadly employed in Linux machines, is more popular in web pages than the original ASCII since 2007.

The Python command to calculate the ASCII code of a character is `ord`. Its inverse is `chr`.

```
for c in 'How are you?':
        print c, '=', ord(c)
```

```
H = 72          H = 01001000
o = 111         o = 01101111
w = 119         w = 01110111
  = 32            = 00100000
a = 97          a = 01100001
r = 114         r = 01110010
e = 101         e = 01100101
  = 32            = 00100000
y = 121         y = 01111001
o = 111         o = 01101111
u = 117         u = 01110101
? = 63          ? = 00111111
```

Use `print c, '=', bin(ord(c))[2:].rjust(8,'0')` to get the 8-digit binary representation of the second column.

[1] We always consider a plaintext message as a sequence of numbers (or a single number) instead of a sequence of characters.

Notation

To encrypt (to hide) a plaintext message we use a key (a secret number) to produce a ciphertext (the encrypted message).

We denote

$\mathcal{K}$ = Space of all possible keys
$\mathcal{M}$ = Space of all possible plaintext messages
$\mathcal{C}$ = Space of all possible ciphertext messages

In many instances $\mathcal{M} = \mathcal{C}$.

With this notation a cryptosystem is a pair of maps

$$\boxed{\begin{array}{l} e : \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \text{ the encryption map} \\ d : \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M} \text{ the decryption map} \end{array}}$$

[For the sake of simplicity use $e_k(m)$ and $d_k(c)$ instead of $e(k,m)$ and $d(k,c)$] Such that for every $k_1 \in \mathcal{K}$, the encryption key, there is a $k_2 \in \mathcal{K}$, the decryption key, satisfying

$$\boxed{d_{k_2}\big(e_{k_1}(m)\big) = m \qquad \forall m \in \mathcal{M}}$$

EXAMPLE: In the affine ciphers we can consider that the key is $k = (a, b)$ and $e_k(x) = ax+b$ and $d_k(x) = a^{-1}x - a^{-1}b$.

Private and Public key cryptosystems

EXAMPLE: In the previous simple encryption algorithms the decryption key k_2 is easily derived from the encryption key k_1. We can in fact assume that $k_1 = k_2$ and implement the easy function on d.

We say that $k_1 = k_2$ (or k_2 easily derived from k_1) corresponds to symmetric cipher or private key cryptosystems.

The notation is self-explanatory:

- symmetric: encrypter and decrypter have equal knowledge.
- private: the encryption key cannot be published

The opposite is called asymmetric cipher or public key cryptosystems.

In this case k_2 cannot be easily computed from k_1

You know how to encrypt **but** You know how to decrypt without extra information

Slipping a note under the closed door of an office is in some sense an example of this phenomenon. Anyone can hide the information in this way but only the owner of the office has the key to recover the information.

At first sight it seems impossible to design a computer based public key cryptosystem (with a channel, www, monitored by attackers). This course will show that this prejudge is wrong.

Requirements (Our wish list for a cryptosystem)

1. $\forall k \in \mathcal{K}$, $\forall m \in \mathcal{M}$ the function $e_k(m)$ must be easy to compute.
2. $\forall k \in \mathcal{K}$, $\forall c \in \mathcal{C}$ the function $d_k(c)$ must be easy to compute.
3. Given $c \in \mathcal{C}$ it must be very difficult to compute $d_k(c)$ without knowledge of k.

---

4. The chosen plaintext attack must be unfeasible

   i.e. given a (small) set of decrypted messages $(c_1, d_k(c_1))$, $(c_2, d_k(c_2)), \ldots$ $(c_n, d_k(c_n))$ it must be difficult to decrypt any other message.

In practice we also assume one of the *Kerckhoff's principles*: The security lies on the key. The algorithm itself "must be able to fall into the hands of the enemy'. This is a polite way of saying that attackers are not stupid and indeed they know every book and result on cryptography.

With respect tp the last requirement note that using the encoding scheme A-Z $\longleftrightarrow$ 0-25 and an affine chiper, if we know that `TRY` is encrypted as `EQN` or equivalently that `EQN` is decrypted as `TRY`, then we know

$$f: \begin{array}{lcl} \texttt{T} = 19 & \longrightarrow & 4 = \texttt{E} \\ \texttt{R} = 17 & \longrightarrow & 16 = \texttt{Q} \\ \texttt{Y} = 24 & \longrightarrow & 13 = \texttt{N} \end{array} \quad \Rightarrow \quad \begin{cases} a \cdot 19 + b = 4 \\ a \cdot 17 + b = 16 \end{cases}$$

Solving the system in $\mathbb{Z}/26\mathbb{Z}$

$$\left.\begin{array}{l} 19a + b = 4 \\ 17a + b = 16 \end{array}\right\} \Rightarrow 2a = -12 \equiv 14 \ (26) \Rightarrow a \equiv 7 \ (26) \Rightarrow b \equiv 1 \ (26)$$

Then the secret key is $a = 7$, $b = 1$ and we can decrypt any other message.

Discrete logarithm

If g is a primitive root mod p (prime) and $p \nmid h$ the sentence in Sage

```
log( Mod(h,p),  Mod(g,p) )
```

gives the discrete logarithm $0 \leq x < p-1$ such that $g^x \equiv h \pmod{p}$.

In order to simplify this heavy notation let us define a function:

```
# It works only when g is a primitive root modulo p
def dlog(h,g,p):
    return log( Mod(h,p),  Mod(g,p))
```

The output of

```
print primitive_root(103)
print dlog(8,5,103)
#of course dlog(0,5,103) does not exists
print dlog(1,5,103)
print dlog(5,5,103)
print dlog(22,5,103)
print dlog(13,5,103)
```

is `5, 30, 0, 1, 3, 72`.

If g is not a primitive root for $\mathbb{F}_p$ then the discrete logarithm may be not well defined.

```
print 'A primitive root=',primitive_root(23)
# 2 is not a primitive root modulo 23 then
# the following sentence raises an error
# print dlog(8,2,23)
# In fact this discrete logarithm is not
# uniquely defined
print Mod(2^3,23)
print Mod(2^14,23)
```

gives

```
A primitive root= 5
8
8
```

Note: In some versions of Sage sentences like `dlog(8,2,23)` do not raise an error and return a valid solution of $2^x \equiv 8 \pmod{23}$.

Discrete logarithm by brute force

```
# Discrete logarithm by brute force
# Example 21^x=29 (107)
for x in range(106):
    if Mod(21,107)^x == Mod(29,107):
        print 'x= ',x
        break
```

```
# Discrete logarithm for g^x=h (p) by brute force

def br_fr(h,g,p):
    for x in range(p-1):
        if Mod(g,p)^x == Mod(h,p):
            print 'x= ',x
            break
```

Comparing the performance with sage built-in function

```
time br_fr(7,3,2^19-1)
time dlog(7,3,2^19-1)
```

The number $2^{19}-1$ is prime and 7 is a primitive root.

```
x=  243983
Time: CPU 7.46 s, Wall: 7.50 s
243983
Time: CPU 0.00 s, Wall: 0.00 s
```

Baby-step and giant-step tables for $3^x \equiv 62\ (101)$

```
# Table baby-step giant-step for 3^x = 62 (101)
n = 10
g = Mod(3,101)
gn = Mod(3,101)^(-n)
print 'i', 'b', 'g'
for i in range(10):
    print i, g^i, 62*gn^i
```

i	0	1	2	3	4	5	6	7	8	9
3^i	1	3	9	27	81	41	22	66	97	89
$62(3^{-10})^i$	62	60	32	44	10	39	41	69	57	91

Baby-step giant-step algorithm

```
# Discrete logarithm for g^x=h (p) by baby-step giant-step

def baby_giant(h,g,p):
    baby = [1]
    giant = [h]
    n = 1+floor(sqrt(p-1))

    for i in range(1,n):
        baby.append( Mod(baby[i-1]*g,p) )

    g = Mod(g,p)^-n
    for j in range(1,n):
        giant.append( Mod(giant[j-1]*g,p) )

    for inters in set(baby).intersection( set(giant) ):
#       print 'i =', baby.index(inters)
#       print 'j =', giant.index(inters)
        print 'x =',  baby.index(inters)+n*giant.index(inters)
```

Checking performance

```
print '# decimal digits =', (2^19-1).ndigits()
print '# binary digits (bits) =', (2^19-1).ndigits(2)
time baby_giant(7,3,2^19-1)
time print dlog(7,3,2^19-1)
print ''
print '# decimal digits =', (2^31-1).ndigits()
print '# binary digits (bits) =', (2^31-1).ndigits(2)
time baby_giant(8,7,2^31-1)
time print dlog(8,7,2^31-1)
```

```
# decimal digits = 6
# binary digits (bits) = 19
x = 243983
Time: CPU 0.08 s, Wall: 0.10 s
243983
Time: CPU 0.00 s, Wall: 0.00 s

# decimal digits = 10
# binary digits (bits) = 31
x = 1454746986
Time: CPU 4.41 s, Wall: 4.51 s
1454746986
Time: CPU 0.00 s, Wall: 0.00 s
```

ElGamal cryptosystem

For ElGamal cryptosystem the encryption and the decryption functions are of the form

$$\begin{array}{ll} e_{k_1} : \mathbb{F}_p^* \longrightarrow \mathbb{F}_p^* \times \mathbb{F}_p^* & k_1 = g^{k_2} \in \mathbb{F}_p^* \quad \text{public encryption key} \\ \text{with} & \\ d_{k_2} : \mathbb{F}_p^* \times \mathbb{F}_p^* \longrightarrow \mathbb{F}_p^* & k_2 \in \mathbb{F}_p^* \quad \text{private decryption key} \end{array}$$

where

$$e_{k_1}(m) = (g^r, mk_1^r) \qquad \text{and} \qquad d_{k_2}(c_1, c_2) = c_2 c_1^{-k_2}$$

with $g \in \mathbb{F}_p^*$ of large order (ideally a generator) and r an arbitrary (random) number. Implicitly a plaintext message is an element $m \in \mathbb{F}_p^*$ and a ciphertext is a pair $(c_1, c_2) \in \mathbb{F}_p^* \times \mathbb{F}_p^*$.

In Sage the encryption function is

```
# ElGamal
# pub_key = public key
# g = generator or high order element
# p = prime
# message = number < p
def elgamal_encrypt(pub_key,g,p,message):
    k = floor( 1+(p-2)*random())
    return (Mod(g,p)^k, message*Mod(pub_key^k,p) )
```

and the decryption function is

```
# ElGamal
# pri_key = private key
# g = generator or high order element
# p = prime
# (m_1,m2) = couple of numbers < p
def elgamal_decrypt(pri_key,g,p,(m1,m2)):
    return Mod(m2,p)*Mod(m1,p)^(-pri_key)
```

If we keep r as a random number the value of $e_{k_1}(m)$ may be different each time that we use the function.

To compare results, let us put `k = 333`. Then for instance for the public key 210904 the message 12345 is encrypted with

```
elgamal_encrypt(210904,3,2^19-1, 12345 )
```

resulting `(29073, 277350)`.

To decrypt we need to know the private key corresponding to 210904. It is 1000 because $3^{1000} \equiv 210904 \pmod{2^{19}-1}$ (check it!).

Now

```
elgamal_decrypt(1000,3,2^19-1, (29073,277350) )
```

gives the right answer 12345.

Check that allowing k to be random the decryption function still works.

Breaking the ElGamal cryptosystem getting the private key k_2 from the public key k_1 requires to solve the DLP and this is considered very hard when p has hundreds of digits.

Quiz:

Take $p = 2^{31} - 1$ and $g = 7$. If the public key is 833 287 206 and the ciphertext is $(1\ 457\ 850\ 878, 2\ 110\ 264\ 777)$. What is the plaintext message?

Quiz:

Take $p = 2^{31} - 1$ and $g = 7$. If the public key is 1659750829 and the ciphertext is $(297629860, 1094924871)$. What is the plaintext message?

Solutions:

```
sage: log( Mod( 833287206,2^31-1), Mod(7,2^31-1))
2011
sage: elgamal_decrypt(2011,3,2^31-1, (1457850878,2110264777) )
23571113

sage: log( Mod( 1659750829,2^31-1), Mod(7,2^31-1))
1001
sage: elgamal_decrypt(1001,3,2^31-1, (297629860,1094924871) )
20110310
```

If we have a long text it is unrealistic to assume that we can encode the message with a single number $m \in \mathbb{F}_p^*$. It leads to some considerations with regard to the encoding schemes.

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 consists in representing 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 following 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 obtain bigger numbers, for instance `encoding('Hello')` is `310939249775`.

The inverse function 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 starts from the 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 into blocks 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 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

```
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`.

We can change the irreducible polynomial giving the modulus and even leave Sage to choose it internally. the method `.modulus()` allows 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 polynomial write `K.<X>=GF(257^20)` and extract the modulus with `p=K(K.modulus())`. The first `K` is to specify that you really want an element of the field, not a polynomial.

RSA

The RSA cryptosystem constitutes historically the first public key cryptosystem and certainly is the most famous. The spaces of all possible plaintext messages and of all possible ciphertext messages are $\mathcal{M} = \mathcal{C} = \big(\mathbb{Z}/N\mathbb{Z}\big)^*$ where $N = pq$ with p and q distinct large primes. The encryption and decryption functions $\big(\mathbb{Z}/N\mathbb{Z}\big)^* \longrightarrow \big(\mathbb{Z}/N\mathbb{Z}\big)^*$ are given by

$$e_{k_1}(m) = m^{k_1} \qquad \text{and} \qquad d_{k_2}(c) = c^{k_2}$$

where the public encryption key k_1 has to be chosen relatively prime to $\phi(N)$, with ϕ the Euler ϕ-function, and the private decryption key k_2 is the inverse of $k_1 \pmod{\phi(N)}$. By Euler-Fermat congruence $d_{k_2}\big(e_{k_1}(m)\big) = m$.

The strength of the cryptosystem depends on the difficulty, with present knowledge, to compute $\phi(N)$ if p and q are not published (it would be easy if we had quick factorization algorithms).

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.

```
#
# 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 `lblock` $= [\log_{256} N]$ blocks.

```
#
# 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.

```
#
# 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:

```
# 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, 31341128073, 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 `secur`) we get a smaller number of blocks. We mention quickly the results corresponding to 30-digit primes. The indented paragraphs are the outputs.

```
new_rsa(30)
```

```
      (Modulus, public key, private key) =
      (178822531229628350654373866803690748889688461703402698680689,
      157265418534171569110276840013576630523458187664904640362369,
      158449061285455078635939913349462061699994698649250183999969)
```

```
rsa_encrypt( 'This is the message that I want to encrypt',
178822531229628350654373866803690748889688461703402698680689,
157265418534171569110276840013576630523458187664904640362369)
```

```
      [983237424234245473082395942684825886414805977767634163444 56,
      172780601364428468575515034163911203354386676628954809107325]
```

```
rsa_decrypt(
[983237424234245473082395942684825886414805977767634163444 56,
172780601364428468575515034163911203354386676628954809107325],
178822531229628350654373866803690748889688461703402698680689,
158449061285455078635939913349462061699994698649250183999969)
```

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

Primality tests

Pseudoprimes. Recall that n is a pseudoprime to the base a coprime to n if $a^{n-1} \equiv 1 \pmod{n}$ but n is composite.

The following program makes a list of pseudoprimes to the base 2. Change 2000 by the upper limit of the list.

```
#
# pseudoprimes to the base 2
#
for n in range(3,2000,2):
    if Mod(2,n)^(n-1)==1:
        if is_prime(n)==False:
            print n, 'passes the test but is not prime'
```

Preserving 2000 the obtained list is short:

```
341 passes the test but is not prime
561 passes the test but is not prime
645 passes the test but is not prime
1105 passes the test but is not prime
1387 passes the test but is not prime
1729 passes the test but is not prime
1905 passes the test but is not prime
```

and it is even shorter if we impose n to be also pseudoprimes to the base 3

```
#
# pseudoprimes to the base 2 and 3
#
for n in range(5,2000,2):
    if (Mod(2,n)^(n-1)==1) and (Mod(3,n)^(n-1)==1):
        if is_prime(n)==False:
            print n, 'passes the test but is not prime'
```

```
1105 passes the test but is not prime
1729 passes the test but is not prime
```

A variation is considering a list of bases

```
#
# pseudoprimes to the bases of a list
#

def pseud(n):
    for k in list:
        if Mod(k,n)^(n-1) != 1:
            return False
    return True

list =[2,3,5,7]
for n in range(5,50000,2):
    if (is_prime(n)==False) and (pseud(n)==True):
        print n, 'passes the test but is not prime'
```

The result is again meager although the upper limit is now 50000.

```
29341 passes the test but is not prime
46657 passes the test but is not prime
```

Strong pseudoprimes. Given n odd we can write $n-1 = 2^k q$ with q odd.

The following lines of code compute q and k for a given n

```
def q_and_k(n):
    q = n-1
    k = 0
    while Mod(q,2)==0:
        k += 1
        q = q/2
    return q,k
```

Running

```
print q_and_k(23)
print q_and_k(57)
print q_and_k(65537)
```

we get

`(11, 1)`		$23-1 = 2^1 \cdot 11$
`(7, 3)`	meaning	$57-1 = 2^3 \cdot 7$
`(1, 16)`		$65537-1 = 2^{16}$

A *strong pseudoprime to the base* a is an odd composite number n such that either $a^q \equiv 1 \pmod{n}$ or $\exists\ 0 \leq t < k$ such that $a^{2^t q} \equiv -1 \pmod{n}$.

With the following program we check if a number n passes the test to be a strong pseudoprime to the base a.

```
def strong_pseud(n,a):
    (q,k) = q_and_k(n)
    b = Mod(a,n)^q
    if b==1:
        return True

    for i in range(k):
        if b==-1:
            return True
        b = Mod(b,n)^2
    return False
```

For instance, the output of

```
print strong_pseud(172947529,17)
print strong_pseud(172947529,23)
```

is `True` and `False`. We have $172947529 = 307 \cdot 613 \cdot 919$.

Miller-Rabin test. This is a test broadly employed in practice. It simply checks if an odd number passes the test for strong pseudoprime to many bases a.

These bases are commonly chosen as the first primes or as random numbers.

In the second version it is known that assuming a conjecture in analytic number theory (the Generalized Riemann Hypothesis) there are not composite numbers n passing the test for more than $2(\log n)^2$.

These two flavors of the test are contained in the following functions:

```
#
# Miller-Rabin
#
def miller_rabin(n,secur):
    for p in primes_first_n(secur):
        # Small primes
        if Mod(n,p)==0:
            if n==p:
                return 'Prime'
            return 'Composite'
        # Check strong pseudoprimes
        if strong_pseud(n,p)==False:
            return 'Composite'
    return 'likely prime'

def miller_rabin2(n,secur):
    a = 0
    for i in range(secur):
        a = 2+ZZ.random_element( n-3 )
        if Mod(n,a)==0:
            return 'Composite'
        if strong_pseud(n,a)==False:
            return 'Composite'
    return 'likely prime'
```

The parameter `secur` indicates the number of bases taken into consideration.

With `secur = 2` we get a couple of failures of the primality tests for $n < 50000$

```
secur = 2
for n in range(3,50000,2):
    if (is_prime(n)==False)and(miller_rabin2(n,secur)=='likely prime'):
        print n, 'passes the test but is not prime'
```

```
5461 passes the test but is not prime
31621 passes the test but is not prime
```

and these exceptions disappear taking `secur = 3`.

For `secur = 9` it can be checked that there are not exceptions less than $3.8 \cdot 10^{18}$. In fact the first exception is $3825123056546413051 = 149491 \cdot 747451 \cdot 34233211$ that can be ruled out with `secur = 12`.

Although its efficiency in practice, Miller-Rabin test is not a *deterministic* primality test. Its output is 'composite' or 'very likely prime'.

There are some deterministic tests, the most important is the *cyclotomic test* (Adleman-Pomerance-Rumely 1983) that runs in time $(\log n)^{O(\log\log\log n)}$. It is not very simple and requires methods of algebraic number theory.

There exists also a deterministic test (Agrawal-Kayal-Saxena 2004) running in $O\big((\log n)^k\big)$ with some constant k (it may be taken $k = 7.5$) but so far it is not a competitor to the cyclotomic test.

Commonly the best option (regarding to performance) is to apply very sophisticated primality tests only when we have tried direct division for some number and Miller-Rabin test because they allow to rule out the most of the composite number with almost no effort.

Factorization algorithms

We consider the problem of getting a nontrivial factor of a composite number. Factorization algorithms appeal recursively to the solution of this problem and combined with primality tests give the full prime factorization of a number.

Fermat's factorization. It is an extremely simple method essentially based on the relation $x^2 - y^2 = (x - y)(x + y)$. If we can find y such that $n + y^2 = x^2$ then $(x - y)|n$.

The following program uses this technique to obtain a nontrivial factor. Some lines at the beginning are included to detect primes and even numbers.

```
def fermat_factor(n):
    if Mod(n,2)==0:
        return 2
    if is_prime(n):
        return n
    y = 0
    while( is_square(n+y^2)==False ):
        y += 1
    return sqrt(n+y^2)-y
```

If $n = pq$ with p and q odd primes, as in RSA, then $n = x^2 - y^2$ with $x = (p + q)/2$, $y = (p - q)/2$. If p and q are very close then y is small and this simple method gives the factorization even for gigantic numbers.

For instance

```
p = random_prime(10^101,True)
q = next_prime(p)
n = p*q
print 'The number is', n
print 'It has', n.ndigits(), 'digits'

print '\nA factor is ', fermat_factor(p*q)
```

can factor in no time a 200 digits number of this form. The moral of the story is that in RSA close prime numbers have to be avoided.

```
The number is
659230925587256723667156710893739926206106725832901190192513650209261568\
920507606065172489394540316660737278499039707243803129856565681278595584\
288485116218855653949555427266986679356929385964479997673 3
It has 202 digits

A factor is
811930369913120520211362870245508687326172438342983987822130172511668081\
443829540242254414528977528 19
```

Pollard's $p-1$ algorithm. It is not useful for all numbers but it allows to factorize some extremely large special numbers. It computes $P(B) = \gcd(a^{B!} - 1, n)$ for increasing values of B. Of course, if $1 < P(B) < n$ for some B, we have got a nontrivial factor. Actually $B!$ is a simplification of the original algorithm, a slightly better choice of the exponent is $\mathrm{lcm}(1, 2, 3, \ldots, B)$. We shall take initially $a = 2$.

The theory suggests that this is a good algorithm if there are prime factors p such that the prime factorization of $p-1$ only contains small prime powers. Here 'good' means that the value of B is reasonably small.

This function computes the values of $P(B)$ for $B <$ `b` and return a nontrivial factor if it finds it.

```
def pollard_p(n,b):
    a = 2
    for i in range(1,b+1):
        a = Mod(a,n)^i
        d = gcd (a-1,n)
        if (d!=1) and (d!=n):
            return d
```

Note that $a_B = a^{B!}$ is computed by the recurrence $a_B = (a_{B-1})^B$ and, of course, we work modulo n, otherwise the size of a_B would be unmanageable for a computer.

With `pollard_p(10403,10)` we get the factor `101` and `None` if `10` is substituted by a smaller number.

A slight variation tries bigger and bigger values of B up to getting a nontrivial factor

```
def pollard_p_auto(n):
    a = 2
    i = 0
    d = n
    if is_prime(n):
        return d
    while (d==1) or (d==n):
        i += 1
        a = Mod(a,n)^i
        d = gcd (a-1,n)
    return d
```

One has to be careful with this program because for instance `pollard_p_auto(65)` enters into an infinite loop because $2^{B!} = 2^{12k}$ for $B > 3$ and $2^{12} \equiv 1 \pmod{65}$.

We avoid this problem changing the basis and starting up if at some point $a^{B!}$ becomes 1 modulo n.

```
def pollard_p_auto2(n):
    aa =2
    a = aa
    i = 0
    d = n
    if is_prime(n):
        return d
    while (d==1) or (d==n):
        i += 1
        a = Mod(a,n)^i
        d = gcd (a-1,n)
        if a == 1:
            aa += 1
            a = aa
            i = 0
    return d
```

It is interesting to check numerically the performance of the algorithm for $n = pq$ in terms of the factorization of the $p - 1$ where p is the output of `pollard_p_auto2(n)`. To this end we consider

```
k = 7
for i in range(20):
    p = random_prime(10^k, True)
    q = random_prime(10^k, True)
    t = cputime()
    f = pollard_p_auto2(p*q)
    dt = cputime(t)
    print factor(f-1), ' Time:', dt
```

that prints the factorization of $p - 1$ and the interval of time `dt` required by `pollard_p_auto2(n)` to get p.

```
2^2 * 3^2 * 139 * 347  Time: 0.043994
2^2 * 3 * 245261  Time: 30.766322
2^2 * 3^7 * 5 * 109  Time: 0.014998
2 * 17 * 19 * 8923  Time: 1.134828
2^4 * 3^2 * 23 * 1423  Time: 0.173974
2^4 * 5 * 157 * 503  Time: 0.061991
2^2 * 5 * 13 * 43 * 401  Time: 0.049991
2 * 7 * 11 * 4597  Time: 0.555916
2 * 3^2 * 31 * 4451  Time: 0.563914
2 * 3^2 * 13^2 * 19 * 79  Time: 0.010998
2 * 17^2 * 17159  Time: 2.112679
2^2 * 3 * 7 * 101149  Time: 12.303129
2^3 * 3^2 * 19 * 37 * 41  Time: 0.00699899999995
2^3 * 11 * 19 * 1993  Time: 0.241964
2 * 3 * 7 * 11 * 15199  Time: 1.845719
2 * 3^3 * 5 * 27743  Time: 3.366488
```

```
2 * 3 * 11 * 151 * 647  Time: 0.077988
2^2 * 31 * 157 * 199  Time: 0.024996
2^3 * 17 * 19 * 29 * 113  Time: 0.0149980000001
2 * 5 * 131113  Time: 16.300522
```

Note that the biggest number in this list corresponds to $p-1 = 2^2 \cdot 3 \cdot 245261$ having the unbalanced prime factor 245261. On the other hand, the best performance is for $p-1 = 2 \cdot 3^2 \cdot 13^2 \cdot 19 \cdot 79$ with many small small prime power factors.

Running the program with higher values of `k` (this is typically like one half of the number of digits) we realize that Pollard's $p-1$ algorithm is not convenient as a single method for general numbers.

For instance a table for `k=10` included some extreme values like

```
2^2 * 5 * 17 * 27685279  Time: 3471.164302
2 * 347 * 6327889  Time: 793.400386
```

The group law in elliptic curves

Elliptic curves. A first not very general definition of elliptic curve over a field K, $\mathrm{char}(K) \neq 2, 3$ is an algebraic curve of the form

$$E:\ y^2 = x^3 + ax + b \qquad \text{with } a, b \in K \quad , \text{ such that } 4a^3 + 27b^2 \neq 0$$

and the 'point at infinity' conventionally denoted by O. It makes sense in the framework of projective geometry.

In Sage there are several forms of introducing an elliptic curve. We consider here the easiest one matching the previous definition: `EllipticCurve(K,[a,b])`

If K is a finite field you can see the points running a `for` loop. For instance

```
E = EllipticCurve(GF(5),[-6,5])
for P in E:
    print P
```

produces the following list of points

```
(0 : 0 : 1)
(0 : 1 : 0)
(1 : 0 : 1)
(2 : 1 : 1)
(2 : 4 : 1)
(3 : 2 : 1)
(3 : 3 : 1)
(4 : 0 : 1)
```

These are the points of $E:\ y^2 = x^3 - 6x + 5$ belonging to $\mathbb{F}_5$, often denoted by $E(\mathbb{F}_5)$. They are in projective notation $(a : b : c)$ means $(a/c, b/c)$ when $c \neq 0$ and the only instance with vanishing last coordinate corresponds to the point at infinity.

Changing the first line to

```
E = EllipticCurve(GF(5^2,'a'),[-6,5])
```

we obtain the result as before plus new points. Remember that $\mathbb{F}_5 \hookrightarrow \mathbb{F}_{5^2}$.

An error is raisen trying to replace $\mathbb{F}_5$ or $\mathbb{F}_{5^2}$ by $\mathbb{F}_7$ or $\mathbb{F}_{7^2}$ because in these fields the condition $4a^3 + 27b^2 \neq 0$ does not hold.

The effect of `show(E)` is displaying the equation of E. This is useless with our presentation of elliptic curves but it is not with others.

The group law. Given P and Q in an elliptic curve over $\mathbb{R}$ we define $P + Q$ as the mirror image respect to the X-axis of the third intersection of the straight line passing through P and Q.

The following code shows it in a picture

```
var('x,y')
graph = implicit_plot(x^3-6*x+5-y^2, (x, -5,5),  (y, -5,5) )
graph += plot( x-1, x, -3,4)
graph += plot( x-1, x, -2,2, thickness=2)
graph += point([1,0],size=30) + point([2,1],size=30)
graph += point([-2,-3],size=30) + point([-2,3],size=30)
graph += line([(-2,3),(-2,-3)], linestyle = '--', thickness=2)
graph += text("P",(2.3,0.4),fontsize=20)
graph += text("Q",(1.3,-0.7),fontsize=20)
graph += text("R",(-2,-3.8),fontsize=20)
graph += text("P+Q",(-2.4,4),fontsize=20)

show(graph)
```

If $P = Q$ we consider that the straight line is the tangent line. If $P = (x, y)$ we define $P = (x, -y)$ and $P + (-P) = O$ (there is not third intersection, it is at infinity). We complete these formulas with $P + O = P$ and $O + P = P$.

It turns out that an elliptic curve E endowed with the operation $+$ is an abelian group.

Using the previous geometric interpretation if $P, Q \neq O$ and $Q \neq -P$ the explicit formula to add $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ is

$$P + Q = (x_3, m(x_1 - x_3) - y_1) \qquad \text{with } m = \frac{y_2 - y_1}{x_2 - x_1} \text{ and } x_3 = m^2 - x_1 - x_2.$$

If $P = Q$, m has to be replaced by $m = (3x_1^2 + a)/2y_1$ which is the slope of the tangent line.

These formulas can be extended to any K (losing the geometrical interpretation) and completed with the trivial cases.

Therefore the following function computes $P + Q$ (the group law)

```
#
# Group law in the elliptic curve y^2= x^3+a*x+b
# ('a' has to be defined in advance)
#
def g_l( P, Q ):
    if P == 'O':
        return Q
    if Q == 'O':
        return P

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return 'O'

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        m = (3*P[0]^2+a)/2/P[1]
    else:
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1]]
```

Taking `a=Mod(-6,13)` and `b=Mod(5,13)` (the last one is not necessary) the output of

```
print g_l( [2,1], [2,-1] )
print g_l( [2,1], [1,0] )
print g_l( [2,1], [2,1] )
print g_l( [5,-10], [5,-10] )
print g_l( 'O', [5,-10] )
print g_l(g_l(g_l( g_l( [5,-10],[5,-10] ),[5,-10]),[5,-10]),[5,-10])
```

is

```
O
[-2, 3]
[5, 3]
[2, 12]
[5, -10]
O
```

For instance, the last line means that $P+P+P+P+P=O$. We abbreviate this as $5P=O$.

According to the notation of group theory, we say that P has order 5.

In general the following function gives the order of a point P

```
def ord_p(P):
    k=1
    Q=P
    while Q!='O':
        k += 1
        Q = g_l(P,Q)
    return k
```

but this is not very efficient if the order is large. There are some shortcuts (not discussed here) using for instance the baby-step giant step algorithm.

The command `E.abelian_group()` computes the structure of the abelian group. For instance

```
E = EllipticCurve(GF(5),[-6,5])
print E.abelian_group()
```

inform us that for $E:\ y^2=x^3-6x+5$ over $\mathbb{F}_5$ the group is isomorphic (i.e. the same up to changing names) to $\mathbb{Z}/4\mathbb{Z}\times\mathbb{Z}/2\mathbb{Z}$. The exact output is

```
(Multiplicative Abelian Group isomorphic to C4 x C2, ((3 : 2 : 1), (4 : 0 : 1)))
```

The last part of the output indicates the generators. With

```
a = Mod(-6,5)
b = Mod(5,5)
print ord_p([3,2])
print ord_p([4,0])
```

we check that really the points $(3,2)$ and $(4,0)$ have order 4 and 2, respectively.

Built-in functions in Sage. Actually the functions introduced above in connection to group law are already implemented in Sage.

The point $(x, y) \in E$ in Sage is indicated by `E([x,y])` except the point at infinity that has the special notation `E(0)`.

The sum and the multiplication by an integer is written as usual. For instance the computations that we performed before on the elliptic curve $E: \ y^2 = x^3 - 6x + 5$ over $\mathbb{F}_{13}$ are shortened now with a lighter notation simply as

```
E = EllipticCurve(GF(13),[-6,5])
P = E([2,1])
Q = E([5,-10])

print P+(-P)
print P+E([1,0])
print 2*P
print 2*Q
print E(0)+P
print 5*P
```

Giving the expected result

```
(0 : 1 : 0)
(11 : 3 : 1)
(5 : 3 : 1)
(2 : 12 : 1)
(2 : 1 : 1)
(0 : 1 : 0)
```

The order of P is computed with `P.additive_order()`. Note the new notation with respect to the `multiplicative_order()` that we employed for the group of units of $\mathbb{Z}/N\mathbb{Z}$.

```
E = EllipticCurve(GF(5),[-6,5])
P = E([3,2])
print P.additive_order()
Q = E([4,0])
print Q.additive_order()
```

Gives 4 and 2 as before.

Lenstra's elliptic curve factorization algorithm

Repeated duplication method. To compute nP the obvious method is to apply n times the group law, i.e. $nP = P + \overset{n \text{ times}}{\ldots} + P$

```
def easy_mult(n,P):
    result = 'O'
    for i in range(n):
        result = g_l( P, result )
    return result
```

But this is useless when n is very large, say hundred of digits.

The following program applies the analog of the repeated squaring algorithm in $\mathbb{F}_p$. It is actually the same algorithm changing the multiplicative notation by the additive notation.

```
def mult_2(n,P):
    result = 'O'
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l( pow_2P, result )
        n //=2
        pow_2P = g_l( pow_2P, pow_2P )
    return result
```

Comparing both algorithms one have to reject the first one even for not very high values of n

```
E = EllipticCurve(GF(1000000007),[-6,5])
P = E([2,1])

a = Mod(-6,5)
b = Mod(5,5)

P=[2,1]
time easy_mult(10^6,P)
time mult_2(10^6,P)
```

gives

```
Time: CPU 8.97 s, Wall: 9.25 s
Time: CPU 0.00 s, Wall: 0.00 s
```

Factorization. In principle it is not possible to define an elliptic curve over a ring if we can save the group law. In fact the following line in Sage

```
E = EllipticCurve(GF(10403),[-6,5])
```

(note that 10403 is not prime) raises the error

```
ValueError: the order of a finite field must be a prime power
```

Let us see in an example what happens when we apply our function to add points in the ring $\mathbb{Z}/10403\mathbb{Z}$.

Example:

```
#
# Example P=(0,1) y^2= x^3+x+1, n = 10403
#

P= [0,1]
a= Mod(1, 10403)
print mult_2( factorial(7), P )

def mult_2(n,P):
    result = 'O'
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l( pow_2P, result )
        n //=2
        pow_2P = g_l( pow_2P, pow_2P )
    return result

def g_l( P, Q ):
    if P == 'O':
        return Q
    if Q == 'O':
        return P

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return 'O'

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        m = (3*P[0]^2+a)/2/P[1]
    else:
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1]]
```

Introducing at the beginning of the definition of the function the sentence

```
print P,Q
```

we learn that the error appears when adding

```
[9696, 506] [7878, 10200]
```

The reason is that when computing the slope m we have to invert `Q[0]-P[0]` $= -1818$ and this is not possible because 1818 and $n = 10403$ are not coprime.

Lenstra elliptic curve factorization It is the analog of Pollard's $p-1$ method. It consists in computing $1!P, 2!P, \ldots, B!P$ in an elliptic curve $E \pmod n$. If an error arises in the group law then it can be employed to get a factor of n. The power of the method is based on the fact that if the the factor is trivial or no error appears, one can easily change the elliptic curve. In some sense is like a Pollard's $p-1$ method with varying abelian groups.

Firstly we have to hack the group law to detect the cases in which the group law is not well-defined. We employ the following notation for the points on E. The last case is the output corresponding to an error in the group law.

```
#
# "Normal" points [x,y,1]
# Point at infinity [0,1,0]
# Fake points [0,0,d] with d not coprime to the modulus.
#
```

The modified group law function is:

```
#
# Group law
#
def g_l_l( P, Q, a ):
    if P[2] != 1:
        if P[1]==1:
            return Q
        return P
    if Q[2] != 1:
        if Q[1]==1:
            return P
        return Q

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return [0,1,0]

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        if P[1].is_unit()==False:
            return [0,0,P[1]]
        m = (3*P[0]^2+a)/2/P[1]
    else:
        if (Q[0]-P[0]).is_unit()==False:
            return [0,0,Q[0]-P[0]]
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1],1]
```

and the modified multiplication function is:

```
#
# Same multiplication routine
# changing 0 and g_l by g_l_l
#
def mult_2_l(n,P,a):
    result = [0,1,0]
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l_l( pow_2P, result, a )
        n //=2
        pow_2P = g_l_l( pow_2P, pow_2P, a )
    return result
```

For instance, the result of

```
g_l_l([9696, 506,1],[7878, 10200,1],1)
```

is now

```
[0, 0, -1818]
```

We integrate this functions in Lenstra's algorithm. We use $y^2 = x^3 + ax + 1$ as varying elliptic curve, because for any value of a the point $P = (0, 1)$ (that we take as starting point) is on it.

```
#
# Lenstra's algorithm
#
def lenstra(n,bound_a,bound_b):
    if is_prime(n):
        print n,'is prime'
        return n
    if n%2==0:
        return 2
    if n%3==0:
        return 3

    for a in range(bound_a):
        # Consider only elliptic curves
        if Mod(4*a^3+27,n)==0:
            continue

        f_point = [Mod(0,n),Mod(1,n),1]
        for b in range(bound_b):
            # compute factorial
            f_point = mult_2_l(b+1,f_point,a)
            if f_point[2]==0:
                break
            if f_point[2]>1:
                print a,b
                return gcd( f_point[2], n)
    print 'Increase the values of bound_a and bound_b'
```

The parameters `bound_a` and `bound_b` in the function `lenstra` give the upper bound for a, the number of elliptic curves, and B, the number of multiplications $n!P$. Of course the algorithm is stronger but slower taking large values of `bound_a` and `bound_b`.

For example

```
# therea are twenty-one 3's in the first example
time print lenstra(133333333333333333333,200,100)
time print lenstra( (10^8+7)*(9*10^8+11),200,100)
time print lenstra( 10^20+699,200,100)
time print lenstra( 10^30+427,300,1000)
```

In a standard computer the results have been

```
43 78 -> 4363363
Time: CPU 1.02 s, Wall: 1.04 s
74 30 -> 900000011
Time: CPU 1.63 s, Wall: 1.66 s
156 20 -> 32935987639
Time: CPU 3.70 s, Wall: 3.74 s
223 756 -> 852759062050499
Time: CPU 89.23 s, Wall: 90.27 s
```

The algorithm has its limitations,

```
time print lenstra( next_prime(10^18)*next_prime(10^19),300,3000)
```

produces

```
Increase the values of bound_a and bound_b
None
Time: CPU 421.03 s, Wall: 423.92 s
```

Remark. It is possible to program Lenstra's algorithm using only Sage functions but it requires some deeper knowledge of Python and Sage. Essentially one cheats Sage forcing it to consider $\mathbb{Z}/n\mathbb{Z}$ as a field and one employs the exception handling in Python to redirect the flow after an error.

The following program was written by Professor W. Stein, the lead developer of Sage, and included in his book *Elementary Number Theory: Primes, Congruences, and Secrets.*

Note that it introduces a twist with respect to our previous program, now the elliptic curve is chosen at random.

```
def ecm(N, B=10^3, trials=10):
    m = prod([p^int(math.log(B)/math.log(p))
              for p in prime_range(B+1)])
    R = Integers(N)
    # Make Sage think that R is a field:
    R.is_field = lambda : True
    for _ in range(trials):
        while True:
            a = R.random_element()
            if gcd(4*a.lift()^3 + 27, N) == 1: break
        try:
            m * EllipticCurve([a, 1])([0,1])
        except ZeroDivisionError, msg:
            # msg: "Inverse of <int> does not exist"
            return gcd(Integer(str(msg).split()[2]), N)
    return 1
```

Now

```
time print ecm(133333333333333333333,200,100)
time print ecm( (10^8+7)*(9*10^8+11),200,100)
time print ecm( 10^20+699,200,100)
time print ecm( 10^30+427,300,1000)
```

gives

```
4363363
Time: CPU 0.79 s, Wall: 0.89 s
100000007
Time: CPU 0.90 s, Wall: 1.00 s
3036192541
Time: CPU 0.96 s, Wall: 1.06 s
1
Time: CPU 47.94 s, Wall: 49.03 s
```

The random choice of the elliptic curve can give different results when running the program several times.

Elliptic curve cryptography

The elliptic curve discrete logarithm problem. Recall that the DLP consists in solving $g^x = h$ in $\mathbb{F}_p^*$ for given g and h. The elliptic curve discrete logarithm problem ECDLP is the analogue changing the multiplicative group operation by the group law in the elliptic curve. It consists in finding $x \in \mathbb{Z}$ such that $xG = H$ where G and H are points on a given elliptic curve over a finite field. We say that x is the discrete logarithm of H to the base G.

In Sage it can be solved with `G.dicrete_log(H)`. For instance

```
E = EllipticCurve(GF(103), [1,1])
G = E([0,1])
H = 20*G
print G.discrete_log(H)
```

prints `20`. If we replace `20` by `100` the result is `13` because the order of G is 87. By the way, the latter value is obtained with `additive_order(G)`.

No algorithm is known to compute discrete logarithms in an elliptic curve over $\mathbb{F}_p$ in less than $\sqrt{p}$ steps. This means that using p with a hundred digits (or even much less) is safe. In the previous listing changing `GF(next_prime(103))` by `GF(next_prime(10^20))` could be too much for Sage running in a usual computer.

Of course in applications one looks for G having large order. In Sage the structure of the abelian group of an elliptic curve `E` is given by `E.abelian_group()`. On the other hand, `E.gens()` gives a list with the generators in such a way that the first one has maximal order.

```
E = EllipticCurve(GF(47), [1,1])
print E.abelian_group()
print E.gens()
print 'Element of maximal order =',E.gens()[0]
```

A possible output for this listing is:

```
(Multiplicative Abelian Group isomorphic to C30 x C2, ((44 : 21 : 1),(35 : 0 :
1)))
((44 : 21 : 1), (35 : 0 : 1))
Element of maximal order = (44 : 21 : 1)
```

The format of `E.abelian_group()` can vary from a version of Sage to another. The previous output means that $P = (44, 21)$ and $Q = (35, 0)$ are points of order 30 and 2, respectively and any point on E can be written as $mP + nQ$ with $m, n \in \mathbb{Z}$.

The elliptic curve ElGamal cryptosystem. In principle one can copy the classic ElGamal cryptosystem changing the multiplicative structure of $\mathbb{F}_p^*$ by the group law in an elliptic curve E over $\mathbb{F}_p$ (or a finite field).

A point $G \in E$ of large order and E itself are public information. The private key is an integer k_2 less than the order of G and the public key is $K_1 = k_2 G$. The hardness of ECDLP assures that it is difficult to recover k_2 from K_1.

The set of plaintext messages is the set of points over the given field. The encryption and decryption functions are

$$\begin{aligned} e_{K_1}(M) &= (rG, M + rK_1) \qquad r = \text{ random number} \\ d_{k_2}(C_1, C_2) &= C_2 - k_2 C_1 \end{aligned}$$

A technical problem is how to encode characters into points of an elliptic curve (note that $M \in E$).

There is a variation of the cryptosystem sometimes called MV-ElGamal (MV stands for Menezes and Vanstone) that avoids this technical problem. In this version a message M is divided into two blocks m_1 and m_2 modulo p, i.e. $\mathbb{F}_p \times \mathbb{F}_p$ is the set of plaintext messages (and the encoding is very easy).

The encryption function is given by

$$e_{K_1}(M) = (rG, c_1, c_2) \in E \times \mathbb{F}_p \times \mathbb{F}_p$$

where $c_1 \equiv xm_1 \pmod{p}$, $c_2 \equiv ym_2 \pmod{p}$, with $(x, y) = rK_1$. We assume $x, y \neq 0$, otherwise we choose another random r. The corresponding decryption function is

$$d_{k_2}(C_0, c_1, c_2) = (c_1 x^{-1}, c_2 y^{-1}) \qquad \text{where } (x, y) = k_2 C_0.$$

For instance, if we choose

```
#
# Choose the elliptic curve modulo p = large prime
# and G a point of high order
#
p = next_prime(10^10)
E = EllipticCurve( GF( p ), [2011,1])
G = E([0,1])
print G.additive_order()
```

The output is `3333330247`, then the order G is quite large.

The functions e_{K_1} and d_{k_2} introduced before can be coded as:

```
#
# Encryption and decryption functions
#
def encrypt_mv_eg(Kpub,m1,m2):
    x,y = 0,0
    while( (x==0) or (y==0) ):
        r = floor( p*random() )
        x = (r*Kpub)[0]
        y = (r*Kpub)[1]
    return r*G, m1*x, m2*y

def decrypt_mv_eg(kpri,enc):
    x = (kpri*enc[0])[0]
    y = (kpri*enc[0])[1]
    return enc[1]*x^-1, enc[2]*y^-1
```

If it is a valid cryptosystem then $d_{k_2}\big(e_{K_1}(M)\big) = M$

```
#
# Example
#
private_key = 12345
public_key = private_key*G
decrypt_mv_eg(private_key, encrypt_mv_eg(public_key,10101,33333))
```

We recover the original message `(10101,33333)`.

Recall that we can convert strings of characters into integers thanks to the following simple encoding and decoding functions:

```
# 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):
        number = Integer(number)
        result = ''
        for i in  number.digits(256):
                result = chr(i) + result
        return result
```

Actually in our case we need to divide into an even number of blocks. If we think in a character as a number < 256 (its ASCII code) and we employ $\mathbb{F}_p$ as a field then we can encode at most $\log_{256} p$ characters in each block.

```
#
# TABLE for a long text
# 1st column: Decoded and decrypted text (original message)
# 2nd, 3rd: encoded blocks
# rest: encrypted blocks
#
text = 'This is a long text to be subdivided into blocks'
k = floor( log(p,256) )
key = 12345
for i in range(0, len(text), 2*k):
    m1 = encoding(text[i:i+k])
    m2 = encoding(text[i+k:i+2*k])
    enc = encrypt_mv_eg(key*G,m1,m2)
    d1 = decoding( decrypt_mv_eg(key, enc)[0] )
    d2 = decoding( decrypt_mv_eg(key, enc)[1] )
    print d1+d2, m1,m2, enc
```

```
This is | 1416128883 543781664  | ((6085741895 :  8254518770 :  1), 7312388880, 5371594140)
a long t | 1629514863 1852252276 | ((8649855487 :  1362971917 :  1), 286631972, 6170749646)
ext to b | 1702392864 1953439842 | ((9600714213 :  1592560103 :  1), 7774722895, 1581078717)
e subdiv | 1696625525 1650747766 | ((6309572051 :  9204716993 :  1), 4678543272, 9009446437)
ided int | 1768187236 543780468  | ((5659728365 :  447382763 :  1), 6143475220, 9237109331)
o blocks | 1864393324 1868786547 | ((434505921 :  4258432774 :  1), 8775161069, 8407264212)
```

Part II

Home assignments and quizzes

Deadline: March 1st

Name:

Exercises

1) What is the last (least significant) digit of $201120122013^{201320122011}$?

2) I have bought pens costing 1.01€ each one and pencil sharpeners costing 1.40€ each one. In total I have expended 29.93€. How many pens and pencil sharpeners have I bought?

3) Prove that $a^5 - a$ is divisible by 30 for every a. *Hint*: $30 = 2 \cdot 3 \cdot 5$.

4) Find a 3-digit number leaving remainder 4 when divided by 7, 9 or 11.

5) Compute all the primitive roots modulo 17 and compute the multiplicative order of 2.

Challenges

Challenges are voluntary. If you solve a number of them you can skip the final exam. The solutions must include the arguments that lead to it.

1) I have unwrapped the strip from the home-made scytale that I showed you during the second lecture (there is a photo in my web page). I read in the strip

```
ENOADYAMGIRNTNTCOILEONAIICDEDIESRNDAROTIEUUNNEMACDCM
AIGENNOEMSEAHRACSONCLTAEFECORANCHOHFIETONTTTOTMISIFR
```

What is the secret message? Note: The sentence is unfinished.

2) I have encrypted a message assigning the numbers 0-25 to the letters `A-Z` and using a map of the form $x \mapsto ax \pmod{26}$. Decipher the result

```
LPQIECDUQGNIDSVLEYDAVPS
```

Deadline: April 7th

Name:

Exercises

1) Write the scheme of the baby-step giant-step algorithm to solve $2^x \equiv 15 \pmod{19}$.

2) Suppose that a wise attacker A has invented a machine to solve Diffie-Hellman problem, i.e. A knows an efficiently computable function f such that $f(g^a, g^b) = g^{ab}$. Show that A can break the ElGamal cryptosystem using the public key.

3) Write the computations to get $5^{101} \pmod{127}$ by the repeated squaring method (fast powering algorithm).

4) Suppose you know that a message has been encryted with the ElGamal cryptosystem using a random exponent less than 20. How would you try to cryptanalyze it? Note: We assume that g, p and the public key are public domain.

5) Consider $\mathbb{F}_8$ in the form $\mathbb{F}_2[X]/\langle X^3 + X + 1\rangle$. Check that X is a generator of $\mathbb{F}_8^*$ and write the complete table of logarithms of the elements of $\mathbb{F}_8^*$ to base X.

Challenges

Challenges are voluntary. If you solve a number of them you can skip the final exam. The solutions must include the arguments that lead to it.

1) Read the attached text written by Gauss in 1801 proving the existence of primitive roots modulo p (prime) and re-write it briefly in your own words using modern notation and mathematical symbols. *Allowed languages: English, Italian and Spanish.*

2) Find the secret string of characters `text` knowing that the output of the following lines

```
for c in text:
    k = floor( 1000*random() )
    print Mod(17,911)^k, ord(c)*Mod(123,911)^k
```

has been:

```
4 712 604 166 196 535 46 401 394 211 563 141 410 854 319 859 797 735 126 630 412 574
105 849 747 291 808 610 305 549 459 151 535 867 896 364
```

Deadline: May 5th

Name:

Exercises

1) Solve the equation $x^{2011} \equiv 1234 \pmod{1625}$. Note: It is not admitted to try all classes modulo 1625 with the computer.

2) Consider a RSA cryptosystem with an encryption key $k \neq \pm 1 \pmod{n}$, $n = pq$. Can the encrypting and the decrypting function coincide, i.e. $e_k(e_k(m)) = m$, $\forall m \in \mathcal{M}$? In the affirmative provide and example and in the negative provide a proof.

3) Find all bases for which 15 is a pseudoprime to the base a.

4) Given $n = p_1 p_2 \cdots p_r$ with p_i distinct primes, $r > 1$, prove that if n is a Carmichael number then $p_i - 1$ divides $n - 1$ for every $1 \leq i \leq r$. Hint: Use primitive roots modulo p_i. Note: Recall that a Carmichael number n is a pseudoprime to every base coprime to n.

5) Describe the calculations to decide if $n = 1 + 367 \cdot 10^3$ is prime using Pocklington primality test.

Challenges

Challenges are voluntary. If you solve a number of them you can skip the final exam. The solutions must include the arguments that lead to it.

1) In class we made the `fermat_fact(n)` function returning a factor of n using Fermat factorization. Employ `fermat_fact(n)` and `is_prime(n)` to define the functions[1]:

a) (80%) `fermat_factor_a(n)` printing a (non-necessarily ordered) list of the prime factors of `n` repeated according multiplicities. For instance, for `n=630` it could give `2,3,7,3,5`.

b) (+20%) `fermat_factor_b(n)` printing the factorization of `n` in the usual (ordered) way as in the function `factor` in Sage. For instance, for `n=630` it has to give `2 * 3^2 * 5 * 7`.

2) In a library the PIN is a string, say `pin`, of three characters encrypted in the magnetic stripe as a number given by the formula

```
Mod( 256^2*ord(pin[0]) + 256*ord(pin[1]) + ord(pin[2]) , 18121121)^7919
```

What is the PIN corresponding to `16479305`?

[1]Please send the answer to this challenge by email to `fernando.chamizo@uam.es` in a text file or in a Sage worksheet. The part b) requires some knowledge of Python. There is a Python cheat sheet in my web site.

Deadline: May 31th

Name:

Exercises

1) Write the complete addition table of $E:\ y^2 = x^3 + x + 3$ over $\mathbb{F}_5$ without using the computer.

2) Suppose that in elliptic Diffie-Hellman key exchange with $E:\ y^2 = x^3 + 1$ over $\mathbb{F}_5$ and $G = (2, 3)$ both parties send $(0, 1)$. What is the shared key?

3) We impose in the definition of elliptic curve the condition $4a^3 + 27b^2 \neq 0$. Consider for instance $E:\ y^2 = x^3 + 2x + 2$ over $\mathbb{F}_5$ that does not fulfill this condition and show that it does not give a coherent group law.

4) Consider an elliptic curve $E:\ y = x^3 + ax + b$ over $\mathbb{Q}$ at let n_2 the number of points of order exactly 2. Prove that $n_2 \neq 2$.

5) An elliptic curve E over a finite field K contains 1089 points (including the point at infinity), $E(K) = \{P_0 = O, P_1, P_2, \ldots, P_{1088}\}$. Suppose that $P_n = nP_1$ for $1 \leq n < 1088$. How many elements are there of each order?

Challenges

Challenges are voluntary. If you solve a number of them you can skip the final exam. The solutions must include the arguments that lead to it.

1) Write the calculations to get a nontrivial factor of 4221089 using $E:\ y^2 = x^3 + x + 7$ and the starting point $P = (1, 3) \in E$. Note: The program typed in class was specialized for $y^2 = x^3 + ax + 1$ and $P = (0, 1)$ but you can still use the function for adding points.

2) Guess the secret message | L_1 | L_2 | L_3 | | L_4 | L_5 | L_6 | where L_i is a letter with `ord(`L_i`)=Ai` knowing that the output of the program

```
E = EllipticCurve(GF(6091541), [0,5622139])
G = E([3353686,4066380])
Ppub = E([5894715,2653441])
k = floor( 10^6*random() )
print k*G, E([256^2*A1+256*A2+A3,256^2*A4+256*A5+A6]) + k*Ppub
```

has been:

```
(3452962 :  2418876 :  1) (1041155 :  5388088 :  1)
```

Quizzes

Quiz 1. We get

```
UXUSTYXVUFFUQCJX
```

using the affine cipher $x \mapsto 5x + 6$. What is the original message?

Hint: The inverse of $n \pmod m$ is `n.inverse_mod(m)`

Quiz 2. Consider ElGamal cryptosystem with $p = 2^{31} - 1$ and $g = 7$. If the public key is 1659750829 and the ciphertext is $(297629860, 1094924871)$. What is the plaintext message?

Quiz 3. With the help of the computer, find all the common pseudoprimes to the bases 2 and 3 less than 50000.

Quiz 4. Given the elliptic curve

$$E: \ y^2 = x^3 + x + 1 \qquad \text{over } \mathbb{F}_{13}$$

compute $P + Q$ for $P = (1, 4)$ and $Q = (4, 2)$.

Part III

Some programs

Selected programs

Affine Cipher

```
def encrypt(message, key1,key2):
    alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    encrypted = ''
    for c in message:
        loc = alph.find(c)
        encrypted += alph[Mod(key1*loc+key2,26)]
    print message,'->',encrypted


# ENCRYPT MESSAGES
encrypt('MYSECRET', 3,0)
encrypt('KUCMGZMF', 3.inverse_mod(26),0)
```

Hill Cipher

```
def encrypt_hill(message, key11, key12, key21, key22, key1, key2):
    alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

    encrypted = ''
    for i in range(0,len(message),2):
        digraph = message[i:i+2]
        encrypted += alph[Mod(key11*alph.find(digraph[0])
                    +key12*alph.find(digraph[1]) + key1,26)]
        encrypted += alph[Mod(key21*alph.find(digraph[0])
                    +key22*alph.find(digraph[1]) + key2,26)]
    print message,'->',encrypted


# encrypt using the matrix A=[3, 5; 7, 2] and b=[1,1]
encrypt_hill('SECRET', 3,5,7,2, 1,1)

# decrypt using its inverse A^-1=[8, 19; 11, 25]
# and -A^-1b=[25,16]  (26)
encrypt_hill('XFOXEP', 8,19,11,25, 25,16)
```

Brute force for DLP

```
# Discrete logarithm for g^x=h (p) by brute force

def br_fr(h,g,p):
    for x in range(p-1):
        if Mod(g,p)^x == Mod(h,p):
            print 'x= ',x
            break
```

Baby-step giant-step for DLP

```
# Discrete logarithm for g^x=h (p) by baby-step giant-step
```

```
def baby_giant(h,g,p):
    baby = [1]
    giant = [h]
    n = 1+floor(sqrt(p-1))

    for i in range(1,n):
        baby.append( Mod(baby[i-1]*g,p) )

    g = Mod(g,p)^-n
    for j in range(1,n):
        giant.append( Mod(giant[j-1]*g,p) )

    for inters in set(baby).intersection( set(giant) ):
#         print 'i =', baby.index(inters)
#         print 'j =', giant.index(inters)
        print 'x =',  baby.index(inters)+n*giant.index(inters)
```

ElGamal cryptosystem: encryption

```
# ElGamal
# pub_key = public key
# g = generator or high order element
# p = prime
# message = number < p
def elgamal_encrypt(pub_key,g,p,message):
    k = floor( 1+(p-2)*random())
    return (Mod(g,p)^k, message*Mod(pub_key^k,p) )
```

ElGamal cryptosystem: decryption

```
# ElGamal
# pri_key = private key
# g = generator or high order element (not needed)
# p = prime
# (m_1,m2) = couple of numbers < p
def elgamal_decrypt(pri_key,g,p,(m1,m2)):
    return Mod(m2,p)*Mod(m1,p)^(-pri_key)
```

Encoding Schemes: Number to text

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

Encoding Schemes: Text to number

```
# text to number
```

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

ElGamal cryptosystem in finite fields

```
# 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

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
#########################
# LIMIT 20 CHARACTERS
#########################
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)   ) )
```

RSA including key generation, encryption and decryption

```
# 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


#
# 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)

#
# 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
#
# 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
```

Miller-Rabin primality test

```
#
# Miller-Rabin
#

def q_and_k(n):
     q = n-1
     k = 0
     while Mod(q,2)==0:
         k += 1
         q = q/2
     return q,k

def strong_pseud(n,a):
    (q,k) = q_and_k(n)
    b = Mod(a,n)^q
    if b==1:
        return True

    for i in range(k):
        if b==-1:
            return True
        b = Mod(b,n)^2
    return False

def miller_rabin(n,secur):
    for p in primes_first_n(secur):
        # Small primes
        if Mod(n,p)==0:
            if n==p:
                return 'Prime'
            return 'Composite'
        # Check strong pseudoprimes
        if strong_pseud(n,p)==False:
            return 'Composite'
    return 'likely prime'

def miller_rabin2(n,secur):
    a = 0
    for i in range(secur):
        a = 2+ZZ.random_element( n-3 )
        if Mod(n,a)==0:
            return 'Composite'
        if strong_pseud(n,a)==False:
            return 'Composite'
    return 'likely prime'
```

Fermat factorization algorithm

```
def fermat_factor(n):
    if Mod(n,2)==0:
        return 2
    if is_prime(n):
        return n
    y = 0
    while( is_square(n+y^2)==False ):
        y += 1
    return sqrt(n+y^2)-y
```

Pollard p factorization algorithm (automatic)

```
def pollard_p_auto2(n):
    aa =2
    a = aa
    i = 0
    d = n
    if is_prime(n):
        return d
    while (d==1) or (d==n):
        i += 1
        a = Mod(a,n)^i
        d = gcd (a-1,n)
        if a == 1:
            aa += 1
            a = aa
            i = 0
    return d
```

Plotting the group law for an elliptic curve

```
var('x,y')
graph = implicit_plot(x^3-6*x+5-y^2, (x, -5,5),  (y, -5,5) )
graph += plot( x-1, x, -3,4)
graph += plot( x-1, x, -2,2, thickness=2)
graph += point([1,0],size=30) + point([2,1],size=30)
graph += point([-2,-3],size=30) + point([-2,3],size=30)
graph += line([(-2,3),(-2,-3)], linestyle = '--', thickness=2)
graph += text("P",(2.3,0.4),fontsize=20)
graph += text("Q",(1.3,-0.7),fontsize=20)
graph += text("R",(-2,-3.8),fontsize=20)
graph += text("P+Q",(-2.4,4),fontsize=20)

show(graph)
```

Group law in an elliptic curve

```
#
# Group law in the elliptic curve y^2= x^3+a*x+b
# ('a' has to be defined in advance)
#
def g_l( P, Q ):
```

```
    if P == 'O':
        return Q
    if Q == 'O':
        return P

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return 'O'

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        m = (3*P[0]^2+a)/2/P[1]
    else:
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1]]
```

Fast multiplication by a number in an elliptic curve

```
def mult_2(n,P):
    result = 'O'
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l( pow_2P, result )
        n //=2
        pow_2P = g_l( pow_2P, pow_2P )
    return result
```

Lenstra's factorization algorithm

```
#
# "Normal" points [x,y,1]
# Point at infinity [0,1,0]
# Fake points [0,0,d] with d not coprime to the modulus.
#
#
# Group law
#
def g_l_l( P, Q, a ):
    if P[2] != 1:
        if P[1]==1:
            return Q
        return P
    if Q[2] != 1:
        if Q[1]==1:
            return P
        return Q

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return [0,1,0]

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        if P[1].is_unit()==False:
```

```
            return [0,0,P[1]]
        m = (3*P[0]^2+a)/2/P[1]
    else:
        if (Q[0]-P[0]).is_unit()==False:
            return [0,0,Q[0]-P[0]]
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1],1]

#
# Same multiplication routine
# changing 0 and g_l by g_l_l
#
def mult_2_l(n,P,a):
    result = [0,1,0]
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l_l( pow_2P, result, a )
        n //=2
        pow_2P = g_l_l( pow_2P, pow_2P, a )
    return result

#
# Lenstra's algorithm
#
def lenstra(n,bound_a,bound_b):
    if is_prime(n):
        print n,'is prime'
        return n
    if n%2==0:
        return 2
    if n%3==0:
        return 3

    for a in range(bound_a):
        # Consider only elliptic curves
        if Mod(4*a^3+27,n)==0:
            continue

        f_point = [Mod(0,n),Mod(1,n),1]
        for b in range(bound_b):
            # compute factorial
            f_point = mult_2_l(b+1,f_point,a)
            if f_point[2]==0:
                break
            if f_point[2]>1:
                print a,b
                return gcd( f_point[2], n)
    print 'Increase the values of bound_a and bound_b'
```

Lenstra's factorization algorithm (extra compact version by W. Stein)

```
# Taken from
# "Elementary Number Theory: Primes, Congruences, and Secrets"
# by W. Stein
#
def ecm(N, B=10^3, trials=10):
    m = prod([p^int(math.log(B)/math.log(p))
              for p in prime_range(B+1)])
    R = Integers(N)
    # Make Sage think that R is a field:
    R.is_field = lambda : True
    for _ in range(trials):
        while True:
            a = R.random_element()
            if gcd(4*a.lift()^3 + 27, N) == 1: break
        try:
            m * EllipticCurve([a, 1])([0,1])
        except ZeroDivisionError, msg:
            # msg: "Inverse of <int> does not exist"
            return gcd(Integer(str(msg).split()[2]), N)
    return 1
```

Elliptic DLP

```
#
# Computing a discrete logarithm
#
E = EllipticCurve(GF(103), [1,1])
G = E([0,1])
H = 20*G
print G.discrete_log(H)


#
# Computing an element of maximal order
#
E = EllipticCurve(GF(47), [1,1])
print E.abelian_group()
print E.gens()
print 'Element of maximal order =',E.gens()[0]
```

Elliptic ElGamal cryptosystem

```
#
# Encryption and decryption functions
#

def encrypt_mv_eg(Kpub,m1,m2):
    x,y = 0,0
    while( (x==0) or (y==0) ):
        r = floor( p*random() )
        x = (r*Kpub)[0]
        y = (r*Kpub)[1]
```

```
    return r*G, m1*x, m2*y

def decrypt_mv_eg(kpri,enc):
    x = (kpri*enc[0])[0]
    y = (kpri*enc[0])[1]
    return enc[1]*x^-1, enc[2]*y^-1

# 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):
        number = Integer(number)
        result = ''
        for i in  number.digits(256):
                result = chr(i) + result
        return result


#
# Choose the elliptic curve modulo p = large prime
# and G a point of high order
#
p = next_prime(10^10)
E = EllipticCurve( GF( p ), [2011,1])
G = E([0,1])

#
# TABLE for a long text
# 1st column: Decoded and decrypted text (original message)
# 2nd, 3rd: encoded blocks
# rest: encrypted blocks
#
text = 'This is a long text to be subdivided into blocks'
k = floor( log(p,256) )
key = 12345
for i in range(0, len(text), 2*k):
    m1 = encoding(text[i:i+k])
    m2 = encoding(text[i+k:i+2*k])
    enc = encrypt_mv_eg(key*G,m1,m2)
    d1 = decoding( decrypt_mv_eg(key, enc)[0] )
    d2 = decoding( decrypt_mv_eg(key, enc)[1] )
    print d1+d2, m1,m2, enc
```

Digital signatures

```
def sign(D,g,p,s):
    r = 0
    while( gcd(r, p-1)!=1 ):
```

```
        r = floor(p*random())
    return D, g^r, (Mod(D,p-1)-Mod(s,p-1)*Mod(g^r,p-1))*Mod(r,p-1)^-1

def verif_sign(D,g,p,s1,s2,v):
    if (g^D == v^Mod(s1,p-1)*Mod(s1,p)^Mod(s2,p-1) ):
        return True
    return False


#
# ElGamal signature scheme
#
p = 2011
g = Mod(primitive_root(p), p)


D = 777
# signature key
s = 123
# verification key
v = g^s

# Computes the signature
print sign(D,g,p,s)

# Verifies the signature
print verif_sign(D,g,p,210, 1071,v)
```

Secret sharing

```
def shares(m,pie,thr):
    R = PolynomialRing(GF(p), 'x')
    Pol = R.random_element(thr-1)
    Pol = Pol - Pol(x=0)+m
    for i in range(pie):
        print (i+1, Pol(x=i+1))

def recover( L ):
    R = PolynomialRing(GF(p), 'x')
    for i in range( len(L) ):
        L[i] = ( (Mod(L[i][0],p), Mod(L[i][1],p) ) )
    Pol = R.lagrange_polynomial(L)
    return Pol(x=0)

#
# Secret sharing
#
p = next_prime(10^10)

m = 123454321

pieces = 7
threshold = 3
```

```
shares(m,pieces,threshold)
print 'Secret =',recover( [ (1, 8595447293),(6, 1032447548), \
(3, 5554840297), (2, 405909266) ] )
```

Appendix: Using the computer

Basic Linux commands by example

This is a small subset of the standard Linux commands. Most of them are more powerful than shown in the following examples. A deeper discussion (but still readable and simple) can be found in [1] and in several web sites like [2]. More technical and complete information is available through the `man` command (e.g., `man ls`).

`cat myfile.txt`	Show the content of `myfile.txt`
`cd mydirectory`	Change the current directory to `mydirectory`
`chmod a+x myprogram`	Turn `myprogram` into an executable file
`cp myfile otherfile`	Copy `myfile` to `otherfile`. Use the option `-r` for folders
`chmod u+rw myfile`	Give permission to the user to read and write in `myfile`
`diff myfile1 myfile2`	Compare `myfile1` and `myfile2`
`killall proname`	Kill the processes named `proname`
`ls`	Show the directory of files and folders. Use the option `-l` for more information and `-a` for seeing the hidden files. The color gives indications about the type of the files but it is not always reliable
`man commandname`	Show the information about `commandname`. In some cases it also works with applications or non-standard Linux commands
`mkdir myfolder`	Create the folder `myfolder`
`more myfile.txt`	Show the content of `myfile.txt` page by page
`mv myfile otherfile`	Move (rename) `myfile` to `otherfile`
`ps -A`	Show the processes and their status. Use the option `-aux` for a more complete list
`rm myfile`	Delete the file `myfile`. This operation cannot be undone. Use the graphic interface for moving a file to the Trash. Use the option `-r` for folders
`wc myfile.txt`	Count bytes, words and lines of `myfile.txt`

Some commands to launch applications:

`firefox` Web navigator (`konqueror` is also a a web browser and file manager under KDE)

`gcc` C compiler (use `gcc -o outputfile inputfile.c -lm` here `-lm` links the Math library)

`kwrite` Basic KDE text editor (if not available `kate` is fairly similar)

`python` Python interpreter

`sage` SAGE mathematical package (use `-notebook` for the web interface)

References

[1] P. Sanz Mercado, A. Luna Fernández. Principios y administración de Linux. Documentos de trabajo 83. Ediciones de la Universidad Autónoma de Madrid 2009.

[2] An A-Z index of the Bash command line for Linux. `http://ss64.com/bash/`.

[3] Linux cheat sheet. `http://wiki.typo3.org/Linux_cheat_sheet`.

The basics of compiling and running

The classic book [2] popularized the use of the `Hello World!` program. Its only purpose is to learn how to compose, compile an run a simple program writing on the screen the homonymous message.

In general, compilation and execution are not platform independent. We restrict ourselves to Linux systems (or, more properly, Unix-like). There are also a lot of IDEs (Integrated Development Environments) to ease the management of large projects. We only consider console commands.

See [1] and [3] for a more complete list.

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

`C: Hello World` Type a file called `helloworld.c` with your favorite text editor containing the following lines:

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

To compile the program type in a console terminal (opened in the same path as the file)

```
gcc helloworld.c -o helloworld.out -lm
```

The options after the name of the file are not mandatory. The first one `-o helloworld.out` indicates the name of the resulting executable file. If it is not specified the output file by default is `a.out`. On the other hand, the option `-lm` lnks the `math` library. It is not necessary in our case but it is required to use mathematical functions like sine or exponential. Running the program reduces to type in the console

```
./helloworld.out
```

`C++: Hello World` In some sense C++ programming language is an extension and improvement of C largely caompatible with it. The compile and run process parallels the steps described above. Now the analog file named `helloworld.cpp`

```
#include <iostream>
int main(){
    std::cout << "Hello World!\n";
    return 0;
}
```

The prefix `std::` is cumbersome. The using directive inserted as `using namespace std;` after the first line allows to replace `std::cout` by `cout` and the same for the rest of functions in the library `iostream`. This gives

```
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World!\n";
    return 0;
}
```

To compile the program use

```
g++ helloworld.cpp -o helloworld.out -lm
```

and to run it, proceed as before.

`Java: Hello World` Type the following lines in a text file named `helloworld.java`

```
public class helloworld{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

The command to compile is

```
javac helloworld.java
```

It generates `helloworld.class`, a program to be interpreted and executed by the so-called Java Virtual Machine (abbreviated as JVM) that is launched by

```
java helloworld
```

This is a standalone (an offline) Java program. There are also Java applets, i.e., Java programs embedded in web pages.

`Java applet: Hello World` Let us say that we name `helloworldapplet.java` to the following lines

```
import java.awt.Graphics;
public class helloworldapplet extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 40, 20);
```

```
    }
}
```

We compile the code as before to get `helloworld.class`. But if we try to run it the JVM will complain saying that there is no `main` in the program. We need a web page (in the same path) acting as a launcher including

```
<applet code="helloworldapplet.class"></applet>
```

at some point of its HTML code. In fact it is convenient to include the atributes `height` and `width` overall if (as in our case) the size of the canvas is not fixed in the applet. The full code of a web page `applet.html` containing the previous applet could be

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head>
  <title>An applet</title>
</head><body>
This is the applet:
<applet code="helloworldapplet.class" height="25" width="200"></applet>
</body></html>
```

Naturally `applet.html` will be opened by a java enabled web navigator. For debugging purposes the command

```
appletviewer applet.html
```

allows to visualize the result.

`Maple: Hello World` First of all we need Maple running. There are two modes: the console mode and the graphic mode, usually corresponding to the commands `maple` and `xmaple`, respectively. In the second case we select "Start with Blank Worksheet" in the popup window showing the startup menu (at least in Maple 10) to start from scratch.

In both modes one can type a program line by line, indeed this is the easier in our case, but in general it becomes tedious, overall in the graphic mode (the one that allows to plot functions) whose WYSIWYG (What You See Is What You Get) interface takes too many decisions by itself. For programs exceeding a simple calculation, the best option in both modes is to type in a separate text file. In our case `helloworld.txt` with a single line:

```
printf("\tHello World!\n");
```

To load and run this program, type beside the prompt (`>`) in Maple

```
read "helloworld.txt";
```

The full path is needed if Maple was opened in a different location. When using the graphics mode the interface will try to close the double quotes by itself and the chances of typing by mistake a third pair increase.

This Hello World! program is not the simplest, just `"Hello world!";` in the command line or a separate file does the job. The based on a C `printf` command allows to control format. In our case `\t` refers to the tab character that indents the message with respect to the left margin.

`Matlab/Octave: Hello World` As in Maple, one could type a program line by line in the Matlab or Octave interfaces but reasonably involved programs require separate files. In our case we type in `helloworld.m`

```
fprintf('Hello world!\n');
```

The command

```
helloworld
```

(note the absence of the extension) typed in Matlab or Octave loads and runs the program.

`Python: Hello World` In a text file `helloworld.py` we type

```
# -*- coding: iso-8859-15 -*-
print 'Hello World!\n'
```

The first line is optional and indicates the codification that the programmer is going to use. To run it simply type in a console

```
python helloworld.py
```

Another way of running a Python program is typing `python` in a terminal and after the prompt

```
import helloworld
```

This load the functions defined in the file (in our case there are none) and run the program. This is interesting to add formerly defined functions to our project (reusing is part of Python philosophy). If we try to `import` again a file the program will not be executed because the functions are already supposed to be loaded, then it is not the usual way of running a program.

In Unix-like systems one can convert a Python program in a directly executable command adding the so-called shebang line (according to the Thesaurus dictionary this means an entire system; used in the phrase 'the whole shebang'). It is a heading line indicating how to run Python. In our case the file turns into

```
#!/usr/bin/env python
# -*- coding: iso-8859-15 -*-
print 'Hello World!\n'
```

The shebang line indicates that Python interpreter should be searched at some place in `/usr/bin`. More specific valid paths like `/usr/bin/python` or `/usr/local/bin/python` could be (more) platform dependent. Modifying the file attributes of the new `helloworld.py`, if necessary, to obtain an executable file (`chmod a+x helloworld.py`) it can be run in console with

```
./helloworld.py
```

`Sage: Hello World` Sage is Python-based then after launching Sage (typically with the console command `sage`) one can type simple Python programs (like `print 'Hello World!\n'`). Following the previous philosophy, we explain how to load a program in a file that in this case we name `helloworld.sage` and contains

```
print 'Hello World!\n'
```

After Sage prompt (`sage:`) to run the program it is enough to write

```
attach "helloworld.sage"
```

It is also valid the more naturally named sentence

```
load "helloworld.sage"
```

This works in part like a kind of analog of `import`. It reads the functions only once (but runs always). Then attach is more convenient when debugging code.

A broadly used way of typing, presenting and running Sage programs is through a web based interface. To access to it, launch Sage with `sage -notebook` or type `notebook()` after the Sage's prompt. A new tab will open in the web browser by default. After logging in (if necessary), select New Worksheet and a name. Click on the box (it will become blue) and type the program. To run it use `evaluate` option or press Ctrl+enter. There is a menu with different options to save

Like in Python it is possible to turn a Sage program in a script directly runnable in console inserting a shebang line:

```
#!/usr/local/sage-4.5.1/sage

print 'Hello World!\n'
```

that is executed (after changing the attributes if necessary) with

```
./helloworld_2.sage
```

This creates automatically a Python file with the same name and `.py` extension.

There are several other forms of running Sage code through Python code and viceversa or compiling Sage programs to improve performance even loading external C functions. They are detailed in Chapter 5 of [4].

References

[1] HelloWiki! http://hellowiki.org/

[2] B.W. Kernighan, D.M. Ritchie. The C programming language, Prentice Hall, 1978.

[3] LiteratePrograms http://en.literateprograms.org/Hello_World

[4] Sage Tutorial Release 4.4.4 (by The Sage Development Team), http://www.sagemath.com/pdf/SageTutorial.pdf June 24, 2010.

A Crash Course in Basic Python Commands with Sage

Comments

The symbol `#` at the beginning of a line means that it is a line. The triple double quotes `"""` are reserved for comments along several lines. If you know C/C++, they are equivalent to `//` and `/*...*/`.

```
# Elements of Python Programming Language
"""
more comments
more lines
"""
```

Of course, do not expect any output from this program.

The command `print` complete it with the customary salute.

```
# Elements of Python Programming Language
"""
more comments
more lines
"""
print 'Hello World!'
```

Types

According to the manuals, Python is strongly-typed. If you know what this jargon means, the following program will probably amaze you:

```
############
#TYPES
############
a = 2
print 3*a
a = 'Hi'
print 3*a
```

The output is

```
6
HiHiHi
```

Inserting `print type(a)` after lines 5 and 7, you get `<type 'sage.rings.integer.Integer'>` in the first case and `<type 'str'>` in the second. The first variable is an integer and the second a string

Data structures

The basic data structures built-in in python are lists, tuples and dictionaries.

Lists. A list is in principle a one-dimensional array like in many other programming languages. It is really a list of elements separated by commas and between brackets `[...,...,...]`. Its elements are called inicated its order between brackects, for instance `L[0]` means the first element of the list, `L[1]` is the second element of the list and so on. Like in C/C++ the computers starts counting from zero. The main difference with other languages is that the elements of a list can be objects of different nature (type). For instance you can combine numbers, string and even lists.

The lists also admit slicing like for instance in Matlab or Octave and `L[n:m]` means the elements from `L[n]` to `L[m-1]`. If n or m (or both) are omitted the starting or the finishing point are the beginning or the end of the list. Negative values are identified with

All of these consideration should be clear with the following code

```
##################
#DATA STRUCTURES
##################
# lists
L = [1,2,'three',4,'five']
print L
print L[3]
print L[2:4] # L[2] included, L[4] not included
print L[2:] # From L[2]
print L[:-1] # To the last but one element
```

giving the output

```
[1, 2, 'three', 4, 'five']
4
['three', 4]
['three', 4, 'five']
[1, 2, 'three', 4]
```

Specially important lists are finite arithmetic progressions. They are generated with `range`(n, m, s) where n is the starting point, s the step and m a strict upper limit. If the step is omitted it is assumed $s = 1$. If besides n is omitted it is assumed $n = 0$. For example

```
# special lists
L = range(20) # from 0 to 20 (not included)
print L
L = range(5,20,2) # Same thing starting from 5 and step=2
print L
L = range(5,13) # Integers in [5,13)
print L
```

gives

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[5, 7, 9, 11, 13, 15, 17, 19]
[5, 6, 7, 8, 9, 10, 11, 12]
```

Tuples. Tuples syntax is similar to list syntax changing brackets by parenthesis in the definition. Tuples are immutable lists. This means that they behave in the same way but you cannot redefine their elements. The following program leads to an error uncommenting the last but one line.

```
# tuples vs lists
L = [1,2,'three',4,'five']
print L[2:4]
L[2] = 'trois'
print L
T = (1,2,'three',4,'five')
print T[2:4]
#T[2] = 'trois'
print T
```

Strings and tuples are alike.

```
salute = "how are you doing?"
print salute[2:6]
print salute[:-1]
salute[2]='z'
```

gives the expected answer for the first lines and the expected error message for the last

```
w ar
how are you doing
---------------------------------------------------------------------------
                  [omitted error messages]
TypeError: 'str' object does not support item assignment
```

Dictionaries. Dictionaries are hash tables. A list of names and their meaning. The syntax is a list of `name: meaning` between curly brackets.

```
# dictionaries
D = {'one': 1, 'two': 'deux', 'e': '2.718...', 'three': 'trois'}
print D
print 'Note that there is not a determined order in a dictionary '
print D['two']
print D['e']
print 7*D['one']
```

```
{'three': 'trois', 'e': '2.718...', 'two': 'deux', 'one': 1}
Note that there is not a determined order in a dictionary
deux
2.718...
7
```

Flow control statements

The flow control statements in Python are `for`, `if` (`if-else`) and `while`. The most original convention is that block indentation is mandatory but easy. When you mark the end of a flow control statement when a colon `:` the editor automatically force you to indent.

In flow control statements the keyword `in` appears very often to indicate elements in a list (or tuple or string).

Let us start with a very easy example

```
##################
#FLOW CONTROL
##################
for i in range(10):
    print i,
print
for i in range(3):
    print i,'->',i^2
```

that allows us also to learn that a comma `,` after print avoid a change of line

```
0 1 2 3 4 5 6 7 8 9
0 -> 0
1 -> 1
2 -> 4
```

The `if` statement and `if-else` blocks work like in other languages

```
friends = ['John','Bob','George','Henry','Alice']
print friends
for name in friends:
    print name,
    if name == 'George':
        print '(My best friend) ,',
    elif name == 'Alice':
        print ''
    else:
        print ', ',
```

The first `if` detects our best friend and the second, disguised as `elif` (else if), omits the last comma.

```
['John', 'Bob', 'George', 'Henry', 'Alice']
John ,  Bob ,  George (My best friend) , Henry ,  Alice
```

Finally, we illustrate `while` statement with program comparing pythonic and non-pythonic (despective term applied when one tries to write Python code using other languages philosophy)

```
sentence = 'The last example'
# Non Pythonic
i=0
while i <len(sentence):
```

```
    print sentence[i],
    i+=1 # abbreviation of i=i+1
print ''
# Pythonic
for i in sentence:
        print i,
print ''
```

The output is the same in both cases. Note that (like in C/C++) the abbreviation `i+=k` is available (but `++i` is not). The command `len` indicates the length of the string.

```
T h e   l a s t   e x a m p l e
T h e   l a s t   e x a m p l e
```