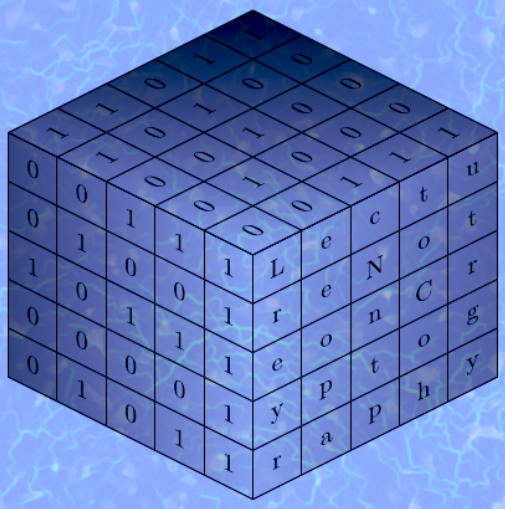


Lecture Note on Cryptography

Author: Szabolcs Tengely

Institute: University of Debrecen

Date: February 9, 2020

Contents

Chapter 1 Introduction

These lecture notes are based on class material offered at University of Debrecen, Hungary. The aim of the lecture note is to help the students to understand and to learn the course material. It provides detailed computations to see all the mathematical objects appearing. The Cryptography class has certain prerequirements that is assumed, Linear Algebra I-II, Algebra, Number Theory. In the computations the computer algebra software package SageMath [23] is used, here also some basic knowledge is assumed. The nice tutorial provided by Michael O'Sullivan is a suggested source to start with: https://mosullivan.sdsu.edu/Teaching/sdsu-sage-tutorial/index.html. Especially the two sections Programming in SageMath and Mathematical Structures may be useful to be able to follow the implementations given in this lecture note.

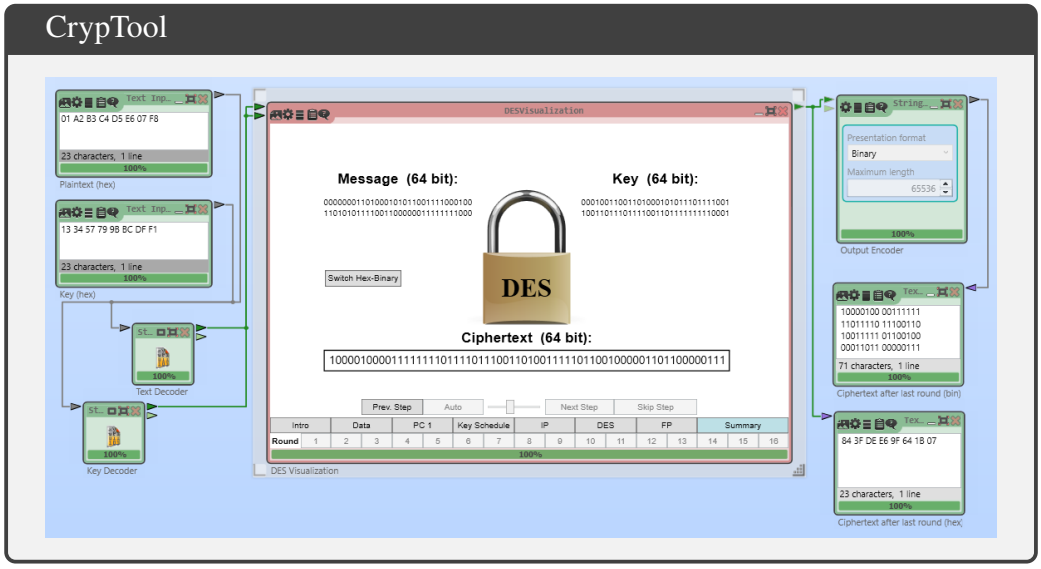
- In Chapter 2 we deal with some simple classical cryptosystems like Caesar's cipher and its generalized versions, Hill ciphers based on matrices and Vigenère ciphers.
- In Chapter 3 the well-known RSA cryptosystem is introduced, a nice number theory based system. Since factorization provides an obvious attack against RSA we present a few algorithms in this direction.
- In Chapter 4 other two factorization based cryptosystems are described, the Rabin cryptosystem and Paillier's cryptosystem.
- In Chapter 5 we deal with the applications of the discrete logarithm problem in cryptography. The Diffie-Hellman key exchange and ElGamal cryptosystem are studied here.
- In Chapter 6 we consider some well-known techniques to handle the discrete logarithm problem such as the Baby-Step Giant-Step algorithm, Pollard's ρ algorithm and index calculus in case of additive and multiplicative groups as well.
- In Chapter 7 we show that difficult mathematical problems may yield cryptosystems that are not secure. As an example we study certain matrix groups and the discrete logarithm problem related to some special subgroups generated by two matrices. It turns out that there exists efficient algorithm to solve the discrete logarithm problem in these cases.
- In Chapter 8 a newer cryptosystem is introduced, the NTRU that works on some special truncated polynomial rings. There are now many variants of the original system. Here we also consider the ITRU system and we show that it can be

efficiently attacked by frequency analysis technique.

- In Chapter 9 an interesting application in cryptography is discussed, namely secret sharing. We present the so-called Shamir's secret sharing.
- In Chapter 10 we consider a knapsack problem based cryptosystem the so-called Merkle-Hellman system. It uses certain superincreasing sequences to make decryption easy, however as we will see there is an efficient attack based on the famous LLL-algorithm.
- In Chapter 11 we provide solutions of the exercises appearing in the lecture note, these are based on the software package SageMath.

Additional sources suggested to use related to the course:

- CrypTool : `https://www.cryptool.org/en/`, very nice educational software in the area of cryptography and cryptoanalysis.



- Mysteri Twister C3 : `https://www.mysterytwisterc3.org/en/`, this is a cipher contest, here one can find all kind of crypto challenges. As an example we may try to solve one of the RSA Factoring Challenges, like RSA-210:

$$N = 245246644900278211976517663573088018467026787678332759743414451715061600830038587216952208399320715491036268271916798640797767232430056005920356312465612184658179041001318592996199338170121493350348758705510 67.$$

Chapter 2 Classical ciphers

Summary

- Shift ciphers
- Affine ciphers
- Hill ciphers
- Substitution ciphers
- Vigenère ciphers
- ADFGX and ADFGVX ciphers
- Playfair cipher
- SageMath summary
- Exercises

2.1 Shift ciphers

In a shift cipher we replace each letter in the message by a letter that is some fixed number of positions further along in the alphabet. Julius Caesar used a special shift cipher to communicate important military information to his military commanders. The length of the shift we are using is called the encryption key. In case of the Caesar cipher it is 3. If we use a 26 letter alphabet, then the encryption can be formulated as follows

$$c_i \equiv E(p_i) \equiv p_i + k \pmod{26},$$

where $p_1, p_2, \ldots$ denotes the sequence of letters in the plaintext and $c_1, c_2, \ldots$ is the sequence of letters in the ciphertext, the k stands for the encryption key. Given a ciphertext character c_i the corresponding plaintext character can be determined by the formula

$$p_i \equiv D(c_i) \equiv c_i - k \pmod{26}.$$

We will use a quote from Adi Shamir as a plaintext:

"Fully secure systems do not exist today and they will not exist in the future."

ShiftCipher

```
S = ShiftCryptosystem(AlphabeticStrings())
print(S)
P = S.encoding("Fully secure systems do not exist today and they will not exist in the future.")
```

```
print("Representation of the plaintext:")
print(P)
k=3
C = S.enciphering(k,P)
print("Encrypted plaintext:")
print(C)
print("Decrypted ciphertext:")
print(S.deciphering(k,C))
```

Output

```
Shift cryptosystem on Free alphabetic string monoid on A-Z
Representation of the plaintext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
Encrypted plaintext:
IXOOBVHFXUHVBVWHPVGRQRWHALVWWRGDBDQGWKHBZLOOQRWHALVWLQWKHIXWXUH
Decrypted ciphertext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
```

There are only 26 possible keys, hence a brute force attack is applicable here.

ShiftCipher

```
S = ShiftCryptosystem(AlphabeticStrings())
print(S)
P = S.encodin
 ↪ g("Fully secure systems do not exist today and they will
not exist in the future.")
print("Representation of the plaintext:")
print(P)
k=3
C = S.enciphering(k,P)
print("Encrypted plaintext:")
print(C)
bf = S.brute_force(C)
sorted(bf.items())
```

Output

```
Shift cryptosystem on Free alphabetic string monoid on A-Z
Representation of the plaintext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
Encrypted plaintext:
IXOOBVHFXUHVBVWHPVGRQRWHALVWWRGDBDQGWKHBZLOOQRWHALVWLQWKHIXWXUH

[(0, IXOOBVHFXUHVBVWHPVGRQRWHALVWWRGDBDQGWKHBZLOOQRWHALVWLQWKHIXWXUH),
 (1, HWNNAUGEWTGUAUVGOUFQPQVGZKUVVQFCACPFVJGAYKNNPQVGZKUVKPVJGHWVWTG),
 (2, GVMMZTFDVSFTZTUFNTEPOPUFYJTUUPEBZBOEUIFZXJMMOPUFYJTUJOUIFGVUVSF),
 (3, FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE),
 (4, ETKKXRDBTQDRXRSDLRCNMNSDWHRSSNCZXZMCSGDXVHKKMNSDWHRSHMSGDETSTQD),
 (5, DSJJWQCASPCQWQRCKQBMLMRCVGQRRMBYWYLBRFCWUGJJLMRCVGQRGLRFCDSRSPC),
 (6, CRIIVPBZROBPVPQBJPALKLQBUFPQQLAXVXKAQEBVTFIIKLQBUFPQFKQEBCRQROB),
 (7, BQHHUOAYQNAOUOPAIOZKJKPATEOPPKZWUWJZPDAUSEHHJKPATEOPEJPDABQPQNA),
 (8, APGGTNZXPMZNTNOZHNYJIJOZSDNOOJYVTVIYOCZTRDGGIJOZSDNODIOCZAPOPMZ),
 (9, ZOFFSMYWOLYMSMNYGMXIHINYRCMNNIXUSUHXNBYSQCFFHINYRCMNCHNBYZONOLY),
 (10, YNEERLXVNKXLRLMXFLWHGHMXQBLMMHWTRTGWMAXRPBEEGHMXQBLMBGMAXYNMNKX),
 (11, XMDDQKWUMJWKQKLWEKVGFGLWPAKLLGVSQSFVLZWQOADDFGLWPAKLAFLZWXMLMJW),
 (12, WLCCPJVTLIVJPJKVDJUFEFKVOZJKKFURPREUKYVPNZCCEFKVOZJKZEKYVWLKLIV),
 (13, VKBBOIUSKHUIOIJUCITEDEJUNYIJJETQOQDTJXUOMYBBDEJUNYIJYDJXUVKJKHU),
 (14, UJAANHTRJGTHNHITBHSDCDITMXHIIDSPNPCSIWTNLXAACDITMXHIXCIWTUJIJGT),
 (15, TIZZMGSQIFSGMGHSAGRCBCHSLWGHHCROMOBRHVSMKWZZBCHSLWGHWBHVSTIHIFS),
 (16, SHYYLFRPHERFLFGRZFQBABGRKVFGGBQNLNAQGURLJVYYABGRKVFGVAGURSHGHER),
 (17, RGXXKEQOGDQEKEFQYEPAZAFQJUEFFAPMKMZPFTQKIUXXZAFQJUEFUZFTQRGFGDQ),
 (18, QFWWJDPNFCPDJDEPXDOZYZEPITDEEZOLJLYOESPJHTWWYZEPITDETYESPQFEFCP),
 (19, PEVVICOMEBOCICDOWCNYXYDOHSCDDYNKIKXNDROIGSVVXYDOHSCDSXDROPEDEBO),
 (20, ODUUHBNLDANBHBCNVBMXWXCNGRBCCXMJHJWMCQNHFRUUWXCNGRBCRWCQNODCDAN),
 (21, NCTTGAMKCZMAGABMUALWVWBMFQABBWLIGIVLBPMGEQTTVWBMFQABQVBPMNCBCZM),
 (22, MBSSFZLJBYLZFZALTZKVUVALEPZAAVKHFHUKAOLFDPSSUVALEPZAPUAOLMBABYL),
 (23, LARREYKIAXKYEYZKSYJUTUZKDOYZZUJGEGTJZNKECORRTUZKDOYZOTZNKLAZAXK),
 (24, KZQQDXJHZWJXDXYJRXITSTYJCNXYYTIFDFSIYMJDBNQQSTYJCNXYNSYMJKZYZWJ),
 (25, JYPPCWIGYVIWCWXIQWHSRSXIBMWXXSHECERHXLICAMPPRSXIBMWXMRXLIJYXYVI)]
```

2.2 Affine ciphers

A generalization of the shift cipher is given by the so-called affine cipher. Here we use a function of the form $ax + b$, where $\gcd(a, 26) = 1$. A ciphertext character c_i is computed via the formula

$$c_i \equiv ap_i + b \pmod{26}.$$

To recover p_i from c_i we apply the congruence given by

$$p_i \equiv (c_i - b) \cdot a^{-1} \pmod{26}.$$

This is the point where we see why the assumption $\gcd(a, 26) = 1$ is important, otherwise the multiplicative inverse of a does not exist modulo 26. Here we use the same Adi Shamir quote as plaintext as in case of the shift cipher.

AffineCipher

```
A = AffineCryptosystem(AlphabeticStrings())
print(S)
```

```
P = S.encodin
 ↪ g("Fully secure systems do not exist today and they will
not exist in the future.")
print("Representation of the plaintext:")
print(P)
(a,b)=(3,11)
C = A.enciphering(a,b,P)
print("Encrypted plaintext:")
print(C)
print("Decrypted ciphertext:")
print(S.deciphering(a,b,C))
```

Output

```
Affine cryptosystem on Free alphabetic string monoid on A-Z
Representation of the plaintext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
Encrypted plaintext:
ATSSFNXRTKXNFNQXVNUBYBQXCJNQQBULFLYUQGXFZJSSYBQXCJNQJYQGXATQTKX
Decrypted ciphertext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
```

A brute force attack can easily break the system, here the possible number of keys is 26×26 even if we count those violating the gcd condition. We only display 40 candidate decipherments.

AffineCipher

```
A = AffineCryptosystem(AlphabeticStrings())
print(S)
P = S.encodin
 ↪ g("Fully secure systems do not exist today and they will
not exist in the future.")
print("Representation of the plaintext:")
print(P)
(a,b)=(3,11)
C = A.enciphering(a,b,P)
```

```
print("Encrypted plaintext:")
print(C)
bf = S.brute_force(C)
sorted(bf.items())[:40]
```

Output

```
Affine cryptosystem on Free alphabetic string monoid on A-Z
Representation of the plaintext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
Encrypted plaintext:
ATSSFNXRTKXNFNQXVNUBYBQXCJNQQBULFLYUQGXFZJSSYBQXCJNQJYQGXATQTKX

[((1, 0), ATSSFNXRTKXNFNQXVNUBYBQXCJNQQBULFLYUQGXFZJSSYBQXCJNQJYQGXATQTKX),
 ((1, 1), ZSRREMWQSJWMEMPWUMTAXAPWBIMPPATKEKXTPFWEYIRRXAPWBIMPIXPFWZSPSJW),
 ((1, 2), YRQQDLVPRIVLDLOVTLSZWZOVAHLOOZSJDJWSOEVDXHQQWZOVAHLOHWOEVYRORIV),
 ((1, 3), XQPPCKUOQHUKCKNUSKRYVYNUZGKNNYRICIVRNDUCWGPPVYNUZGKNGVNDUXQNQHU),
 ((1, 4), WPOOBJTNPGTJBJMTRJQXUXMTYFJMMXQHBHUQMCTBVFOOUXMTYFJMFUMCTWPMPGT),
 ((1, 5), VONNAISMOFSIAILSQIPWTWLSXEILLWPGAGTPLBSAUENNTWLSXEILETLBSVOLOFS),
 ((1, 6), UNMMZHRLNERHZHKRPHOVSVKRWDHKKVOFZFSOKARZTDMMSVKRWDHKDSKARUNKNER),
 ((1, 7), TMLLYGQKMDQGYGJQOGNURUJQVCGJJUNEYERNJZQYSCLLRUJQVCGJCRJZQTMJMDQ),
 ((1, 8), SLKKXFPJLCPFXFIPNFMTQTIPUBFIITMDXDQMIYPXRBKKQTIPUBFIBQIYPSLILCP),
 ((1, 9), RKJJWEOIKBOEWEHOMELSPSHOTAEHHSLCWCPLHXOWQAJJPSHOTAEHAPHXORKHKBO),
 ((1, 10), QJIIVDNHJANDVDGNLDKRORGNSZDGGRKBVBOKGWNVPZIIORGNSZDGZOGWNQJGJAN),
 ((1, 11), PIHHUCMGIZMCUCFMKCJQNQFMRYCFFQJAUANJFVMUOYHHNQFMRYCFYNFVMPIFIZM),
 ((1, 12), OHGGTBLFHYLBTBELJBIPMPELQXBEEPIZTZMIEULTNXGGMPELQXBEXMEULOHEHYL),
 ((1, 13), NGFFSAKEGXKASADKIAHOLODKPWADDOHYSYLHDTKSMWFFLODKPWADWLDTKNGDGXK),
 ((1, 14), MFEERZJDFWJZRZCJHZGNKNCJOVZCCNGXRXKGCSJRLVEEKNCJOVZCVKCSJMFCFWJ),
 ((1, 15), LEDDQYICEVIYQYBIGYFMJMBINUYBBMFWQWJFBRIQKUDDJMBINUYBUJBRILEBEVI),
 ((1, 16), KDCCPXHBDUHXPXAHFXELILAHMTXAALEVPVIEAQHPJTCCILAHMTXATIAQHKDADUH),
 ((1, 17), JCBBOWGACTGWOWZGEWDKHKZGLSWZZKDUOUHDZPGOISBBHKZGLSWZSHZPGJCZCTG),
 ((1, 18), IBAANVFZBSFVNVYFDVCJGJYFKRVYYJCTNTGCYOFNHRAAGJYFKRVYRGYOFIBYBSF),
 ((1, 19), HAZZMUEYAREUMUXECUBIFIXEJQUXXIBSMSFBXNEMGQZZFIXEJQUXQFXNEHAXARE),
 ((1, 20), GZYYLTDXZQDTLTWDBTAHEHWDIPTWWHARLREAWMDLFPYYEHWDIPTWPEWMDGZWZQD),
 ((1, 21), FYXXKSCWYPCSKSVCASZGDGVCHOSVVGZQKQDZVLCKEOXXDGVCHOSVODVLCFYVYPC),
 ((1, 22), EXWWJRBVXOBRJRUBZRYFCFUBGNRUUFYPJPCYUKBJDNWWCFUBGNRUNCUKBEXUXOB),
 ((1, 23), DWVVIQAUWNAQIQTAYQXEBETAFMQTTEXOIOBXTJAICMVVBETAFMQTMBTJADWTWNA),
 ((1, 24), CVUUHPZTVMZPHPSZXPWDADSZELPSSDWNHNAWSIZHBLUUADSZELPSLASIZCVSVMZ),
 ((1, 25), BUTTGOYSULYOGORYWOVCZCRYDKORRCVMGMZVRHYGAKTTZCRYDKORKZRHYBURULY),
 ((3, 0), APGGTNZXPMZNTNOZHNYJIJOZSDNOOJYVTVIYOCZTRDGGIJOZSDNODIOCZAPOPMZ),
 ((3, 1), RGXXKEQOGDQEKEFQYEPAZAFQJUEFFAPMKMZPFTQKIUXXZAFQJUEFUZFTQRGFGDQ),
 ((3, 2), IXOOBVHFXUHVBVWHPVGRQRWHALVWWRGDBDQGWKHBZLOOQRWHALVWLQWKHIXWXUH),
 ((3, 3), ZOFFSMYWOLYMSMNYGMXIHINYRCMNNIXUSUHXNBYSQCFFHINYRCMNCHNBYZONOLY),
 ((3, 4), QFWWJDPNFCPDJDEPXDOZYZEPITDEEZOLJLYOESPJHTWWYZEPITDETYESPQFEFCP),
 ((3, 5), HWNNAUGEWTGUAUVGOUFQPQVGZKUVVQFCACPFVJGAYKNNPQVGZKUVKPVJGHWVWTG),
 ((3, 6), YNEERLXVNKXLRLMXFLWHGHMXQBLMMHWTRTGWMAXRPBEEGHMXQBLMBGMAXYNMNKX),
 ((3, 7), PEVVICOMEBOCICDOWCNYXYDOHSCDDYNKIKXNDROIGSVVXYDOHSCDSXDROPEDEBO),
 ((3, 8), GVMMZTFDVSFTZTUFNTEPOPUFYJTUUPEBZBOEUIFZXJMMOPUFYJTUJOUIFGVUVSF),
 ((3, 9), XMDDQKWUMJWKQKLWEKVGFGLWPAKLLGVSQSFVLZWQOADDFGLWPAKLAFLZWXMLMJW),
 ((3, 10), ODUUHBNLDANBHBCNVBMXWXCNGRBCCXMJHJWMCQNHFRUUWXCNGRBCRWCQNODCDAN),
 ((3, 11), FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE),
 ((3, 12), WLCCPJVTLIVJPJKVDJUFEFKVOZJKKFURPREUKYVPNZCCEFKVOZJKZEKYVWLKLIV),
 ((3, 13), NCTTGAMKCZMAGABMUALWVWBMFQABBWLIGIVLBPMGEQTTVWBMFQABQVBPMNCBCZM)]
```

2.3 Hill ciphers

Affine ciphers use functions of the form $f(x) = ax+b$, so these are one-dimensional ciphers. Hill ciphers act on blocks of k characters. Given a $k \times k$ matrix A such that modulo 26 it has an inverse and b is a k-dimensional vector. A ciphertext block C_i is

computed by using the formula

$$c_i \equiv Ap_i + b \pmod{26}.$$

To recover p_i from c_i we apply the congruence given by

$$p_i \equiv A^{-1}(c_i - b) \pmod{26}.$$

HillCipher

```
S = AlphabeticStrings()
E = HillCryptosystem(S,3)
R = IntegerModRing(26)
M = MatrixSpace(R,3,3)
T=True
while T:
    A = M.random_element()
    if (A.det())%26 in [k for k in [1..25] if gcd(k,26)==1]:
        T=False
print("The matrix A is:")
print(A)
print("It has an inverse mod 26:")
print(A.inverse())
P = S.encodin
 ↪ g("Fully secure systems do not exist today and they will
not exist in the future.")
print("Representation of the plaintext:")
print(P)
C = E.enciphering(A,P)
print("Encrypted plaintext:")
print(C)
print("Decrypted ciphertext:")
print(E.deciphering(A,C))
```

Output

```
The matrix A is:
[16 22  1]
[13 10 15]
[23 11 15]
It has an inverse mod 26:
[ 7  5 24]
[ 8  1 21]
[25  2 12]
Representation of the plaintext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
Encrypted plaintext:
VPCSEREQWKOJPHHKQMJLSRWVCEXPIUCSZEHGTMCWEGUBHRWVCEXFRWTMCXZSJEX
Decrypted ciphertext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
```

2.4 Substitution ciphers

We consider an alphabet $\mathcal{A}$, let say all the capital letters from 'A' to 'Z'. Encryption is based on a substitution that is we take a permutation $\pi : \mathcal{A} \to \mathcal{A}$ and in the plaintext we replace each letter α with $\pi(\alpha)$. To decrypt a ciphertext we simple replace each letter β in the ciphertext with $\pi^{-1}(\beta)$. Exhaustive search is now much more difficult than in case of shift ciphers, here the key space has 26! elements.

SubstitutionCipher

```
A = AlphabeticStrings()
S = SubstitutionCryptosystem(A)
K = S.random_key()
print("The substitution key is:")
print(K)
M = A.encodin
 ↪ g("Fully secure systems do not exist today and they
will not exist in the future.")
print("Representation of the plaintext:")
```

```
print(M)
C = S.enciphering(K, M)
print("Encrypted plaintext:")
print(C)
print("Decrypted ciphertext:")
print(S.deciphering(K,C))
```

Output

```
The substitution key is:
JDGYUIAMCKBPZEQHXLVSWFORNT
Representation of the plaintext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
Encrypted plaintext:
IWPPNVUGWLUVNVSUZVYQEQSURCVSSQYJNJEYSMUNOCPPEQSURCVSCESMUIWSWLU
Decrypted ciphertext:
FULLYSECURESYSTEMSDONOTEXISTTODAYANDTHEYWILLNOTEXISTINTHEFUTURE
```

2.5 Vigenère ciphers

An extended version of affine ciphers. Blaise de Vigenère wrote a book in 1586 in which he described the known ciphers of his time. This cipher is named after him. The Vigenère cipher uses a keyword of any length greater than one. In case of a Vigenère cipher we add each of the index of the plaintext character to the index of the keyword

character using the so-called Vigenère square.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A

Let $M = M_1 M_2 \dots M_n$ denote the message represented by numbers between 0 and 25 ($A = 0, B = 1, C = 2$ etc.), $K = K_1 K_2 \dots K_n$ is the key obtained by repeating the keyword and $C = C_1 C_2 \dots C_n$ is the ciphertext. The encryption and decryption can be described algebraically as follows

$$C_i \equiv M_i + K_i \pmod{26},$$
$$M_i \equiv C_i - K_i \pmod{26}.$$

If the plaintext is 'CRYPTOGRAPHY' and the keyword is 'SEVEN', then we have

C	R	Y	P	T	O	G	R	A	P	H	Y
S	E	V	E	N	S	E	V	E	N	S	E

and here $M_1 = 2$ since the first character in the plaintext is 'C'. In a similar way we get that $K_1 = 18$, since the first letter in the keyword is 'S'. Therefore $C_1 = 2+18 = 20 \pmod{26}$ and the first letter in the ciphertext is 'U'. The complete encryption is SageMath can be done as follows.

Vigenère

```
S = AlphabeticStrings()
E = VigenereCryptosystem(S,5)
print('The keyword is SEVEN')
K = S('SEVEN')
e = E(K)
print('The message is CRYPTOGRAPHY')
```

```
print('The encrypted message is:',e(S("CRYPTOGRAPHY")))
```

Output

```
The keyword is SEVEN
The message is CRYPTOGRAPHY
The encrypted message is: UVTTGGKMECZC
```

Decryption is easy by using the Vigenère square, if we consider the row of the letter 'S' we look for the corresponding ciphertext letter (in our example it is 'U') and we see that it is in the third column, the column of 'C'. Thus the first letter in the plaintext is 'C'.

To break the Vigenère cipher first we need to figure out the length of the keyword. This is done by applying the so-called Kasiski method. It uses the fact that certain plaintext fragments occur quite frequently, while other plaintext fragments occur infrequently. The list of the distances that separate the repetitions has a key role here. The length of the keyword is likely to divide many of these distances. Let us try to use this idea in case of a quote by Anand.

Vigenère2

```
S = AlphabeticStrings()
E = VigenereCryptosystem(S,5)
print('The keyword is CHESS')
K = S('CHESS')
e = E(K)
P = S.encodin
 ↪ g("The broader the chess player you are, the easier it is
to be competitive, and the same seems to be true of
 ↪ mathematics
if you can find links between different branches of
 ↪ mathematics,
it can help you resolve problems. In both mathematics and
 ↪ chess,
you study existing theory and use that to go forward.")
print('The encrypted message is:',e(P))
```

Output

```
The keyword is CHESS
The encrypted message is:
VOITJQHHWJVOIUZGZWHDCFIJQQBEJWVOIWSUPIJAVPWLGDLGGERLXALKCISFFAL
WKCTIKWGTWLGDLXJMGVJESVOIESVPGKAHFSMUCUJAFFSMFCUIILOGLRVAHMIJWP
AFJSPJLWKQMQSLJLQSLKJWALEHRZWNWCGMTLWGDXLTJGDSIEKKUFGLJTELZGTEL
AEZEFVEOIKKAVYKLWKCWPKZXAFIALWGTFEFVWZILZCAXGYQMSJOCYH
```

There are some substrings that occur a few times, for example

$$\begin{aligned} \text{WLGD:} &\quad 42, 72 \\ \text{ESV:} &\quad 83, 88 \\ \text{TEL:} &\quad 181, 186. \end{aligned}$$

Hence we obtain the list of distances $[30, 5, 5]$. We may guess that the length of the keyword is 5. We partition the encrypted message into 5 parts:

VQVGCQVUVDRKFCGDGVVHCFUGHPPQJKENTXDKJGEEAWKITWCQC
OHOZFBOPPLLCATTLVOPFUSILMAJMLJHWLLSUTTZOVKZAFZAMY
IHIWIEIIWGXILIWXJIGSJMIRIFLQQWRCWTIFEEEIYCXLEIXSH
TWUHJJWJLGASWKLJEEKMAFLVJJWSSAZGGJEGLLFKKWAWFLGJ
JJZDQWSAGELFKWGMSSAUFCOAWSKLLLWMDGKLZAVKLPFGVZYO.

Frequency analysis is useful to break shift ciphers. The characteristic frequency probability distribution table of Beker and Piper can be obtained in SageMath using the code:

BekerPiper

```
A = AlphabeticStrings()
table = A.characteristic_frequency(table_name="beker_piper")
sorted(table.items())
```

Output

```
[('A', 0.0820000000000000),
 ('B', 0.0150000000000000),
 ('C', 0.0280000000000000),
 ('D', 0.0430000000000000),
 ('E', 0.127000000000000),
 ('F', 0.0220000000000000),
 ('G', 0.0200000000000000),
 ('H', 0.0610000000000000),
 ('I', 0.0700000000000000),
 ('J', 0.00200000000000000),
 ('K', 0.00800000000000000),
 ('L', 0.0400000000000000),
 ('M', 0.0240000000000000),
 ('N', 0.0670000000000000),
 ('O', 0.0750000000000000),
 ('P', 0.0190000000000000),
 ('Q', 0.00100000000000000),
 ('R', 0.0600000000000000),
 ('S', 0.0630000000000000),
 ('T', 0.0910000000000000),
 ('U', 0.0280000000000000),
 ('V', 0.0100000000000000),
 ('W', 0.0230000000000000),
 ('X', 0.00100000000000000),
 ('Y', 0.0200000000000000),
 ('Z', 0.00100000000000000)]
```

There is an other table by Lewand:

Lewand

```
1 A = AlphabeticStrings()
2 table = A.characteristic_frequency(table_name="lewand")
3 sorted(table.items())
```

Output

```
[('A', 0.0816700000000000),
('B', 0.0149200000000000),
('C', 0.0278200000000000),
('D', 0.0425300000000000),
('E', 0.127020000000000),
('F', 0.0222800000000000),
('G', 0.0201500000000000),
('H', 0.0609400000000000),
('I', 0.0696600000000000),
('J', 0.00153000000000000),
('K', 0.00772000000000000),
('L', 0.0402500000000000),
('M', 0.0240600000000000),
('N', 0.0674900000000000),
('O', 0.0750700000000000),
('P', 0.0192900000000000),
('Q', 0.000950000000000000),
('R', 0.0598700000000000),
('S', 0.0632700000000000),
('T', 0.0905600000000000),
('U', 0.0275800000000000),
('V', 0.00978000000000000),
('W', 0.0236000000000000),
('X', 0.00150000000000000),
('Y', 0.0197400000000000),
('Z', 0.000740000000000000)]
```

Thus characters with a high frequency of occurence are E,A and T. In case of

VQVGCQVUVDRKFCGDGVVHCFUGHPPQJKENTXDKJGEEAWKITWCQC

we obtain the following frequency distribution

Output

```
{A: 0.0204081632653061, C: 0.102040816326531,
 D: 0.0612244897959184, E: 0.0612244897959184,
 F: 0.0408163265306122, G: 0.102040816326531,
 H: 0.0408163265306122, I: 0.0204081632653061,
 J: 0.0408163265306122, K: 0.0816326530612245,
 N: 0.0204081632653061, P: 0.0408163265306122,
 Q: 0.0816326530612245, R: 0.0204081632653061,
 T: 0.0408163265306122, U: 0.0408163265306122,
 V: 0.122448979591837, W: 0.0408163265306122,
 X: 0.0204081632653061}
```

It suggests that here V is E,A or T. The second partition given by

OHOZFBOPPLLCATTLVOPFUSILMAJMLJHWLLSUTTZOVKZAFZAMY

has

Output

```
{A: 0.0816326530612245, B: 0.0204081632653061,
 C: 0.0204081632653061, F: 0.0612244897959184,
 H: 0.0408163265306122, I: 0.0204081632653061,
 J: 0.0408163265306122, K: 0.0204081632653061,
 L: 0.142857142857143, M: 0.0612244897959184,
 O: 0.102040816326531, P: 0.0612244897959184,
 S: 0.0408163265306122, T: 0.0816326530612245,
 U: 0.0408163265306122, V: 0.0408163265306122,
 W: 0.0204081632653061, Y: 0.0204081632653061,
 Z: 0.0816326530612245}
```

Therefore we may expect that L is one of the letters E,A or T. In this way we only need to check a few possible shifts in a given partition. In this concrete example we obtain that

$$\begin{aligned} \text{V} &\longrightarrow \text{T} \\ \text{L} &\longrightarrow \text{E} \\ \text{I} &\longrightarrow \text{E} \\ \text{J} &\longrightarrow \text{R} \\ \text{L} &\longrightarrow \text{T}. \end{aligned}$$

2.6 ADFGX and ADFGVX ciphers

The German Army used the ADFGX cipher during World War I. It is named after the five letters used in the ciphertext: A, D, F, G and X. These letters were chosen in a way to reduce the possibility of operator error, these are very different from each other when transmitted via morse code. To see how the encryption works let us provide the steps involved.

- We need to construct a so-called polybius square, a 5×5 matrix containing the letters from 'a' to 'z' in such a way that 'i' is considered to be the same as 'j'. For example

$$\begin{array}{c} \\ A \\ D \\ F \\ G \\ X \end{array} \begin{array}{c} \begin{array}{ccccc} A & D & F & G & X \end{array} \\ \begin{pmatrix} m & z & t & x & i/j \\ q & u & b & w & c \\ k & g & n & p & d \\ y & e & r & s & o \\ l & h & a & v & f \end{pmatrix} \end{array}$$

- To encode a plaintext we use the above matrix in the following way: $m \to AA, z \to AD, \ldots, f \to XX$. For example the plaintext 'cryptography' is going to be

$$DXGFGAFGAFGXFDGFXFFGXDGA.$$

- A code word is chosen and we write the enciphered plaintext underneath. Following the above example using the code word CIPHER we obtain

C	I	P	H	E	R
D	X	G	F	G	A
F	G	A	F	G	X
F	D	G	F	X	F
F	G	X	D	G	A

- We apply a columnar transposition, that is we sort the code word alphabetically, moving the appropriate columns as well. In the above example we get

C	E	H	I	P	R
D	G	F	X	G	A
F	G	F	G	A	X
F	X	F	D	G	F
F	G	D	G	X	A

- We read the final ciphertext off in columns as follows

$$DFFFGGXGFFFDXGDGGAGXAXFA.$$

An extension based on six letters A, D, F, G, V and X was invented by Colonel Fritz Nebel and introduced in March 1918. It uses a 6 by 6 polybius square containing all the letters and the numbers from 0 to 9. Let us describe the steps in case of the plaintext 'cryptography2020' and with code word 'MOVE'.

- We fix the polybius square as follows

$$\begin{array}{c} \\ A \\ D \\ F \\ G \\ V \\ X \end{array} \begin{array}{c} \begin{array}{cccccc} A & D & F & G & V & X \end{array} \\ \begin{pmatrix} m & j & t & z & 0 & 1 \\ u & 6 & w & 9 & g & d \\ 4 & e & y & r & l & v \\ h & s & a & 3 & x & i \\ q & b & c & k & 2 & 8 \\ 7 & 5 & n & p & o & f \end{pmatrix} \end{array}.$$

- Encoding the plaintext 'cryptography2020' using the above polybius square yields

$$VFFGFFXGAFXVDVFGGFXGGAFFVVAVVVAV.$$

- The code word is 'MOVE' so we have

M	O	V	E
V	F	F	G
F	F	X	G
A	F	X	V
D	V	F	G
G	F	X	G
G	A	F	F
V	V	A	V
V	V	A	V

- We sort the code word alphabetically together with the corresponding columns to obtain the ciphertext

$$VFADGGVVFFFVFAVVFXXFXFAAGGVGGFVV.$$

A possible SageMath implementation with a random polybius square is as follows.

ADFGX

```
def ADFGX(w,codeword):
    S=Set([chr(k) for k in [97..122] if chr(k)!='j'])
    S0=Set([])
    while S.cardinality()!=0:
        s=S.random_element()
        S0=S0.union(Set([s]))
        S=S.difference(Set([s]))
    P=PolynomialRing(Integers(),25,list(S0))
    Polybius=matrix(5,5,list(P.gens()))
    pretty_print(html('The Polybius Square is $%s$'%late
    ↪ x(Polybius)))
    L=['A','D','F','G','X']
    Encode0=''
    for k in w:
        if k=='j': k='i'
        k1=[(i,j) for i in [0..4] for j in [0..4] if
        ↪ Polybius[i,j]==P(k)][0]
        Encode0=Encode0+L[k1[0]]+L[k1[1]]
    pretty_print(html('The first round is %s'%Encode0))
    El=len(Encode0)
    cwl=len(codeword)
    if cwl*(El//cwl)!=El:
        m0=cwl*((El//cwl)+1)-El
        Encode0=Encode0+['A' for _ in [1..m0]]
    R.<A,D,F,G,X>=PolynomialRing(Integers(),5)
    Encode1=[R(k) for k in Encode0]
    Erow=len(Encode1)//cwl
    M=matrix(Erow,cwl,Encode1)
    pretty_print(html('The code word table is $%s$'%latex(M)))
    Cipher=[]
    for k in sorted(list(codeword)):
        Cipher=Cipher+list(M.column(codeword.index(k)))
```

```
    pretty_print(html('The encypted message is $%s$'%late
     ↪  x(Cipher)))
    return Cipher
ADFGX('cryptography','CIPHER')
```

Output

The Polybius Square is $\begin{pmatrix} m & x & z & t & i \\ u & q & b & w & c \\ k & g & n & d & p \\ a & y & e & r & o \\ l & h & s & v & f \end{pmatrix}$

The first round is DXGGGDFXAGGXFDGGGAFXXDGD

The code word table is $\begin{pmatrix} D & X & G & G & G & D \\ F & X & A & G & G & X \\ F & D & G & G & G & A \\ F & X & X & D & G & D \end{pmatrix}$

The encypted message is $[D, F, F, F, G, G, G, G, G, G, G, D, X, X, D, X, G, A, G, X, D, X, A, D]$

2.7 Playfair cipher

The Playfair cipher is named after Lord Lyon Playfair who promoted the use of the cipher [22]. The cipher itself was invented by Charles Wheatstone in 1854. The British Army in World War I used as the standard field system. It is also based on a polybius square like the ABFGX and ABFGVX ciphers, however here one uses a keyword as well to form the appropriate 5×5 square. The two letters I and J are identified out of the 26 capital letters used in the cryptosystem. For example if we have KEYWORD given as a keyword, then the corresponding polybius square is

K	E	Y	W	O
R	D	A	B	C
F	G	H	I	L
M	N	P	Q	S
T	U	V	X	Z

We will demonstrate the rules used in the encryption process using the plaintext COMMUNICATION and the polybius square given above.

- We split the plaintext up into digraphs: CO MM UN IC AT IO N.

- If a digraph consists of the same letter twice, then we insert an X between them. If the length of the plaintext is odd, then we add an X at the end. In our example we have MM as a digraph hence we get CO MX MU NI CA TI ON.
- If the two letters in a digraph appear on the same row in the polybius square, then replace each letter by the letter immediately to the right of it in the polybius square (cycling round if necessary). In our example we have CA in a given row, it is encrypted as RB.
- If the two letters in a digraph appear in the same column in the polybius square, then replace each letter by the letter immediately below it in the polybius square (cycling round if necessary). In the example the digraph CO is encrypted as LC.
- If the two letters are not in the same row or column, then form the rectangle for which the two letters are two opposite corners. We replace each letter with the letter that forms the other corner of the rectangle that lies on the same row. Let us see this latter rule in action in our example. We have

MX	$\longrightarrow$	QT
MU	$\longrightarrow$	NT
NI	$\longrightarrow$	QG
TI	$\longrightarrow$	XF
ON	$\longrightarrow$	ES

.

That is the plaintext COMMUNICATION is encrypted as LCQTNTQGRBXFES.

The following SageMath implementations are due to Alasdair McAndrew (see https://trac.sagemath.org/ticket/8559) and modified by Catalina Camacho-Navarro (see https://wiki.sagemath.org/interact/cryptography#Playfair_Cipher). First we consider the encryption part.

PlayfairEncrypt

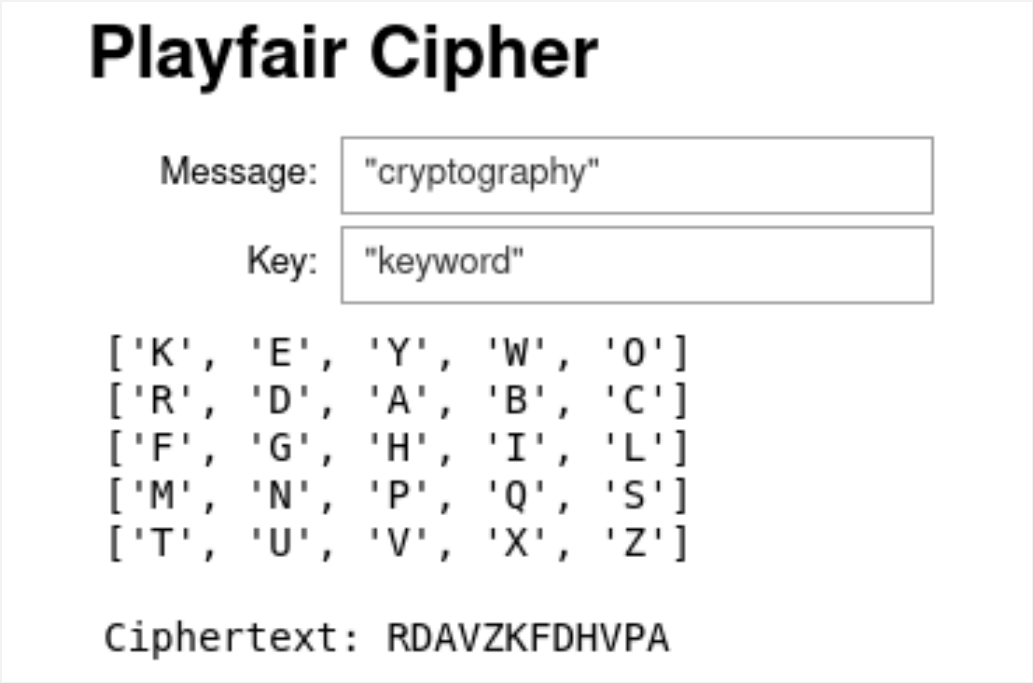
```
def change_to_plain_text(pl):
    plaintext=''
    for ch in pl:
        if ch.isalpha():
            plaintext+=ch.upper()
    return plaintext

def makePF(word1):
    word=change_to_plain_text(word1)
```

```
    alph='ABCDEFGHIKLMNOPQRSTUVWXYZ'
    pf=''
    for ch in word:
        if (ch!="J") & (pf.find(ch)==-1):
            pf+=ch
    for ch in alph:
        if pf.find(ch)==-1:
            pf+=ch
    PF=[[pf[5*i+j] for j in range(5)] for i in range(5)]
    return PF

def pf_encrypt(di,PF):
    for i in range(5):
        for j in range(5):
            if PF[i][j]==di[0]:
                i0=i
                j0=j
            if PF[i][j]==di[1]:
                i1=i
                j1=j
    if (i0!=i1) & (j0!=j1):
        return PF[i0][j1]+PF[i1][j0]
    if (i0==i1) & (j0!=j1):
        return PF[i0][(j0+1)%5]+PF[i1][(j1+1)%5]
    if (i0!=i1) & (j0==j1):
        return PF[(i0+1)%5][j0]+PF[(i1+1)%5][j1]

def insert(ch,str,j):
    tmp=''
    for i in range(j):
        tmp+=str[i]
    tmp+=ch
    for i in range(len(str)-j):
        tmp+=str[i+j]
```

```
    return tmp


def playfair(pl1,word):
    pl=change_to_plain_text(pl1)
    PF=makePF(word)
    pl2=makeDG(pl)
    tmp=''
    for i in range(len(pl2)//2):
        tmp+=pf_encrypt(pl2[2*i]+pl2[2*i+1],PF)
    return tmp

def makeDG(str):
    tmp=str.replace('J','I')
    c=len(tmp)
    i=0
    while (c>0) & (2*i+1<len(tmp)):
        if tmp[2*i]==tmp[2*i+1]:
            tmp=insert("X",tmp,2*i+1)
            c-=1
            i+=1
        else:
            c-=2
            i+=1
    if len(tmp)%2==1:
        tmp+='X'
    return tmp

pretty_print(html("<h1>Playfair Cipher</h1>"))
@interact
def _(Message=input_box(default='"cryptography"',
 ↪  label="Message:"),
      Key=input_box(default='"keyword"', label="Key:")):

```

```
    poly = makePF(Key)
    for i in range(5):
        print(poly[i])

    print('\nCiphertext:', playfair(Message, Key))
```

Output



The decryption part uses the keyword KEYWORD as a default value and the ciphertext is NTFZHQFLRBXFES.

PlayfairDecrypt

```
def change_to_plain_text(pl):
    plaintext=''
    for ch in pl:
        if ch.isalpha():
            plaintext+=ch.upper()
    return plaintext

def makePF(word1):
    word=change_to_plain_text(word1)
    alph='ABCDEFGHIKLMNOPQRSTUVWXYZ'
```

```
    pf=''
    for ch in word:
        if (ch!="J") & (pf.find(ch)==-1):
            pf+=ch
    for ch in alph:
        if pf.find(ch)==-1:
            pf+=ch
    PF=[[pf[5*i+j] for j in range(5)] for i in range(5)]
    return PF

def pf_decrypt(di,PF):
    for i in range(5):
        for j in range(5):
            if PF[i][j]==di[0]:
                i0=i
                j0=j
            if PF[i][j]==di[1]:
                i1=i
                j1=j
    if (i0!=i1) & (j0!=j1):
        return PF[i0][j1]+PF[i1][j0]
    if (i0==i1) & (j0!=j1):
        return PF[i0][(j0-1)%5]+PF[i1][(j1-1)%5]
    if (i0!=i1) & (j0==j1):
        return PF[(i0-1)%5][j0]+PF[(i1-1)%5][j1]

def insert(ch,str,j):
    tmp=''
    for i in range(j):
        tmp+=str[i]
    tmp+=ch
    for i in range(len(str)-j):
        tmp+=str[i+j]
    return tmp
```

```
45
46
47  def playfair_decrypt(pl1,word):
48      pl=change_to_plain_text(pl1)
49      if len(pl1)%2:
50          raise TypeError
            ↪ ('The lenght of the ciphertext is not even')
51      pl2=makeDG(pl)
52      if pl2!=pl:
53          if 'J' in pl:
54              raise TypeError('The ciphertext contains a J')
55          if len(pl2)!=len(pl):
56              raise
                ↪ TypeError('The ciphertext contains digraphs \
57                          with repeated letters')
58
59      PF=makePF(word)
60
61      tmp=''
62      for i in range(len(pl2)//2):
63          tmp+=pf_decrypt(pl2[2*i]+pl2[2*i+1],PF)
64      return tmp
65
66  def makeDG(str):
67      tmp=str.replace('J','I')
68      c=len(tmp)
69      i=0
70      while (c>0) & (2*i+1<len(tmp)):
71          if tmp[2*i]==tmp[2*i+1]:
72              tmp=insert("X",tmp,2*i+1)
73              c-=1
74              i+=1
75          else:
76              c-=2
```

```
            i+=1
    if len(tmp)%2==1:
        tmp+='X'
    return tmp

def playfair_decrypt_options(pl):
    pl_noI=pl.replace('I','J')
    if pl.endswith('X'):
        pl_no_last_X=pl[:-1]
    else: pl_no_last_X=pl
    pl_noX=pl
    for ch in pl_noX:
        if (ch=='X') & (pl.find(ch)!=0):
            if  pl_noX[pl_noX.find(ch)-1]==pl_noX[pl_noX.fin
             d(ch)+1]:
                pl_noX=pl_noX.replace('X','')
    return([pl,pl_noI,pl_noX,pl_no_last_X])

pretty_print(html("<h1>Playfair Cipher</h1>"))
@interact
def _(Ciphertext=input_box(default='"NTFZHQFLRBXFES"',
  label="Message:"),
      Key=input_box(default='"keyword"', label='Key:')):
    print('These are some of the possibilities \
    for the plaintext:')
    print(playfair_decrypt_options(playfair_decrypt(Cipherte
     xt,Key)))
    poly=makePF(Key)
    print('----------------------')
    for i in range(5):
        print(poly[i])
```

Output

Playfair Cipher

Message: "NTFZHQFLRBXFES"

Key: "keyword"

```
These are some of the possibilities for the plaintext:
['MULTIPLICATION', 'MULTJPLJCATJON', 'MULTIPLICATION', 'MULTIPLICATION']
----------------------
['K', 'E', 'Y', 'W', 'O']
['R', 'D', 'A', 'B', 'C']
['F', 'G', 'H', 'I', 'L']
['M', 'N', 'P', 'Q', 'S']
['T', 'U', 'V', 'X', 'Z']
```

2.8 SageMath summary

function	description
AffineCryptosystem()	Create an affine cryptosystem.
AlphabeticStrings()	The free alphabetic string monoid on generators A-Z.
HillCryptosystem()	Create a Hill cryptosystem.
IntegerModRing()	The quotient ring $\mathbb{Z}/N\mathbb{Z}$.
MatrixSpace()	Create a matrix space of all $n \times m$ matrices over a ring R.
ShiftCryptosystem()	Create a shift cryptosystem.
SubstitutionCryptosystem()	Create a substitution cryptosystem.
VigenereCryptosystem()	Create a Vigenere cryptosystem.
brute_force()	Attempt a brute force cryptanalysis of the ciphertext.
characteristic_frequency()	Table of the characteristic frequency probability distribution of the English alphabet.
deciphering	Decrypt the ciphertext.
det()	The determinant of a matrix.
enciphering()	Encrypt the plaintext using a cipher encryption.
encoding()	The encoding of a string in the alphabetic string monoid.
gcd()	The greatest common divisor.
inverse()	The inverse of a matrix.

Exercises

1. Encrypt the message MATHEMATICS with the shift cipher with 7 as the key.
2. Encrypt the message MATHEMATICS with the affine cipher with $(a, b) = (11, 5)$ as the key.
3. Encrypt the message CRYPTOSYSTEM with the Hill cipher with
$$A = \begin{pmatrix} 5 & 15 \\ 24 & 1 \end{pmatrix} \qquad \text{and } b = (3, 5).$$
4. Encrypt the message MATHEMATICS with the Vigenère cipher with ROOT as the keyword.
5. Decrypt the following message, which was encrypted with a shift cipher with 17 as the key: KIZREXCV.
6. Decrypt the following message, which was encrypted with an affine cipher with $(a, b) = (9, 10)$ as the key: ZHEKXMFU.

7. Decrypt the following message, which was encrypted with a Hill cipher with
$$A = \begin{pmatrix} 15 & 18 \\ 14 & 21 \end{pmatrix}: \qquad \text{DXQOTWNW.}$$
8. Decrypt the following message, which was encrypted with a Vigenère cipher with KEY as the keyword: DVGKREVI.
9. It is known that the following ciphertexts were encrypted using shift ciphers. Decrypt the messages and determine the keys that were used.
   - XYNYLGCHUHN,
   - EQCGQZOQ.
10. It is known that the following ciphertexts were encrypted using affine ciphers. Decrypt the messages and determine the keys that were used.
   - PKVQTOMV,
   - ZFBCPODKX.
11. Suppose we intercept the ciphertext "ANGJPWPBKOCRAXWYJSHMTKUUQTWCNVBRIAKOHQUAOVTKJUANDCOGBQSRIDAXCZEVXEWVWMFF" formed by a Hill cipher with some 2×2 key matrix over $\mathbb{Z}_{26}$. We also know that the plaintext starts with "ANEXPERT". Decrypt the remainder of the ciphertext.
12. The ciphertext GPHDRHXVLYCSKTVUPZNQKGUPGFHGNOZJHSKVZCNKUWNCWICQHGPGFHOPDBGUEPDXTRCHLKNG was obtained by encrypting an English text using a Vigenère cipher. Try to decrypt it.
13. Encrypt the message 'polynomial' using the ADFGX cipher with the code word 'RING' and an arbitrary polybius square.
14. Implement the ADFGVX cipher in SageMath and use it to encrypt the message 'cryptography2020' with the code word 'MOVE' and an arbitrary polybius square.
15. Encrypt the message PARALLELEPIPED using the Playfair cipher with the keyword CIPHER.
16. The following ciphertext was encrypted with a Playfair cipher with keyword BISECTOR. Decrypt the message: HPTNISIDESTRACEDKSTAGW.

Chapter 3 The RSA algorithm

Summary

- ❏ Common modulus attack
- ❏ Iterated encryption attack
- ❏ Low public exponent
- ❏ Wiener's attack
- ❏ Davida's attack
- ❏ Elliptic curve factorization
- ❏ Continued fraction factorization
- ❏ Dixon's method
- ❏ Pollard's ρ factorization
- ❏ SageMath summary
- ❏ Exercises

RSA is a widely used public key cryptosystem developed by Rivest, Shamir and Adleman in 1977 [19]. The security of the RSA algorithm is based on the complexity of factorization of (large) integers. The algorithm consists of the following steps.

- Key generation: one generates two large prime numbers p and q and computes $N = pq$, finally picks e such that $\gcd(e, \varphi(N)) = 1$. We remark that $\varphi()$ is a multiplicative function and for a prime number p one has that $\varphi(p) = p-1$, hence $\varphi(N) = \varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1)$. By using the extended Euclidean algorithm one can determine d satisfying

$$ed \equiv 1 \pmod{\varphi(N)}.$$

  The public key is given by (N, e) and the private key is (p, q, d).
- Encryption: for a given public key (N, e) and $x \in \mathbb{Z}_N$ one computes

$$\mathrm{Enc}_{(N,e)}(x) = x^e \pmod{N}.$$

- Decryption: given the private key and an encrypted message $y \in \mathbb{Z}_N$ one determines the message as follows

$$\mathrm{Dec}_{(p,q,d)}(y) = y^d \pmod{N}.$$

To show that the above procedure works correctly we need to prove that

$$(x^e)^d \equiv x \pmod{N}.$$

We use the following well-known result from elementary number theory.

Theorem 3.1. Fermat's little theorem

Let p be a prime number and a is an integer such that $\gcd(a, p) = 1$. One has that

$$a^{p-1} \equiv 1 \pmod{p}.$$

First we prove that $x^{ed} \equiv x \pmod{p}$. Since $ed \equiv 1 \pmod{\varphi(N)}$ one has that $ed = 1 + k\varphi(N) = 1 + k(p-1)(q-1)$ for some integer k. We have that

$$x^{ed} \equiv x^{1+k\varphi(N)} \equiv x^{1+k(p-1)(q-1)} \equiv x \cdot \left(x^{p-1}\right)^{k(q-1)} \pmod{p}.$$

If p divides x, then it is clear that the $x^{ed} \equiv x \pmod{p}$. If p does not divide x, then by Fermat's little theorem it follows that $x^{p-1} \equiv 1 \pmod{p}$, thus

$$x^{ed} \equiv x \cdot 1^{k(q-1)} \equiv x \pmod{p}.$$

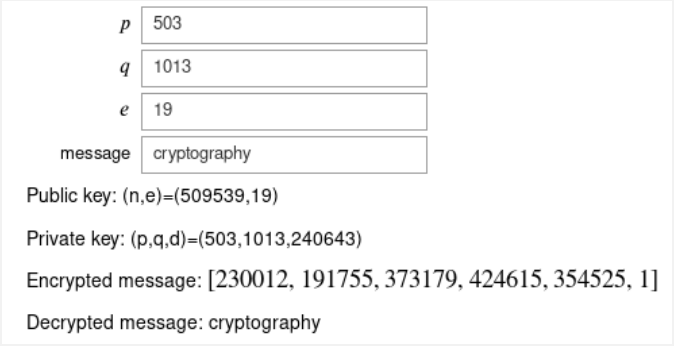
Repeating the above steps with q instead of p one obtains that $x^{ed} \equiv x \pmod{q}$. Since p and q are distinct prime numbers and both divides $x^{ed} - x$ we get that $p \cdot q = N$ divides $x^{ed} - x$, that is $x^{ed} \equiv x \pmod{N}$. To see how the RSA algorithm works in practice we provide a SageMath code.

RSA

```
@interact
def RSA(p1=input_box('503',type = str, label='$p$'),
        q1=input_box('1013',type = str, label='$q$'),
        e1=input_box('19',type = str, label='$e$'),
        message="cryptography"):
    p=ZZ(p1)
    q=ZZ(q1)
    e=ZZ(e1)
    n=p*q
    phi=(p-1)*(q-1)
    d=(1/e)%phi
    pretty_print(html('Public key: (n,e)=(%s,%s)'%(latex(n
    ↪ ),latex(e))))
    pretty_print(html('Private key: (p,q,d)=(%s,%s,%s)'%(late
    ↪ x(p),latex(q),latex(d))))
    m=IntegerRing()([ord(k) for k in message],base=256)
    m1=m.digits(base=n)
    encrypt=[power_mod(t, e, n) for t in m1]
    pretty_print(html('Encrypted message: $%s$'%latex(encryp
    ↪ t)))
    decrypt0=[power_mod(t, Integers()(d), n) for t in encrypt]
    decrypt=IntegerRing()(decrypt0,base=n)
    dm0=[chr(t) for t in decrypt.digits(base=256)]
```

```
21      dm=''.join(dm0)
22      pretty_print(html('Decrypted message: %s'%dm))
```

Output



3.1 Common modulus attack

Suppose that at a company it is decided that they will use the RSA cryptosystem with a given modulus N and each employee will have a personalized public encryption exponent e and corresponding private decryption exponent d. A manager sends the same message m to two of his/her colleagues. It will yield two different ciphertexts

$$\begin{aligned} c_1 &\equiv m^{e_1} \pmod{N}, \\ c_2 &\equiv m^{e_2} \pmod{N}. \end{aligned}$$

Here N, e_1, e_2, c_1 and c_2 are all known and one can read the secret message without knowing one of the private keys d_1, d_2. The approach is based on the extended Euclidean algorithm. We compute s and t for which

$$se_1 + te_2 = 1.$$

The above system of congruences provide that

$$\begin{aligned} c_1^s &\equiv (m^{e_1})^s \pmod{N}, \\ c_2^t &\equiv (m^{e_2})^t \pmod{N}. \end{aligned}$$

It follows that

$$c_1^s c_2^t \equiv m^{se_1+te_2} \equiv m \pmod{N}.$$

Common modulus attack

```
def RSAcommon(m,e1,e2,N):
    c1=(m^e1)%N
    c2=(m^e2)%N
    pretty_print(html('The first ciphertext is %s'%latex(c1)))
    pretty_print(html('The second ciphertext is %s'%late
    ↪ x(c2)))
    g,s,t=xgcd(e1,e2)
    pretty_print(html('We have $s=%s$ and $t=%s$'%(latex(s
    ↪ ),latex(t))))
    M=(c1^s*c2^t)%N
    pretty_print(html('The message is %s'%latex(M)))
    return M
RSAcommon(500,17,5,1591)
```

Output

```
The first ciphertext is 849
The second ciphertext is 22
We have s=-2 and t=7
The message is 500
```

3.2 Iterated encryption attack

This attack is based on Euler's theorem, a generalization of Fermat's little theorem.

Theorem 3.2. Euler

Let N be a positive integer and M is an integer such that $\gcd(M,N)=1$. *We have that*

$$M^{\varphi(N)} \equiv 1 \pmod{N}.$$

Suppose we use the RSA cryptosystem with the public key (N,e) and we would like to send a message $m < N$. We need to compute $c_1 \equiv m^e \pmod{N}$. One can consider

an iterative sequence given by

$$\begin{aligned} c_1 &\equiv m^e \pmod N, \\ c_2 &\equiv c_1^e \pmod N, \\ &\dots \\ c_k &\equiv c_{k-1}^e \pmod N. \end{aligned}$$

This sequence is simply $c_k \equiv m^{e^k} \pmod N$. If $\gcd(N, m) \neq 1$, then we can easily obtain a non-trivial divisor of N. Otherwise we may apply Euler's theorem to get that $m^{\varphi(N)} \equiv 1 \pmod N$. Since $\gcd(e, \varphi(N)) = 1$, we obtain that for some integer k_0 we have

$$e^{k_0} \equiv 1 \pmod{\varphi(N)}.$$

Thus we get that $c_{k_0} = m$.

Iterated attack

```
def RSAiterate(c1,e,N):
    L=[c1]
    ck=1
    pretty_print(html('$c_1=%s$'%latex(c1)))
    k=2
    while len(L)==Set(L).cardinality():
        ck=(c1^e)%N
        pretty_print(html('$c_{%s}=%s$'%(latex(k),latex(ck))))
        L=L+[ck]
        c1=ck
        k=k+1
    pretty_print(htm⌋
    ↪ l('The set of possible messages is $%s$'%latex(Se⌋
    ↪ t(L))))
    return Set(L)
RSAiterate(849,17,1591)
```

Output

```
c1=849
c2=389
c3=997
c4=426
c5=1441
c6=500
c7=849
The set of possible messages is {1441,849,500,389,997,426}
```

3.3 Low public exponent attack

In the RSA cryptosystem the encryption process might be quite costly if the public exponent e is large. It may be a problem in terms of time and battery power on some limited devices like smart cards. In these cases one might choose a small public exponent like $e = 3$. Suppose that Alice is going to send the same message to Bob, Chris and David, say m. She knows the public keys of Bob, Chris and David, let us denote these by N_B, N_C and N_D. Alice computes the ciphertexts as follows

$$\begin{aligned} c_B &\equiv m^3 \pmod{N_B}, \\ c_C &\equiv m^3 \pmod{N_C}, \\ c_D &\equiv m^3 \pmod{N_D}. \end{aligned}$$

Assume that Eve, an eavesdropper, obtains these ciphertexts. Let us see how to recover m. If N_B, N_C and N_D are not pairwise relatively prime numbers, then Eve can factor at least two of them and easily computes the private keys. So we may assume that those numbers are pairwise relatively prime. In this case Eve applies the Chinese Remainder Theorem to determine c for which $c \equiv m^3 \pmod{N_B N_C N_D}$. Since m is less than N_B, N_C and N_D, we get that $m^3 < N_B N_C N_D$. Thus instead of a congruence we have equality over the integers, that is $c = m^3$. Taking the cubic root of c over the integers yields the message m.

Iterated attack

```
1 def LowExponent(NB,NC,ND,m):
2     pretty_print(html('The message is %s'%latex(m)))
3     cB=(m^3)%NB
```

```
    cC=(m^3)%NC
    cD=(m^3)%ND
    pretty_print(html('Bob receives: %s'%latex(cB)))
    pretty_print(html('Chris receives: %s'%latex(cC)))
    pretty_print(html('David receives: %s'%latex(cD)))
    M3=CRT_list([cB,cC,cD], [NB,NC,ND])
    M=(M3)^(1/3)
    pretty_print(html('The attacker obtains: %s'%latex(M)))
    return M
LowExponent(2257,2581,4223,123)
```

Output

```
The message is 123
Bob receives: 1099
Chris receives: 2547
David receives: 2747
The attacker obtains: 123
```

3.4 Wiener's attack

If the private exponent d is small compared to N, then there is an attack based on continued fractions. A finite continued fraction is defined as follows. Let $a_0 \in \mathbb{Z}$ and $a_1, a_2, \ldots, a_n \in \mathbb{N}$. The expression

$$[a_0; a_1, a_2, \ldots, a_n] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \frac{1}{a_n}}}}$$

is called a finite continued fraction. Clearly, $[a_0; a_1, a_2, \ldots, a_n]$ defines a rational number. If $u/v = [a_0; a_1, a_2, \ldots, a_n]$, then the rational numbers

$$[a_0], [a_0; a_1], [a_0; a_1, a_2], \ldots, [a_0; a_1, a_2, \ldots, a_n] = u/v$$

are called the convergents to the number u/v. For a given rational number u_0/v_0 with $\gcd(u_0, v_0) = 1$ the expansion can be computed via the Euclidean algorithm as follows

$$u_k = u_{k+1}a_k + u_{k+2} \quad \text{where } 0 \le u_{k+2} \le u_{k+1} - 1 \quad \text{for } k = 0, 1, \ldots, n-1$$

and $u_n = a_n$. For example in case of $1234/2019$ we have

$$[0;1,1,1,1,2,1,36,1,2] = 0 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{36 + \cfrac{1}{1 + \frac{1}{2}}}}}}}}}$$

and the convergents are given by

$[0]$	$[0,1]$	$[0,1,1]$	$[0,1,1,1]$	$[0,1,1,1,1]$	$[0,1,1,1,1,2]$	$[0,1,1,1,1,2,1]$
0	1	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{3}{5}$	$\frac{8}{13}$	$\frac{11}{18}$

$[0,1,1,1,1,2,1,36]$	$[0,1,1,1,1,2,1,36,1]$	$[0,1,1,1,1,2,1,36,1,2]$
$\frac{404}{661}$	$\frac{415}{679}$	$\frac{1234}{2019}$

In case of rational numbers we have the classical result (see e.g. Chapter X in [10]).

Theorem 3.3

Let $a, b, u, v \in \mathbb{Z}$ such that $\gcd(a,b) = \gcd(u,v) = 1$ and $1 \le b \le v$. If

$$\left|\frac{u}{v} - \frac{a}{b}\right| < \frac{1}{2b^2},$$

then a/b is a convergent of u/v.

If the private exponent d is small, then one can efficiently determine the factorization of N, a result due to Wiener [24].

Theorem 3.4. Wiener

Let $N = pq$ with primes p, q satisfying $q < p < 2q$. Let $d < 1/3N^{1/4}$. Given a public key (N, e) with $ed - 1 = k\varphi(N)$ one has

$$\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{1}{2d^2}.$$

Proof We have that $N = pq > q^2$, that is $q < \sqrt{N}$. It follows that $0 < N - \varphi(N) = p + q - 1 < 3q < 3\sqrt{N}$. Let us consider some approximations of e/N. We obtain that

$$\left|\frac{e}{N} - \frac{k}{d}\right| = \left|\frac{1 + k(\varphi(N) - N)}{dN}\right| \le \frac{3k}{d\sqrt{N}}.$$

Since $e < \varphi(N)$ and $k\varphi(N) = ed - 1 < ed$ we get that $k < d < 1/3N^{1/4}$. Hence it follows that

$$\left|\frac{e}{N} - \frac{k}{d}\right| \le \frac{1}{dN^{1/4}} < \frac{1}{2d^2}.$$

This is a very efficient algorithm for recovering the private exponent d. By the classical approximation relation we know that k/d is a convergent. It remains to determine which convergent's denominator is the secret key. One encrypts a phrase using the public key (N, e) and tries to decrypt it using N and the denominator of a convergent. Knowing d yields the value of k and from e, d and k one can easily compute $\varphi(N)$ and thus p and q. The following SageMath implementation determines d, p and q, that is the private key.

Wiener's attack

```
@interact
def Wiener(p1=input_box('2027',type = str, label='$p$'),
        q1=input_box('3571',type = str, label='$q$'),
        message="attack"):
    p=ZZ(p1)
    q=ZZ(q1)
    N=p*q
    phi=(p-1)*(q-1)
    d=Set([k for k in [2..floor(N^(1/4)/3)] if
     ↪  gcd(k,phi)==1]).random_element()
    e=(1/d)%phi
    pretty_print(html('Public key: (N,e)=(%s,%s)'%(late⌋
     ↪  x(N),latex(e))))
    pretty_print(html('Private key: (p,q,d)=(%s,%s,%s)'%(late⌋
     ↪  x(p),latex(q),latex(d))))
    cf=continued_fraction(e/N)
    conv=cf.convergents()
    m0=0
    M0=IntegerRing()([ord(k) for k in message],base=256)
    m1=M0.digits(base=N)
    encrypt=[power_mod(k, e, N) for k in m1]
    t=0
    while m0!=M0:
        if conv[t]!=0:
            k0=conv[t].numerator()
            d0=conv[t].denominator()
```

```
            print 'possible k and d:',k0,d0
            decrypt0=[power_mod(k, Integers()(d0), N) for k
            ↪  in encrypt]
            m0=IntegerRing()(decrypt0,base=N)
            t=t+1
        else:
            t=t+1
    print 'we have that (k,d)=',(k0,d0)
    phi0=(e*d0-1)//k0
    pretty_print(html(r'$\varphi(N)=\frac{ed-1}{k}=%s$'%late
    ↪  x(phi0)))
    poly=x^2+(phi0-N-1)*x+N
    pretty_print(html('$p$ is a solution of $%s$'%latex(p
    ↪  oly)))
    sol=solve(poly,x)
    print 'the solutions are:',sol[0].rhs(),',',sol[1].rhs()
```

Output

p 2027

q 3571

message attack

Public key: (N,e)=(7238417,5007337)

Private key: (p,q,d)=(2027,3571,13)

```
possible k and d: 1 1
possible k and d: 2 3
possible k and d: 9 13
we have that (k,d)= (9, 13)
```

$\varphi(N) = \frac{ed-1}{k} = 7232820$

p is a solution of $x^2 - 5598\,x + 7238417$

```
the solutions are: 3571 , 2027
```

3.5 Davida's attack

This special attack due to Davida [6] depends on the possibility to get access to recipient's garbage. The basic idea is that the attacker intercepts a ciphertext c and transforms it. The transformed ciphertext will be meaningless when the receiver decrypts

it. Therefore it will be discarded. The attacker can recover the original plaintext from the discarded one in a clever way. The steps of this attack are given below.

- The attacker knows the public key of Alice, let say (N, e) and also gets a ciphertext message c sent to Alice.
- The attacker uses an arbitrary number k and determines

$$c_1 \equiv c \cdot k^e \pmod{N}.$$

  This c_1 will be sent to Alice as a ciphertext.
- Alice uses her private key d to recover the plaintext that is she computes

$$m_1 \equiv c_1^d \pmod{N}.$$

  It is very likely to be a meaningless message, hence Alice will discard it.
- The attacker accesses the discarded plaintext m_1 and computes

$$m_1 \cdot k^{-1} \equiv c_1^d \cdot k^{-1} \equiv c^d \pmod{N}.$$

  Thus the attacker will know $c^d \pmod{N}$ corresponding to the original message.

DavidaAttack

```
def Davida(N,e,m,k):
    c=(m^e)%N
    pretty_print(html('The message is %s'%latex(m)))
    pretty_print(html('The ciphertext is %s'%latex(c)))
    c1=(c*k^e)%N
    pretty_print(html('The modified ciphertext is %s'%late⌋
     ↪  x(c1)))
    phiN=euler_phi(N)
    d=(1/e)%phiN
    m1=(c1^d)%N
    pretty_print(html('The discarded plaintext is %s'%latex(m⌋
     ↪  1)))
    M=(m1/k)%N
    pretty_print(html('The attacker obtains: %s'%latex(M)))
    return M
Davida(8137,23,100,37)
```

Output

```
The message is 100
The ciphertext is 7527
The modified ciphertext is 8128
The discarded plaintext is 3700
The attacker obtains: 100
```

3.6 Elliptic curve factorization

As we saw RSA cryptography is based on the difficulty of factoring large integers. There are many algorithms that can factor very large numbers of a certain form, however general purpose efficient algorithm is still unknown. Now we introduce Lenstra's elliptic curve factorization method [14].

An elliptic curve over a field K, char$K \neq 2, 3$ is given by as follows

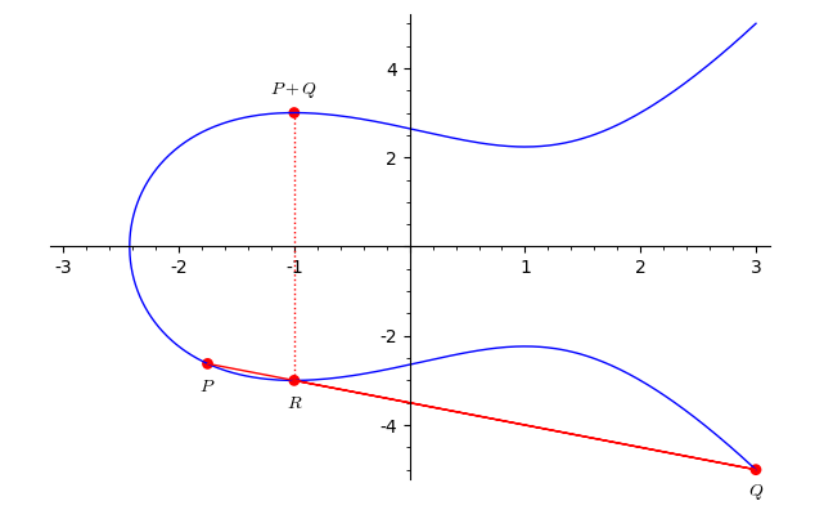
$$E: \quad y^2 = x^3 + ax + b \quad \text{with } a, b \in K, \text{ such that } 4a^3 + 27b^2 \neq 0$$

together with a point denoted by O, the so-called point at infinity. Let us see how to define a group law in a special case. Given two points P and Q on 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. As a concrete example consider the elliptic curve defined by

$$y^2 = x^3 - 3x + 7 \quad \text{and let } P = (-7/4, -21/8), Q = (3, -5).$$

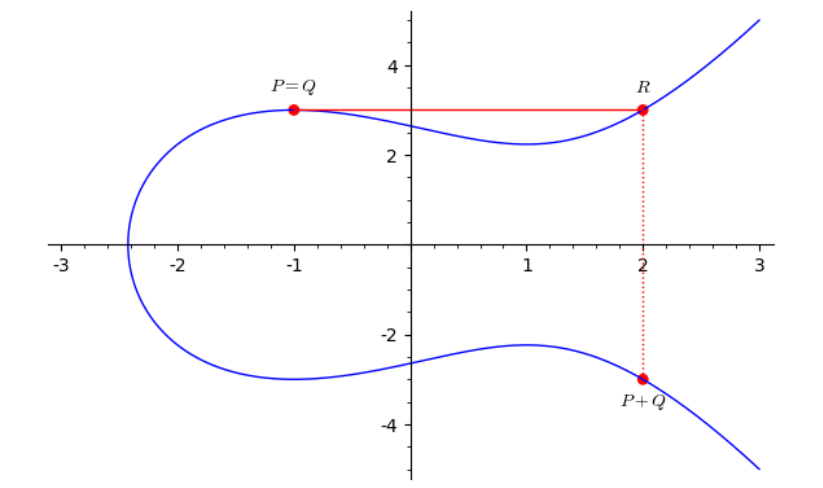
The third intersection point of the curve and the line is $R = (-1, -3)$. Therefore $P + Q = (-1, 3)$.

Group law I.



If $P = Q$ we consider the tangent line. If $P = (x, y)$ we define $-P = (x, -y)$ and we have $P + (-P) = 0, P + O = O + P = P$. Let us see this construction in case of the previously defined elliptic curve with the point $P = (-1, 3)$. In this case $R = (2, 3)$, hence $P + P = 2P = (2, -3)$.

Group law II.



By means of the geometric interpretation we may provide the sum of the two points $P = (x_1, y_1), Q = (x_2, y_2)$ as

$$P + Q = (x_3, m(x_1 - x_3) - y_1,$$

where $m = \frac{y_2 - y_1}{x_2 - x_1}$ if $P \neq Q$, $m = \frac{3x_1^2 + a}{2y_1}$ if $P = Q$ and $x_3 = m^2 - x_1 - x_2$. Lenstra's elliptic curve factorization method consists of computing $(B!)P$ on an elliptic curve $E \pmod N$ for some given point P and given integer B. If an error arises in the group law then it can be used to factor N. The advantage of the method is that we can change the

point and also the elliptic curve if no error arises.

EllipticCurveFactorization

```
def GroupLaw(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]
def multP(n,P,a):
    P0 = [0,1,0]
    nP = P
    while n!=0:
        if (n%2)==1:
            P0 = GroupLaw(nP,P0,a)
        n//=2
        nP = GroupLaw(nP,nP,a)
    return P0
def ECFactor(N,A,B):
    RN=Integers(N)
```

```
    for a in range(A):
        if (4*a^3+27)%N==0:
            continue
        point=[RN(0),RN(1),RN(1)]
        for b in range(B):
            point = multP(b+1,point,a)
            if point[2]==0:
                break
            if point[2]>1:
                print a,b
                return gcd(point[2],N)
    print 'failed'
p1=next_prime(123456789)
p2=next_prime(987654321)
N=p1*p2
print N
ECFactor(N,100,20)
```

Output

```
121932633334857493
49 18
123456791
```

In this example we have $N = 121932633334857493$ and it turns out that that the elliptic curve factorization algorithm works, for example taking the curve $y^2 = x^3 + 49x + 1$ and the point $P = (0, 1)$ we obtain that N is divisible by 123456791, in fact

$$N = 123456791 \cdot 987654323.$$

3.7 Continued fraction factorization method

It was the first integer factoring algorithm with a sub-exponential running time. The basic idea behind the algorithm is that if we have two integers x and y for which

$$x^2 \equiv y^2 \pmod{N},$$

then N divides $x^2 - y^2 = (x - y)(x + y)$ and there is a chance that $\gcd(N, x - y) \neq 1$ or $\gcd(N, x + y) \neq 1$. So we compute the continued fraction expansion of $\sqrt{N}$ and the

convergents, denoted by P_n/Q_n for an integer n. Here we have that

$$|P_n^2 - NQ_n^2| = Q_n^2 \left| \frac{P_n}{Q_n} - \sqrt{N} \right| \left| \frac{P_n}{Q_n} + \sqrt{N} \right| < \left| \frac{P_n}{Q_n} + \sqrt{N} \right| < 2\sqrt{N} + 1.$$

It follows that $C_n = P_n^2 - NQ_n^2$ is relatively "small", a fact that can be used to search for numbers having only small prime divisors. We clearly have that $C_n \equiv P_n^2 \pmod{N}$. We generate congruences for which C_n has only prime divisors less than or equal B (these are the so-called B-smooth numbers) for some fixed integer B. That is we obtain

$$P_i^2 \equiv C_i = \prod_{p<B} p^{a_{i,p}} \pmod{N}$$

for $i = 1, 2, \ldots, s$. We would like to determine a subset S of $\{1, 2, \ldots, s\}$ such that

$$\prod_{i \in S} P_i^2 \equiv \prod_{i \in S} C_i = \prod_{p<B} p^{\sum_{i \in S} a_{i,p}} \pmod{N},$$

where $p^{\sum_{i \in S} a_{i,p}}$ is even for all $p < B$. This latter part is linear algebra modulo 2. In SageMath a possible implementation is as follows.

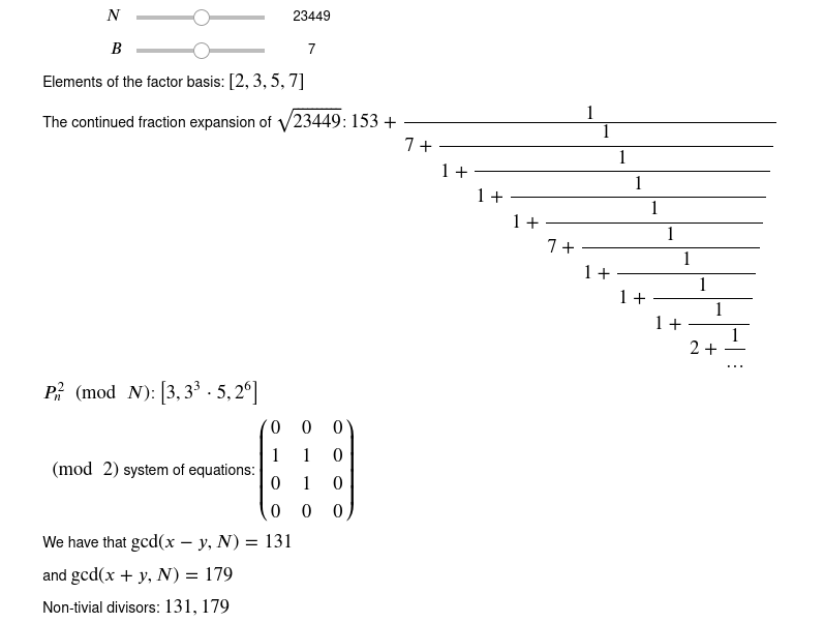
ContinuedFractionFactorization

```
@interact
def _(n=('$N$',23449),B=('$B$',7)):
    w=[]
    for i in [1..B]:
        if is_prime(i):
            w=w + [i]
    pretty_print(htm⌋
    ↪ l('Elements of the factor basis: $%s$'%latex(w)))
    sn=sqrt(n)
    cfn=continued_fraction(sn)
    t=1
    x=cfn.numerator(t)
    pretty_print(htm⌋
    ↪ l('The continued fraction expansion of $ \sqrt{%s} $: $%s$'
    ↪ %(latex(n), latex(continued_fraction(sqrt(n))))))
    A=[]
    K=[]
    T=True
    while T:
        x=cfn.numerator(t)
```

```
        a=(x^2)%n
        p=a/prod([k^valuation(a,k) for k in w])
        if p==1:
            q=[valuation(a,k) for k in w]
            A.append(q)
            K.append(x)
            Q=matrix(GF(2),A).transpose()
            if Q.right_kernel().dimension()>0:
                for Qv in Q.right_kernel().basis():
                    Av=sum([vector(A[k]) for k in
                    ↪  [0..len(A)-1] if Qv[k]==1])
                    av=prod([w[s]^Av[s] for s in
                    ↪  [0..(len(w)-1)]]).nth_root(2)
                    Kv=prod([K[k] for k in [0..len(A)-1] if
                    ↪  Qv[k]==1])
                    d1=gcd(Kv-av,n)
                    d2=gcd(Kv+av,n)
                    if Set([1,n])!=Set([d1,d2]) and
                    ↪  Set([d1,d2])!=Set([1]) and
                    ↪  Set([d1,d2])!=Set([n]):
                        T=False
                        pretty_print(htm⌋
                        ↪  l('$P_n^2\pmod{N}$: $%s$'
                        ↪  %latex([(k^2%n).factor() for k in
                        ↪  K])))
                        pretty_print(htm⌋
                        ↪  l('$\pmod{2}$ system of equations: $%s$'
                        ↪  %latex(Q)))
                        pretty_print(htm⌋
                        ↪  l('We have that $ \gcd(x-y,N) = %s$ '
                        ↪  %latex(d1)))
                        pretty_print(htm⌋
                        ↪  l('and $ \gcd(x+y, N) = %s $'
                        ↪  %latex(d2)))
```

```
                    pretty_print(htm
                    ↪ l('Non-tivial divisors: $ %s, %s$ '
                    ↪ %(latex(n),latex(d1),latex(d2))))
            t=t+1
        else:
            t=t+1
```

Output



3.8 Dixon's method

The continued fraction factorization method is a variant of Dixon's algorithm, the only difference is the way to obtain appropriate congruence relations. Here we take a sequence x_n of distinct random integers for which $\sqrt{N} < x_n < N$. We search for congruences

$$x_i^2 \equiv \prod_{p<B} p^{a_{i,p}} \pmod N$$

for $i = 1, 2, \ldots, s$, where B is a bound for the element of the factor base like in case of the continued fraction factorization.

Dixon

```
@interact
def _(n=('$N$',4279339),B=('$B$',13)):
    w=[]
    for i in [1..B]:
        if is_prime(i):
            w=w + [i]
    x=ceil(sqrt(n))
    a=(x^2)%n
    q=[valuation(a,k) for k in w]
    A=[]
    K=[]
    T=True
    while T:
        p=0
        while p!=1:
            x += 1
            a=(x^2)%n
            p=a/prod([k^valuation(a,k) for k in w])
            q=[valuation(a,k) for k in w]
        K.append(x)
        A.append(q)
        p=0
        Q=matrix(GF(2),A).transpose()
        if Q.right_kernel().dimension()>0:
            Qv=sum(Q.right_kernel().basis())
            Av=sum([vector(A[k]) for k in [0..len(A)-1] if
             ↪  Qv[k]==1])
            av=prod([w[s]^Av[s] for s in
             ↪  [0..(len(w)-1)]]).nth_root(2)
            Kv=prod([K[k] for k in [0..len(A)-1] if Qv[k]==1])
            d1=gcd(Kv-av,n)
            d2=gcd(Kv+av,n)
            if Set([1,n])!=Set([d1,d2]):
```

```
            T=False
        DH=[(k,n//k) for k in [d1,d2] if k!=1 and k!=n]
pretty_print(htm」
↪ l('Elements in the factor basis $B$: $%s$'%latex(w)))
pretty_print(html('Exponent vectors: $%s$ '%latex(A)))
pretty_print(htm」
↪ l('The modulo 2 matrix is given by $%s$'%latex(Q)))
pretty_print(htm」
↪ l('We obtain that $\displaystyle \sqrt{\prod_{ b_i \in B }
b_i^{ a_{i,j} }}$  = $ %s $' %latex(av) ) )
pretty_print(htm」
↪ l('We also have that $\displaystyle \prod_{i=1}^k p_i $ =
 $ %s$' %latex(Kv)))
pretty_print(htm」
↪ l(r'Compute $\gcd\displaystyle \left( \prod_{i=1}^k p_i \pm
\sqrt{\prod_{ b_i \in B} b_i^{a_{i,j}}} , N \right): $ '))
pretty_print(html('$\gcd(%s+%s,%s)$ and $\gcd(%s-%s,%s)$'
↪ %(latex(Kv), latex(av), latex(n), latex(Kv),
↪ latex(av), latex(n))))
pretty_print(html('Non-trivial divisors of $N:$  $%s$'
↪ %latex(DH[0])))
```

Output

N 4279339

B 13

Elements in the factor basis B: $[2, 3, 5, 7, 11, 13]$

Exponent vectors: $[[1, 2, 0, 2, 1, 0], [1, 2, 1, 1, 2, 0], [0, 1, 2, 2, 1, 1], [1, 3, 2, 2, 0, 1]]$

The modulo 2 matrix is given by $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$

We obtain that $\sqrt{\prod_{b_i \in B} b_i^{a_{i,j}}} = 66216150$

We also have that $\prod_{i=1}^{k} p_i = 10291345744$

Compute $\gcd\left(\prod_{i=1}^{k} p_i \pm \sqrt{\prod_{b_i \in B} b_i^{a_{i,j}}}, N\right)$:

$\gcd(10291345744 + 66216150, 4279339)$ and $\gcd(10291345744 - 66216150, 4279339)$

Non-trivial divisors of N : $(433, 9883)$

3.9 Pollard's ρ factorization

John Pollard in 1975 introduced a factorization method in the article "A Monte Carlo method for factorization" [18]. This factorization algorithm is based on congruences using a polynomial and involves some probabilistic ideas. Brent and Pollard [4] applied this method to factor the eighth Fermat number

$$2^{2^8} + 1.$$

Let us describe the algorithm to factor an integer N. Let $f(x)$ be a polynomial (usually we take $f(x) = x^2 + c$ for some integer c). We define two sequences x_n and y_n as follows. Let $x_0 = y_0 = 2$ and

$$\begin{aligned} x_{n+1} &\equiv f(x_n) \pmod{N}, \\ y_{n+1} &\equiv f(f(y_n)) \pmod{N}. \end{aligned}$$

We iterate these sequences until

$$g = \gcd(|y_n - x_n|, N) > 1.$$

If $g < N$, then we determined a non-trivial divisor of N (since by construction $g > 1$).

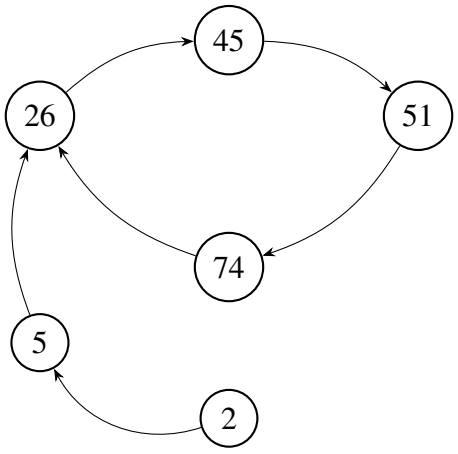
Otherwise the procedure fails, we get that $g = N$. In case of $N = 6557$ we have

$x_n \mod N$	$x_n \mod 79$	$y_n \mod 79$
2	2	2
5	5	26
26	26	51
677	45	26
5897	51	51
2839	74	26
1369	26	51
5417	45	26
1315	51	51
4735	74	26
1843	26	51

.

Since there are only finitely many possible values modulo 79 the two sequences x_n and y_n will cycle. For example we see that $x_n \pmod{79}$ will follow the graph given below. We also see that $x_4 \equiv y_4 \pmod{79}$ and the method gives that

$$\gcd(|5897 - 1315|, 6557) = 79.$$



PollardRhoFactorization

```
@interact
def _(n=('$n$',4279339)):
    def fv(x,n): return mod(x^2-1,n)
    a=2
    b=2
    d=1
    rows = []
```

```
    while d == 1:
        a=fv(a,n)
        b=fv(fv(b,n),n)
        d=gcd(a-b,n)
        rows.append([a,b,d])
    pretty_print(htm
    ↪ l("We use the quadratic function $%s$"%latex(x^2-1)))
    pretty_print(table(rows,header_r
    ↪ ow=["$x_n$","$y_n$","$\gcd$"],frame=True))
    if d == n:
        print("trivial divisor:",d)
        return -1
    else:
        print (n,"is divisible by",d)
```

Output

n 4279339

We use the quadratic function $x^2 - 1$

x_n	y_n	gcd
3	8	1
8	3968	1
63	3562378	1
3968	2493195	1
2907006	1734297	1
3562378	2583494	1
3154179	3104366	1
2493195	1492448	1
3253489	24139	1
1734297	3548216	1
1315990	2156556	1
2583494	237473	1
3282464	2669342	1
3104366	1731891	1
3995938	3008265	433

```
4279339 is divisible by 433
```

We note that changing the quadratic function may help to reduce the time of the computation. For example if we try to factor the same integer by using the function $f(x) = x^2 + x + 11$ instead of $f(x) = x^2 - 1$ we obtain the following.

Output

n 4279339

We use the quadratic function $x^2 + x + 11$

x_n	y_n	gcd
17	317	1
317	738192	1
100817	4221677	1
738192	2601824	433

```
4279339 is divisible by 433
```

3.10 SageMath summary

function	description
IntegerRing()	Returns the integer ring.
Integers()	Returns the integer ring.
ceil()	The ceiling function.
continued_fraction()	Returns the continued fraction.
convergents()	Returns the (partial) convergents of a number.
denominator()	Returns the denominator of a fraction.
digits()	Returns the digits of a number in a given base.
dimension()	Returns the dimension.
is_prime()	Returns true if the number is prime, false otherwise.
is_unit()	Returns true if the element is a unit.
next_prime()	Returns the next prime number.
nth_root()	Returns the (possibly truncated) n-th root.
numerator()	Returns the numerator of a fraction.
power_mod()	Returns the n-th power of a modulo the integer m.
right_kernel()	Computes the space of vectors w so that $A \cdot w = 0$.
solve()	Solves certain types of equations/systems of equations.
sqrt()	Returns the square root of an element.
transpose()	Returns the transpose of a matrix.
valuation()	Returns the valuation.

Exercises

1. In RSA we have $(n, e) = (5352499, 3516607)$. Encrypt the message "The only way to learn mathematics is to do mathematics".
2. In RSA we know $(p, q, d) = (12227, 35569, 136215539)$, and we receive the encrypted message

   $[158079363, 173377019, 373536605, 97680494, 144518909, 1942499, 413795444, 147133032]$.

   Determine the original message.
3. Factor the integer $N = 5352499$ by using elliptic curves.
4. Factor the integer $N = 5352499$ by using continued fractions.
5. Factor the integer $N = 5352499$ by applying the quadratic sieve.

6. Factor the integer $N = 5352499$ by applying Pollard's ρ algorithm with 2 different quadratic functions.
7. In RSA we know $(N, e_1, e_2, c_1, c_2) = (8137, 7, 23, 7155, 2626)$. Apply the common modulus attack to recover the secret message.
8. In RSA apply the iterated encryption attack to obtain a list containing the possible values of the secret message. We have that $(N, e, c) = (1591, 5, 22)$.
9. In RSA we have $(N, e) = (24739307, 7526327)$. Apply Wiener's attack to recover the secret key d.
10. Bob, Chris and David have RSA public keys given by $(N_B, e_B) = (6527, 7)$, $(N_C, e_C) = (11537, 7)$ and $(N_D, e_D) = (10123, 7)$, respectively. Alice sends the same message to both of them, the ciphertexts are as follows $c_B = 2268, c_C = 3442$ and $c_D = 4737$. Determine the message by means of the low public exponent attack.

Chapter 4 The Rabin and Paillier cryptosystems

Summary

4.1 The Rabin cryptosystem

There is an other cryptosystem based on the difficulty of factoring large integers due to Rabin. If $N = p \cdot q$ is a product of two large primes, then extracting square roots modulo N is a challenging computational task. The encryption and decryption is similar to the one used in RSA, however in case of the RSA exponent e we have that $\gcd(e, \varphi(N)) = 1$. Here the exponent is fixed, it is 2. The steps involved in the encryption are as follows.

- Alice obtains Bob's public key: N,
- Alice represents the message as a number m from the set $\{0, 1, \ldots, N-1\}$,
- Alice sends to Bob the value $c \equiv m^2 \pmod{N}$.

The decryption works as described below.

- Bob computes the square root of c modulo N.
- There are 4 roots, hence it is important to have some redundancy in the message, for example the least significant k bits of the binary representation of m are all ones.

The decryption is especially easy if we choose primes p and q such that $p \equiv q \equiv 3 \pmod 4$. In this case Euler's criterion provides that

$$(\pm c^{(p+1)/4})^2 \equiv c^{(p+1)/2} \equiv c^{(p-1)/2} \cdot c \equiv c \pmod{p}$$

and similarly for q we get that

$$(\pm c^{(q+1)/4})^2 \equiv c^{(q+1)/2} \equiv c^{(q-1)/2} \cdot c \equiv c \pmod{q}.$$

Having two square roots modulo p and other two modulo q yields four square roots modulo N by applying the Chinese Reminder Theorem.

Suppose that Bob's public key is $N = 11 \cdot 19 = 209$ and Alice wants to send the

message $m = 79$. Alice computes

$$c \equiv 79^2 \equiv 180 \pmod{N}.$$

Bob receives $c = 180$ and determines the possible square roots, these are given by

$$\begin{aligned} c^{(11+1)/4} &\equiv 9 \pmod{11}, \\ -c^{(11+1)/4} &\equiv 2 \pmod{11}, \\ c^{(19+1)/4} &\equiv 16 \pmod{19}, \\ -c^{(19+1)/4} &\equiv 3 \pmod{19}. \end{aligned}$$

Finally, we determine the solutions of the system of congruences by means of the Chinese Reminder Theorem:

$$\begin{aligned} x \equiv 9 \pmod{11} \quad x \equiv 16 \pmod{19}, \\ x \equiv 9 \pmod{11} \quad x \equiv 3 \pmod{19}, \\ x \equiv 2 \pmod{11} \quad x \equiv 16 \pmod{19}, \\ x \equiv 2 \pmod{11} \quad x \equiv 3 \pmod{19}. \end{aligned}$$

We obtain that

$$35^2 \equiv 79^2 \equiv 130^2 \equiv 174^2 \equiv 180 \pmod{N}.$$

Rabin

```
p=1
while not p%4==3:
    p=random_prime(1000,lbound=100)
q=1
while not q%4==3:
    q=random_prime(1000,lbound=100)
N=p*q
pretty_print(html('The public key is %s'%latex(N)))
m=randint(2,N)
pretty_print(html('The message is %s'%latex(m)))
c=(m^2)%N
pretty_print(html('The encrypted message is %s'%latex(c)))
s1=(c^((p+1)/4))%p
s2=p-s1
s3=(c^((q+1)/4))%q
```

```
s4=q-s3
S=[s1,s2,s3,s4]
pretty_print(html('The four square roots are $%s$'%latex(S)))
CRT1=crt(s1,s3,p,q)
CRT2=crt(s1,s4,p,q)
CRT3=crt(s2,s3,p,q)
CRT4=crt(s2,s4,p,q)
possible=[CRT1,CRT2,CRT3,CRT4]
pretty_print(html('The four possible messages are $%s$'%late
    x(possible)))

```

Output

```
The public key is 83261
The message is 68246
The encrypted message is 62698
The four square roots are [136,3,40,559]
The four possible messages are [82702,68246,15015,559]
```

4.2 Paillier's cryptosystem

In 1999 Paillier published an article [16] in which he introduced a cryptosystem based on composite residuosity classes. As in case of the RSA cryptosystem we fix two large primes p and q. The public key is given by (N, g), where $N = p \cdot q$ and g is an element of $\mathbb{Z}_{N^2}^*$. Let us denote by λ the Carmichael function ($\lambda(p \cdot q) = \text{lcm}(p-1, q-1)$) and

$$\begin{aligned} L : \mathbb{Z}_{N^2}^* &\to \mathbb{Z}_N \\ u &\longmapsto \frac{u-1}{N}. \end{aligned}$$

The element g has to satisfy that $\gcd(N, L(g^\lambda \bmod N^2)) = 1$. To encrypt a message Alice follows the steps given below.

- The message m is represented as an element of $\mathbb{Z}_N$,
- Alice picks a random number $r \in \mathbb{Z}_N^*$,
- the ciphertext is in the set $\mathbb{Z}_{N^2}$, it is computed via the formula

$$c \equiv g^m \cdot r^N \pmod{N^2}.$$

Bob can decrypt the ciphertext using his private key (p, q) as follows:

$$m \equiv \frac{L(c^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \pmod{N}.$$

To show that the decryption in fact works we need a theorem of Carmichael.

Theorem 4.1. Carmichael

If a and N are relatively prime numbers, then

$$a^{\lambda(N)} \equiv 1 \pmod{N}.$$

The element g comes from $\mathbb{Z}^*_{N^2}$, so $g^{\lambda(N)} \equiv 1 \pmod{N}$. Therefore there exists a $k \in \mathbb{Z}_N$ such that

$$g^{\lambda(N)} \equiv 1 + kN \pmod{N^2}.$$

We also need to compute $c^{\lambda(N)} \pmod{N^2}$. Here we obtain

$$c^{\lambda(N)} \equiv g^{m\lambda(N)} \cdot r^{N\lambda(N)} \pmod{N^2}.$$

Clearly we have $\varphi(N^2) = N\varphi(N) = N(p-1)(q-1)$, therefore $r^{N\lambda(N)} \equiv 1 \pmod{N^2}$. That is we get that

$$c^{\lambda(N)} \equiv g^{m\lambda(N)} \pmod{N^2}.$$

Putting together the above results we have

$$c^{\lambda(N)} \equiv (1 + kN)^m \equiv 1 + kmN \pmod{N^2}.$$

The remaining part is as follows

$$\frac{L(c^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \equiv \frac{km}{k} \equiv m \pmod{N}.$$

Consider an example with $N = p \cdot q = 17 \cdot 29 = 493$ and $g = 35$. Alice wants to send the message $m = 28$ to Bob and she chooses the random integer $r = 9$. Alice computes

$$c \equiv 35^{28} \cdot 9^{493} \equiv 43562 \pmod{N^2}.$$

Bob may decrypt it by evaluating $L(c^\lambda \bmod N^2)$ and $L(g^\lambda \bmod N^2)$. He obtains

$$\begin{aligned} L(c^\lambda \bmod N^2) &= 313, \\ L(g^\lambda \bmod N^2) &= 64. \end{aligned}$$

So it remains to determine

$$\frac{L(c^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \equiv \frac{313}{64} \equiv 28 \pmod{N},$$

we see that the message is decrypted.

Paillier

```
p=random_prime(1000,lbound=100)
q=random_prime(1000,lbound=100)
N=p*q
lpq=lcm(p-1,q-1)
a1=0
while not gcd(N,a1)==1:
    a1=randint(2,N)
a2=0
while not gcd(N,a2)==1:
    a2=randint(2,N)
g=((1+a1*N)*a2^N)%(N^2)
pretty_print(html('The public key is $[N,g]=%s$'%late
↪ x([N,g])))
m=randint(2,N)
pretty_print(html('The message is %s'%latex(m)))
r=0
while not gcd(N,r)==1:
    r=randint(2,N)
pretty_print(html('The random integer $r$ is %s'%latex(r)))
c=(g^m*r^N)%(N^2)
pretty_print(html('The encrypted message is %s'%latex(c)))
Lc=((c^lpq)%(N^2)-1)/N
Lg=((g^lpq)%(N^2)-1)/N
pretty_print(html(r'$L(c^{\lambda}\pmod{N^2})=%s$'%latex(Lc)))
pretty_print(html(r'$L(g^{\lambda}\pmod{N^2})=%s$'%latex(Lg)))
pretty_print(html('The decrypted message is $%s$'%late
↪ x((Lc/Lg)%N)))

```

Output

The public key is $[N, g] = [169511, 7504933656]$

The message is 113006

The random integer r is 1744

The encrypted message is 15521868369

$L(c^{\lambda} \pmod{N^2}) = 152763$

$L(g^{\lambda} \pmod{N^2}) = 12561$

The decrypted message is 113006

4.3 SageMath summary

function	description
`crt()`	Applies the Chinese Reminder Theorem.
`lcm()`	Returns the least common multiple.
`randint()`	Returns a random integer from a range.
`random_prime()`	Returns a random prime number from a range.

Exercises

1. Bob uses a Rabin public-key cryptosystem with $N = 1817 = 23 \cdot 79$. Alice tells Bob that the two-digit message will be padded with starting digits "11" and she sends 882 as encrypted message. Decode the message.
2. The Paillier's cryptosystem has an additive homomorphic property that we check in case of a concrete example. Let $(N, g) = (2501, 92)$ and $(m_1, r_1) = (34, 5)$, $(m_2, r_2) = (16, 7)$. Compute the two encoded messages c_1, c_2 and show that $c_1 \cdot c_2$ is the same as the ciphertext of $m_1 + m_2$ with random integer $r = 35$.

Chapter 5 Applications of the discrete logarithm problem

Summary

- Diffie-Hellman key exchange
- ElGamal cryptosystem
- Massey-Omura encryption
- The AA_β cryptosystem
- SageMath summary
- Exercises

The discrete logarithm problem can be stated as follows (in multiplicative and in additive groups as well). If G is a multiplicative group, then we try to find x for which

$$g^x = h, \quad g, h \in G.$$

If H is an additive group, then we look for n such that

$$nP = Q, \quad P, Q \in H.$$

Computing discrete logarithms is a computationally difficult problem. So it is natural to use this problem as the basis for cryptographic protocols.

5.1 Diffie-Hellman key exchange

In 1976 Diffie and Hellman published a key exchange procedure. The goal of the Diffie-Hellman key exchange algorithm is as follows. Alice and Bob want to share a secret key for use in a cipher. However their only means of communication is insecure. The difficulty of the discrete logarithm problem provides here a possible solution. The first step is for Alice and Bob to agree on a large prime p and a nonzero g mod p. Alice and Bob make the values of p and g public.

- Alice picks a secret integer a and computes g^a mod p,
- Bob picks a secret integer b and computes g^b mod p,
- Alice sends g^a to Bob,
- Bob sends g^b to Alice,
- Alice and Bob use their secret keys to compute $(g^b)^a$ mod p and $(g^a)^b$ mod p, respectively. The common value

$$g^{ab} \equiv (g^b)^a \equiv (g^a)^b \pmod{p}$$

is their exchanged key.

The Diffie-Hellman problem is the computation of the value $g^{ab} \bmod p$ from the known values of $g^a \bmod p$ and $g^b \bmod p$.

The Diffie-Hellman key exchange can be generalized to any number of participants. Let us consider the case with three participants. Alice, Bob and Carol wish to construct a shared key together. They agree on a fixed prime p and primitive root $g \bmod p$, and choose their own encryption exponents a, b and c.

- Alice publishes g^a, Bob publishes g^b and Carol makes public g^c modulo p.
- Alice uses g^b and g^c to compute g^{ab} and g^{ac} modulo p. Bob computes $g^{bc} \bmod p$.
- Alice sends g^{ab} to Carol and Carol computes $(g^{ab})^c$. Alice sends g^{ac} to Bob and he computes $(g^{ac})^b$.
- Bob sends g^{bc} to Alice and she uses it to get $(g^{bc})^a$.
- The secret key is

$$g^{abc} \equiv (g^{bc})^a \equiv (g^{ac})^b \equiv (g^{ab})^c \pmod{p}.$$

The Diffie-Hellman key exchange can be used with additive groups as well. Alice and Bob agree on a finite field $\mathbb{F}$ and an elliptic curve over it and also they fix a point on the elliptic curve, let say P.

- Alice chooses a secret value a and computes $[a]P$,
- Bob chooses a secret value b and computes $[b]P$,
- Alice sends $[a]P$ to Bob and he computes $[b]([a]P)$,
- Bob sends $[b]P$ to Alice and she computes $[a]([b]P)$.
- The secret key is

$$[ab]P = [b]([a]P) = [a]([b]P).$$

Let us consider a multiplicative group example in SageMath.

Diffie-Hellman

```
p=random_prime(10000,false,1000)
F = GF(p)
g = F(primitive_root(p))
pretty_print(htm
↪  l('The publicly known values: $p=%s$ and $g=%s$'%(latex(p
↪  ),latex(g))))
a=F.random_element()
b=F.random_element()
```

```
pretty_print(html('The secret value of Alice: $a=%s$'%late
    x(a)))
pretty_print(html('Bob picks the value: $b=%s$'%latex(b)))
pretty_print(html('Alice computes $g^a$: $g^a=%s$'%late
    x(g^a)))
pretty_print(html('Bob computes $g^b$: $g^b=%s$'%latex(g^b)))
pretty_print(html('Alice determines $(g^b)^a$: $%s$'%late
    x((g^b)^a)))
pretty_print(html('Bob determines $(g^a)^b$: $%s$'%late
    x((g^a)^b)))
key=(g^a)^b
pretty_print(html('The common value is: $%s$'%latex(key)))

```

Output

```
The publicly known values: p=8521 and g=13
The secret value of Alice: a=749
Bob picks the value: b=5157
Alice computes g^a: g^a=5312
Bob computes g^b: g^b=7286
Alice determines (g^b)^a: 1100
Bob determines (g^a)^b: 1100
The common value is: 1100
```

We also provide an example using elliptic curves.

Diffie-HellmanEC

```
p=random_prime(10000,false,1000)
F = GF(p)
A=F.random_element()
B=F.random_element()
E=EllipticCurve([A,B])
P=E.random_element()
pretty_print(htm
    l('The publicly known objects: $p=%s, E=%s$ and $P=%s$'%(late
    x(p),latex(E),latex(P))))
```

```
a=ZZ(F.random_element())
b=ZZ(F.random_element())
pretty_print(html('The secret value of Alice: $a=%s$'%late
↪ x(a)))
pretty_print(html('Bob picks the value: $b=%s$'%latex(b)))
pretty_print(html('Alice computes $[a]P$: $%s$'%latex(a*P)))
pretty_print(html('Bob computes $[b]P$: $%s$'%latex(b*P)))
pretty_print(html('Alice determines $[a]([b]P)$: $%s$'%late
↪ x((a*b*P))))
pretty_print(html('Bob determines $[b]([a]P)$: $%s$'%latex((b
↪ *a*P))))
key=a*b*P
pretty_print(html('The common value is: $%s$'%latex(key)))

```

Output

```
The publicly known objects: p=8423,E=y^2=x^3+1270x+6029
and P=(4862:5749:1)
The secret value of Alice: a=2175
Bob picks the value: b=5954
Alice computes [a]P: (4195:2098:1)
Bob computes [b]P: (5839:6553:1)
Alice determines [a]([b]P): (5779:7571:1)
Bob determines [b]([a]P): (5779:7571:1)
The common value is: (5779:7571:1)
```

5.2 ElGamal cryptosystem

In 1985 Taher ElGamal [7] described a public key cryptosystem based on the discrete logarithm problem. Let us provide some details of the procedure. Bob creates a key and publish it using the following steps.

- Bob selects a large prime p and a primitive root $g \pmod{p}$.
- Bob chooses a random element a such that $1 < a < p - 1$.

- Bob computes g^a and the public key is given by

$$(p, g, g^a).$$

To encrypt a message M Alice needs to break it into blocks $(m_1, m_2, \ldots)$ such that $0 \leq m_i \leq p-1$. There are many ways of doing this. Alice also picks a random exponent $0 < k < p-1$ and computes $g^k \pmod p$ and $m_i \cdot (g^a)^k \pmod p$. Finally, Alice sends g^k and $m_i \cdot (g^a)^k$ to Bob. To decrypt the ciphertext Bob computes

$$(g^k)^{p-1-a} \equiv g^{-ak} \pmod p.$$

From here Bob only needs to determine

$$(m_i \cdot (g^a)^k) \cdot g^{-ak} \pmod p.$$

In SageMath we can follow these steps based on the code given below.

ElGamal

```
p=random_prime(10000,false,1000)
F = GF(p)
g = F(primitive_root(p))
S=Set([2..p-2])
Sm=Set([0..p-1])
a=S.random_element()
pretty_print(htm
↪ l('The publicly known values: $p=%s,g=%s$ and $g^a=%s$' %
↪ (latex(p) , latex(g) , latex(g^a))))
pretty_print(html('The secret exponent is $a=%s$'%latex(a)))
m=Sm.random_element()
k=S.random_element()
pretty_print(htm
↪ l('Alice wants to send the message: $m=%s$'%latex(m)))
pretty_print(htm
↪ l('Alice chooses a random exponent: $k=%s$'%latex(k)))
pretty_print(htm
↪ l(r'Alice sends $g^k=%s$ and $m\cdot (g^a)^k=%s$' %
↪ (latex(g^k) , latex(m*(g^a)^k))))
pretty_print(html('Bob computes $(g^k)^{p-1-a}=%s$'
↪ %latex((g^k)^(p-1-a))))
m0=(g^k)^(p-1-a)*m*(g^a)^k
```

```
pretty_print(htm⌋
↪ l(r'Bob obtains the plaintext via $(g^k)^{p-1-a} \cdot
(m(g^a)^k)=%s$' %latex(m0)))

```

Output

The publicly known values: $p = 4657$, $g = 15$ and $g^a = 4239$

The secret exponent is $a = 1631$

Alice wants to send the message: $m = 4087$

Alice chooses a random exponent: $k = 2851$

Alice sends $g^k = 3241$ and $m \cdot (g^a)^k = 3594$

Bob computes $(g^k)^{p-1-a} = 3444$

Bob obtains the plaintext via $(g^k)^{p-1-a} \cdot (m(g^a)^k) = 4087$

Let us see now how to provide an ElGamal cryptosystem using elliptic curves over finite fields.

- Bob chooses a finite field $\mathbb{F}_q$ and an elliptic curve $E : y^2 = x^3 + Ax + B$ over this finite field. These are public. The order of the Mordell-Weil group is denoted by N. The private key is a and the public key is (E, P, aP), where P is a point on the elliptic curve.
- Alice fixes a random exponent k and computes $kP, m + k(aP)$, where m is the message represented as a point on the elliptic curve.
- Alice send kP and $m + k(aP)$ to Bob.
- Bob determines $-a(kP)$ and than computes $(m + k(aP)) - a(kP) = m$.

ElGamalElliptic

```
p=random_prime(10000,false,1000)
F = GF(p)
A=F.random_element()
B=F.random_element()
E=EllipticCurve([A,B])
N=E.order()
```

```
P=E.gens()[0]
S=Set([2..N-1])
a=S.random_element()
pretty_print(htm
    ↪ l('The publicly known objects: $p=%s, E=%s$$'%(latex(p
    ↪ ),latex(E))))
pretty_print(htm
    ↪ l('The two given points $P=%s, [a]P=%s$'%(latex(P),late
    ↪ x(a*P))))
pretty_print(html('The private key is $a=%s$'%latex(a)))
m=E.random_element()
k=S.random_element()
pretty_print(htm
    ↪ l('Alice wants to send the message: $m=%s$'%latex(m)))
pretty_print(htm
    ↪ l('Alice chooses a random integer: $k=%s$'%latex(k)))
pretty_print(htm
    ↪ l(r'Alice sends $[k]P=%s$ and $m+[k]([a]P)=%s$'%(late
    ↪ x(k*P),latex(m+k*a*P))))
pretty_print(html('Bob computes $-[a]([k]P)=%s$'%latex(-
    ↪ a*k*P)))
m0=m+k*a*P-a*k*P
pretty_print(html(r'Bob obtains the plaintext $%s$'%latex(m
    ↪ 0)))

```

Output

The publicly known objects: $p = 9901$, $E = y^2 = x^3 + 8471x + 4389$

The two given points $P = (5163 : 5552 : 1)$, $[a]P = (3083 : 174 : 1)$

The private key is $a = 71$

Alice wants to send the message: $m = (3266 : 4715 : 1)$

Alice chooses a random integer: $k = 3509$

Alice sends $[k]P = (9602 : 5250 : 1)$ and $m + [k]([a]P) = (2404 : 4241 : 1)$

Bob computes $-[a]([k]P) = (5544 : 1366 : 1)$

Bob obtains the plaintext $(3266 : 4715 : 1)$

5.3 Massey-Omura encryption

Massey-Omura encryption is based on the discrete logarithm problem and it does not use shared keys. The method works in any finite group. We describe it in a simple case first.

- Alice and Bob share (public) a prime p.
- They choose private keys (e_A, d_A) and (e_B, d_B) satisfying
$$e_A d_A \equiv 1 \pmod{p-1} \quad \text{and } e_B d_B \equiv 1 \pmod{p-1}.$$
- Alice wants to send the message m. She computes $m^{e_A} \pmod{p}$ and sends it to Bob.
- Bob computes $(m^{e_A})^{e_B} \pmod{p}$ and sends it back to Alice.
- Alice determines $((m^{e_A})^{e_B})^{d_A} \pmod{p}$ and sends it to Bob.
- Bob can read the message by computing
$$(((m^{e_A})^{e_B})^{d_A})^{d_B} \pmod{p}.$$

Here we have by the choice of e_A, d_A, e_B, d_B that

$$e_A d_A = K_A(p-1) + 1, \qquad e_B d_B = K_B(p-1) + 1.$$

Therefore

$$(m^{e_A d_A})^{e_B d_B} \equiv (m^{K_A(p-1)+1})^{K_B(p-1)+1} \equiv m^{K_B(p-1)+1} \equiv m \pmod{p}.$$

In SageMath we may compute examples using the following implementation.

MasseyOmura

```
def MasseyOmura(p):
    R=Integers(p-1)
    RU=R.unit_group()
    g=list(RU.gens_values())
    go=list(RU.gens_orders())
    l=len(g)
    eA=1
    for k in [0..l-1]:
        eA=(eA*g[k]^randint(1,go[k]))%(p-1)
    dA=(1/eA)%(p-1)
    eB=1
    for k in [0..l-1]:
        eB=(eB*g[k]^randint(1,go[k]))%(p-1)
    dB=(1/eB)%(p-1)
    eA=ZZ(eA)
    dA=ZZ(dA)
    eB=ZZ(eB)
    dB=ZZ(dB)
    print("The public prime is", p)
    print("Alice chooses eA and dA as follows:",(eA,dA))
    print("Bob chooses eB and dB as follows:",(eB,dB))
    m=randint(1,p-1)
    print("Alice would like to send the message:",m)
    m1=power_mod(m,eA,p)
    pretty_print(htm
    ↪ l("Alice computes and sends to Bob $m^{eA}=%s$"%late
    ↪ x(m1)))
    m2=power_mod(m,eA*eB,p)
    pretty_print(htm
    ↪ l("Bob computes and sends to Alice $m^{eAeB}=%s$"%late
    ↪ x(m2)))
    m3=power_mod(m,eA*eB*dA,p)
```

```
    pretty_print(htm⌋
    ↪ l("Alice computes and sends to Bob $m^{eAeBdA}=%s$"%late⌋
    ↪ x(m3)))
    m4=power_mod(m,eA*eB*dA*dB,p)
    pretty_print(html("Bob determines $m^{eAeBdAdB}=%s$"%late⌋
    ↪ x(m4)))
MasseyOmura(2027)
```

Output

```
The public prime is 2027
Alice chooses eA and dA as follows: (781, 441)
Bob chooses eB and dB as follows: (1425, 745)
Alice would like to send the message: 1117
```

Alice computes and sends to Bob $m^{eA} = 1960$

Bob computes and sends to Alice $m^{eAeB} = 1513$

Alice computes and sends to Bob $m^{eAeBdA} = 1132$

Bob determines $m^{eAeBdAdB} = 1117$

An other interesting version is based on elliptic curves over finite fields. The are similar, but here we work with points on elliptic curves.

- Alice and Bob choose a finite field $\mathbb{F}_q$ and an elliptic curve E over this finite field. These are public. Let us denote the order of the group by N, that is $N = \#E(\mathbb{F}_q)$.
- They choose private keys (e_A, d_A) and (e_B, d_B) satisfying

$$e_A d_A \equiv 1 \pmod{N} \quad \text{and } e_B d_B \equiv 1 \pmod{N}.$$

- Alice wants to send the message $M \in E(\mathbb{F}_q)$ She computes $eA \cdot M$ and sends it to Bob.
- Bob computes $e_B(e_A \cdot M)$ and sends it back to Alice.
- Alice determines $d_A(e_B(e_A \cdot M))$ and sends it to Bob.
- Bob can read the message by computing

$$d_B(d_A(e_B(e_A \cdot M))).$$

Here we have that

$$e_A d_A = K_A N + 1, \qquad e_B d_B = K_B N + 1.$$

Thus it follows that

$$d_A e_A \cdot M = (K_A N + 1)M = K_A \infty + M = M,$$

where we used Lagrange's theorem. We may implement these computations in SageMath as follows.

MasseyOmuraElliptic

```
def MasseyOmuraEC(p):
    A=randint(1,p-1)
    B=randint(1,p-1)
    Fp=GF(p)
    E=EllipticCurve([Fp(A),Fp(B)])
    N=E.order()
    R=Integers(N)
    RU=R.unit_group()
    g=list(RU.gens_values())
    go=list(RU.gens_orders())
    l=len(g)
    eA=1
    for k in [0..l-1]:
        eA=(eA*g[k]^randint(1,go[k]))%N
    dA=(1/eA)%N
    eB=1
    for k in [0..l-1]:
        eB=(eB*g[k]^randint(1,go[k]))%N
    dB=(1/eB)%N
    eA=ZZ(eA)
    dA=ZZ(dA)
    eB=ZZ(eB)
    dB=ZZ(dB)
    print("The public elliptic curve is", E)
    print("Alice chooses eA and dA as follows:",(eA,dA))
    print("Bob chooses eB and dB as follows:",(eB,dB))
    m=E.random_point()
```

```
    print("Alice would like to send the message:",m)
    m1=eA*m
    pretty_print(htm⌟
    ↪ l(r"Alice computes and sends to Bob $eA\cdot M = %s$ "
    ↪ %latex(m1)))
    m2=eA*eB*m
    pretty_print(html(r"Bob computes and sends to Alice
    $eB\cdot eA\cdot M= %s$ " %latex(m2)))
    m3=eA*eB*dA*m
    pretty_print(html(r"Alice computes and sends to Bob
    $dA\cdot eB\cdot eA\cdot M =%s$ " %latex(m3)))
    m4=eA*eB*dA*dB*m
    pretty_print(html(r"Bob determines
    $dB\cdot dA\cdot eB\cdot eA\cdot M =%s$ " %latex(m4)))
MasseyOmuraEC(3251)
```

Output

```
The public elliptic curve is Elliptic Curve defined by y^2 = x^3 + 1982*x + 752 over Finite Field of size 3251
Alice chooses eA and dA as follows: (2527, 1426)
Bob chooses eB and dB as follows: (859, 232)
Alice would like to send the message: (1306 : 3125 : 1)
```

Alice computes and sends to Bob $eA \cdot M = (67 : 1471 : 1)$

Bob computes and sends to Alice $eB \cdot eA \cdot M = (249 : 1583 : 1)$

Alice computes and sends to Bob $dA \cdot eB \cdot eA \cdot M = (3111 : 2035 : 1)$

Bob determines $dB \cdot dA \cdot eB \cdot eA \cdot M = (1306 : 3125 : 1)$

5.4 The AA_β cryptosystem

Ariffin and Abu [1] proposed a new cryptosystem in 2009 called AA_β cryptosystem. It is a Diffie-Hellman type key agreement scheme. The idea behind the cryptosystem is that the discrete logarithm problem modulo 1 is difficult to solve. If we consider real numbers $x, y \in [0, 1]$, then we would like to find an integer n such that the fractional part of nx is the same as the fractional part of y. Let us now provide details of the key agreement. Given two public integers a and b, also a public real number is known $x \in [0, 1)$ (as it was pointed out by Ariffin and Abu, real numbers cannot be represented on computers, hence certain approximation is supposed to be used here). In the procedure

two matrices are used, these are as follows

$$A_0 = \begin{pmatrix} 1 & a \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad A_1 = \begin{pmatrix} b & 1 \\ 1 & 0 \end{pmatrix}.$$

- Alice chooses a random k bit binary string $b_1 b_2 \dots b_k$. She computes the matrix product

  $$A_{b_k} A_{b_{k-1}} \cdot A_{b_2} A_{b_1}.$$

  The top left entry n_A of the resulting matrix is her private integer. In a similar way Bob fixes a random k bit binary string and computes the corresponding matrix product. The top left entry n_B of this matrix is his private key.
- Alice determines $r_A = n_A \cdot x \pmod 1$ and send it to Bob. Similarly, Bob computes $r_B = n_B \cdot x \pmod 1$ and sends it to Alice.
- Alice computes $n_A r_B \pmod 1$ and Bob calculates $n_B r_A \pmod 1$, so the shared key is

  $$n_A n_B x \pmod 1.$$

Consider a SageMath code to reproduce the example from [1].

AriffinAbu

```
def AAbeta(a,b,A,B,t):
    A0=matrix(2,2,[1,a,1,0])
    A1=matrix(2,2,[b,1,1,0])
    EA=matrix(2,2,[1,0,0,1])
    EB=matrix(2,2,[1,0,0,1])
    M=[A0,A1]
    K=len(A)
    for k in [1..K]:
        EA=EA*M[A[-k]]
    for k in [1..K]:
        EB=EB*M[B[-k]]
    pretty_print(html('Alice has the matrix $%s$'%latex(EA)))
    pretty_print(html('Bob has the matrix $%s$'%latex(EB)))
    nA=EA[0,0]
    nB=EB[0,0]
    pretty_print(html('The value of $nA$ is $%s$'%latex(nA)))
    pretty_print(html('The value of $nB$ is $%s$'%latex(nB)))
    rA=(nA*t).frac()
```

```
    rB=(nB*t).frac()
    pretty_print(html('Alice computes $rA=%s$'%latex(rA)))
    pretty_print(html('Bob computes $rB=%s$'%latex(rB)))
    pretty_print(htm⌋
    ↪ l(r'The shared key is $nA\cdot rB=nB\cdot rA=%s$'%late⌋
    ↪ x((nA*rB).frac())))
    return (nA*rB).frac(),(nB*rA).frac()
AAbeta(2 , 3 , [1,0,1,1,0,0,1,1] , [1,0,1,1,0,1,1,1] ,
↪ 0.782170876860618 59)
```

Output

Alice has the matrix $\begin{pmatrix} 2415 & 533 \\ 734 & 162 \end{pmatrix}$

Bob has the matrix $\begin{pmatrix} 3725 & 823 \\ 1127 & 249 \end{pmatrix}$

The value of nA is 2415

The value of nB is 3725

Alice computes $rA = 0.9426676183938980$

Bob computes $rB = 0.5865163058042526$

The shared key is $nA \cdot rB = nB \cdot rA = 0.4368785172700882$

In 2010, Blackburn [2] showed that one can efficiently solve the discrete logarithm problem modulo 1, that is determining an integer n for which $nx \equiv y \pmod 1$, where $x, y \in [0, 1)$. The solution is based on continued fraction expansion. Certainly, if $y = 0$, then $n = 0$ is a solution. So we may assume that $y \neq 0$. Blackburn's method can be described as follows. We compute convergents of x, let say a/b, where $\gcd(a, b) = 1$. We determine the nearest integer to by, say s. We calculate the unique integer n_1 such that

$$n_1 \equiv s \cdot a^{-1} \pmod b$$

and $0 \leq n_1 < b$. If $n_1 x \equiv y \pmod 1$, then we have a solution. If not, then we take the next convergent of x. So it remains to show that this approach stops at some point. Since

a/b is a convergent we have that

$$|x - \frac{a}{b}| < \frac{1}{b^2}.$$

Therefore $|\varepsilon| < 1/b^2$, where $\varepsilon = x - a/b$. There exists an integer m such that $nx = y + m$, since $nx \equiv y \pmod 1$. We have that

$$\begin{aligned} na - mb &= nb(a/b) - mb = nb(x - \varepsilon) - mb = \\ &= by + bm - nb\varepsilon - mb = by - nb\varepsilon. \end{aligned}$$

Since $na - mb$ is an integer and $|nb\varepsilon| < nb/b^2 \leq 1/2$, we get that the nearest integer s to by is $na - mb$. Thus $s \equiv na \pmod b$. It yields that $n \equiv n_1 \equiv s \cdot a^{-1} \pmod b$. The denominators in convergents a/b for any real number grow exponentially, hence we will obtain values for which $n < b$ and in case of n_1 we have that $n_1 < b$. These numbers are congruent modulo b and both are less than b, so we get that $n = n_1$.

AAbetaDLP

```
def AAbetaDLP(x,y):
    cf=continued_fraction(x)
    T=True
    k=1
    while T:
        c=cf.convergent(k)
        pretty_print(html('The convergent is $%s$'%latex(c)))
        cn=c.numerator()
        cd=c.denominator()
        a=round(cd*y)
        n=(a/cn)%cd
        pretty_print(html('The value of $a$ is $%s$'%late⌋
        ↪ x(a)))
        pretty_print(html('Possible solution: $n=%s$'%latex(n⌋
        ↪ )))
        if (n*x).frac()==y:
            pretty_print(html('$n$ is a solution'))
            T=False
            return n
        else:
            pretty_print(html('$n$ is not a solution'))
            k=k+1
```

```
AAbetaDLP(0.78217087686061859,0.94266761839389801)
```

Output

The convergent is 1

The value of a is 1

Possible solution: $n = 0$

n is not a solution

The convergent is $\frac{3}{4}$

The value of a is 4

Possible solution: $n = 0$

n is not a solution

The convergent is $\frac{4}{5}$

The value of a is 5

Possible solution: $n = 0$

n is not a solution

The convergent is $\frac{7}{9}$

The value of a is 8

Possible solution: $n = 5$

n is not a solution

The convergent is $\frac{18}{23}$

The value of a is 22

Possible solution: $n = 14$

n is not a solution

The convergent is $\frac{61}{78}$

The value of a is 74

Possible solution: $n = 14$

n is not a solution

The convergent is $\frac{79}{101}$

The value of a is 95

Possible solution: $n = 37$

n is not a solution

The convergent is $\frac{1009}{1290}$

The value of a is 1216

Possible solution: $n = 1024$

n is not a solution

The convergent is $\frac{1088}{1391}$

The value of a is 1311

Possible solution: $n = 1125$

n is not a solution

The convergent is $\frac{2097}{2681}$

The value of a is 2527

Possible solution: $n = 2415$

n is a solution

5.5 SageMath summary

function	description
`EllipticCurve()`	Returns an elliptic curve.
`GF()`	Returns a finite field.
`gens()`	Returns generators of a given structure.
`gens_orders()`	Returns the orders of the generator elements.
`gens_values()`	Returns the values of the generator elements.
`order()`	Returns the order of a group.
`primitive_root()`	Returns a primitive root.
`unit_group()`	Returns the group of invertable elements.

Exercises

1. In a Diffie-Hellman key-exchange the group is $(\mathbb{F}^*_{1117}, \cdot)$ with a generator $g = 29$. Compute the public key belonging to Bob's secret key $b = 123$. Alice's public key is 321. Compute the shared key.
2. Alice and Bob exchange a key using the Diffie-Hellman protocol. They publish an elliptic curve $E : y^2 = x^3 + 13x + 10 \pmod{101}$. They also publish the point $P = (51, 2)$. Alice chooses $a = 28$, Bob has $b = 77$. Compute the points aP, bP and abP.
3. Alice and Bob are using the ElGamal cryptosystem. The public key of Alice is $(p, g, g^a) = (3571, 2, 2905)$. Bob encrypts the message $m_1 = 567$ with $k = 111$. What does Bob send to Alice?
4. Alice uses the ElGamal cryptosystem. She publishes the curve $E : y^2 = x^3 + 13x + 10 \pmod{101}$ and the point $P = (51, 2)$ of order 106. She also chooses a secret number $a = 37$ and publishes the point aP. Bob wants to send to Alice a message corresponding to the point $P_m = (85, 7)$. Compute aP. Cipher the message using $k = 19$. Decipher the message.
5. Alice and Bob use the Massey-Omura cryptosystem. Describe the communication and the intermediate results for the transmission of the message $m = 44$ between Alice and Bob, given $p = 113, e_A = 39$ and $e_B = 75$.
6. Alice and Bob use the elliptic curve Massey-Omura cryptosystem. They use the elliptic curve $E : y^2 = x^3 + 13x + 10 \pmod{101}$. The private keys are given by $(e_A, d_A) = (35, 103)$ and $(e_B, d_B) = (77, 95)$. Alice would like to send the message $M = (31 : 45) \in E(\mathbb{F}_{101})$. Describe the involved steps.

7. Alice chooses the binary string 111001 and Bob's binary string is 101010. They use the AA_β cryptosystem. Compute their shared key if the public real number is $0.14159265358979323846264$ and $a = 5, b = 7$.
8. Using the computation of the previous exercise. Eve could get the values of rA and rB and she also knows the public real number $x = 0.14159265358979323846264$. Apply the continued fraction expansion to solve the discrete logarithm modulo 1 and therefore the values of nA and nB.

Chapter 6 General attacks on the discrete logarithm problem

Summary

We study algorithms to determine solutions of discrete logarithm problems in multiplicative and additive groups.

6.1 Baby-Step Giant-Step algorithm

Let G be a multiplicative group of order p, we try to find x for which

$$g^x = h, \quad g, h \in G.$$

In order to find a solution x we fix an integer N for which $N^2 > p$. We determine the elements of two lists. The list corresponding to the baby-steps contains elements of the form

$$g^i \quad \text{for } i = 0, 1, \ldots, N-1.$$

The second list contains the elements related to the giant-steps

$$hg^{-jN} \quad \text{for } j = 0, 1, \ldots, N-1.$$

We compare the two lists to find an element that is on both lists: $g^i \equiv hg^{-jN} \pmod{p}$. We obtain a solution $x = i + jN$. The two lists have an element in common since if we write x in base N, then x has at most two digits (we note that $N^2 > p > x$). Therefore $x = x_0 + x_1 N$ for some $0 \leq x_0, x_1 \leq N-1$. Now we provide an example in SageMath.

BabyStep-GiantStep

```
p=random_prime(300,false,100)
F = GF(p)
g = F(primitive_root(p))
h = F.random_element()
```

```
N = ceil(sqrt(p))
pretty_print(htm
    l(r'We solve the DLP: $%s^x\equiv %s\pmod{%s}$'%(late
    x(g),latex(h),latex(p))))
pretty_print(html('The value of $N$: $%s$'%latex(N)))
L1=[g^i for i in [0..N-1]]
L2=[h*g^(-j*N) for j in [0..N-1]]
pretty_print(html('The list of baby-steps: $L_1=%s$'%late
    x(L1)))
pretty_print(html('The list of giant-steps: $L_2=%s$'%late
    x(L2)))
common=Set(L1).intersection(Set(L2))[0]
pretty_print(html('Common element: $%s$'%latex(common)))
I=L1.index(common)
J=L2.index(common)
pretty_print(htm
    l('We obtain that $g^{%s}\equiv hg^{-%s\cdot N}$'%(late
    x(I),latex(J))))
X=I+J*N
pretty_print(html('Solution of the DLP $x=%s$'%latex(X)))

```

Output

We solve the DLP: $3^x \equiv 200 \pmod{283}$

The value of N: 17

The list of baby-steps: $L_1 = [1, 3, 9, 27, 81, 243, 163, 206, 52, 156, 185, 272, 250, 184, 269, 241, 157]$

The list of giant-steps: $L_2 = [200, 266, 167, 174, 22, 250, 191, 138, 76, 169, 171, 168, 31, 95, 282, 143, 210]$

Common element: 250

We obtain that $g^{12} \equiv hg^{-5\cdot N}$

Solution of the DLP $x = 97$

In a similar way we can apply the method in additive groups. In case of elliptic curves we use the SageMath code below to provide an example.

BabyStep-GiantStepEC

```
p=random_prime(300,false,100)
F = GF(p)
A=F.random_element()
B=F.random_element()
E=EllipticCurve([A,B])
P=E.gens()[0]
Q=E.random_element()
N=ceil(sqrt(E.order()))
pretty_print(html('The prime is: $%s$'%latex(p)))
pretty_print(html('The elliptic curve is: $%s$'%latex(E)))
pretty_print(html(r'We solve the DLP: $n %s= %s$'%(late
    x(P),latex(Q))))
pretty_print(html('The value of $N$: $%s$'%latex(N)))
L1=[i*P for i in [0..N-1]]
L2=[Q+(-j*N)*P for j in [0..N-1]]
pretty_print(html('The list of baby-steps: $L_1=%s$'%late
    x(L1)))
pretty_print(html('The list of giant-steps: $L_2=%s$'%late
    x(L2)))
common=Set(L1).intersection(Set(L2))[0]
pretty_print(html('Common element: $%s$'%latex(common)))
I=L1.index(common)
J=L2.index(common)
pretty_print(htm
    l('We obtain that ${%s}P=Q-{%s N\cdot P}$'%(latex(I),late
    x(J))))
n=I+J*N
pretty_print(html('Solution of the additive DLP $n=%s$'%late
    x(n)))
pretty_print(html('We have that $nP=%s$ and $Q=%s$'%(latex(n
    *P),latex(Q))))
```

Output

The prime is: 199

The elliptic curve is: $y^2 = x^3 + 136x + 134$

We solve the DLP: $n\,(166 : 107 : 1) = (181 : 49 : 1)$

The value of N: 15

The list of baby-steps:
$L_1 = [(0 : 1 : 0), (166 : 107 : 1), (112 : 164 : 1), (21 : 149 : 1), (47 : 74 : 1), (76 : 30 : 1), (69 : 18 : 1), (109 : 13 : 1), (63 : 7 : 1), (35 : 196 : 1), (175 : 89 : 1), (61 : 81 : 1), (70 : 184 : 1), (49 : 79 : 1), (98 : 161 : 1)]$

The list of giant-steps:
$L_2 = [(181 : 49 : 1), (127 : 13 : 1), (23 : 107 : 1), (135 : 18 : 1), (179 : 89 : 1), (141 : 78 : 1), (0 : 1 : 0), (141 : 121 : 1), (179 : 110 : 1), (135 : 181 : 1), (23 : 92 : 1), (127 : 186 : 1), (181 : 150 : 1), (7 : 6 : 1), (40 : 83 : 1)]$

Common element: $(0 : 1 : 0)$

We obtain that $0P = Q - 6N \cdot P$

Solution of the additive DLP $n = 90$

We have that $nP = (181 : 49 : 1)$ and $Q = (181 : 49 : 1)$

6.2 The Pollard's ρ algorithm

Pollard's ρ algorithm is a Monte Carlo type probabilistic method to solve a discrete logarithm problem $g^x \equiv h \pmod{p}$. The basic idea is to determine numbers u and v for which

$$g^u \equiv h^v \pmod{p}.$$

Let us define a sequence x_n in the following way, $x_0 = 1$ and

$$x_{n+1} = \begin{cases} gx_n & \text{if } 0 < x_n \leq p/3, \\ x_n^2 & \text{if } p/3 < x_n \leq 2p/3, \\ hx_n & \text{if } 2p/3 < x_n < p. \end{cases}$$

It is clear that

$$x_n \equiv g^{u_n} h^{v_n} \pmod{p}.$$

We can also provide formulas for the sequences appearing in the exponents, we have $u_0 = v_0 = 0$ and

$$u_{n+1} = \begin{cases} u_n + 1 & \text{if } 0 < x_n \leq p/3, \\ 2u_n & \text{if } p/3 < x_n \leq 2p/3, \\ u_n & \text{if } 2p/3 < x_n < p, \end{cases}$$

while in case of v_n we obtain

$$v_{n+1} = \begin{cases} v_n & \text{if } 0 < x_n \leq p/3, \\ 2v_n & \text{if } p/3 < x_n \leq 2p/3, \\ v_n + 1 & \text{if } 2p/3 < x_n < p. \end{cases}$$

In the algorithm we store elements of vectors of the form

$$(x_i, u_i, v_i, x_{2i}, u_{2i}, v_{2i}).$$

We compute such vectors until we get one with $x_i = x_{2i}$ for some i. In this case we have

$$g^{u_i} h^{v_i} \equiv g^{u_{2i}} h^{v_{2i}} \pmod{p},$$

therefore it follows that

$$g^{u_i - u_{2i}} \equiv h^{v_{2i} - v_i} \pmod{p}.$$

So we may take $u \equiv u_i - u_{2i} \pmod{p-1}$ and $v \equiv v_{2i} - v_i \pmod{p-1}$. If $v \equiv 0 \pmod{p-1}$, then the algorithm fails. Assume that it is not the case. Compute $d = \gcd(v, p-1)$ and also s and t for which

$$d = sv + t(p-1).$$

We get that

$$g^{su} \equiv h^{sv} \equiv h^{d - t(p-1)} \equiv h^d \equiv (g^x)^d \pmod{p}.$$

It implies that $su \equiv dx \pmod{p-1}$, hence

$$dx = su + w(p-1).$$

Since d is a divisor of $p-1$, we have that d also divides su. Thus

$$x = \frac{su + w(p-1)}{d} \text{ for some } w \in \{0, 1, \ldots, d\}.$$

PollardRho

```
def PollardRho(g,h,p):
    S0=[k for k in [0..p//3]]
    S1=[k for k in [p//3+1..2*p//3]]
    S2=[k for k in [2*p//3+1..p-1]]
    T1=[1,0,0]
    T2=[1,0,0]
    def w(t):
        if t[0] in S0:
            return [(g*t[0])%p,(t[1]+1)%(p-1),t[2]]
        elif t[0] in S1:
            return [t[0]^2,(2*t[1])%(p-1),(2*t[2])%(p-1)]
        else:
            return [(h*t[0])%p,t[1],(t[2]+1)%(p-1)]
    jo=True
```

```
    while jo:
        T1=w(T1)
        T2=w(w(T2))
        if T1[0]==T2[0]:
            jo=False
    pretty_print(htm
    ↪ l(r'We solve the DLP: $%s^x\equiv %s\pmod{%s}$ '
    %(latex(g),latex(h),latex(p))))
    pretty_print(htm
    ↪ l(r'The triple $[x_i,\alpha_i,\beta_i]$: $%s$ '
    ↪ %latex(T1)))
    pretty_print(htm
    ↪ l(r'The triple $[x_{2i},\alpha_{2i},\beta_{2i}]$: $%s$ '
    ↪ %latex(T2)))
    g1=(T1[1]-T2[1])%(p-1)
    h1=(T2[2]-T1[2])%(p-1)
    pretty_print(htm
    ↪ l(r'We obtain that $%s^{%s}\equiv %s^%s\pmod{%s}$'
    %(latex(g),latex(g1),latex(h),latex(h1),latex(p))))
    d,s,t=xgcd(h1,p-1)
    possible=[((s*g1+i*(p-1))/d)%(p-1) for i in [0..d-1]]
    pretty_print(htm
    ↪ l(r'The possible list of solutions is $%s$'%latex(p
    ↪ ossible)))
    sol0=[j for j in possible if (g^j)%p==h]
    pretty_print(html('Solution of the DLP $x=%s$'%latex(s
    ↪ ol0[0])))
    return sol0[0]
PollardRho(3,200,283)
```

Output

We solve the DLP: $3^x \equiv 200 \pmod{283}$

The triple $[x_i, \alpha_i, \beta_i]$: $[9801, 0, 134]$

The triple $[x_{2i}, \alpha_{2i}, \beta_{2i}]$: $[9801, 264, 140]$

We obtain that $3^{18} \equiv 200^6 \pmod{283}$

The possible list of solutions is $[3, 50, 97, 144, 191, 238]$

Solution of the DLP $x = 97$

Consider the elliptic curve discrete logarithm problem, that is we have $Q \in \langle P \rangle$ for some $P \in E(\mathbb{F}_q)$. Our goal is to determine n such that $nP = Q$. Denote by N the order of the subgroup generated by the point P. We define a sequence of points of the form $R_n = a_n P + b_n Q$. Let $R_0 = P$ (we may also use some random integers $a_0, b_0 \in \{0, \ldots, N-1\}$). We need to partition the subgroup generated by P into three parts, here we apply the rule

$$\begin{aligned} S_1 &= \{R : R \in E(\mathbb{F}_q), y(R) \pmod 3 = 0\}, \\ S_2 &= \{R : R \in E(\mathbb{F}_q), y(R) \pmod 3 = 1\}, \\ S_3 &= \{R : R \in E(\mathbb{F}_q), y(R) \pmod 3 = 2\}, \end{aligned}$$

where $R = (x(R), y(R))$, that is $y(R)$ is the y-coordinate of the point R. The sequence R_n is defined as

$$R_{n+1} = \begin{cases} P + R_n & \text{if } R_n \in S_1, \\ 2R_n & \text{if } R_n \in S_2, \\ Q + R_n & \text{if } R_n \in S_3. \end{cases}$$

The two sequences a_n and b_n can be describes as follows. We have that $a_0 = 1$ (since $R_0 = P$) and $b_0 = 0$. The recurrence definitions are given by

$$a_{n+1} = \begin{cases} a_n + 1 & \text{if } R_n \in S_1, \\ 2a_n & \text{if } R_n \in S_2, \\ a_n & \text{if } R_n \in S_3, \end{cases}$$

while in case of b_n we have

$$b_{n+1} = \begin{cases} b_n & \text{if } R_n \in S_1, \\ 2b_n & \text{if } R_n \in S_2, \\ b_n + 1 & \text{if } R_n \in S_3. \end{cases}$$

We keep computing the vectors

$$(R_n, a_n, b_n, R_{2n}, a_{2n}, b_{2n})$$

until $R_n = R_{2n}$. In that case the definition of the sequence R_n yields that

$$a_n P + b_n Q = a_{2n} P + b_{2n} Q,$$

that is

$$(a_n - a_{2n})P = (b_{2n} - b_n)Q.$$

Therefore we obtain that

$$n = \log_P(Q) \equiv \frac{a_n - a_{2n}}{b_{2n} - b_n} \pmod{N}.$$

If $\gcd(N, b_{2n} - b_n) \neq 1$, then we follow the steps provided in case of the Pollard's ρ method in $\mathbb{F}_p$ to obtain a short list of possible solutions.

PollardRhoElliptic

```
def PollardRhoEC(P,Q):
    N=P.order()
    T1=[P,1,0]
    T2=[P,1,0]
    SEQ=[]
    def w(t):
        if ZZ(t[0][1])%3==0:
            return [P+t[0],(t[1]+1)%N,t[2]]
        elif ZZ(t[0][1])%3==1:
            return [2*t[0],(2*t[1])%N,(2*t[2])%N]
        else:
            return [Q+t[0],t[1],(t[2]+1)%N]
    jo=True
    while jo:
        T1=w(T1)
        T2=w(w(T2))
        SEQ.append([T1,T2])
```

```
        if T1[0]==T2[0]:
            jo=False
    pretty_print(html(r'We solve the DLP: $n%s=%s$'
    %(latex(P),latex(Q))))
    pretty_print(html(table(SEQ)))
    pretty_print(html(r'The triple $[R_i,a_i,b_i]$: $%s$'
    %latex(T1)))
    pretty_print(htm⌋
     ↪ l(r'The triple $[R_{2i},a_{2i},b_{2i}]$: $%s$'
    %latex(T2)))
    a1=(T1[1]-T2[1])%N
    b1=(T2[2]-T1[2])%N
    g,t,s=xgcd(N,b1)
    sol=[ZZ((s*a1+k*N)/g) for k in [0..g-1]]
    pretty_print(html(r'We obtain that $%sP=%sQ$'
    %(latex(a1),latex(b1))))
    pretty_print(htm⌋
     ↪ l(r'Possible solutions of the DLP $n\in %s$'
    %latex(sol)))
    sol1=[k for k in sol if k*P==Q]
    pretty_print(html('The solution is $n=%s$'%latex(s⌋
     ↪ ol1[0])))
    return sol1[0]
E=EllipticCurve([GF(743)(11),GF(743)(101)])
pretty_print(html('The elliptic curve E is $%s$'%latex(E)))
P=E.gens()[0]
Q=321*P
PollardRhoEC(P,Q)
```

Output

The elliptic curve E is $y^2 = x^3 + 11x + 101$

We solve the DLP: $n\,(83 : 520 : 1) = (443 : 608 : 1)$

$[(715 : 311 : 1), 2, 0]$	$[(592 : 661 : 1), 2, 1]$
$[(592 : 661 : 1), 2, 1]$	$[(734 : 4 : 1), 8, 4]$
$[(144 : 286 : 1), 4, 2]$	$[(640 : 416 : 1), 32, 16]$
$[(734 : 4 : 1), 8, 4]$	$[(23 : 337 : 1), 33, 17]$
$[(701 : 115 : 1), 16, 8]$	$[(93 : 181 : 1), 132, 68]$
$[(640 : 416 : 1), 32, 16]$	$[(434 : 76 : 1), 164, 272]$
$[(440 : 381 : 1), 32, 17]$	$[(398 : 429 : 1), 328, 181]$
$[(23 : 337 : 1), 33, 17]$	$[(159 : 709 : 1), 329, 182]$
$[(682 : 625 : 1), 66, 34]$	$[(454 : 675 : 1), 224, 0]$
$[(93 : 181 : 1), 132, 68]$	$[(297 : 17 : 1), 226, 0]$
$[(600 : 400 : 1), 264, 136]$	$[(715 : 311 : 1), 88, 2]$
$[(434 : 76 : 1), 164, 272]$	$[(144 : 286 : 1), 176, 6]$
$[(265 : 89 : 1), 328, 180]$	$[(701 : 115 : 1), 340, 24]$
$[(398 : 429 : 1), 328, 181]$	$[(440 : 381 : 1), 316, 49]$
$[(39 : 524 : 1), 329, 181]$	$[(682 : 625 : 1), 270, 98]$
$[(159 : 709 : 1), 329, 182]$	$[(600 : 400 : 1), 352, 28]$
$[(426 : 301 : 1), 294, 0]$	$[(265 : 89 : 1), 316, 112]$
$[(454 : 675 : 1), 224, 0]$	$[(39 : 524 : 1), 317, 113]$
$[(459 : 132 : 1), 225, 0]$	$[(426 : 301 : 1), 270, 228]$
$[(297 : 17 : 1), 226, 0]$	$[(459 : 132 : 1), 177, 92]$
$[(141 : 187 : 1), 226, 1]$	$[(141 : 187 : 1), 178, 93]$

The triple $[R_i, a_i, b_i]$: $[(141 : 187 : 1), 226, 1]$

The triple $[R_{2i}, a_{2i}, b_{2i}]$: $[(141 : 187 : 1), 178, 93]$

We obtain that $48P = 92Q$

Possible solutions of the DLP $n \in [48, 139, 230, 321]$

The solution is $n = 321$

6.3 Index calculus

We introduce the index calculus algorithm to solve the discrete logarithm problem $a^x \equiv b \pmod{p}$. We consider a sequence $x_1, x_2, \ldots$ of distinct random integers from the set $\{0, 1, \ldots, p-2\}$ and we fix a factor base containing small primes $B = \{p_1, p_2, \ldots, p_k\}$. We compute $a^{x_i} \pmod{p}$ and check if the resulting value is only divisible by primes from the set B. That is we try to find x_i such that

$$a^{x_i} \equiv p_1^{u_{i1}} p_2^{u_{i2}} \cdots p_k^{u_{ik}} \pmod{p}.$$

If we find such a congruence, then we have

$$x_i \equiv u_{i1} \log_a(p_1) + u_{i2} \log_a(p_2) + \ldots + u_{ik} \log_a(p_k) \pmod{p-1}.$$

That is we obtain a linear equation in the unknowns $\log_a(p_1), \ldots, \log_a(p_k)$. We have k unknowns so we only need to determine appropriately many linear relations to be able to compute the discrete logarithms of the primes contained in B. Let us suppose that we solved this linear algebra exercise. Now we take random integers m from the set $\{0, 1, \ldots, p-2\}$ and we compute $a^m b \pmod{p}$. If this latter number is only divisible by primes from B, then we get

$$\log_a(b) + m \equiv m_{i1} \log_a(p_1) + m_{i2} \log_a(p_2) + \ldots + m_{ik} \log_a(p_k) \pmod{p-1}.$$

Thus

$$\log_a(b) \equiv m_{i1} \log_a(p_1) + m_{i2} \log_a(p_2) + \ldots + m_{ik} \log_a(p_k) - m \pmod{p-1}.$$

Let us provide an implementation in SageMath.

IndexCalculus

```
def IndexCalculus(a,b,p,B):
    S=Set([0..p-2])
    pd=prime_divisors(p-1)
    T=True
    M=[]
    V=[]
    dimM=0
    while T:
        xi=S.random_element()
        S=S.difference(Set([xi]))
        q=[valuation((a^xi)%p,k) for k in B]
```

```
        if prod([k^valuation((a^xi)%p,k) for k in B])==(a^xi)%p:
            pretty_print(html("We take $x_i=%s$"%latex(xi)))
            pretty_print(html("Exponent vector is given by $%s$"%latex(x(q))))
            M0=M+[q]
            if all(matrix(GF(x),M0).rank()==dimM+1 for x in pd):
                M.append(q)
                V.append(xi)
                dimM=dimM+1
                pretty_print(html("The new matrix is $%s$"%latex(matrix(M))))
                pretty_print(html("The new vector is $%s$"%latex(vector(V))))
            else:
                pretty_print(html("Rank problem."))
    if dimM==len(B):
        T=False
LOGS=matrix(Integers(p-1),M).inverse()*vector(V)
pretty_print(html("The discrete logarithms of the $%s$ primes:$%s$"%(latex(B),latex(LOGS))))
S=Set([0..p-2])
T=True
while T:
    xi=S.random_element()
    S=S.difference(Set([xi]))
    q=[valuation((a^xi*b)%p,k) for k in B]
    if prod([k^valuation((a^xi*b)%p,k) for k in B])==(a^xi*b)%p:
```

```
36              pretty_print(html("We take $m_i=%s$"%latex(xi)))
37              pretty_print(htm⌋
                ↪ l("Exponent vector is given by $%s$"%late⌋
                ↪ x(q)))
38              DL=(sum(q[k]*LOGS[k] for k in
                ↪ [0..len(q)-1])-xi)%(p-1)
39              pretty_print(htm⌋
                ↪ l("The discrete logarithm of $b$ is $%s$"%late⌋
                ↪ x(DL)))
40              return DL
41  IndexCalculus(3,37,1217,[2,3,5,7])
```

Output

We take $x_i = 1148$

Exponent vector is given by $[1, 0, 1, 1]$

The new matrix is $\begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}$

The new vector is (1148)

We take $x_i = 216$

Exponent vector is given by $[1, 0, 0, 0]$

The new matrix is $\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$

The new vector is $(1148,\ 216)$

We take $x_i = 253$

Exponent vector is given by $[3, 2, 1, 0]$

The new matrix is $\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix}$

The new vector is $(1148,\ 216,\ 253)$

We take $x_i = 113$

Exponent vector is given by $[0, 0, 0, 1]$

Rank problem.

We take $x_i = 435$

Exponent vector is given by $[2, 3, 0, 0]$

The new matrix is $\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 2 & 3 & 0 & 0 \end{pmatrix}$

The new vector is $(1148,\ 216,\ 253,\ 435)$

The discrete logarithms of the $[2, 3, 5, 7]$ primes:$(216,\ 1,\ 819,\ 113)$

We take $m_i = 1061$

Exponent vector is given by $[2, 1, 0, 0]$

The discrete logarithm of b is 588

6.4 Pohlig-Hellman algorithm

The Pohlig-Hellman algorithm [17] is very useful in practice if the order of the group in which we would like to solve a given discrete logarithm problem is smooth, that is it has only "small" prime divisors. Hence the discrete logarithm problem in a group G is as hard as the discrete logarithm problem in the largest subgroup of prime order in G. Suppose we have a group $G = \langle g \rangle$ such that

$$\mathrm{ord}(g) = \prod_{k=1}^{n} p_k^{e_k}$$

for some primes $p_1, p_2, \ldots, p_n$ and positive integers $e_1, e_2, \ldots, e_n$. Our goal is to determine an x for which $g^x = h$. Instead of solving the general discrete logarithm problem we try to compute $x \pmod{p_i^{e_i}}$ and than we apply the Chinese Reminder Theorem to obtain $x \pmod{N}$. The group G is a cyclic group so we know from group theory that there is a group isomorphism given by

$$\Phi : G \to C_{p_1^{e_1}} \times C_{p_2^{e_2}} \times \cdots \times C_{p_n^{e_n}},$$

where the groups $C_{p_i^{e_i}}$ are cyclic of order $p_i^{e_i}$. Let us also consider the projection of Φ to the component $C_{p_i^{e_i}}$ denoted by $\Phi_i : G \to C_{p_i^{e_i}}$ and defined by

$$a \longmapsto a^{N/p_i^{e_i}}.$$

This is a group homomorphism so from the equality $h = g^x$ it follows that $\Phi_i(h) = \Phi_i(g)^x$, this latter relation provides a discrete logarithm problem but not in G, but $C_{p_i^{e_i}}$. Thus we get $x \pmod{p_i^{e_i}}$. We do it for all the prime divisors of N to get a system of congruences

$$\begin{aligned} x &\equiv x_1 \pmod{p_1^{e_1}} \\ x &\equiv x_2 \pmod{p_2^{e_2}} \\ &\vdots \\ x &\equiv x_n \pmod{p_n^{e_n}}. \end{aligned}$$

A standard application of the Chinese Reminder Theorem yields

$$x \pmod{p_1^{e_1} p_2^{e_2} \cdots p_n^{e_n}} \to x \pmod{N}.$$

It remains to describe how to solve the discrete logarithm problem in C_{p^e}. Let say we would like to determine x for which $g^x = h$ in C_{p^e}. The unknown exponent has a base p representation as follows

$$x = b_0 + b_1 p + b_2 p^2 + \ldots + b_{e-1} p^{e-1}.$$

We proceed in a recurrent way, suppose we know x modulo p^k for some $0 \leq k \leq e-1$, say

$$x_k \equiv b_0 + b_1 p + b_2 p^2 + \ldots + b_{k-1} p^{k-1} \pmod{p^k}.$$

We have that $x \equiv x_k + b_k p^k \pmod{p^{k+1}}$, that is

$$h = g^{x_k} (g^{p^k})^{b_k}.$$

Let $h_k = h g^{-x_k}$ and $g_k = g^{p^k}$. The reduced discrete logarithm problem is $h_k = g_k^{b_k}$.

PohligHellman

```
def PohligHellman(g,h,p):
    pretty_print(html('The prime $p$ is $%s$'%latex(p)))
    F=GF(p)
    g1=F(g)
    h1=F(h)
    N=p-1
    qi=[r^N.valuation(r) for r in prime_divisors(N)]
    pretty_print(htm
     ↪ l('Prime power divisors of $p-1: %s$'%latex(qi)))
    lqi=len(qi)
    Nqi=[N/q for q in qi]
    gi=[g1^r for r in Nqi]
    hi=[h1^r for r in Nqi]
    xi=[discrete_log(hi[i],gi[i]) for i in range(lqi)]
    pretty_print(html('Discrete logarithms $x_i=%s$'%late
     ↪ x(xi)))
    x=CRT(xi,qi)
    pretty_print(html(r'We have that $\log_g h=%s$'%latex(x)))
    return x
PohligHellman(2,33,next_prime(1111))
```

Output

The prime p is 1117

Prime power divisors of $p-1$: $[4, 9, 31]$

Discrete logarithms $x_i = [1, 3, 19]$

We have that $\log_g h = 453$

Basicly the same idea can be used in case of additive groups. For example if we have an elliptic curve E over a finite field, then the discrete logarithm problem is given by $nP = Q$, where P and Q are points on the curve. We compute

$$\begin{aligned} n &\equiv n_1 \pmod{p_1^{e_1}} \\ n &\equiv n_2 \pmod{p_2^{e_2}} \\ &\vdots \\ n &\equiv n_m \pmod{p_m^{e_m}}, \end{aligned}$$

where the primes $p_1, \ldots, p_m$ are the divisors of the order of the group generated by P. For a given prime p we write

$$n = b_0 + b_1 p + b_2 p^2 + \ldots + b_{e-1} p^{e-1}.$$

First we set $P_0 = (N/p)P$ and $Q_0 = (N/p)Q$. The order of P_0 is p. Here we have

$$Q_0 = \frac{N}{p}Q = \frac{N}{p}nP = n(\frac{N}{p}P) = nP_0.$$

Since the order of P_0 is p it follows that $n \equiv b_0 \pmod{p}$ for some $b_0 \in \{0, 1, \ldots, p-1\}$. The next step is to determine $Q_1 = (N/p^2)(Q - b_0 P)$. We get that

$$Q_1 = \frac{N}{p^2}(Q - b_0 P) = \frac{N}{p^2}(n - b_0)P = (b_0 + b_1 p - b_0)\frac{N}{p^2}P = b_1 P_0,$$

for some $b_1 \in \{0, 1, \ldots, p-1\}$. In general we determine b_i from

$$Q_i = \frac{N}{p^{i+1}}(Q - b_0 P - b_1 pP - \ldots - b_{i-1} p^{i-1} P).$$

PohligHellmanEC

```
1 F = FiniteField(next_prime(8811))
2 E = EllipticCurve(F,[20,20])
3 pretty_print(html('The elliptic curve is $%s$'%latex(E)))
```

```
P = E.gens()[0]
Q = 3333*P
pretty_print(html('The point $P$ is $%s$'%latex(P)))
pretty_print(html('The point $Q$ is $%s$'%latex(Q)))
pretty_print(html('We look for $n$ such that $nP=Q$'))
pretty_print(html('The order of $P$ factors as $%s$'%late
    x(P.order().factor())))
facs,exps = zip(*factor(P.order()))
primes = [facs[i]^exps[i] for i in range(len(facs))]
dlogs = []
for fac in primes:
    t = ZZ(P.order())//ZZ(fac)
    dlog = discrete_log(t*Q,t*P,operation="+")
    dlogs += [dlog]
    pretty_print(htm
        l(r'We have that $n\pmod{%s}\equiv %s$'%(late
        x(fac),latex(dlog))))
n = crt(dlogs,primes)
pretty_print(html('$n=%s$'%latex(n)))
```

Output

The elliptic curve is $y^2 = x^3 + 20x + 20$

The point P is $(6179 : 154 : 1)$

The point Q is $(1909 : 6431 : 1)$

We look for n such that $nP = Q$

The order of P factors as $2^6 \cdot 3^3 \cdot 5$

We have that $n \pmod{64} \equiv 5$

We have that $n \pmod{27} \equiv 12$

We have that $n \pmod{5} \equiv 3$

$n = 3333$

6.5 SageMath summary

function	description
`difference()`	Returns the difference of two sets.
`index()`	Returns the position of an element in a list.
`intersection()`	Returns the intersection of two sets.
`rank()`	Returns the rank of a matrix.
`xgcd()`	Applies the extended Euclidean algorithm.

Exercises

1. Solve the discrete logarithm problem $3^x \equiv 224 \pmod{1013}$ using the Baby-Step Giant-Step algorithm.

2. Solve the discrete logarithm problem $n \cdot (51 : 2) = (62 : 28)$ in case of the elliptic curve $E : y^2 = x^3 + 13x + 10$ over $\mathbb{F}_{101}$ using the Baby-Step Giant-Step algorithm.
3. Use the Pollard's ρ method to attack the discrete logarithm problem given by $g = 3, h = 245$ in $\mathbb{F}_{1013}^*$. That is find an integer x such that $g^x \equiv h \pmod{1013}$.
4. Given the elliptic curve $E : y^2 = x^3 + 13x + 32$ over the finite field $\mathbb{F}_{2027}$ and two points $P = (1381 : 839), Q = (6 : 1229)$. Determine a solution of the elliptic curve discrete logarithm problem $nP = Q$ by applying the Pollard's ρ method.
5. Apply the index calculus algorithm to solve the discrete logarithm problem $5^x \equiv 20 \pmod{503}$.
6. Let

$$p = 1747808567274271495694237474897031552001$$

and $(g, h) = (13, 2020)$. Apply the Pohlig-Hellman algorithm to compute $\log_g(h) \pmod{p}$.

Chapter 7 Non-abelian discrete logarithm

Summary

- Discrete logarithm with special generators
- General case in $PSL(2, \mathbb{F}_p)$
- SageMath summary
- Exercises

7.1 Discrete logarithm with special generators

The discrete logarithm problem forms the basis of many cryptographic primitives. Ilić and Magliveras [12] showed that the generalized discrete logarithm problem in certain non-abelian groups is easy to solve. Here we provide details and numerical examples based on results by Ilić and Magliveras.

The general linear group of degree n over $\mathbb{F}_q$ (denoted by $GL(n, \mathbb{F}_q)$) is the set of $n \times n$ invertible matrices, together with the operation of ordinary matrix multiplication. The special linear group $SL(n, \mathbb{F}_q)$ of degree n over $\mathbb{F}_q$ is the set of $n \times n$ matrices with determinant 1. The center $Z(GL(n, \mathbb{F}_q))$ of $GL(n, \mathbb{F}_q)$ consists of all matrices of the form

$$\lambda I,$$

where $\lambda \in \mathbb{F}_q^*$ and I is the $n \times n$ identity matrix. The quotient group

$$PSL(n, \mathbb{F}_q) = SL(n, \mathbb{F}_q)/Z(SL(n, \mathbb{F}_q))$$

is called the projective special linear group of degree n over $\mathbb{F}_q$. Consider the group $PSL(2, \mathbb{F}_p)$ where p is an odd prime. Let

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

These matrices are of order p and non-commuting. We will work in the group $G =< A, B >$. Suppose that

$$M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \in G, \ m_{11}, m_{12}, m_{21}, m_{22} \in \mathbb{F}_p.$$

Solving the generalized discrete logarithm problem here means the determination of

non-negative integers (i, j, k, l) such that $M = A^i B^j A^k B^l$. It is easy to see that

$$A^i = \begin{pmatrix} 1 & i \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad B^j = \begin{pmatrix} 1 & 0 \\ j & 1 \end{pmatrix}.$$

Hence it follows that

$$A^i B^j A^k B^l = \begin{pmatrix} 1 & i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ j & 1 \end{pmatrix} \begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ l & 1 \end{pmatrix}.$$

Putting together the above computations we obtain that

$$\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} 1 + ij + l((1 + ij)k + i) & (1 + ij)k + i \\ j + l(jk + 1) & jk + 1 \end{pmatrix}.$$

The matrix equation can be written as a system of equations as follows

$$\begin{aligned} 1 + ij + l((1 + ij)k + i) &= m_{11}, \\ (1 + ij)k + i &= m_{12}, \\ j + l(jk + 1) &= m_{21}, \\ jk + 1 &= m_{22}. \end{aligned}$$

The Groebner basis of the ideal

$$\begin{aligned} I = \langle\, 1 + ij + l((1 + ij)k + i) - m_{11}, \\ (1 + ij)k + i - m_{12}, \\ j + l(jk + 1) - m_{21}, \\ jk + 1 - m_{22} \,\rangle \end{aligned}$$

over the rationals is given by

$$\begin{aligned} GB = [l - jim_{21} + jm_{11} - m_{21}, \\ k + im_{22} - m_{12}, \\ jim_{12}m_{21} + ji + jm_{11}m_{12} - m_{11} + m_{12}m_{21} + 1, \\ jim_{22} - jm_{12} + m_{22} - 1, \\ m_{11}m_{22} - m_{12}m_{21} - 1]. \end{aligned}$$

Hence we need to handle the system of equations

$$\begin{aligned} l - jim_{21} + jm_{11} - m_{21} &= 0, \\ k + im_{22} - m_{12} &= 0, \\ jim_{22} - jm_{12} + m_{22} - 1 &= 0. \end{aligned}$$

In practice it can be solved efficiently for i, j, k and l. Let us consider a concrete example.

Over $\mathbb{F}_7$ we take the matrices

$$M = \begin{pmatrix} 5 & 2 \\ 6 & 4 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

We get the system of equations given by

$$\begin{aligned} 1 + ij + l((i + ij)k + i) &= 5 \\ (i + ij)k + i &= 2 \\ j + l(jk + 1) &= 6 \\ jk + 1 &= 4. \end{aligned}$$

A solution of the above system is $(i, j, k, l) = (0, 5, 2, 2)$ that is we have

$$M = A^0 B^5 A^2 B^2.$$

A SageMath implementation of the above computation is as follows.

DLPNonAbelian

```
@interact(layout=[['Px','M']])
def DLP_Non_Abelian(Px=input_grid(1,1, default = [[7]],
    label='$p=$'),
      M=input_grid(2,2, default = [[5,2],[6,4]],
        label='$M=$')):
    p=Px[0][0]; M=matrix(GF(p),2,2,M)
    if M not in SL(2,p):
        pretty_print(html(r'$%s \ \notin \ SL(2,%s) $'%(late
            x(M),latex(p))))
    else:
        P.<i,j,k,l>=PolynomialRing(GF(p),4)
        PZ.<X1,X2,X3,X4>=PolynomialRing(ZZ,4)
        A=matrix(GF(p),[[1,1],[0,1]])
        B=matrix(GF(p),[[1,0],[1,1]])
        pretty_print(html('Given the following matrices:'))
        pretty_print(html(r'$M=%s, A=%s, B=%s$'%(late
            x(M),latex(A),latex(B))))
        pretty_print(htm
            l('Our goal is to determine $(i,j,k,l):$'))
        pretty_print(html(r'$%s^i%s^j%s^k%s^l=%s$'
            %(latex(A),latex(B),latex(A),latex(B),latex(M))))
```

```
        pretty_print(htm⌋
        ↪ l('We obtain the system of equations:'))
        Ai=matrix(P,[[1,i],[0,1]]); Ak=matrix(P,[[1,k],[0,1]])
        Bj=matrix(P,[[1,0],[j,1]]); Bl=matrix(P,[[1,0],[l,1]])
        Mi=Ai*Bj*Ak*Bl
        I=ideal(Mi[v][w]-M[v][w] for v in [0..1] for w in
        ↪ [0..1])
        for v in [0..3]:
            pretty_print(Mi.list()[v],'=',M[v//2][v%2])
        pretty_print(htm⌋
        ↪ l('Let $I$ be the ideal genereted by:'))
        IList=[Mi.list()[w]-M[w//2][w%2] for w in [0..3]]
        pretty_print(IList)
        pretty_print(htm⌋
        ↪ l('The Groebner basis is as follows:'))
        GB=I.groebner_basis()
        pretty_print(GB)
        var('x1,x2,x3,x4')

        ↪ S=[PZ(g(i=X1,j=X2,k=X3,l=X4))(X1=x1,X2=x2,X3=x3,X4=x4)==0
        ↪ for g in GB]
        pretty_print(html('The solutions are given by:'))
        Sol=solve_mod(S,p)
        for t in [0..len(Sol)-1]:
            pretty_print(Sol[t])
```

Output

$p =$ 7

$M =$ 5 2
6 4

Given the following matrices:

$$M = \begin{pmatrix} 5 & 2 \\ 6 & 4 \end{pmatrix}, A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Our goal is to determine non-negative integers (i, j, k, l) :

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^i \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^j \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^k \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^l = \begin{pmatrix} 5 & 2 \\ 6 & 4 \end{pmatrix}$$

We obtain the system of equations:

$$ijkl + ij + il + kl + 1 = 5$$

$$ijk + i + k = 2$$

$$jkl + j + l = 6$$

$$jk + 1 = 4$$

Let I be the ideal genereted by:

$$[ijkl + ij + il + kl + 3, ijk + i + k + 5, jkl + j + l + 1, jk + 4]$$

The Groebner basis is as follows:

$$[kl + 2k + 6, i + 2k + 3, j + 4l + 1]$$

The solutions are given by:

$$(0, 5, 2, 2)$$

$$(1, 2, 5, 1)$$

$$(2, 3, 1, 6)$$

$$(3, 6, 4, 0)$$

$$(5, 1, 3, 3)$$

$$(6, 4, 6, 4)$$

7.2 General case in $PSL(2, \mathbb{F}_p)$

The above method can be extended to more general groups. Let C, D be order p, non-commuting elements of $PSL(2, \mathbb{F}_p)$. Consider the group $G =< C, D >$. We show that the discrete logarithm problem can be reduced to the previous case, that is we can solve it in $< A, B >$ instead of $< C, D >$. To do so we try to determine $g \in G = PSL(2, \mathbb{F}_p)$ for which

$$C = g^{-1} A^s g \quad \text{and} \quad D = g^{-1} B^t g,$$

where $s, t < p$ are non-negative integers. Let

$$g = \begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix}.$$

To compute g_1, g_2, g_3 and g_4 we may use the system of linear equations $gC = A^s g$ and $gD = B^t g$. Suppose that we have determined such a matrix g, than we have

$$\begin{aligned} M &= C^i D^j C^k D^l \\ &= (g^{-1}A^s g)^i (g^{-1}B^t g)^j (g^{-1}A^s g)^k (g^{-1}B^t g)^l \\ &= (g^{-1}A^{si} g)(g^{-1}B^{tj} g)(g^{-1}A^{sk} g k (g^{-1}B^{tl} g) \\ &= g^{-1}A^{si}B^{tj}A^{sk}B^{tl} g. \end{aligned}$$

Therefore we need to solve the discrete logarithm problem

$$M_1 = A^x B^y A^v B^w,$$

where $M_1 = gMg^{-1}$ and $x = si, y = tj, v = sk, w = tl$. Thus the problem is reduced and we can apply the method introduced previously. Let us describe a concrete example. Consider the matrices given by

$$M = \begin{pmatrix} 3 & 5 \\ 2 & 6 \end{pmatrix}, \quad C = \begin{pmatrix} 5 & 1 \\ 5 & 4 \end{pmatrix}, \quad D = \begin{pmatrix} 2 & 5 \\ 4 & 0 \end{pmatrix}.$$

First we need to find an appropriate matrix g such that

$$\begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix}\begin{pmatrix} 5 & 1 \\ 5 & 4 \end{pmatrix} = \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix}\begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix}$$

$$\begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix}\begin{pmatrix} 2 & 5 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ t & 1 \end{pmatrix}\begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix}.$$

The above matrix equations yield the system of equations

$$\begin{aligned} g_1 + 4g_2 &= 0 \\ 5g_1 - g_2 &= 0 \\ -g_1 t + g_3 - g_4 &= 0 \\ -g_2 t + 5g_3 - g_4 &= 0 \\ -g_3 s + 4g_1 + g_2 &= 0 \\ -g_4 s + 4g_1 + 3g_2 &= 0 \\ 4g_3 + 5g_4 &= 0 \\ g_3 + 3g_4 &= 0. \end{aligned}$$

By means of Groebner basis technique it simplifies as follows

$$g_4 s + g_2 = 0$$
$$g_2 t + 2g_4 = 0$$
$$g_1 + 4g_2 = 0$$
$$g_3 + 3g_4 = 0.$$

We obtain that

$$g \cdot C - A^s \cdot g = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \quad \text{and} \quad g \cdot D - B^t \cdot g = \begin{pmatrix} 0 & 0 \\ \frac{4g_2 st - g_2}{s} & \frac{-g_2 st + g_2}{s} \end{pmatrix}.$$

It follows easily that

$$(s,t) \in \{(1,2),(6,5),(3,3),(5,6),(4,4),(2,1)\}.$$

If $s = 1$ and $t = 2$, then we have $g = \begin{pmatrix} 6 & 2 \\ 6 & 5 \end{pmatrix}$ and $M_1 = g \cdot M \cdot g^{-1} = \begin{pmatrix} 3 & 3 \\ 1 & 6 \end{pmatrix}$. It remains to solve the discrete logarithm problem given by $M_1 = A^x B^y A^v B^w$, where

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

The approach we provided earlier gives

$$(x,y,v,w) \in \{(0,4,3,3),(1,3,4,2),(2,1,5,0),(3,2,6,1),(5,5,1,4),(6,6,2,5)\}.$$

Thus the complete set of solutions of the original discrete logarithm problem $M = C^i D^j C^k D^l$ is given by

$$(i,j,k,l) \in \{(0,2,3,5),(1,5,4,1),(2,4,5,0),(3,1,6,4),(5,6,1,2),(6,3,2,6)\}.$$

We can also check the computation by using the SageMath implementation given below.

DLPNonAbelianGeneral

```
def sortfv(L):
    SL=[]
    TL=[L[i][0] for i in [0..len(L)-1]]
    for j in [0..len(L)-1]:
        SL.append(L[TL.index(max(TL))])
        L.remove(L[TL.index(max(TL))])
        TL.remove(TL[TL.index(max(TL))])
    return SL

@interact(layout=[['Pmx','M','C','D']])
def _(Pmx=input_grid(1,1, default = [[7]], label='$p=$'),
```

```
    M=input_grid(2,2, default = [[3,5],[2,6]],
     ↪  label='$M=$'),
    C=input_grid(2,2, default = [[5,1],[5,4]],
     ↪  label='$C=$'),
    D=input_grid(2,2, default = [[2,5],[4,0]],
     ↪  label='$D=$')):
p=Pmx[0][0]; M=matrix(GF(p),2,2,M); C=matrix(GF(p),2,2,C);
 ↪  D=matrix(GF(p),2,2,D)
if C.multiplicative_order()!=p or
 ↪  D.multiplicative_order()!=p:
    pretty_print(html('Provide an appropriate matrix.'))
else:
    P.<g1,g2,g3,g4,s,t>=PolynomialRing(GF(p),6)
    P.<i,j,k,l>=PolynomialRing(GF(p),4)
    Ptmp.<X1,X2,X3,X4,Y1,Y2>=PolynomialRing(ZZ,6)
    GVar=[g1,g2,g3,g4,s,t]
    var('x1,x2,x3,x4,S,T')
    G=matrix([[g1,g2],[g3,g4]])
    As=matrix([[1,s],[0,1]])
    Bt=matrix([[1,0],[t,1]])
    N=matrix(2)

    pretty_print(html('Given the matrices:'))
    pretty_print('M=',M,', C=',C,', D=',D,', G=',G)
    pretty_print(html('<br>We determine $(i,j,k,l):$'))
    pretty_print(html(r'$%s=%s^i%s^j%s^k%s^l$'%
     ↪  (latex(M),latex(C),latex(D),latex(C),latex(D))))
    pretty_print(html('<br>Reduction step:'))
    pretty_print(htm⌋
     ↪  l(r'$%s%s=%s%s \quad$ and $\quad %s%s=%s%s$'%(late⌋
     ↪  x(G),latex(C),
     ↪  latex(As),latex(G),latex(G),latex(D),late⌋
     ↪  x(Bt),latex(G))))

```

```
W1=G*D-Bt*G; W2=G*C-As*G
iw=ideal([W1[0,0],W1[0,1],W1[1,0],W1[1,1],W2[0,0],
↪ W2[0,1],W2[1,0],W2[1,1]])
pretty_print(html('<br>System of equation:'))
for y in [0..1]:
    for z in [0..1]:
        pretty_print(W1[y][z],'=',0)
for y in [0..1]:
    for z in [0..1]:
        pretty_print(W2[y][z],'=',0)

pretty_print(html('<br>Groebner basis computation:'))
GB=iw.groebner_basis()
pretty_print(GB)
GBList=[]
for i in GB:
    GBList.append(Ptmp(i(g1=X1,g2=X2,g3=X3,g4=X4,s」
    ↪ =Y1,t=Y2))
    ↪ (X1=x1,X2=x2,X3=x3,X4=x4,Y1=S,Y2=T))
XVar=[x1,x2,x3,x4]

V1=[y.variables() for y in GBList]
V2=[]
for y in [0..len(XVar)-1]:
    n=0
    for z in [0..len(GBList)-1]:
        if XVar[y] in V1[z]:
            n+=1
    V2.append((n,XVar[y]))
V3=sortfv(copy(V2));
x=V3[0][1]; g=Ptmp」
↪ (x(x1=X1,x2=X2,x3=X3,x4=X4))(X1=g1,X2=g2,X3=g3,X4=g4)
↪ #x:special xi element, g: x in GF(p)

```

```
        PrevC=[]
        ESol=[]
        for y in [0..len(GBList)-1]:
            VarT=list(V1[y])
            z=0
            lh=len(VarT)-1
            while z<=lh:
                if bool(VarT[z]==x) or (VarT[z] in PrevC) or
                ↪  (VarT[z] not in XVar):
                    r=VarT.index(VarT[z])
                    VarT.remove(VarT[r])
                    lh-=1
                else:
                    z+=1
            if len(VarT)!=0:
                ESol.append((solve(GBList[y],VarT[0]))[0])
            if len(VarT)!=0:
                PrevC.append(VarT[0])
        pretty_print(ESol)

        X=matrix(2,2,[x1,x2,x3,x4])
        for i in [1..3]:
            for y in ESol:
                X=X.substitute({y.lhs():y.rhs()})

        H=[Ptmp(a.numerator」
        ↪  ()(x1=X1,x2=X2,x3=X3,x4=X4,S=Y1,T=Y2))
        ↪  (X1=g1,X2=g2,X3=g3,X4=g4,Y1=s,Y2=t)
        ↪  /Ptmp(a.denominator()(x1=X1,x2=X2,x3=X3,x4=X4,
        ↪  S=Y1, T=Y2))(X1=g1, X2=g2, X3=g3, X4=g4,
        ↪  Y1=s,Y2=t) for a in copy(X.list())]
        V=matrix(2,2,H)
        G1=V*C-As*V; G2=V*D-Bt*V;
        GmatList=[G1,G2]
```

```
93          pretty_print(htm」
             ↪ l('<br>Substitution into $G1=g\cdot C-A^s\cdot g$ and'))
94          pretty_print(htm」
             ↪ l('<br>$G2=g\cdot D-B^t\cdot g$ yields:'))
95          pretty_print(htm」
             ↪ l('$G1=%s \quad$ and $\quad G2=%s$'%(late」
             ↪ x(G1),latex(G2))))
96          N=[]
97
98          for I in GmatList:
99              if I!=matrix(2):
100                 for y in [0..1]:
101                     for w in [0..1]:
102                         if I[y][w]!=0:
103                             N.append(solve_mod(Ptmp」
                                 ↪ ((I[y][w].numerator()/g)
                                 ↪ (g1=X1, g2=X2, g3=X3, g4=X4,
                                 ↪ s=Y1, t=Y2))(X1=x1, X2=x2,
                                 ↪ X3=x3 ,X4=x4, Y1=S, Y2=T),p))
104
105         for i in [0..len(N)-1]:
106             N[i]=Set(N[i])
107         ST=N[0]
108         for y in [1..len(N)-1]:
109             ST=ST.intersection(N[y])
110         pretty_print(htm」
             ↪ l('<br>Conditions for $s$ and $t$ provides:'))
111         pretty_print(ST)
112
113         T1=W1(s=ST[0][0],t=ST[0][1]).list()
114         for y in [0..len(T1)-1]:
115             T1[y]=Ptmp(T1[y](g1=X1,g2=X2,g3=X3,g4=X4,s」
                 ↪ =Y1,t=Y2))(X1=x1,X2=x2,X3=x3,X4=x4,Y1=S,Y2=T)
116         S1=Set(solve_mod(T1,p))
```

```
        T2=W2(s=ST[0][0],t=ST[0][1]).list()
        for y in [0..len(T2)-1]:
            T2[y]=Ptmp(T2[y](g1=X1,g2=X2,g3=X3,g4=X4,s
             ↪  =Y1,t=Y2))(X1=x1,X2=x2,X3=x3,X4=x4,Y1=S,Y2=T)
        S2=Set(solve_mod(T2,p))
        GEm=(S1.intersection(S2)).list() #GElements
        if (0,0,0,0) in GEm:
            GEm.remove((0,0,0,0))
        pretty_print(htm
         ↪  l('<br>Let us choose $s=%s, t=%s$'%(late
         ↪  x(ST[0][0]),latex(ST[0][1]))))
        pretty_print(htm
         ↪  l('<br>Possible values of $g1,g2,g3,g4:$'))
        for y in GEm:
            pretty_print(y)
        G=matrix(GF(p),2,2,GEm[0])
        pretty_print(htm
         ↪  l('<br>Fix the solution to be $%s,$ then $g$ is: '%(late
         ↪  x(GEm[0]))))
        pretty_print(html('$g=%s$'%(latex(G))))
        W1=G*D-Bt*G
        W2=G*C-As*G

        P.<i,j,k,l>=PolynomialRing(GF(p),4)
        M1=G*M*G.inverse(); MOr=M; M=M1

        pretty_print(htm
         ↪  l('We apply the method developed for $A,B$ with:'))
        pretty_print(html(r'$M_1=%s=%s %s %s$'%(late
         ↪  x(M1),latex(G),latex(MOr),latex(G.inverse()))))
        Ai=matrix(P,[[1,i],[0,1]]); Ak=matrix(P,[[1,k],[0,1]])
        Bj=matrix(P,[[1,0],[j,1]]); Bl=matrix(P,[[1,0],[l,1]])
        Mi=Ai*Bj*Ak*Bl

```

```
        I=ideal(Mi[v][w]-M[v][w] for v in [0..1] for w in
        ↪ [0..1])
        IList=[Mi.list()[w]-M[w//2][w%2] for w in [0..3]]
        GB=I.groebner_basis()
        LIS=[Ptmp」
        ↪ (g(i=X1,j=X2,k=X3,l=X4))(X1=x1,X2=x2,X3=x3,X4=x4)==0
        ↪ for g in GB]
        Sol=solve_mod(LIS,p)
        s=ST[0][0]; t=ST[0][1]
        SolFn=[]
        for y in [0..len(Sol)-1]:
            SolFn.append((Sol[y][0]*inverse_mod(int(s),p」
            ↪ ),Sol[y][1]*inverse_mod(int(t),p」
            ↪ ),Sol[y][2]*inverse_mod(int(s),p」
            ↪ ),Sol[y][3]*inverse_mod(int(t),p)))
        pretty_print(html('<br>Solutions $(i,j,k,l):$'))
        for t in [0..len(Sol)-1]:
            pretty_print(SolFn[t])
```

7.3 SageMath summary

function	description
groebner_basis()	Returns the Groebner basis of an ideal.
ideal()	Returns an ideal generated by polynomials in a ring.
multiplicative_order()	Returns the multiplicative order of an element.
solve_mod()	Solves certain equations/systems of equations modulo an integer.

Exercises

1. Let

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

be matrices over $\mathbb{F}_{13}$. Determine all solutions of the discrete logarithm problem

$$A^i B^j A^k B^l = \begin{pmatrix} 0 & 1 \\ 12 & 4 \end{pmatrix}.$$

2. Let

$$C = \begin{pmatrix} 9 & 3 \\ 7 & 10 \end{pmatrix}, \quad D = \begin{pmatrix} 0 & 12 \\ 7 & 2 \end{pmatrix}$$

be matrices over $\mathbb{F}_{17}$. Determine all solutions of the discrete logarithm problem

$$C^i D^j C^k D^l = \begin{pmatrix} 14 & 12 \\ 3 & 16 \end{pmatrix}.$$

3.

4.

5.

Chapter 8 The NTRU cryptosystem

Summary

- Lattice based attack on NTRU
- CTRU: polynomials over $\mathbb{F}_2$
- ITRU: a variant of NTRU
- SageMath summary
- Exercises

The NTRU cryptosystem was developed in 1996 by Hoffstein, Pipher and Silverman [11]. NTRU is a public key cryptosystem not based on factorization or discrete logarithm problems. NTRU is based on the shortest vector problem in a lattice. The NTRU public key cryptosystem is one of the fastest known public key cryptosystem. NTRU works in the ring of truncated polynomials

$$R_q = \frac{\mathbb{Z}_q[X]}{X^N - 1},$$

where N is a fixed positive integer and $\mathbb{Z}_q$ denotes the ring of integers modulo q. In this ring a polynomial f can be written as

$$f = (f_0, f_1, \ldots, f_{N-1}) = \sum_{k=0}^{N-1} f_k X^k.$$

The addition of two polynomials $f = (f_0, f_1, \ldots, f_{N-1})$ and $g = (g_0, g_1, \ldots, g_{N-1})$ is defined as pairwise addition of the coefficients of the same degree, that is

$$f + g = (f_0 + g_0, f_1 + g_1, \ldots, f_{N-1} + g_{N-1}).$$

The multiplication of two polynomials f and g is given by a convolution multiplication as follows

$$f \star g = h = (h_0, h_1, \ldots, h_{N-1}), \text{ where } h_k = \sum_{i+j \equiv k \bmod N} f_i g_j.$$

As an example let us compute

$$(x^4 + 2x^3 + 3) \star (x^4 + 3x^2 + x + 2) = x^8 + 2x^7 + 3x^6 + 7x^5 + 7x^4 + 4x^3 + 9x^2 + 3x + 6.$$

In the polynomial ring $\frac{\mathbb{Z}_3[X]}{X^5-1}$ it can be written as

$$x^3 + 2x^2 + 3x + 7 + 7x^4 + 4x^3 + 9x^2 + 3x + 6 = x^4 + 2x^3 + 2x^2 + 1.$$

For a given positive integer d define the set

$$B(d) = \left\{ \sum_{k=0}^{N-1} f_k X^k : f_i = 0 \text{ or } 1, \sum_{k=0}^{N-1} f_k = d \right\}.$$

In NTRU the parameters are chosen as follows

- N is a sufficiently large prime,
- p and q are relatively prime numbers such that q is much larger than p.
- d_f, d_g and d_r are integers such that the polynomials from which the private keys are selected are from the set $B(d_f)$ and $B(d_g)$. The set $B(d_r)$ contains the polynomials from which the blinding value used during encryption is selected.
- $\frac{\mathbb{Z}_p[X]}{X^N-1}$ is the plaintext space.

Consider now the key generation.

- Randomly choose a polynomial $f \in B(d_f)$ such that $f$has an inverse modulo p and q as well,
- set $f_p \equiv f^{-1} \pmod{p}$ and $f_q \equiv f^{-1} \pmod{q}$.
- Randomly choose a polynomial $g \in B(d_g)$.
- Compute $h \equiv g \star f_q \pmod{q}$.
- public key: (N, h), private key (f, f_p).

Let us describe the encryption.

- The message is represented as a polynomial m from the plaintext space.
- Randomly choose a polynomial $r \in B(d_r)$.
- Encrypt m using the rule $e \equiv p \star r \star h + m \pmod{q}$.

The steps used in the decryption can be given as follows.

- Compute $a \equiv f \star e \pmod{q}$.
- Transform a to a polynomial with coefficients in the interval $[-q/2, q/2[$.
- Compute $m \equiv f_p \star a \pmod{p}$.

Let us check the computation to see why the above procedure works. We have

$$\begin{aligned} a &\equiv f \star e \pmod{q} \\ &\equiv f \star (p \star r \star h + m) \pmod{q} \\ &\equiv p \star r \star g \star f \star f_q + f \star m \pmod{q} \\ &\equiv p \star r \star g + f \star m \pmod{q}. \end{aligned}$$

We obtain that

$$f_p \star a \pmod{p} \equiv (p \star r \star g + f \star m) \star f_p \pmod{p} \equiv m \pmod{p}.$$

To illustrate the NTRU encryption/decryption let us see an example with

$$N = 7, p = 3, q = 41,$$
$$f = X^6 - X^4 + X^3 + X^2 - 1,$$
$$g = X^6 + X^4 - X^2 - X,$$
$$m = -X^5 + X^3 + X^2 - X + 1,$$
$$r = X^6 - X^5 + X - 1.$$

Here we get

$$f_p = X^6 + 2X^5 + X^3 + X^2 + X + 1,$$
$$f_q = 8X^6 + 26X^5 + 31X^4 + 21X^3 + 40X^2 + 2X + 37,$$
$$h = 19X^6 + 38X^5 + 6X^4 + 32X^3 + 24X^2 + 37X + 8,$$
$$e = 31X^6 + 19X^5 + 4X^4 + 2X^3 + 40X^2 + 3X + 25.$$

In SageMath it can be implemented as follows.

NTRU

```
def cmap(t,p):
    if (ZZ(t)%p)>(p//2):
        return ((ZZ(t)%p)-p)
    else:
        return ZZ(t)%p

N=7
p=3
q=41
print "(N,p,q)=",(N,p,q)
Zx.<X> = ZZ[]
f=X^6-X^4+X^3+X^2-1
g=X^6+X^4-X^2-X
print "f=",f
print "g=",g
Pp.<b>=PolynomialRing(GF(p))
Pq.<c>=PolynomialRing(GF(q))
f3=Pp(f).inverse_mod(b^N-1)
f41=Pq(f).inverse_mod(c^N-1)
print "f_p=",f3(b=X)
```

```
print "f_q=",f41(c=X)
h=(p*f41*Pq(g))%(c^N-1)
print "public key h: ",h(c=X)
r=X^6-X^5+X-1
print "r=",r
m=-X^5+X^3+X^2-X+1
print "message m:",m
em=(Pq(r)*h)%(c^N-1)+Pq(m)%(c^N-1)
print "encrypted m: ", em(c=X)
A=(Pq(f)*em)%(c^N-1)
print A(c=X)
A1=[cmap(k,q) for k in A.list()]
print Zx(A1)
print "decrypted message m: ", Zx([cmap(k,p) for k in
↪  Pp((Zx(A1)*Zx(f3))%(X^N-1)).list()])

```

Output

```
(N,p,q)= (7, 3, 41)
f= X^6 - X^4 + X^3 + X^2 - 1
g= X^6 + X^4 - X^2 - X
f_p= X^6 + 2*X^5 + X^3 + X^2 + X + 1
f_q= 8*X^6 + 26*X^5 + 31*X^4 + 21*X^3 + 40*X^2 + 2*X + 37
public key h:  19*X^6 + 38*X^5 + 6*X^4 + 32*X^3 + 24*X^2 + 37*X + 8
r= X^6 - X^5 + X - 1
message m: -X^5 + X^3 + X^2 - X + 1
encrypted m:  31*X^6 + 19*X^5 + 4*X^4 + 2*X^3 + 40*X^2 + 3*X + 25
X^6 + 10*X^5 + 33*X^4 + 40*X^3 + 40*X^2 + X + 40
X^6 + 10*X^5 - 8*X^4 - X^3 - X^2 + X - 1
decrypted message m:  -X^5 + X^3 + X^2 - X + 1
```

8.1 Lattice based attack on NTRU

The public key satisfies

$$h \equiv g \star f_q \pmod{q}$$

hence we have that $f \star h \equiv g \pmod{q}$. Consider the lattice defined by

$$\Lambda = \left\{(F_1, F_2) \in R_q \times R_q : F_1 \star h \equiv F_2 \pmod{q}\right\}.$$

Obviously $(f, g) \in \Lambda$. The relation $f \star h \equiv g \pmod{q}$ can be written as

$$f \star h - u \star q = g \text{ for some } u \in R_q.$$

The above equation is the same as

$$\begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ h & q \end{pmatrix} \begin{pmatrix} f \\ -u \end{pmatrix}$$

or in a more useful form

$$\begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \\ g_0 \\ g_1 \\ \vdots \\ g_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ h_0 & h_1 & \cdots & h_{N-1} & q & 0 & \cdots & 0 \\ h_{N-1} & h_0 & \cdots & h_{N-2} & 0 & q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \cdots & h_0 & 0 & 0 & \cdots & q \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \\ -u_0 \\ -u_1 \\ \vdots \\ -u_{N-1} \end{pmatrix}$$

Let us remark that the coefficients of the polynomials f and g are small, therefore (f, g) is a short vector in the lattice Λ. That is the LLL-algorithm can be applied. Let us apply the above ideas in case of the example we considered. The appropriate lattice can be obtained in SageMath as follows.

NTRU-Lattice

```
N=7
p=3
q=41
Zx.<X> = ZZ[]
h=19*X^6 + 38*X^5 + 6*X^4 + 32*X^3 + 24*X^2 + 37*X + 8
M = matrix(2*N)
for i in [0..N-1]: M[i,i] = 1
for i in [N..2*N-1]: M[i,i] = q
```

```
for i in [0..N-1]:
    for j in [0..N-1]:
        M[i+N,j] = ((Zx(GF(q)(1/p)*h)*X^i)%(X^N-1))[j]
M

```

Output

```
[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0]
[30 26  8 38  2 40 20 41  0  0  0  0  0  0]
[20 30 26  8 38  2 40  0 41  0  0  0  0  0]
[40 20 30 26  8 38  2  0  0 41  0  0  0  0]
[ 2 40 20 30 26  8 38  0  0  0 41  0  0  0]
[38  2 40 20 30 26  8  0  0  0  0 41  0  0]
[ 8 38  2 40 20 30 26  0  0  0  0  0 41  0]
[26  8 38  2 40 20 30  0  0  0  0  0  0 41]
```

We apply the LLL-algorithm to find short vectors in the lattice.

NTRU-LLL

```
print M.transpose().LLL()
print f.coefficients(sparse=False)
print g.coefficients(sparse=False)

```

Output

```
[ -1  -1  -1  -1  -1  -1  -1   0   0   0   0   0   0   0]
[ -1   0   1  -1  -1   0   1  -1   0  -1   0   1   1   0]
[  1  -1  -1   0   1  -1   0  -1   0   1   1   0  -1   0]
[  0   1   0   1  -1   0   1   0  -1  -1   0   0   2   0]
[  0   1  -1   0   1  -1  -1   1   0  -1   0  -1   0   1]
[  1  -2   0   0   0  -1  -1   0   0   1  -1   0  -1   1]
[  1   0   1  -1   0   1   0  -1  -1   0   0   2   0   0]
[-10  -1   0   0   8   1  -1  -3  -5   8  -5  -1   5   1]
[  1  -2  -9   0   0  -1   9   5   2  -3  -6   7  -5   0]
[  0   0   8   1  -1 -10  -1   8  -5  -1   5   1  -3  -5]
[ -1   9   1  -2  -9   0   0  -5   0   5   2  -3  -6   7]
[  9   1  -2  -9   0   0  -1   0   5   2  -3  -6   7  -5]
[ -1   0   0   8   1  -1 -10  -5   8  -5  -1   5   1  -3]
[ -3  -1   6  -2   6  -2  -2  -6 -11  -3  -4  -9   1  -9]


[-1, 0, 1, 1, -1, 0, 1]
[0, -1, -1, 0, 1, 0, 1]
```

8.2 CTRU: using polynomials over $\mathbb{F}_2$

Gaborit, Ohler and Solé [8] introduced a variant of NTRU for which lattice based attacked can be avoided. Let N be a positive integer and $R = A[X]/(X^N - 1)$, where A denotes the ring of polynomials over $\mathbb{F}_2$. Two irreducible polynomials P and Q in $A[X]$ are fixed. Let us denote the degrees of these polynomials by s and t, that is $\deg P = s, \deg Q = t$. These numbers should satisfy that $2 \leq s \leq t$ and $\gcd(s,t) = 1$. This latter gcd condition is needed to have $\mathbb{F}_{2^s} \cap \mathbb{F}_{2^t} = \mathbb{F}_2$. For a given polynomial

$$F = F_0(T) + F_1(T)X + \ldots + F_{N-1}X^{N-1} \in R$$

the maximum degree in T of the coefficients of X is denoted by $\deg_T(F)$. Define the set $\mathcal{L}(d)$ in the following way

$$\mathcal{L}(d) = \{F \in R : \deg_T(F) < d\}.$$

For a given X^i we have the coefficient polynomial

$$F_{i,0} + F_{i,1}T + \ldots + F_{i,d-1}T^{d-1},$$

where $F_{i,j} \in \mathbb{F}_2$. Hence the number of possible coefficient polynomials is 2^d. There are N coefficients, therefore the size of the set $\mathcal{L}(d)$ is 2^{dN}. In CTRU we use three additional parameters d_f, d_g and d_ϕ, these are integers between 1 and $t-1$. Alice and Bob follow the steps given below.

- Alice chooses a polynomial $f \in \mathcal{L}(d_f + 1)$ such that it has an inverse modulo P and also modulo Q. Denote by f_P the inverse of f modulo P and by f_Q the inverse modulo Q. She also picks a polynomial $g \in \mathcal{L}(d_g + 1)$. The public key is
$$h = g * f_Q \pmod{Q}.$$
- Bob needs two polynomials to send a message, the first one is m, the polynomial representing the message and the second one is a random polynomial from $\mathcal{L}(d_\phi + 1)$ denoted by ϕ. He encrypts the message using the formula
$$e = P * \phi * h + m \pmod{Q}.$$
- Alice receives e and she computes
$$a = e * f = P * \phi * h * f + m * f = P * \phi * g + m * f \pmod{Q}$$
and
$$a * f_P = P * \phi * g * f_P + m * f * f_P = m \pmod{P}.$$

8.3 ITRU: a variant of NTRU

In 2017 Gaithuru, Salleh and Mohamad [9] introduced a variant of NTRU called ITRU. Instead of working in a truncated polynomial ring ITRU is based on the ring of integers. The parameters and the main steps of ITRU are as follows.

- The value of p is suggested to be 1000.
- Random integers f, g and r are chosen such that f is invertible modulo p.
- A prime q is fixed satisfying $q > p \cdot r \cdot g + f \cdot m$, where m is the representation of the message in decimal form. The suggested conversion is based on ASCII conversion tables, that is the one with $a \longrightarrow 97$.
- One computes $F_p = f^{-1} \pmod{p}$ and $F_q = f^{-1} \pmod{q}$. These computations can be done by using the extended Euclidean algorithm.
- The public key is $h \equiv p \cdot F_q \cdot g \pmod{q}$ and q.
- The encryption is similar to the one applied in NTRU, one generated a random integer r and computes
$$e \equiv r \cdot h + m \pmod{q}.$$
- To get the plaintext from the ciphertext one determines $a \equiv f \cdot e \pmod{q}$.

Recovering the message is done by computing

$$F_p \cdot a \pmod{p}.$$

We need to show that we obtain the original message at the end. We have

$$a \equiv f \cdot e \equiv f(r \cdot h + m) \equiv f(r \cdot p \cdot F_q \cdot g + m) \equiv r \cdot p \cdot g + f \cdot m \pmod{q}.$$

Here we used the fact that $f \cdot F_q \equiv 1 \pmod{q}$. It remains to compute $F_p \cdot a \pmod{p}$. We obtain that

$$F_p \cdot a \equiv F_p(r \cdot p \cdot g + f \cdot m) \equiv F_p \cdot f \cdot m \equiv m \pmod{p}.$$

We note that to fix q one needs a bound for the largest possible value of the representation, so here if one only uses the letters from 'A' to 'Z' and 'a' to 'z', then the maximum is 122. In the SageMath implementation we will use 255.

ITRU

```
s='cryptography'
pretty_print('The message is : ', s)
r=8
p=1000
F=Set([k for k in range(2,1000) if gcd(k,1000)==1])
f=F.random_element()
S=Set([2..1000])
g=S.random_element()
m=[ord(k) for k in s]
pretty_print('The ASCII code of the message :',m)
q=next_prime(p*r*g+255*f)
Fp=(1/f)%p
Fq=(1/f)%q
h=(p*Fq*g)%q
pretty_print('Large modulus :', q)
pretty_print('Public key :',h)
pretty_print('Private key pair :', (f,Fp))
e=[((r*h)+m[i])%q for i in [0..len(m)-1]]
pretty_print('The encrypted message :',e)
a=[(f*e[i])%q for i in [0..len(e)-1]]
pretty_print(html(r'$f\cdot e\pmod{q}$ is : $%s$'%latex(a)))
C=[(Fp*a[l])%p for l in [0..len(a)-1]]
```

```
23 pretty_print(html(r'$F_p\cdot a\pmod{q}$ is : $%s$'%latex(C)))
24 D=[chr(k) for k in C]
25 pretty_print('The original message :', ''.join(D))
26
```

Output

The message is :cryptography

The ASCII code of the message : [99, 114, 121, 112, 116, 111, 103, 114, 97, 112, 104, 121]

Large modulus :3699583

Public key :1470026

Private key pair : (861, 741)

The encrypted message : [661558, 661573, 661580, 661571, 661575, 661570, 661562, 661573, 661556, 661571, 661563, 661580]

$f \cdot e \pmod{q}$ is :
[3565239, 3578154, 3584181, 3576432, 3579876, 3575571, 3568683, 3578154, 3563517, 3576432, 3569544, 3584181]

$F_p \cdot a \pmod{q}$ is : [99, 114, 121, 112, 116, 111, 103, 114, 97, 112, 104, 121]

The original message :cryptography

Let us encrypt a paragraph from the article describing ITRU [9] (without spaces):

```
'ThegoalofthisstudyistopresentavariantofNTRUwhichisbasedontheringof
integersasopposedtousingthepolynomialringwithintegercoefficients.
WeshowthatNTRUbasedontheringofintegers(ITRU),hasasimpleparameter
selectionalgorithm,invertibilityandsuccessfulmessagedecryption.
Wedescribeaparameterselectionalgorithmandalsoprovideanimplementation
ofITRUusinganexample.ITRUisshowntohavesuccessfulmessagedecryption,
whichprovidesmoreassuranceofsecurityincomparisontoNTRU.'
```

If the large modulus is 1104427 and the public key is given by 37619, then the ciphertext starts as

301036, 301056, 301053, 301055, 301063, 301049, 301060, 301063, 301054 . . . .

There are 32 different numbers appearing in the ciphertext these are between 300992 and 301073. A simple frequency analysis provides the following data

```
[(301056, 0.0380313199105145), (301057, 0.0850111856823266),
(301060, 0.0313199105145414), (301061, 0.0290827740492170),
(301062, 0.0648769574944072), (301063, 0.0738255033557047),
(301064, 0.0313199105145414), (301066, 0.0536912751677852),
(301067, 0.0850111856823266), (301068, 0.0693512304250559),
```

```
(301069, 0.0201342281879195), (301070, 0.0111856823266219),
(301071, 0.0111856823266219), (301072, 0.00223713646532438),
(301073, 0.0134228187919463), (300992, 0.00223713646532438),
(300993, 0.00223713646532438), (300996, 0.00671140939597315),
(300998, 0.00894854586129754), (301025, 0.00671140939597315),
(301030, 0.00671140939597315), (301034, 0.0134228187919463),
(301036, 0.0156599552572707), (301037, 0.0134228187919463),
(301039, 0.00447427293064877), (301049, 0.0693512304250559),
(301050, 0.00894854586129754), (301051, 0.0357941834451902),
(301052, 0.0246085011185682), (301053, 0.109619686800895),
(301054, 0.0223713646532438), (301055, 0.0290827740492170)]
```

That is the number 301053 appears the most in the ciphertext. Our guess is that 301053 represents either 'e', 'a' or 't'. If it is 'e', then we apply the formula $c_i - 300952$ and we get a sequence of numbers starting with

$$84, 104, 101, 103, 111, 97, 108, 111.$$

If we consider it as a sequence of ASCII codes and determine the corresponding plaintext, then we get the encoded message.

8.4 SageMath summary

function	description
`LLL()`	Returns an LLL reduced system of vectors.
`chr()`	Returns an ASCII character given by a code.
`coefficients()`	Returns coefficients of a polynomial.
`ord()`	Returns the ASCII code of a given character.

Exercises

1. Let $N = 11$, $p = 3$ and $q = 32$. Determine polynomials f_p and f_q such that $f \star f_p \equiv 1 \pmod{p}$ and $f \star f_q \equiv 1 \pmod{q}$, where $f = -1 + X + X^2 + X^4 - X^5 + X^8 - X^{10}$ is an element of $\mathbb{F}[X]/(X^{11} - 1)$.
2. NTRU encryption. Let $N = 11$, $p = 3$ and $q = 23$. We choose $f = x^{10} - x^9 + x^7 + x^4 - 1$ and $g = x^{10} + x^6 - x^3 - x^2 + x + 1$. Alice would like to send the message $m = x^{10} + 2x^9 + x^5 + x + 1$ to Bob. Alice chooses the random polynomial $r = x^9 - x^7 + x^6 + x^3 - x - 1$. Compute the polynomial that Alice sends to Bob.

3. NTRU decryption. Decrypt the message sent by Alice to Bob as it is described in the previous exercise.
4. Try to determine f and g by means of the LLL-algorithm in case of the NTRU cryptosystem described in the previous exercises, that is $N = 11, q = 23$ and
$$h = 12x^{10} + 21x^8 + 4x^7 + 12x^5 + 17x^4 + 22x^3 + 6x^2 + 16x + 7.$$
5. Encrypt and decrypt the message "Fully secure systems do not exist today and they will not exist in the future" by means of the ITRU cryptosystem with $f = 557, g = 806, r = 17$ and $q = 13844041$.

Chapter 9 Shamir's secret sharing

Summary

- Basic setup
- Example and implementation
- SageMath summary
- Exercises

9.1 Basic setup

In 1979 Blakley [3] and Shamir [21] independently introduced the idea of secret sharing. A (t, n) threshold secret sharing scheme is a method for n parties to distribute a secret such a way that at least t of them together can reconstruct it. Let us describe the steps of Shamir's secret sharing scheme. Let $\mathbb{F}_q$ be a finite field and the secret is $s \in \mathbb{F}_q$. We take $t-1$ random field elements $a_1, a_2, \ldots, a_{t-1}$ and we define the polynomial

$$f(x) = a_{t-1}x^{t-1} + \ldots + a_1 x + s.$$

That is we have that the secret is $f(0)$. The shares $(i, f(i))$ are distributed to the n distinct parties. To reconstruct the secret we may use a classical algorithm called Lagrange interpolation.

Theorem 9.1. Lagrange interpolation

Given t distinct points (x_i, y_i) than there is a polynomial $f(x)$ of degree less than t such that $f(x_i) = y_i$. The polynomial is as follows

$$f(x) = \sum_{i=1}^{t} y_i \prod_{\substack{1 \le j \le t \\ i \neq j}} \frac{x - x_j}{x_i - x_j}.$$

9.2 Example and SageMath implementation

Let us consider a concrete example. We work in the finite field $\mathbb{F}_{37}$ and suppose that using the shares

$$(3, 13), (4, 5), (10, 6), (13, 24), (22, 22), (30, 31)$$

one can reconstruct the secret. We obtain that

$$f(0) = \frac{13\cdot 4\cdot 10\cdot 13\cdot 22\cdot 30}{(4-3)(10-3)(13-3)(22-3)(30-3)} + \frac{5\cdot 3\cdot 10\cdot 13\cdot 22\cdot 30}{(3-4)(10-4)(13-4)(22-4)(30-4)} + \frac{6\cdot 3\cdot 4\cdot 13\cdot 22\cdot 30}{(3-10)(4-10)(13-10)(22-10)(30-10)} + \frac{24\cdot 3\cdot 4\cdot 10\cdot 22\cdot 30}{(3-13)(4-13)(10-13)(22-13)(30-13)} + \frac{22\cdot 3\cdot 4\cdot 10\cdot 13\cdot 30}{(3-22)(4-22)(10-22)(13-22)(24-22)} + \frac{31\cdot 3\cdot 4\cdot 10\cdot 13\cdot 22}{(3-31)(4-31)(10-31)(13-31)(22-31)} = 8.$$

Therefore the secret is 8. We may also apply linear algebra to reconstruct the secret. Since $t = 6$ we need to consider a polynomial of the form

$$f(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + s.$$

The shares yield the system of linear equations

$$\begin{aligned}
31 = f(3) &= a_5 3^5 + a_4 3^4 + a_3 3^3 + a_2 3^2 + a_1 3 + s\\
5 = f(4) &= a_5 4^5 + a_4 4^4 + a_3 4^3 + a_2 4^2 + a_1 4 + s\\
6 = f(10) &= a_5 10^5 + a_4 10^4 + a_3 10^3 + a_2 10^2 + a_1 10 + s\\
24 = f(13) &= a_5 13^5 + a_4 13^4 + a_3 13^3 + a_2 13^2 + a_1 13 + s\\
22 = f(22) &= a_5 22^5 + a_4 22^4 + a_3 22^3 + a_2 22^2 + a_1 22 + s\\
31 = f(30) &= a_5 30^5 + a_4 30^4 + a_3 30^3 + a_2 30^2 + a_1 30 + s.
\end{aligned}$$

It follows that

$$\begin{pmatrix} 1 & 3 & 9 & 27 & 7 & 21\\ 1 & 4 & 16 & 27 & 34 & 25\\ 1 & 10 & 26 & 1 & 10 & 26\\ 1 & 13 & 21 & 14 & 34 & 35\\ 1 & 22 & 3 & 29 & 9 & 13\\ 1 & 30 & 12 & 27 & 33 & 28 \end{pmatrix}\begin{pmatrix} s\\ a_1\\ a_2\\ a_3\\ a_4\\ a_5 \end{pmatrix} = \begin{pmatrix} 13\\ 5\\ 6\\ 24\\ 22\\ 31 \end{pmatrix}$$

and the solution is given by $(8, 15, 19, 30, 5, 29)$. Thus the polynomial is $f(x) = 29x^5 + 5x^4 + 30x^3 + 19x^2 + 15x + 8$. The secret is the constant term, that is 8.

Now we provide a SageMath code to generate appropriate polynomials and shares and also a code to reconstruct the secret from given shares.

Shamir's secret sharing

```
@interact
def ShamirSecret(q=prime_range(19,80),n=slider(range(3,11),
    default=5)):
```

```
    gf=GF(q)
    S=gf.random_element()
    while S==0:
        S=gf.random_element()
    an=gf.random_element()
    while an==0:
        an=gf.random_element()
    A=[an]+[gf.random_element() for k in [1..n-1]]+[S]
    P.<X>=gf[]
    F=sum([A[k]*X^(len(A)-1-k) for k in [0..len(A)-1]])
    pretty_print(html('Randomly polynomial: $%s$'%latex(F)))
    pretty_print(html('The secret: $%s$'%latex(S)))
    T=Set([])
    while T.cardinality()<n+1:
        t=gf.random_element()
        while t==0:
            t=gf.random_element()
        T=T.union(Set([t]))
    Shares=[(k,F(X=k)) for k in T]
    pretty_print(html('Generated shares: $%s$'%latex(Shares)))
```

Output

q 19

n 5

Randomly generated polynomial: $18X^5 + 8X^4 + 13X^3 + 4X^2 + 10X + 16$

The secret: 16

Generated shares: $[(1, 12), (2, 5), (4, 0), (10, 14), (12, 6), (14, 11)]$

Reconstruction

```
@interact
def Reconstruct(q=('q',19),S0=input_box(defau
↪   lt='[(1,12),(2,5),(4,0),(10,14),(12,6),(14,11)]', type =
↪   str, label = 'Shares')):
    %display latex
    Shares=sage_eval(S0)
```

```
m=matrix(GF(q),[[j[0]^k for k in [0..len(Shares)-1]] for j in Shares])
v=vector(GF(q),len(Shares),[j[1] for j in Shares])
P=PolynomialRing(GF(q),len(Shares),'a')
aa=vector(P,[k for k in P.gens()]).column()
pretty_print(html('Linear algebra implies the equation:'))
pretty_print(html('$%s\cdot%s=%s$'%(latex(m),latex(aa),latex(v.column()))))
tvek=m.inverse()*v
pretty_print(html('We obtain that $%s=%s$'%(latex(aa),latex(tvek.column()))))
pretty_print(html('The secret is the constant term,'))
m3str='that is $S=a_0,$ in this case: $S=%s.$'
pretty_print(html(m3str%latex(tvek[0])))
R = PolynomialRing(GF(q), "x")
LagPol=R.lagrange_polynomial(Shares, algorithm="neville")[-1]
pretty_print(html('Other approach is Lagrange-interpolation:'))
pretty_print(html('Let R=PolynomialRing(GF(q),"x").'))
m4str='R.lagrange_polynomial(Shares,algorithm="neville")[-1],'
pretty_print(html(m4str))
pretty_print(html('where Shares=$%s.$'%latex(Shares)))
pretty_print(html('The polynomial is: $%s.$'%latex(LagPol)))
pretty_print(html('Hence the secret is: $%s.$'%latex(LagPol(x=0))))
```

Output

q 19

Shares [(1,12),(2,5),(4,0),(10,14),(12,6),(1·

Linear algebra implies the equation:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 13 \\ 1 & 4 & 16 & 7 & 9 & 17 \\ 1 & 10 & 5 & 12 & 6 & 3 \\ 1 & 12 & 11 & 18 & 7 & 8 \\ 1 & 14 & 6 & 8 & 17 & 10 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} 12 \\ 5 \\ 0 \\ 14 \\ 6 \\ 11 \end{pmatrix}$$

We obtain that $\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} 16 \\ 10 \\ 4 \\ 13 \\ 8 \\ 18 \end{pmatrix}$

The secret is the constant term,

that is $S = a_0$, in this case: $S = 16$.

Other approach is Lagrange-interpolation:

Let Pq=PolynomialRing(GF(q),"x").

Pq.lagrange_polynomial(Shares,algorithm="neville")[-1],

where Shares=$[(1, 12), (2, 5), (4, 0), (10, 14), (12, 6), (14, 11)]$.

The polynomial is: $18x^5 + 8x^4 + 13x^3 + 4x^2 + 10x + 16$.

Hence the secret is: 16.

9.3 SageMath summary

function	description
`cardinality()`	Returns the number of elements of a set.
`lagrange_polynomial()`	Returns the Lagrange polynomial corresponding to some points.
`union()`	Returns the union of two sets.

Exercises

1. Use Shamir's secret sharing to share $s = 19$ over $\mathbb{F}_{79}$ such that 4 users can

reconstruct the secret. Generate at least 6 shares.

2. Shamir's secret sharing with $p = 31$. We know the following shares and we also know that 3 parties can reconstruct the secret

$$S_1 = 16, \qquad S_4 = 16,$$
$$S_2 = 5, \qquad S_5 = 7,$$
$$S_3 = 5, \qquad S_6 = 9.$$

Recover the secret by using the shares S_1, S_2, S_3 and S_4, S_5, S_6.

Chapter 10 Knapsack cryptosystems

Summary

- Merkle–Hellman cryptosystem
- Attack based on LLL
- Chor-Rivest cryptosystem
- SageMath summary
- Exercises

10.1 Merkle–Hellman cryptosystem

The Knapsack problem is as follows. Given a collection of objects having both a weight and a kind of usefulness. Our goal is to fill a bag maximizing the usefulness of the items contained while restricted to an upper weight limit. General knapsack problems are difficult to solve, there is no known polynomial-time algorithm to handle these computations. However, in case of certain families the problem is easy to solve. Given objects with weights $\omega_1, \omega_2, \ldots, \omega_n$ it is our goal to find binary variables $v_1, v_2, \ldots, v_n$ such that

$$\sum_{i=1}^{n} v_i \omega_i = S,$$

where S is the weight limit to reach. A super-increasing weight system satisfies the following inequalities

$$\sum_{i=1}^{k-1} \omega_i < \omega_k \text{ for } k = 2, \ldots, n.$$

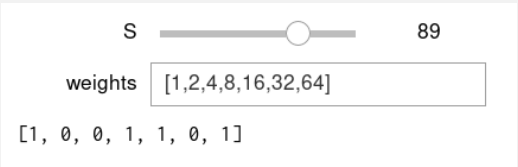
For example if we take $\omega_i = 2^{i-1}$ for $i = 1, 2, \ldots, n$, then we obtain a super-increasing sequence. A greedy type algorithm is provided below implemented in SageMath [23].

Super-increasing knapsack

```
@interact
def SuperIncreasing(S=slider(range(1,128),default=89),
    w=input_box(default='[1,2,4,8,16,32,64]', type=str,
    label='weights')):
    W=sage_eval(w)
    n=len(W)
```

```
    v=[0 for k in [1..n]]
    for k in [n-1,n-2..0]:
        if S>=W[k]:
            v[k]=1
            S=S-W[k]
    return v
```

Output



Here we obtain that $89 = 1 \cdot 1 + 0 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 0 \cdot 32 + 1 \cdot 64$. Thus the super-increasing knapsack is easy and general knapsacks are difficult that suggested the idea behind the Merkle-Hellman knapsack encryption [15]. Let us see the details of this cryptosystem.

- Alice chooses a super-increasing sequence $\omega_1, \omega_2, \ldots, \omega_n$ and a prime p such that
$$p > \sum_{i=1}^{n} \omega_i.$$
She also fix an integer $1 < r < p$.
- The public weights $\Omega_1, \Omega_2, \ldots, \Omega_n$ are computed by the formula
$$\Omega_i \equiv r \cdot \omega_i \pmod{p}.$$
- Bob only knows the general knapsack weights $\Omega_1, \Omega_2, \ldots, \Omega_n$ and he wants to send the (binary) message $[b_1, b_2, \ldots, b_n]$. He computes
$$M = \sum_{i=1}^{n} b_i \Omega_i,$$
that is the encrypted message. To encrypt longer messages, multiple blocks are encrypted.
- Alice receives M and she can easily compute $r^{-1} \pmod{p}$ by means of the extended Euclidean algorithm. She determines $r^{-1}M$ and solves the super-increasing knapsack problem.

A SageMath implementation of the above procedure is as follows.

Super-increasing knapsack

```
@interact
def MerkleHellman(n=slider(range(5,11),default=8)):
    o=[ZZ.random_element(1,100)]
    while len(o)!=n:
        o=o+[ZZ.random_element(sum(o),2*sum(o))]
    p=ZZ.random_element(sum(o),2*sum(o)).next_prime()
    r=ZZ.random_element(2,p-1)
    rinv=(1/r)%p
    O=[(k*r)%p for k in o]
    print "super-increasing privite sequence: ",o
    print "p: ",p
    print "r: ",r
    print "public key: ",O
    b=[ZZ.random_element(0,2) for _ in range(n)]
    print "binary message: ",b
    M=[b[k]*O[k] for k in [0..n-1]]
    print "encrypted message: ", sum(M)
    return o,(rinv*sum(M))%p
```

It creates a random super-increasing sequences and an appropriate prime number p and an integer $1 < r < p$. A random binary message is also composed and the corresponding encrypted message is computed. A possible output is as follows.

Output

```
n  8

super-increasing private sequence:  [42, 62, 137, 451, 1287, 2935, 6335, 12293]
p:  25447
r:  2786
public key:  [15224, 20050, 25424, 9583, 23002, 8423, 14539, 22083]
binary message:  [0, 1, 0, 0, 0, 1, 1, 0]
encrypted message:  43012

([42, 62, 137, 451, 1287, 2935, 6335, 12293], 9332)
```

In this example Alice receives the encrypted message 25036 and she needs to solve the super-increasing knapsack with respect of the sequence $[78, 136, 283, 899, 2533, 7588, 22503, 40389]$. Using the SageMath code `SuperIncreasing` one obtains the binary message $[0, 0, 0, 0, 1, 0, 1, 0]$. That is we have $25036 = 2533 + 22503$.

10.2 Attack based on the LLL-algorithm

Unfortunately, the knapsack cryptosystem that has been introduced is not secure against cryptanalysis attacks as pointed out by Shamir [20]. The problem can be reduced to find short vectors in lattices. The purpose of the Lenstra-Lenstra-Lovász (LLL) algorithm [13] is exactly this. Let us consider the example given above. We define vectors in the following form

$$\begin{aligned}
V_1 &= (1,0,0,0,0,0,0,0,78),\\
V_2 &= (0,1,0,0,0,0,0,0,136),\\
V_3 &= (0,0,1,0,0,0,0,0,283),\\
V_4 &= (0,0,0,1,0,0,0,0,899),\\
V_5 &= (0,0,0,0,1,0,0,0,2533),\\
V_6 &= (0,0,0,0,0,1,0,0,7588),\\
V_7 &= (0,0,0,0,0,0,1,0,22503),\\
V_8 &= (0,0,0,0,0,0,0,1,40389),\\
V_9 &= (0,0,0,0,0,0,0,0,-25036).
\end{aligned}$$

Our lattice is given by

$$\left\{V: \quad V=\sum_{i=1}^{9} z_i V_i, z_i \in \mathbb{Z}\right\}.$$

If we have a solution of the knapsack problem, then there exists a vector in the lattice having coordinates from the set $\{0,1\}$, that is a short vector. Let us see a SageMath code to apply the LLL-algorithm to the knapsack problem.

LLL application to knapsack

```
@interact
def LLLknapsack(v=input_box(type = str, label =
    'knapsack:',defau⌋
    lt='[78, 136, 283, 899, 2533, 7588, 22503, 40389,-25036]'
    )):
    %display latex
    V=sage_eval(v)
    n=len(V)
    k=matrix (ZZ,n,1,V)
    A = identity_matrix(ZZ,n-1)
```

```
    B = matrix(ZZ,n-1,1)
    A = A.augment(B)
    pretty_print(A.transpose().augment(k))
    return (A.transpose( )).augment(k).LLL()
```

Output

knapsack: [15224, 20050, 25424, 9583, 230(

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15224 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 20050 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 25424 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 9583 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 23002 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 8423 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 14539 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 22083 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -43012 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ -2 & -1 & 1 & 0 & 2 & 0 & 0 & 1 & 1 \\ -1 & -1 & 1 & 0 & 0 & 2 & -2 & 1 & 1 \\ -1 & 0 & -3 & 0 & -1 & 0 & -1 & 0 & -1 \\ -1 & -3 & -1 & 0 & -1 & 2 & 0 & -1 & -1 \\ -1 & -2 & -1 & 0 & -1 & 1 & 2 & 3 & 0 \\ -1 & 1 & -1 & 3 & 0 & 0 & -2 & -1 & 2 \\ -2 & 2 & 0 & -2 & -2 & -1 & 0 & -1 & 0 \\ 2 & -2 & -2 & 0 & 1 & 1 & 2 & 0 & 3 \end{pmatrix}$$

In the first row of the second matrix we see the binary message appearing.

10.3 Chor-Rivest cryptosystem

Cryptosystems based on knapsack problems used to be popular in public key cryptography, however many variants have been shown insecure by lattice reduction techniques. The Chor-Rivest [5] cryptosystem is one of the exceptions. We consider a finite field $\mathbb{F}_q$, where $q = p^h$ for some prime number p and positive integer $h \leq p$. To represent elements of $\mathbb{F}_q$ we fix a monic irreducible polynomial $f(x)$ of degree h over $\mathbb{F}_p$. The polynomials of degree at most $h-1$ over $\mathbb{F}_p$ are the elements of $\mathbb{F}_q$. The product of two elements $f_1(x)$ and $f_2(x)$ is the polynomial $f_1(x)f_2(x) \pmod{f(x)}$ of degree less than h. We determine a generator $g(x)$ of $\mathbb{F}_q$. We compute certain discrete logarithms of the

form

$$a_i = \log_{g(x)}(x+i),$$

where $i \in \mathbb{F}_p$. We also take a random permutation π of the set $\{0, 1, \ldots, p-1\}$ and a random integer d for which $0 \leq d \leq p^h - 2$. We compute

$$c_i \equiv (a_{\pi(i)} + d) \pmod{p^h - 1}, \text{ where } 0 \leq i \leq p-1.$$

The private key of Alice is given by $(f(x), g(x), d, \pi)$ and the public key is $(p, h, (c_0, c_1, \ldots, c_{p-1}))$. Let us describe the encryption.

- Bob would like to send to Alice a binary message $M = (M_0, M_1, \ldots, M_{p-1})$ containing exactly h ones.
- He computes

$$c = \sum_{i=0}^{p-1} M_i c_i.$$

  The ciphertext is c.

Alice receives c and follows the decryption algorithm given below.

- She determines $r \equiv (c - hd) \pmod{p^h - 1}$.
- She computes $s(x) \equiv g(x)^r \pmod{f(x)}$.
- She takes the monic polynomial $t(x) = f(x) + s(x)$ of degree h.
- She factors the polynomial $t(x)$, let say she has

$$t(x) = \prod_{j=1}^{h} (x + t_j).$$

- She applies the inverse of the permutation π to recover the positions of the ones in M.

This algorithm indeed will yield the original binary vector. This fact follows from the steps given below. We have that

$$\begin{aligned} s(x) &\equiv g(x)^r \pmod{f(x)} \equiv \\ g(x)^{c-hd} &\equiv g(x)^{\sum_{i=0}^{p-1} M_i c_i - hd} \equiv \\ g(x)^{\sum_{i=0}^{p-1} M_i((a_{\pi(i)}+d))-hd} &\equiv g(x)^{\sum_{i=0}^{p-1} M_i a_{\pi(i)}} \equiv \\ \prod_{i=0}^{p-1} (g(x)^{a_{\pi(i)}})^{M_i} &\equiv \prod_{i=0}^{p-1} (x + \pi(i))^{M_i} \pmod{f(x)}. \end{aligned}$$

We obtain that

$$t(x) = s(x) + f(x) = \prod_{i=0}^{p-1} (x + \pi(i))^{M_i}$$

and applying the inverse of π to the roots of $t(x)$ gives the coordinates of M that are 1.

ChorRivest

```
p=11
h=6
F.<q>=GF(p^h)
f=F.modulus()
pretty_print(html('We have that $(p,h)=%s$'%latex((p,h))))
pretty_print(html('The polynomial $f$ is $%s$'%latex(f)))
P=f.parent()
g = F.multiplicative_generator()
a = [discrete_log(q+i,g) for i in [0..p-1]]
pretty_print(htm
    ↪ l(r'The discrete logarithms $\log_{g(x)}(x+i)$ are $%s$'%late
    ↪ x(a)))
d = randint(0,p^h-2)
pretty_print(html('The random integer $d$ is $%s$'%latex(d)))
P11=Permutations(11)
p11=P11.random_element()
p11inv=p11.inverse()
pretty_print(htm
    ↪ l(r'The random permutation $\pi$ is $%s$'%latex(p11)))
pretty_print(htm
    ↪ l(r'The inverse permutation of $\pi$ is $%s$'%latex(p11in
    ↪ v)))
c = [(a[i-1]+d)%(p^h-1) for i in p11]
pretty_print(htm
    ↪ l(r'The public key $(c_0,c_1,\ldots,c_{p-1})$ is $%s$'%late
    ↪ x(c)))
M = [1,1,1,0,1,0,1,0,0,0,1]
pretty_print(html(r'The binary message $M$ is $%s$'%latex(M)))
C = sum(M[i]*c[i] for i in [0..p-1])
pretty_print(html(r'The ciphertext $c$ is $%s$'%latex(C)))
r = (C-h*d)%(p^h-1)
pretty_print(html(r'We compute $r=c-hd:$ it is $%s$'%latex(r
    ↪ )))
```

```
s=g^r
pretty_print(html(r'The polynomial $s(x)$ is $%s$'%latex(P(s.polynomial())%f)))
pretty_print(html(r'The polynomial $t(x)$ is $%s$'%latex(P(s.polynomial())%f+f)))
pretty_print(html(r'We obtain that $t(x)=%s$'%latex((P(s.polynomial())+f).factor())))
S=[-k[0] for k in (P(s.polynomial())+f).roots()]
M0=[0 for _ in [0..p-1]]
for k in S:
    M0[p11inv[k]-1]=1
pretty_print(html(r'The decrypted message is $%s$'%latex(M0)))
```

Output

We have that $(p, h) = (11, 6)$

The polynomial f is $x^6 + 3x^4 + 4x^3 + 6x^2 + 7x + 2$

The discrete logarithms $\log_{g(x)}(x + i)$ are
$[1, 101410, 703772, 279323, 1053498, 1565140, 1622352, 633225, 820891, 1765613, 1042680]$

The random integer d is 1031177

The random permutation π is $[4, 11, 8, 6, 3, 7, 5, 10, 1, 9, 2]$

The inverse permutation of π is $[9, 11, 5, 1, 7, 4, 6, 3, 10, 8, 2]$

The public key $(c_0, c_1, \ldots, c_{p-1})$ is
$[1310500, 302297, 1664402, 824757, 1734949, 881969, 313115, 1025230, 1031178, 80508, 1132587]$

The binary message M is $[1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1]$

The ciphertext c is 6457850

We compute $r = c - hd$: it is 270788

The polynomial $s(x)$ is $5x^5 + 8x^4 + 10x^3 + 7x^2 + 7x + 6$

The polynomial $t(x)$ is $x^6 + 5x^5 + 3x^3 + 2x^2 + 3x + 8$

We obtain that $t(x) = (x + 1) \cdot (x + 2) \cdot (x + 3) \cdot (x + 4) \cdot (x + 7) \cdot (x + 10)$

The decrypted message is $[1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1]$

10.4 SageMath summary

function	description
augment()	Returns a new matrix formed by appending the given matrix.
identity_matrix()	Returns the identity matrix of a given size.

Exercises

1. Generate a superincreasing sequence $\{w_1, w_2, \ldots, w_6\}$ such that $w_i \leq 2^i - 2^{i-1}$.
2. We have the superincreasing sequence $\{1, 2, 4, 10, 20, 40\}$. Pack a knapsack weighing 53.
3. Given a superincreasing sequence $S = \{2, 3, 7, 14, 30, 57, 120, 251\}$. Transform S into a general knapsack with $r = 41$ and $p = 491$. Encrypt the message $m = 10110111$.
4. We have the general knapsack $\{82, 123, 287, 83, 248, 373, 10, 471\}$ and a ciphertext is 548. Apply the LLL-algorithm to recover the original message.
5. Alice and Bob use the Chor-Rivest cryptosystem with $p = 13, h = 4, f(x) = x^4 + 3x^2 + 12x + 2$ and $g(x) = x, d = 3118$. The private permutation is given by $\pi = [1, 5, 6, 9, 0, 10, 11, 3, 2, 7, 8, 4, 12]$. Determine $(c_0, c_1, \ldots, c_{p-1})$ and encrypt the message $M = (1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0)$.
6. Decrypt the ciphertext obtained in the previous exercise.

Chapter 11 Solutions

11.1 Classical ciphers

Exercise 1

Encrypt the message MATHEMATICS with the shift cipher with 7 as the key.

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
```

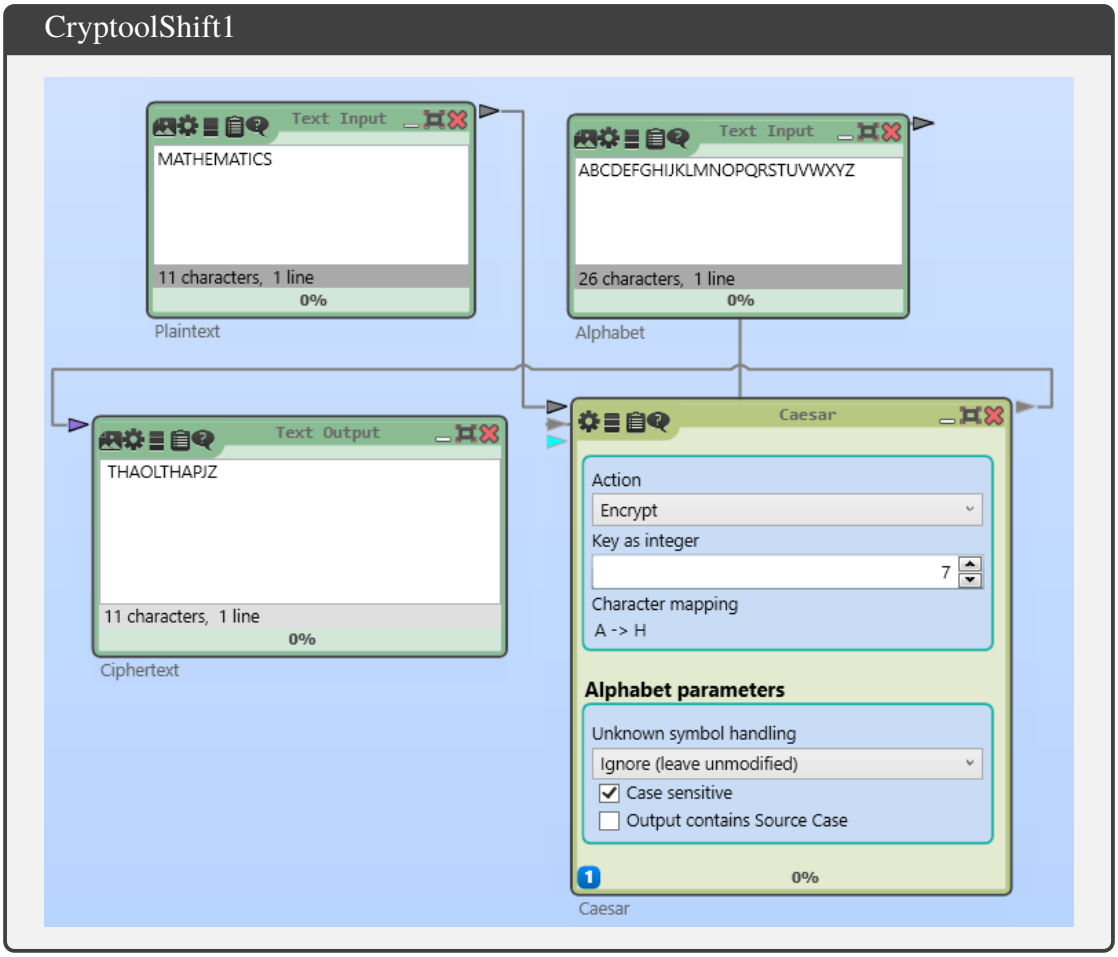
```
sage: P = S.encoding("mathematics")
```

```
sage: k=7
```

```
sage: C = S.enciphering(k,P)
```

```
sage: C
```

THAOLTHAPJZ



Exercise 2

Encrypt the message MATHEMATICS with the affine cipher with $(a, b) = (11, 5)$ as the key.

```
sage: A = AffineCryptosystem(AlphabeticStrings())
```

```
sage: P = A.encoding("mathematics")
```

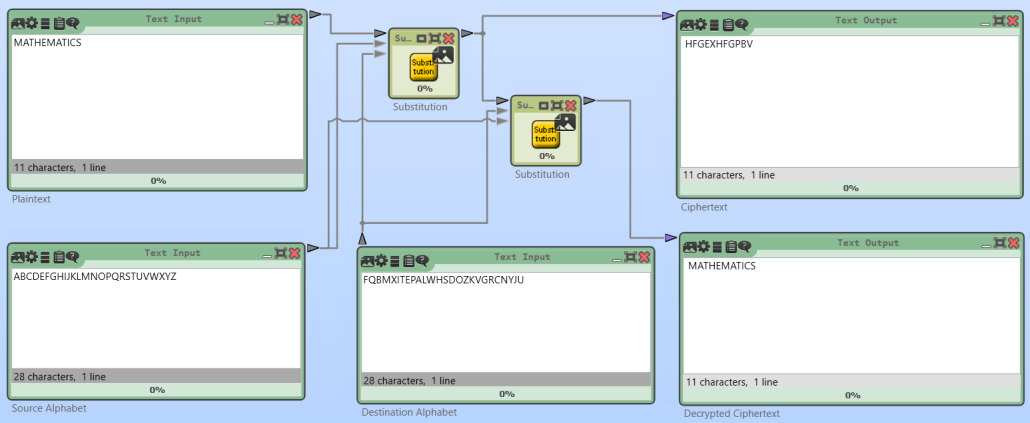
```
sage: (a,b)=(11,5)
```

```
sage: C = A.enciphering(a,b,P)
```

```
sage: C
```

HFGEXHFGPBV

CryptoolShift1



Exercise 3

Encrypt the message CRYPTOSYSTEM with the Hill cipher with

$$A = \begin{pmatrix} 5 & 15 \\ 24 & 1 \end{pmatrix} \qquad \text{and } b = (3, 5).$$

```
sage: R=Integers(26)
```

```
sage: A=matrix(R,2,2,[5,15,24,1])
```

```
sage: b=vector(R,[3,5])
```

```
sage: m=[ord(k)-65 for k in 'CRYPTOSYSTEM']
```

```
sage: vm=matrix(R,6,2,m)
```

```
sage: conv=[A*vm.row(k)+b for k in [0..5]]
```

```
sage: ct=''
```

```
sage: for k in conv: ct=ct+chr(ZZ(k[0])+65); ct=ct+chr(ZZ(k[1])
    +65)
```

```
sage: ct
```

```
ISKYWHLTOOVJ
```

Exercise 4

Encrypt the message MATHEMATICS with the Vigenère cipher with ROOT as the keyword.

```
sage: S = AlphabeticStrings()
```

```
sage: E = VigenereCryptosystem(S,4)
```

```
sage: K = S('ROOT')
```

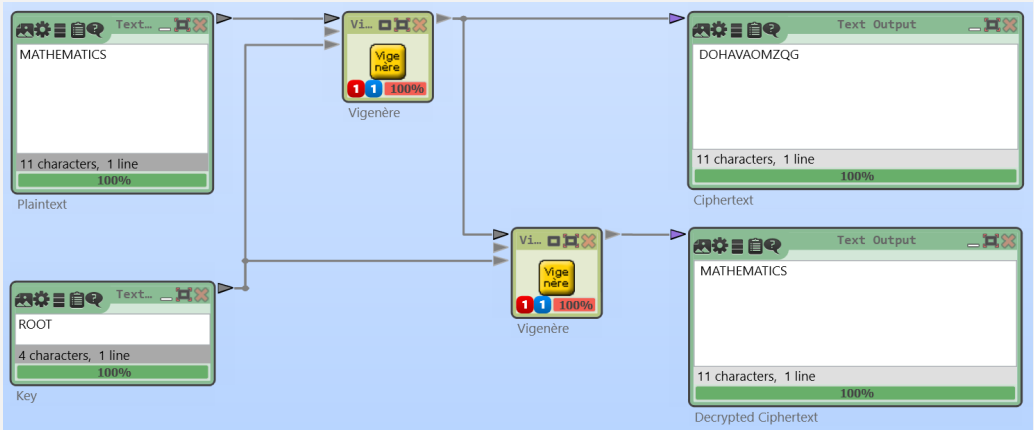
```
sage: e = E(K)
```

```
sage: P = S.encoding("mathematics")
```

```
sage: e(P)
```

DOHAVAOMZQG

CryptoolVigenère



Exercise 5

Decrypt the following message, which was encrypted with a shift cipher with 17 as the key: KIZREXCV.

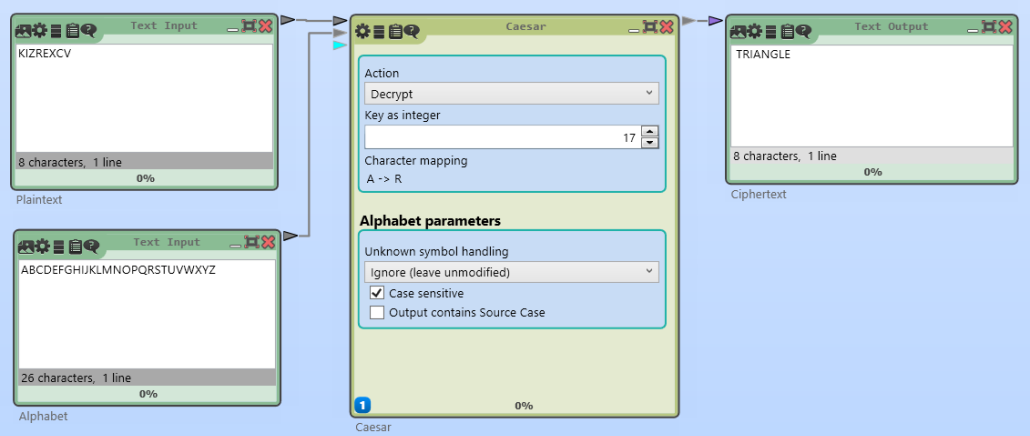
```
sage: S = ShiftCryptosystem(AlphabeticStrings())
```

```
sage: C = S.encoding("KIZREXCV")
```

```
sage: S.deciphering(17,C)
```

TRIANGLE

CryptoolShiftDecrypt



Exercise 6

Decrypt the following message, which was encrypted with an affine cipher with $(a, b) = (9, 10)$ as the key: ZHEKXMFU.

```
sage: A = AffineCryptosystem(AlphabeticStrings())
```

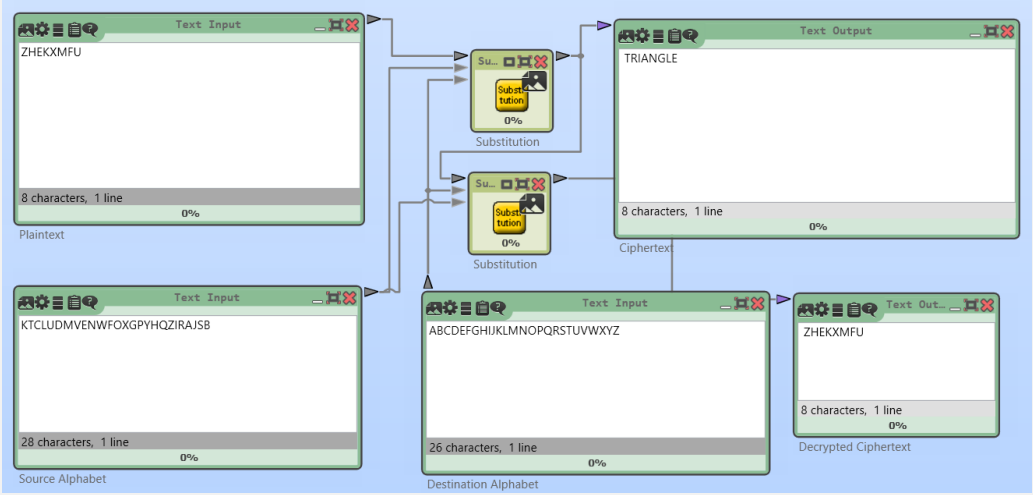
```
sage: C = A.encoding("ZHEKXMFU")
```

```
sage: (a,b)=(9,10)
```

```
sage: A.deciphering(a,b,C)
```

TRIANGLE

CryptoolShift1



Exercise 7

Decrypt the following message, which was encrypted with a Hill cipher with

$$A = \begin{pmatrix} 15 & 18 \\ 14 & 21 \end{pmatrix}: \qquad \text{DXQOTWNW}.$$

```
sage: S = AlphabeticStrings()
```

```
sage: H = HillCryptosystem(S,2)
```

```
sage: R = IntegerModRing(26)
```

```
sage: A = matrix(R,2,2,[15,18,14,21])
```

```
sage: C = S.encoding("DXQOTWNW")
```

```
sage: H.deciphering(A,C)
```

TRIANGLE

Exercise 8

Decrypt the following message, which was encrypted with a Vigenère cipher with KEY as the keyword: DVGKREVI.

```
sage: S = AlphabeticStrings()
```

```
sage: V = VigenereCryptosystem(S,3)
```

```
sage: K = S('KEY')
```

```
sage: C = S('DVGKREVI')
```

```
sage: V.deciphering(K,C)
```

$$TRIANGLE$$

Exercise 9

It is known that the following ciphertexts were encrypted using shift ciphers. Decrypt the messages and determine the keys that were used.

- XYNYLGCHUHN,
- EQCGQZOQ.

```
sage: A = AlphabeticStrings()
```

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
```

```
sage: C = A('XYNYLGCHUHN')
```

```
sage: bf = S.brute_force(C)
```

```
sage: sorted(bf.items())[20:21]
```

$$[(20, DETERMINANT)]$$

```
sage: A = AlphabeticStrings()
```

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
```

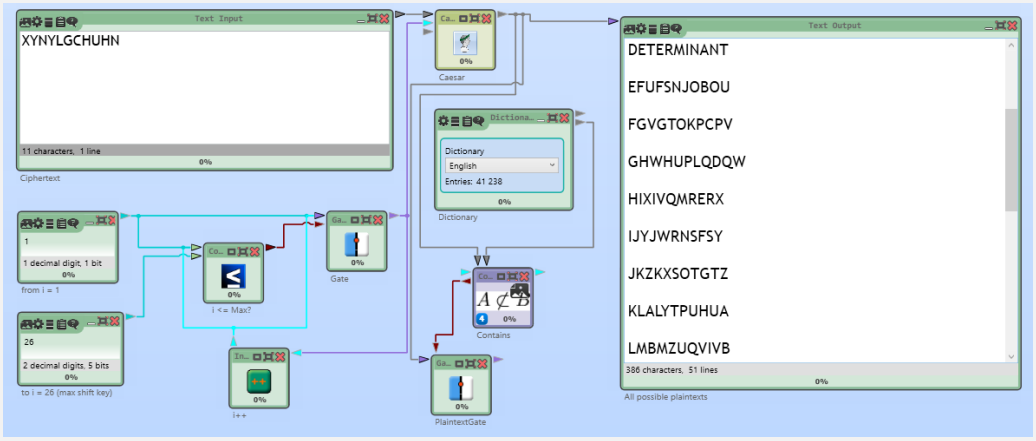
```
sage: C = A('EQCGQZOQ')
```

```
sage: bf = S.brute_force(C)
```

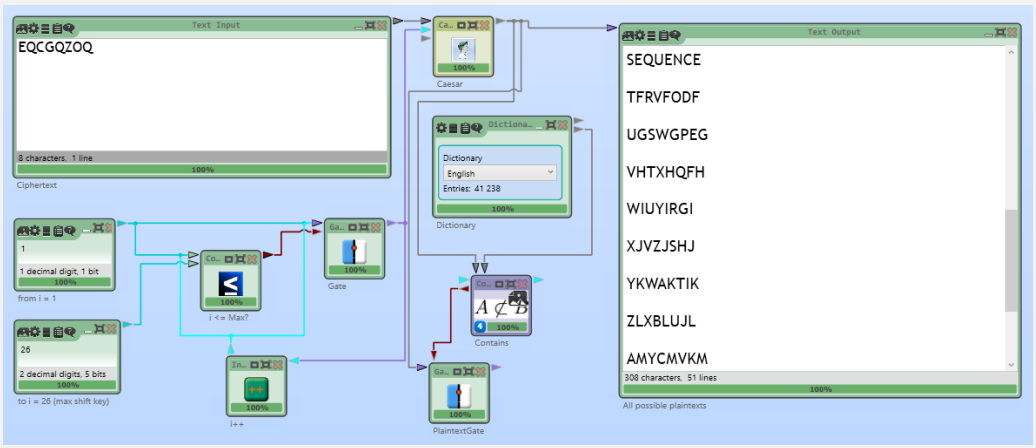
```
sage: sorted(bf.items())[12:13]
```

$$[(12, SEQUENCE)]$$

CryptoolShiftBruteForce



CryptoolShiftBruteForce



Exercise 10

It is known that the following ciphertexts were encrypted using affine ciphers. Decrypt the messages and determine the keys that were used.

- PKVQTOMV,
- ZFBCPODKX.

```
sage: A = AlphabeticStrings()
```

```
sage: AF = AffineCryptosystem(AlphabeticStrings())
```

```
sage: C = A('PKVQTOMV')
```

```
sage: bf = AF.brute_force(C)
```

```
sage: sorted(bf.items())[190:191]
```

$$[((17,8),FUNCTION)]$$

```
sage: A = AlphabeticStrings()
```

```
sage: AF = AffineCryptosystem(AlphabeticStrings())
```

```
sage: C = A('ZFBCPODKX')
```

```
sage: bf = AF.brute_force(C)
```

```
sage: sorted(bf.items())[171:172]
```

$$[((15,15),SIGNATURE)]$$

Exercise 11

Suppose we intercept the ciphertext "ANOLNKZDOSORYZCULSNWTUOQITWEBHRHCCOSHGSSYRTUVKANBWOMZYSVEDYZCLCVDGATUQXR" formed by a Hill cipher with some 2×2 key matrix over $\mathbb{Z}_{26}$. We also know that the plaintext starts with "ANEXPERT". Decrypt the remainder of the ciphertext.

```
sage: R=IntegerModRing(26)
```

```
sage: p1=[ord(k)-65 for k in 'ANEXPERT']
```

```
sage: c1=[ord(k)-65 for k in 'ANGJPWPB']
```

```
sage: p=[[p1[2*k],p1[2*k+1]] for k in [0..3]]
```

```
sage: c=[[c1[2*k],c1[2*k+1]] for k in [0..3]]
```

```
sage: Mp=matrix(R,4,2,p).transpose()
```

```
sage: Mc=matrix(R,4,2,c).transpose()
```

```
sage: A=Mp.solve_left(Mc)
```

```
sage: C='ANGJPWPBKOCRAXWYJSHMTKUUQTWCNVBRIAKOH'
```

```
sage: C=C+'QUAOVTKJUANDCOGBQSRIDAXCZEVXEWVWMFF'
```

```
sage: C1=[ord(k)-65 for k in C]
```

```
sage: C2=[[C1[2*k],C1[2*k+1]] for k in [0..len(C1)//2-1]]
```

```
sage: MC=matrix(R,len(C2),2,C2).transpose()
```

```
sage: X0=A.inverse()*MC
```

```
sage: X1=matrix([[ZZ(elt)+65 for elt in row] for row in X0.rows()
   ])
```

```
sage: L=[]
```

```
sage: for k in X1.columns(): L=L+list(k)
```

```
sage: ''.join([chr(k) for k in L])
```

ANEXPERTISAPERSONWHOHASMADEALLTHEMISTAKESTHATCANBEMADEINAVERYNARROWFIELD

Exercise 12

The ciphertext GPHDRHXVLYCSKTVUPZNQKGUPGFHGNOZJHSKVZCNKUWNCWICQHGPGFHOPDBGUEPDXTRCHLKNG was obtained by encrypting an English text using a Vigenère cipher. Try to decrypt it.

```
sage: s='GPHDRHXVLYCSKTVUPZNQKGUPGFHGNOZJHS'
```

```
sage: s=s+'KVZCNKUWNCWICQHGPGFHOPDBGUEPDXTRCHLKNG'
```

```
sage: d={}
```

```
sage: for sublen in range(3,int(len(s)/2)):
....:     for i in range(0,len(s)-sublen):
....:         sub = s[i:i+sublen]
....:         cnt = s.count(sub)
```

```
....:         if cnt >= 2 and sub not in d:
....:             d[sub] = cnt
```

```
sage: L=[]
```

```
sage: sorted(d)
```

[GFH, PGF, PGFH]

```
sage: for k in sorted(d):
....:     k0=s.find(k,0)
....:     k1=s.find(k,k0+1)
....:     L.append(k1-k0)
```

```
sage: L
```

[27, 27, 27]

We determined repeated substrings of length at least 3 and we also computed the distances between these repetitions. The length of the keyword is very likely divides these numbers, so we expect that it is either 3,9 or 27. Assume that the length is 3. We divide the ciphertext into 3 groups based on this length. In these groups we only need to find the key of a shift cipher.

```
sage: s1=''.join([s[3*k] for k in [0..23]])
```

```
sage: s2=''.join([s[3*k+1] for k in [0..23]])
```

```
sage: s3=''.join([s[3*k+2] for k in [0..23]])
```

```
sage: A = AlphabeticStrings()
```

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
```

```
sage: (l1,K1)=S.brute_force(A(s1),ranking="chisquare")[0]
```

```
sage: (l2,K2)=S.brute_force(A(s2),ranking="chisquare")[0]
```

```
sage: (l3,K3)=S.brute_force(A(s3),ranking="chisquare")[0]
```

```
sage: T=['' for k in [1..72]]
```

```
sage: for k in [0..23]:
....:     T[3*k]=str(K1)[k]
....:     T[3*k+1]=str(K2)[k]
....:     T[3*k+2]=str(K3)[k]
```

```
sage: (l1,l2,l3)
```

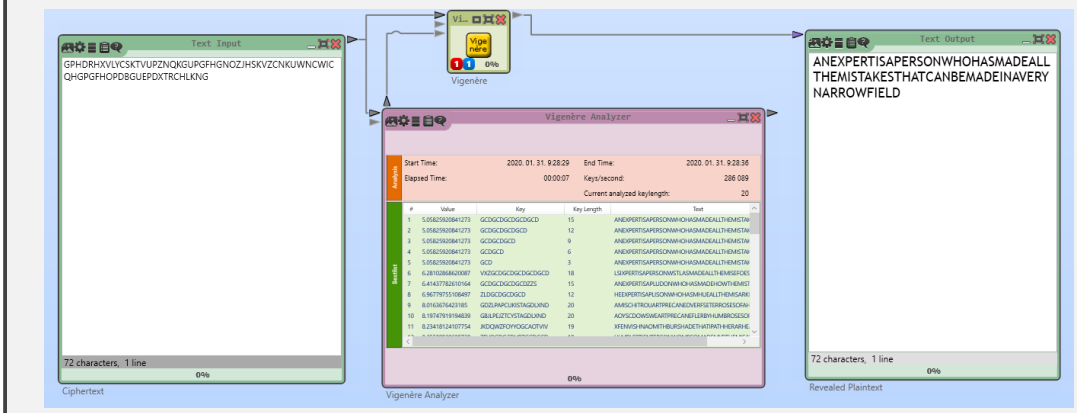
$$(6, 2, 3)$$

```
sage: ''.join(T)
```

```
ANEXPERTISAPERSONWHOHASMADEALLTHEMISTAKESTHATCANBEMADEINAVERYNARROWFIELD
```

We could decrypt the ciphertext and we saw that the 3 keys in the 3 shift ciphers are given by 6,2 and 3.

CryptoolVigenèreAnalysis



Exercise 13

Encrypt the message 'polynomial' using the ADFGX cipher with the code word 'RING' and an arbitrary polybius square.

```
sage: w='polynomial'
```

```
sage: cw='RING'
```

```
sage: S=Set([chr(k) for k in [97..122] if chr(k)!='j'])
```

```
sage: S0=Set([])
```

```
sage: while S.cardinality()!=0:
```

```
....:     s=S.random_element()
....:     S0=S0.union(Set([s]))
....:     S=S.difference(Set([s]))
```

```
sage: P=PolynomialRing(Integers(),25,list(S0))
```

```
sage: Polybius=matrix(5,5,list(P.gens()))
```

```
sage: 'Polybius Square:'
```

Polybius Square:

```
sage: Polybius
```

$$\begin{pmatrix} a & b & u & q & w \\ d & e & l & g & k \\ s & p & h & f & y \\ o & v & t & n & i \\ r & m & x & c & z \end{pmatrix}$$

```
sage: L=['A','D','F','G','X']
```

```
sage: Encode0=''
```

```
sage: for k in w:
....:     if k=='j': k='i'
....:     k1=[(i,j) for i in [0..4] for j in [0..4] if Polybius[i,j
   ]==P(k)][0]
....:     Encode0=Encode0+L[k1[0]]+L[k1[1]]
```

```
sage: R.<A,D,F,G,X>=PolynomialRing(Integers(),5)
```

```
sage: Encode1=[R(k) for k in Encode0]
```

```
sage: cwl=len(cw)
```

```
sage: Erow=len(Encode1)//cwl
```

```
sage: M=matrix(Erow,cwl,Encode1)
```

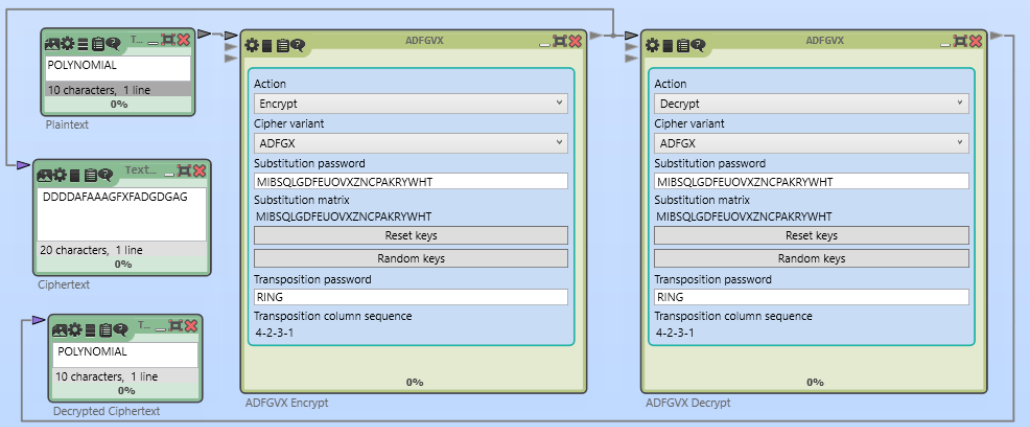
```
sage: Cipher=[]
```

```
sage: for k in sorted(list(cw)):
....:     Cipher=Cipher+list(M.column(cw.index(k)))
```

```
sage: Cipher
```

$$[A, X, A, X, F, D, F, G, D, A, G, F, G, G, D, F, D, G, X, A]$$

CryptoolADFGX



Exercise 14

Implement the ADFGVX cipher in SageMath and use it to encrypt the message 'cryptography2020' with the code word 'MOVE' and an arbitrary polybius square.

```
sage: def ADFGVX(w,codeword):
....:     S=Set([chr(k) for k in [97..122]] + ['A'+chr(k) for k in
    [48..57]])
....:     S0=Set([])
....:     while S.cardinality()!=0:
....:         s=S.random_element()
....:         S0=S0.union(Set([s]))
....:         S=S.difference(Set([s]))
....:     P=PolynomialRing(Integers(),36,list(S0))
....:     Polybius=matrix(6,6,list(P.gens()))
....:     L=['A','D','F','G','V' ,'X']
....:     Encode0=''
```

```
....:     for k in w:
....:         if ord(k) in [48..57]:
....:             k='A'+k
....:         k1=[(i,j) for i in [0..5] for j in [0..5] if Polybius[i
    ,j]==P(k)][0]
....:         Encode0=Encode0+L[k1[0]]+L[k1[1]]
....:     Encode0
....:     El=len(Encode0)
....:     cwl=len(codeword)
....:     if cwl*(El//cwl)!=El:
....:         m0=cwl*((El//cwl)+1)-El
....:         Encode0=Encode0+['A' for _ in [1..m0]]
....:     R.<A,D,F,G,V,X>=PolynomialRing(Integers(),6)
....:     Encode1=[R(k) for k in Encode0]
....:     Erow=len(Encode1)//cwl
....:     M=matrix(Erow,cwl,Encode1)
....:     M
....:     Cipher=[]
....:     for k in sorted(list(codeword)):
....:         Cipher=Cipher+list(M.column(codeword.index(k)))
....:     Cipher
....:     return Polybius, Cipher
```

```
sage: PS,C=ADFGVX('cryptography2020','MOVE')
```

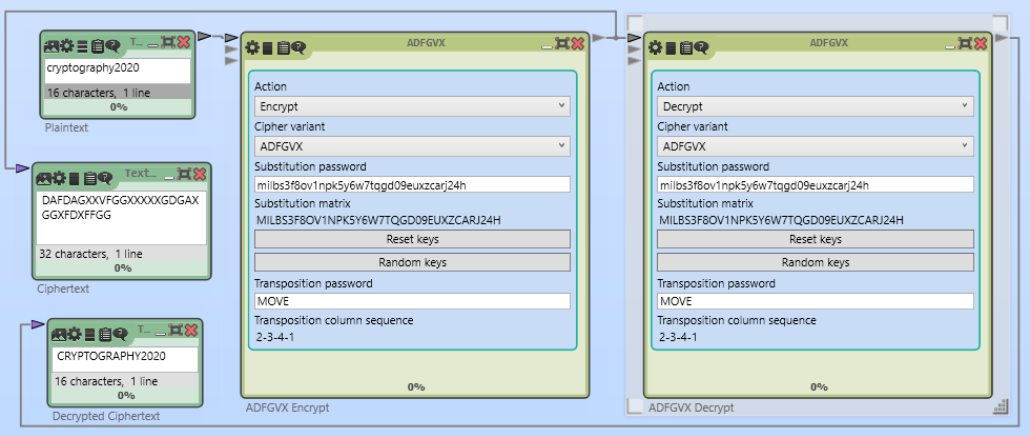
```
sage: PS
```

$$\begin{pmatrix} b & a & q & w & l & A_9 \\ z & g & k & A_3 & s & j \\ h & A_5 & o & v & t & n \\ r & m & x & A_6 & f & u \\ d & A_4 & A_2 & p & A_1 & A_8 \\ y & A_0 & i & A_7 & c & e \end{pmatrix}$$

```
sage: C
```

$$[A, G, F, A, G, A, D, D, X, X, F, D, A, F, V, V, V, A, V, D, D, A, F, F, G, V, F, G, V, X, X, X]$$

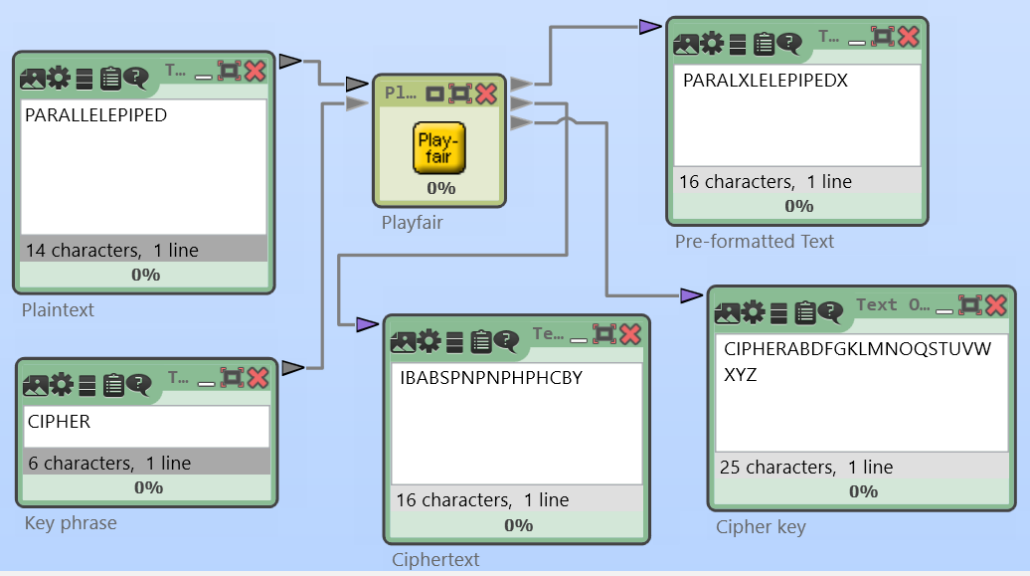
CryptoolADFGX



Exercise 15

Encrypt the message PARALLELEPIPED using the Playfair cipher with the keyword CIPHER.

CryptoolPlayfair



Exercise 16

The following ciphertext was encrypted with a Playfair cipher with keyword BISECTOR. Decrypt the message: HPTNISIDESTRACEDKSTAGW.

11.2 The RSA algorithm

Exercise 1

In RSA we have $(n, e) = (5352499, 3516607)$. Encrypt the message "The only way to learn mathematics is to do mathematics".

```
sage: n=5352499
```

```
sage: e=3516607
```

```
sage: message="The␣only␣way␣to␣learn␣mathematics␣is␣to␣do␣
   mathematics"
```

```
sage: m=IntegerRing()([ord(k) for k in message],base=256)
```

```
sage: m1=m.digits(base=n)
```

```
sage: encrypt=[power_mod(t, e, n) for t in m1]
```

```
sage: encrypt[0:9]
```

[2075984, 646969, 1459402, 676493, 3029933, 3773348, 2497354, 2831392, 2461840]

```
sage: encrypt[10:19]
```

[2047865, 2903393, 2067217, 3066198, 4615359, 2873947, 4134709, 1234154, 4942563]

Exercise 2

In RSA we know $(p, q, d) = (12227, 35569, 136215539)$, and we receive the encrypted message

[158079363, 173377019, 373536605, 97680494, 144518909, 1942499, 413795444, 147133032].

Determine the original message.

```
sage: p=12227
```

```
sage: q=35569
```

```
sage: d=136215539
```

```
sage: n=p*q
```

```
sage: cipher=[158079363,173377019,373536605,97680494]
```

```
sage: cipher=cipher+[144518909,1942499,413795444,147133032]
```

```
sage: decrypt0=[power_mod(t, Integers()(d), n) for t in cipher]
```

```
sage: decrypt=IntegerRing()(decrypt0,base=n)
```

```
sage: dm0=[chr(t) for t in decrypt.digits(base=256)]
```

```
sage: ''.join(dm0)
```

```
Was it a car or a cat I saw?
```

Exercise 3

Factor the integer $N = 5352499$ by using elliptic curves.

```
sage: N=5352499
```

```
sage: E=EllipticCurve(Integers(N),[13,1])
```

```
sage: P=E(0,1)
```

```
sage: n=factorial(18)
```

```
sage: try:
....:     n*P
....: except ZeroDivisionError as err:
....:     d = Integer(err.args[0].split()[2])
....:     g = gcd(d,N)
```

```
sage: g,N//g
```

$$(1237, 4327)$$

Exercise 4

Factor the integer $N = 5352499$ by using continued fractions.

```
sage: N=5352499
```

```
sage: cfN=continued_fraction(sqrt(N))
```

```
sage: L=[(cfN.numerator(k),((cfN.numerator(k)^2)%N).factor()) for
   k in [0..150] if Set(((cfN.numerator(k)^2) %N).prime_divisors()
   ).issubset(Set([2,3,5,7,11]))]
```

```
sage: L[1]
```

$$\left(221970938961378646013, 2 \cdot 3 \cdot 5^3\right)$$

```
sage: L[3]
```

$$(39557631731449261482041799790653351370163149108414543, 2 \cdot 3 \cdot 5)$$

```
sage: gcd(N,L[1][0]*L[3][0]-2*3*5^2),N//gcd(N,L[1][0]*L
   [3][0]-2*3*5^2)
```

$$(1237, 4327)$$

Exercise 5

Factor the integer $N = 5352499$ by applying Dixon's method.

```
sage: N=5352499
```

```
sage: x0=ceil(sqrt(N))
```

```
sage: L=[(k,((k^2)%N).factor()) for k in [x0..x0+2500] if Set(((k
   ^2)%N).prime_divisors()).issubset(Set([2,3,5,7,11]))]
```

```
sage: L[1]
```

$$\left(4019, 2^4 \cdot 7^2 \cdot 11^2\right)$$

```
sage: gcd(N,4019-2^2*7*11),N//gcd(N,4019-2^2*7*11)
```

$$(1237, 4327)$$

Exercise 6

Factor the integer $N = 5352499$ by applying Pollard's ρ algorithm with 2 different quadratic functions.

The first function we try to use is given by $x \longrightarrow x^2 + 7 \pmod{N}$.

```
sage: def fv(x,n): return mod(x^2+7,n)
```

```
sage: N=5352499
```

```
sage: a=2
```

```
sage: b=2
```

```
sage: d=1
```

```
sage: rows=[]
```

```
sage: while d == 1:
....:     a=fv(a,N)
....:     b=fv(fv(b,N),N)
....:     d=gcd(a-b,N)
....:     rows.append([a,b,d])
```

```
sage: d
```

$$1237$$

```
sage: table(rows,header_row=["$x_n$","$y_n$",r"$\gcd$"],frame=True
   )
```

x_n	y_n	gcd
11	128	1
128	1039938	1
16391	5229639	1
1039938	4810797	1
3973400	2738339	1
5229639	1653472	1237

The second one we try is given by $x \to x^2 + 8 \pmod{N}$.

```
sage: def fv(x,n): return mod(x^2+8,n)
```

```
sage: N=5352499
```

```
sage: a=2
```

```
sage: b=2
```

```
sage: d=1
```

```
sage: rows=[]
```

```
sage: while d == 1:
....:     a=fv(a,N)
....:     b=fv(fv(b,N),N)
....:     d=gcd(a-b,N)
....:     rows.append([a,b,d])
```

```
sage: d
```

1237

```
sage: table(rows,header_row=["$x_n$","$y_n$",r"$\gcd$"],frame=True
   )
```

x_n	y_n	gcd
12	152	1
152	4267151	1
23112	440220	1
4267151	4210249	1

Exercise 7

In RSA we know $(N, e_1, e_2, c_1, c_2) = (8137, 7, 23, 7155, 2626)$. Apply the common modulus attack to recover the secret message.

```
sage: c1=7155
```

```
sage: c2=2626
```

```
sage: g,s,t=xgcd(7,23)
```

```
sage: (c1^s*c2^t)%8137
```

123

Exercise 8

In RSA apply the iterated encryption attack to obtain a list containing the possible values of the secret message. We have that $(N, e, c) = (1591, 5, 22)$.

```
sage: L=[22]
```

```
sage: c1=22
```

```
sage: ck=1
```

```
sage: while len(L)==Set(L).cardinality():
....:     ck=(c1^5)%1591
....:     L=L+[ck]
....:     c1=ck
```

```
sage: Set(L)
```

$\{720, 819, 500, 997, 22, 383\}$

Exercise 9

In RSA we have $(N, e) = (24739307, 7526327)$. Apply Wiener's attack to recover the secret key d.

```
sage: N,e=24739307, 7526327
```

```
sage: cf=continued_fraction(e/N)
```

```
sage: uv=cf.convergents()
```

```
sage: c0=(11^e)%N
```

```
sage: t=0
```

```
sage: while not (c0^(uv[t]).denominator())%N==11:
....:   t=t+1
```

```
sage: (uv[t]).denominator()
```

23

Exercise 10

Chris and David have RSA public keys given by $(N_B, e_B) = (6527, 7)$, $(N_C, e_C) = (11537, 7)$ and $(N_D, e_D) = (10123, 7)$, respectively. Alice sends the same message to both of them, the ciphertexts are as follows $c_B = 2268, c_C = 3442$ and $c_D = 4737$. Determine the message by means of the low public exponent attack.

```
sage: NB=6527
```

```
sage: NC=11537
```

```
sage: ND=10123
```

```
sage: cB=2268
```

```
sage: cC=3442
```

```
sage: cD=4737
```

```
sage: CRT_list([2268,3442,4737],[NB,NC,ND])^(1/3)
```

345

11.3 The Rabin and Paillier cryptosystems

Exercise 1

Bob uses a Rabin public-key cryptosystem with $N = 1817 = 23 \cdot 79$. Alice tells Bob that the two-digit message will be padded with starting digits "11" and she sends 882 as encrypted message. Decode the message.

```
sage: p=23
```

```
sage: q=79
```

```
sage: c=882
```

```
sage: s1=(c^((p+1)/4))%p
```

```
sage: s2=p-s1
```

```
sage: s3=(c^((q+1)/4))%q
```

```
sage: s4=q-s3
```

```
sage: S=[s1,s2,s3,s4]
```

```
sage: CRT1=crt(s1,s3,p,q)
```

```
sage: CRT2=crt(s1,s4,p,q)
```

```
sage: CRT3=crt(s2,s3,p,q)
```

```
sage: CRT4=crt(s2,s4,p,q)
```

```
sage: possible=[CRT1,CRT2,CRT3,CRT4]
```

```
sage: possible
```

$$[979, 680, 1137, 838]$$

Among the four possible plaintexts only 1137 starts with 11, hence the original message is 37.

Exercise 2

Let $(N, g) = (2501, 92)$ and $(m_1, r_1) = (34, 5), (m_2, r_2) = (16, 7)$. Compute the two encoded messages c_1, c_2 and show that $c_1 \cdot c_2$ is the same as the ciphertext of $m_1 + m_2$ with random integer $r = 35$.

```
sage: N=2501
```

```
sage: g=92
```

```
sage: m1=34
```

```
sage: r1=5
```

```
sage: m2=16
```

```
sage: r2=7
```

```
sage: m=m1+m2
```

```
sage: r=35
```

```
sage: c1=(g^m1*r1^N)%(N^2)
```

```
sage: c2=(g^m2*r2^N)%(N^2)
```

```
sage: "c1:␣",c1
```

$$(\texttt{c1:}\quad, 1129735)$$

```
sage: "c2:␣",c2
```

(c2: ,5140305)

```
sage: "c1*c2:␣",(c1*c2)%(N^2)
```

(c1*c2: ,2010769)

```
sage: c=(g^m*r^N)%(N^2)
```

```
sage: "c:␣",c
```

(c: ,2010769)

11.4 Applications of the discrete logarithm problem

Exercise 1

In a Diffie-Hellman key-exchange the group is $(\mathbb{F}^*_{1117}, \cdot)$ with a generator $g = 29$. Compute the public key belonging to Bob's secret key $b = 123$. Alice's public key is 321. Compute the shared key.

```
sage: g=Integers(1117)(29)
```

```
sage: b=123
```

```
sage: "Bob's␣public␣key:"
```

Bob's public key:

```
sage: g^b
```

173

```
sage: ga=Integers(1117)(321)
```

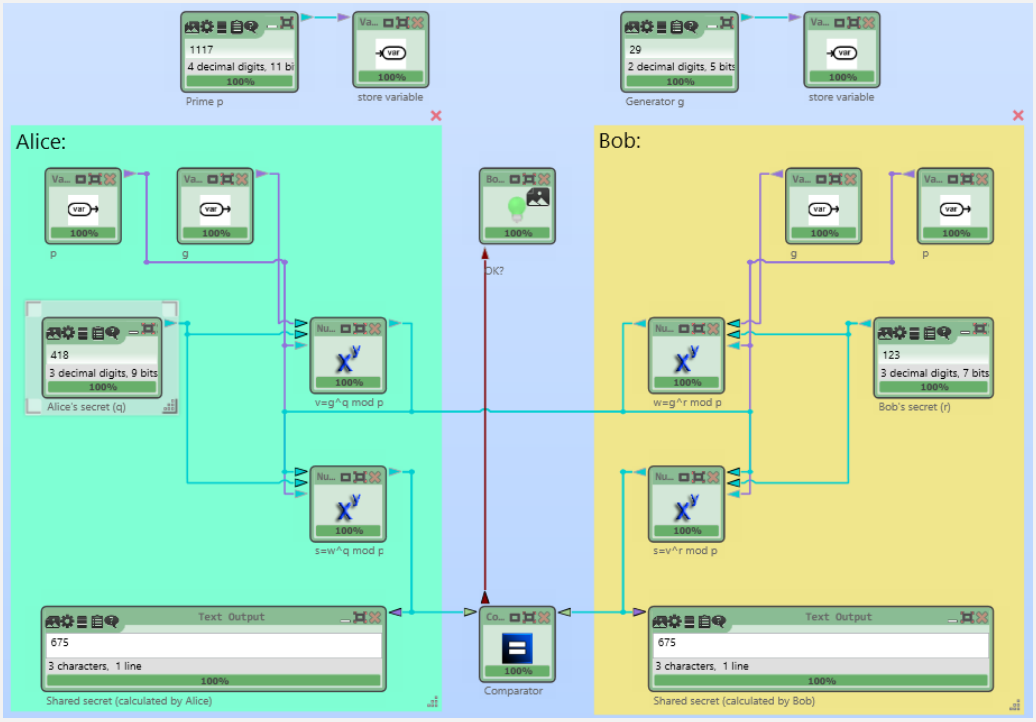
```
sage: "Shared␣key:"
```

Shared key:

```
sage: ga^b
```

675

CryptoolDiffieHellman



Exercise 2

Alice and Bob exchange a key using the Diffie-Hellman protocol. They publish an elliptic curve $E : y^2 = x^3 + 13x + 10 \pmod{101}$. They also publish the point $P = (51, 2)$. Alice chooses $a = 28$, Bob has $b = 77$. Compute the points aP, bP and abP.

```
sage: E=EllipticCurve(GF(101),[13,10])
```

```
sage: P=E(51,2)
```

```
sage: "P␣=",P
```

$$(\texttt{P =}, (51 : 2 : 1))$$

```
sage: a=28
```

```
sage: b=77
```

```
sage: "aP="
```

aP=

```
sage: a*P
```

$$(44 : 44 : 1)$$

```
sage: "bP="
```

bP=

```
sage: b*P
```

$$(42 : 56 : 1)$$

```
sage: "abP="
```

abP=

```
sage: a*b*P
```

$$(15 : 34 : 1)$$

Exercise 3

Alice and Bob are using the ElGamal cryptosystem. The public key of Alice is $(p, g, g^a) = (3571, 2, 2905)$. Bob encrypts the message $m_1 = 567$ with $k = 111$. What does Bob send to Alice?

```
sage: g=Integers(3571)(2)
```

```
sage: ga=Integers(3571)(2905)
```

```
sage: k=111
```

```
sage: m1=Integers(3571)(567)
```

```
sage: "Bob␣sends␣$(g^k,m1*(g^a)^k)$:"
```

Bob sends $(g^k,m1*(g^a)^k)$:

```
sage: g^k,m1*ga^k
```

$$(959, 3436)$$

Exercise 4

Alice uses the ElGamal cryptosystem. She publishes the curve $E : y^2 = x^3 + 13x + 10 \pmod{101}$ and the point $P = (51, 2)$ of order 106. She also chooses a secret number $a = 37$ and publishes the point aP. Bob wants to send to Alice a message corresponding to the point $P_m = (85, 7)$. Compute aP. Cipher the message using $k = 19$. Decipher the message.

```
sage: E=EllipticCurve(GF(101),[13,10])
```

```
sage: P=E(51,2)
```

```
sage: a=37
```

```
sage: "aP="
```

aP=

```
sage: a*P
```

(52 : 55 : 1)

```
sage: k=19
```

```
sage: Pm=E(85,7)
```

```
sage: "kP="
```

kP=

```
sage: k*P
```

(53 : 55 : 1)

```
sage: "The encrypted message is:"
```

The encrypted message is:

```
sage: Pm+k*a*P
```

(28 : 56 : 1)

```
sage: "-a(kP)="
```

-a(kP)=

```
sage: -a*k*P
```

(67 : 82 : 1)

```
sage: "The␣decrypted␣message␣is:"
```

The decrypted message is:

```
sage: Pm+k*a*P-a*k*P
```

(85 : 7 : 1)

Exercise 5

Alice and Bob use the Massey-Omura cryptosystem. Describe the communication and the intermediate results for the transmission of the message $m = 44$ between Alice and Bob, given $p = 113, e_A = 39$ and $e_B = 75$.

```
sage: p=113
```

```
sage: R=Integers(p)
```

```
sage: eA=39
```

```
sage: eB=75
```

```
sage: m=44
```

```
sage: dA=(1/eA)%(p-1)
```

```
sage: dB=(1/eB)%(p-1)
```

```
sage: "Alice␣sends␣m^eA␣to␣Bob:␣",R(m^eA)
```

(Alice sends m^eA to Bob: , 18)

```
sage: "Bob␣sends␣back␣(m^eA)^eB␣to␣Alice:␣",R((m^eA)^eB)
```

(Bob sends back (m^eA)^eB to Alice: ,69)

```
sage: "Alice␣sends␣((m^eA)^eB)^dA␣to␣Bob:␣",R(((m^eA)^eB)^dA)
```

(Alice sends ((m^eA)^eB)^dA to Bob: ,95)

```
sage: "Bob␣computes␣(((m^eA)^eB)^dA)^dB␣to␣obtain␣the␣message␣:",R
    ((((m^eA)^eB)^dA)^dB)
```

(Bob computes (((m^eA)^eB)^dA)^dB to obtain the message :,44)

Exercise 6

Alice and Bob use the elliptic curve Massey-Omura cryptosystem. They use the elliptic curve $E : y^2 = x^3 + 13x + 10 \pmod{101.}$ The private keys are given by $(e_A, d_A) = (35, 103)$ and $(e_B, d_B) = (77, 95)$. Alice would like to send the message $M = (31 : 45) \in E(\mathbb{F}_{101})$. Describe the involved steps.

```
sage: E=EllipticCurve(GF(101),[13,10])
```

```
sage: M=E(31,45)
```

```
sage: N=E.order()
```

```
sage: R=Integers(N)
```

```
sage: eA=35
```

```
sage: dA=(1/eA)%N
```

```
sage: eB=77
```

```
sage: dB=(1/eB)%N
```

```
sage: "Alice␣sends␣eA*M␣to␣Bob:␣",eA*M
```

(Alice sends eA*M to Bob: ,(26 : 59 : 1))

```
sage: "Bob␣sends␣back␣eB*(eA*M)␣to␣Alice:␣",eB*(eA*M)
```

(Bob sends back eB*(eA*M) to Alice: ,(95 : 76 : 1))

```
sage: "Alice␣sends␣dA*(eB*(eA*M))␣to␣Bob:␣",dA*(eB*(eA*M))
```

(Alice sends dA*(eB*(eA*M)) to Bob: , (33 : 97 : 1))

```
sage: "Bob␣computes␣dB*(dA*(eB*(eA*M)))␣to␣obtain␣the␣message␣:",
   dB*(dA*(eB*(eA*M)))
```

(Bob computes dB*(dA*(eB*(eA*M))) to obtain the message :, (31 : 45 : 1))

Exercise 7

Alice chooses the binary string 111001 and Bob's binary string is 101010. They use the AA_β cryptosystem. Compute their shared key if the public real number is 0.14159265358979323846264 and $a = 5, b = 7$.

```
sage: a=5
```

```
sage: b=9
```

```
sage: A=[1,1,1,0,0,1]
```

```
sage: B=[1,0,1,0,1,0]
```

```
sage: t=0.14159265358979323846264
```

```
sage: A0=matrix(2,2,[1,a,1,0])
```

```
sage: A1=matrix(2,2,[b,1,1,0])
```

```
sage: EA=matrix(2,2,[1,0,0,1])
```

```
sage: EB=matrix(2,2,[1,0,0,1])
```

```
sage: M=[A0,A1]
```

```
sage: K=len(A)
```

```
sage: for k in [1..K]:
....:     EA=EA*M[A[-k]]
```

```
sage: for k in [1..K]:
....:     EB=EB*M[B[-k]]
```

```
sage: nA=EA[0,0]
```

```
sage: nB=EB[0,0]
```

```
sage: rA=(nA*t).frac()
```

```
sage: rB=(nB*t).frac()
```

```
sage: (nA*rB).frac()
```

0.4776266964772029636685

Exercise 8

Using the computation of the previous exercise. Eve could get the values of rA and rB and she also knows the public real number $x = 0.14159265358979323846264$ and integers $a = 5, b = 7$. Apply the continued fraction expansion to solve the discrete logarithm modulo 1 and therefore the values of nA and nB.

```
sage: x=0.14159265358979323846264
```

```
sage: y=0.8640524548074799343884
```

```
sage: cf=continued_fraction(x)
```

```
sage: T=True
```

```
sage: k=1
```

```
sage: Eold=10^(-3)
```

```
sage: while T:
....:     c=cf.convergent(k)
....:     cn=c.numerator()
....:     cd=c.denominator()
....:     a=round(cd*y)
```

```
....:     n=(a/cn)%cd
....:     Enew=(n*x).frac()-y
....:     OldNew=Eold/Enew
....:     Eold=Enew
....:     if abs(OldNew)>10^10:
....:         T=False
....:     else:
....:         k=k+1
```

```
sage: "nA",n
```

$$(\mathrm{nA}, 45185)$$

```
sage: x=0.1415926535897932384626 4
```

```
sage: y=0.4859240373286815802332
```

```
sage: cf=continued_fraction(x)
```

```
sage: T=True
```

```
sage: k=1
```

```
sage: Eold=10^(-3)
```

```
sage: while T:
....:     c=cf.convergent(k)
....:     cn=c.numerator()
....:     cd=c.denominator()
....:     a=round(cd*y)
....:     n=(a/cn)%cd
....:     Enew=(n*x).frac()-y
....:     OldNew=Eold/Enew
....:     Eold=Enew
....:     if abs(OldNew)>10^10:
....:         T=False
....:     else:
....:         k=k+1
```

```
sage: "nB",n
```

(nB, 3005)

11.5 Attacks on the discrete logarithm problem

Exercise 1

Solve the discrete logarithm problem $3^x \equiv 224 \pmod{1013}$ using the Baby-Step Giant-Step algorithm.

```
sage: R=Integers(1013)
```

```
sage: g=R(3)
```

```
sage: h=R(224)
```

```
sage: N=ceil(sqrt(1013))
```

```
sage: L=[g^k for k in [0..N-1]]
```

```
sage: j=0
```

```
sage: while not (h*g^(-j*N)) in L:
....:     j=j+1
```

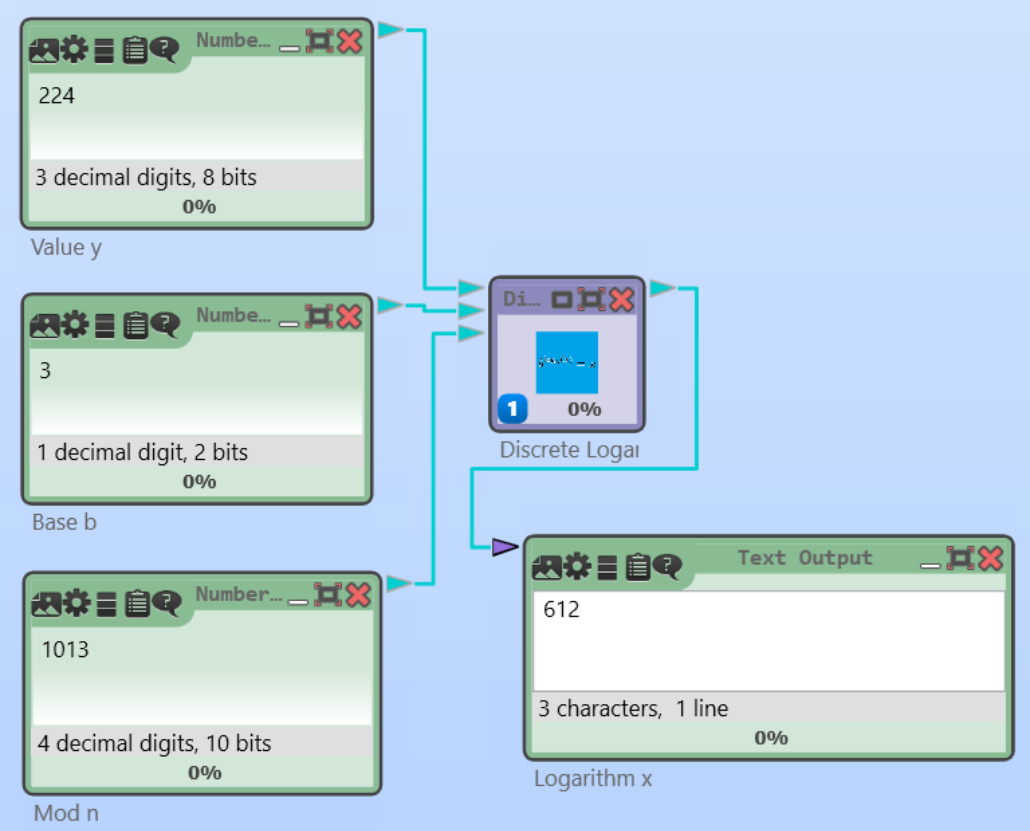
```
sage: h*g^(-j*N)
```

81

```
sage: L.index(h*g^(-j*N))+j*N
```

612

CryptoolDiscreteLogarithm



Exercise 2

Solve the discrete logarithm problem $n \cdot (51 : 2) = (62 : 28)$ in case of the elliptic curve $E : y^2 = x^3 + 13x + 10$ over $\mathbb{F}_{101}$ using the Baby-Step Giant-Step algorithm.

```
sage: E=EllipticCurve(GF(101),[13,10])
```

```
sage: EO=E.order()
```

```
sage: N=ceil(sqrt(EO))
```

```
sage: P=E(51,2)
```

```
sage: Q=E(62,28)
```

```
sage: L=[k*P for k in [0..N-1]]
```

```
sage: j=0
```

```
sage: while not (Q-j*N*P) in L:
....:     j=j+1
```

```
sage: (Q-j*N*P)
```

$$(1 : 23 : 1)$$

```
sage: L.index((Q-j*N*P))+j*N
```

$$76$$

Exercise 3

Use the Pollard's ρ method to attack the discrete logarithm problem given by $g = 3, h = 245$ in $\mathbb{F}_{1013}^*$. That is find an integer x such that $g^x \equiv h \pmod{1013}$.

```
sage: p=1013
```

```
sage: g=3
```

```
sage: h=245
```

```
sage: S0=[k for k in [0..p//3]]
```

```
sage: S1=[k for k in [p//3+1..2*p//3]]
```

```
sage: S2=[k for k in [2*p//3+1..p-1]]
```

```
sage: T1=[1,0,0]
```

```
sage: T2=[1,0,0]
```

```
sage: tbl=[([1,0,0],[1,0,0])]
```

```
sage: def w(t):
....:     if t[0] in S0:
....:         return [(g*t[0])%p,(t[1]+1)%(p-1),(t[2])%(p-1)]
....:     elif t[0] in S1:
....:         return [(t[0]^2)%p,(2*t[1])%(p-1),(2*t[2])%(p-1)]
```

```
....:     else:
....:         return [(h*t[0])%p,(t[1])%(p-1),(t[2]+1)%(p-1)]
```

```
sage: jo=True
```

```
sage: while jo:
....:     T1=w(T1)
....:     T2=w(w(T2))
....:     tbl.append((T1,T2))
....:     if T1[0]==T2[0]:
....:         jo=False
```

```
sage: table(tbl)
```

[1, 0, 0]	[1, 0, 0]
[3, 1, 0]	[9, 2, 0]
[9, 2, 0]	[81, 4, 0]
[27, 3, 0]	[729, 6, 0]
[81, 4, 0]	[951, 7, 1]
[243, 5, 0]	[15, 8, 2]
[729, 6, 0]	[135, 10, 2]
[317, 6, 1]	[932, 22, 4]
[951, 7, 1]	[15, 44, 10]
[5, 7, 2]	[135, 46, 10]
[15, 8, 2]	[932, 94, 20]
[45, 9, 2]	[15, 188, 42]
[135, 10, 2]	[135, 190, 42]

```
sage: g1=(T1[1]-T2[1])%(p-1)
```

```
sage: h1=(T2[2]-T1[2])%(p-1)
```

```
sage: d,s,t=xgcd(h1,p-1)
```

```
sage: possible=[((s*g1+i*(p-1))/d)%(p-1) for i in [0..d-1]]
```

```
sage: [j for j in possible if (g^j)%p==h][0]
```

Exercise 4

Given the elliptic curve $E : y^2 = x^3 + 13x + 32$ over the finite field $\mathbb{F}_{2027}$ and two points $P = (1381 : 839), Q = (6 : 1229)$. Determine a solution of the elliptic curve discrete logarithm problem $nP = Q$ by applying the Pollard's ρ method.

```
sage: E=EllipticCurve(GF(2027),[13,32])
```

```
sage: P=E(1381,839)
```

```
sage: Q=E(6,1229)
```

```
sage: N=E.order()
```

```
sage: T1=[P,1,0]
```

```
sage: T2=[P,1,0]
```

```
sage: SEQ=[]
```

```
sage: def w(t):
....:     if ZZ(t[0][1])%3==0:
....:         return [P+t[0],(t[1]+1)%N,t[2]]
....:     elif ZZ(t[0][1])%3==1:
....:         return [2*t[0],(2*t[1])%N,(2*t[2])%N]
....:     else:
....:         return [Q+t[0],t[1],(t[2]+1)%N]
```

```
sage: jo=True
```

```
sage: while jo:
....:     T1=w(T1)
....:     T2=w(w(T2))
....:     SEQ.append([T1,T2])
....:     if T1[0]==T2[0]:
....:         jo=False
```

```
sage: a1=(T1[1]-T2[1])%N
```

```
sage: b1=(T2[2]-T1[2])%N
```

```
sage: g,t,s=xgcd(N,b1)
```

```
sage: sol=[ZZ((s*a1+k*N)/g) for k in [0..g-1]]
```

```
sage: [k for k in sol if k*P==Q][0]
```

567

Exercise 5

Apply the index calculus algorithm to solve the discrete logarithm problem $5^x \equiv 20 \pmod{503}$.

```
sage: p=503
```

```
sage: g=5
```

```
sage: h=20
```

```
sage: B=[2,3,5,7]
```

```
sage: v=[157,258,202,407]
```

```
sage: M=matrix(Integers(p-1),[[valuation((g^xi)%p,k) for k in B]
   for xi in v])
```

```
sage: M
```

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 3 \\ 1 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 \end{pmatrix}$$

```
sage: V=vector(Integers(p-1),v)
```

```
sage: V
```

(157, 258, 202, 407)

```
sage: LOGS=M.inverse()*V
```

```
sage: LOGS
```

(202, 156, 1, 86)

```
sage: xi=143
```

```
sage: q=[valuation((g^xi*h)%p,k) for k in B]
```

```
sage: (sum(q[k]*LOGS[k] for k in [0..len(q)-1])-xi)%(p-1)
```

405

Exercise 6

Let

$$p = 1747808567274271495694237474897031552001$$

and $(g, h) = (13, 2020)$. Apply the Pohlig-Hellman algorithm to compute $\log_g(h) \pmod{p}$.

```
sage: p=1747808567274271495694237474897031552001
```

```
sage: F=GF(p)
```

```
sage: g=F(13)
```

```
sage: h=F(2020)
```

```
sage: N=p-1
```

```
sage: qi=[r^N.valuation(r) for r in prime_divisors(N)]
```

```
sage: lqi=len(qi)
```

```
sage: Nqi=[N/q for q in qi]
```

```
sage: gi=[g^r for r in Nqi]
```

```
sage: hi=[h^r for r in Nqi]
```

```
sage: xi=[discrete_log(hi[i],gi[i]) for i in range(lqi)]
```

```
sage: CRT(xi,qi)
```

$$1567923366711914963612701098288689036134$$

11.6 Non-abelian discrete logarithm problem

Exercise 1

Let

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

be matrices over $\mathbb{F}_{13}$. Determine all solutions of the discrete logarithm problem

$$A^i B^j A^k B^l = \begin{pmatrix} 0 & 1 \\ 12 & 4 \end{pmatrix}.$$

```
sage: P.<i,j,k,l>=PolynomialRing(GF(13),4)
```

```
sage: A1=matrix([[1,i],[0,1]])
```

```
sage: B1=matrix([[1,0],[j,1]])
```

```
sage: A2=matrix([[1,k],[0,1]])
```

```
sage: B2=matrix([[1,0],[l,1]])
```

```
sage: AB=A1*B1*A2*B2
```

```
sage: M=matrix(P,[[0,1],[12,4]])
```

```
sage: S=AB-M
```

```
sage: I=ideal(P,S.list()+[prod([i-k for k in [0..12]])])
```

```
sage: v=I.variety()
```

```
sage: table([[w[k] for k in [i,j,k,l]] for w in v],header_row=["
```

```
    $i$", "$j$", "$k$","$l$"], frame=True)
```

i	j	k	l
1	12	10	0
3	8	2	1
2	7	6	11
5	6	7	8
7	10	12	7
9	4	4	2
12	11	5	10
8	2	8	9
6	1	3	6
4	5	11	5
11	9	9	4
0	3	1	12

Exercise 2

Let

$$C = \begin{pmatrix} 9 & 3 \\ 7 & 10 \end{pmatrix}, \quad D = \begin{pmatrix} 0 & 12 \\ 7 & 2 \end{pmatrix}$$

be matrices over $\mathbb{F}_{17}$. Determine all solutions of the discrete logarithm problem

$$C^i D^j C^k D^l = \begin{pmatrix} 14 & 12 \\ 3 & 16 \end{pmatrix}.$$

```
sage: P.<g1,g2,g3,g4,s,t>=PolynomialRing(GF(17),6)
```

```
sage: G=matrix([[g1,g2],[g3,g4]])
```

```
sage: As=matrix([[1,s],[0,1]])
```

```
sage: Bt=matrix([[1,0],[t,1]])
```

```
sage: C=matrix(GF(17),[[9,3],[7,10]])
```

```
sage: D=matrix(GF(17),[[0,12],[7,2]])
```

```
sage: S1=As*G-G*C
```

```
sage: S2=Bt*G-G*D
```

```
sage: I=ideal(P,S1.list()+S2.list()+[prod([s-k for k in [0..16]])
   ]+[G.determinant()-1])
```

```
sage: v=I.variety()
```

```
sage: v0=v[0]
```

```
sage: G0=matrix([[v0[g1],v0[g2]],[v0[g3],v0[g4]]])
```

```
sage: s0=v0[s]
```

```
sage: t0=v0[t]
```

```
sage: M=matrix(GF(17),[[14,12],[3,16]])
```

```
sage: M1=G0*M*G0.inverse()
```

```
sage: P.<i,j,k,l>=PolynomialRing(GF(17),4)
```

```
sage: A1=matrix([[1,i],[0,1]])
```

```
sage: B1=matrix([[1,0],[j,1]])
```

```
sage: A2=matrix([[1,k],[0,1]])
```

```
sage: B2=matrix([[1,0],[l,1]])
```

```
sage: AB=A1*B1*A2*B2
```

```
sage: I=ideal(P,(AB-M1).list()+[prod([i-k for k in [0..16]])])
```

```
sage: v=I.variety()
```

```
sage: sol=[[w[k] for k in [i,j,k,l]] for w in v]
```

```
sage: SOL=[[(k[0]/s0)%17, (k[1]/t0)%17,(k[2]/s0)%17,(k[3]/t0)%17]
   for k in sol]
```

```
sage: table(SOL,header_row=["$i$", "$j$", "$k$","$l$"], frame=True
    )
```

i	j	k	l
0	15	5	0
13	12	2	13
1	16	10	7
4	3	8	1
6	7	1	12
5	11	13	6
15	2	12	11
10	6	4	5
9	10	16	16
11	14	9	10
14	1	7	4
2	5	15	15
12	9	14	9
8	13	11	3
7	4	6	8
3	8	3	2

11.7 The NTRU cryptosystem

Exercise 1

Let $N = 11, p = 3$ and $q = 32$. Determine polynomials f_p and f_q such that $f \star f_p \equiv 1 \pmod{p}$ and $f \star f_q \equiv 1 \pmod{q}$, where $f = -1 + X + X^2 + X^4 - X^5 + X^8 - X^{10}$ is an element of $\mathbb{F}[X]/(X^{11} - 1)$.

```
sage: P.<X>=PolynomialRing(GF(3))
```

```
sage: R=P.quo(X^11-1)
```

```
sage: f=-1+X+X^2+X^4-X^5+X^8-X^10
```

```
sage: d,fp,F=xgcd(f,X^11-1)
```

```
sage: "fp␣=␣",fp
```

$$\left(\texttt{fp = },\ X^{10}+X^9+X^7+X^6+2X^5+X^3+X^2+2X\right)$$

```
sage: R(f*fp)
```

$$1$$

```
sage: P.<X>=PolynomialRing(GF(32))
```

```
sage: R=P.quo(X^11-1)
```

```
sage: f=-1+X+X^2+X^4-X^5+X^8-X^10
```

```
sage: d,fq,F=xgcd(f,X^11-1)
```

```
sage: "fq␣=␣",fq
```

$$\left(\texttt{fq = },\ X^7+X^3+X\right)$$

```
sage: R(f*fq)
```

$$1$$

Exercise 2

NTRU encryption. Let $N = 11, p = 3$ and $q = 23$. We choose $f = x^{10} - x^9 + x^7 + x^4 - 1$ and $g = x^{10} + x^6 - x^3 - x^2 + x + 1$. Alice would like to send the message $m = x^{10} + 2x^9 + x^5 + x + 1$ to Bob. Alice chooses the random polynomial $r = x^9 - x^7 + x^6 + x^3 - x - 1$. Compute the polynomial that Alice sends to Bob.

```
sage: Pp.<x>=PolynomialRing(GF(3))
```

```
sage: Rp.<X>=Pp.quo(x^11-1)
```

```
sage: Pq.<y>=PolynomialRing(GF(23))
```

```
sage: Rq.<Y>=Pq.quo(y^11-1)
```

```
sage: f=x^10-x^9+x^7+x^4-1
```

```
sage: g=x^10+x^6-x^3-x^2+x+1
```

```
sage: f1=y^10-y^9+y^7+y^4-1
```

```
sage: g1=y^10+y^6-y^3-y^2+y+1
```

```
sage: dp,fp,Fp=xgcd(f,x^11-1)
```

```
sage: dq,fq,Fq=xgcd(f1,y^11-1)
```

```
sage: h=Rq(g1*fq)
```

```
sage: m=x^10+2*x^9+x^5+x+1
```

```
sage: m1=Y^10+2*Y^9+Y^5+Y+1
```

```
sage: r=x^9-x^7+x^6+x^3-x-1
```

```
sage: r1=Y^9-Y^7+Y^6+Y^3-Y-1
```

```
sage: E=3*r1*h+m1
```

```
sage: E
```

$$10Y^{10} + 8Y^9 + 13Y^8 + 20Y^7 + 9Y^6 + 3Y^5 + 6Y^4 + Y^3 + 10Y^2 + 8Y + 10$$

Exercise 3

NTRU decryption. Decrypt the message sent by Alice to Bob as it is described in the previous exercise.

```
sage: def cmod(n):
....:     if not ZZ(n) in [-11..11]:
....:         return ZZ(n)-23
....:     else:
....:         return ZZ(n)
```

```
sage: Pp.<x>=PolynomialRing(GF(3))
```

```
sage: Rp.<X>=Pp.quo(x^11-1)
```

```
sage: Pq.<y>=PolynomialRing(GF(23))
```

```
sage: Rq.<Y>=Pq.quo(y^11-1)
```

```
sage: f=x^10-x^9+x^7+x^4-1
```

```
sage: g=x^10+x^6-x^3-x^2+x+1
```

```
sage: f1=y^10-y^9+y^7+y^4-1
```

```
sage: g1=y^10+y^6-y^3-y^2+y+1
```

```
sage: dp,fp,Fp=xgcd(f,x^11-1)
```

```
sage: dq,fq,Fq=xgcd(f1,y^11-1)
```

```
sage: h=Rq(g1*fq)
```

```
sage: m=x^10+2*x^9+x^5+x+1
```

```
sage: m1=Y^10+2*Y^9+Y^5+Y+1
```

```
sage: r=x^9-x^7+x^6+x^3-x-1
```

```
sage: r1=Y^9-Y^7+Y^6+Y^3-Y-1
```

```
sage: E=3*r1*h+m1
```

```
sage: a=Rq(f1)*E
```

```
sage: Rp(Pp([cmod(k) for k in a.list()])*fp)
```

$$X^{10} + 2X^9 + X^5 + X + 1$$

Exercise 4

Try to determine f and g by means of the LLL-algorithm in case of the NTRU cryptosystem described in the previous exercises, that is $N = 11, q = 23$ and

$$h = 12x^{10} + 21x^8 + 4x^7 + 12x^5 + 17x^4 + 22x^3 + 6x^2 + 16x + 7.$$

```
sage: N=11
```

```
sage: p=3
```

```
sage: q=23
```

```
sage: ZY.<Y> = ZZ[]
```

```
sage: h=12*Y^10 + 21*Y^8 + 4*Y^7 + 12*Y^5 + 17*Y^4 + 22*Y^3 + 6*Y^2 + 16*
    Y + 7
```

```
sage: M = matrix(2*N)
```

```
sage: for i in [0..N-1]: M[i,i] = 1
```

```
sage: for i in [N..2*N-1]: M[i,i] = q
```

```
sage: for i in [0..N-1]:
....:     for j in [0..N-1]:
....:         M[i+N,j] = ((ZY(GF(q)(1)*h)*Y^i)%(Y^N-1))[j]
```

```
sage: M
```

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 16 & 6 & 22 & 17 & 12 & 0 & 4 & 21 & 0 & 12 & 23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
12 & 7 & 16 & 6 & 22 & 17 & 12 & 0 & 4 & 21 & 0 & 0 & 23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 12 & 7 & 16 & 6 & 22 & 17 & 12 & 0 & 4 & 21 & 0 & 0 & 23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
21 & 0 & 12 & 7 & 16 & 6 & 22 & 17 & 12 & 0 & 4 & 0 & 0 & 0 & 23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 21 & 0 & 12 & 7 & 16 & 6 & 22 & 17 & 12 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 4 & 21 & 0 & 12 & 7 & 16 & 6 & 22 & 17 & 12 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 & 0 & 0 & 0 \\
12 & 0 & 4 & 21 & 0 & 12 & 7 & 16 & 6 & 22 & 17 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 & 0 & 0 \\
17 & 12 & 0 & 4 & 21 & 0 & 12 & 7 & 16 & 6 & 22 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 & 0 \\
22 & 17 & 12 & 0 & 4 & 21 & 0 & 12 & 7 & 16 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 \\
6 & 22 & 17 & 12 & 0 & 4 & 21 & 0 & 12 & 7 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 0 \\
16 & 6 & 22 & 17 & 12 & 0 & 4 & 21 & 0 & 12 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23
\end{pmatrix}$$

```
sage: LM=M.transpose().LLL(delta=0.99,algorithm='fpLLL:proved',fp='xd')
```

```
sage: LM
```

$$\begin{pmatrix}
1 & -1 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 \\
1 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
-1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & 0 & 0 \\
0 & 1 & -1 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & -1 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & -1 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\
1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & -1 \\
0 & 1 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & 0 \\
0 & 0 & 0 & -1 & 1 & -1 & 0 & 1 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
-1 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 1 & 1 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & -1 & 0 & -2 & -1 & -2 & 0 & -3 & -2 & -2 & -1 & 2 & 2 & -2 & 0 & 3 & 1 & -3 & 2 & -2 & -1 \\
2 & 1 & 2 & 0 & 3 & 2 & 2 & -1 & -1 & 1 & 0 & 0 & -3 & -1 & 3 & -2 & 2 & 1 & 1 & -2 & -2 & 2 \\
0 & -3 & -2 & -2 & 1 & 1 & -1 & 0 & -2 & -1 & -2 & -3 & 2 & -2 & -1 & -1 & 2 & 2 & -2 & 0 & 3 & 1 \\
0 & -2 & -1 & -2 & 0 & -3 & -2 & -2 & 1 & 1 & -1 & -2 & 0 & 3 & 1 & -3 & 2 & -2 & -1 & -1 & 2 & 2 \\
3 & 2 & 2 & -1 & -1 & 1 & 0 & 2 & 1 & 2 & 0 & -2 & 2 & 1 & 1 & -2 & -2 & 2 & 0 & -3 & -1 & 3 \\
-1 & 1 & 0 & 2 & 1 & 2 & 0 & 3 & 2 & 2 & -1 & -2 & -2 & 2 & 0 & -3 & -1 & 3 & -2 & 2 & 1 & 1 \\
-1 & -2 & 0 & -3 & -2 & -2 & 1 & 1 & -1 & 0 & -2 & 3 & 1 & -3 & 2 & -2 & -1 & -1 & 2 & 2 & -2 & 0 \\
2 & 2 & -1 & -1 & 1 & 0 & 2 & 1 & 2 & 0 & 3 & 2 & 1 & 1 & -2 & -2 & 2 & 0 & -3 & -1 & 3 & -2 \\
2 & -1 & -1 & 1 & 0 & 2 & 1 & 2 & 0 & 3 & 2 & 1 & 1 & -2 & -2 & 2 & 0 & -3 & -1 & 3 & -2 & 2 \\
-1 & 0 & -2 & -1 & -2 & 0 & -3 & -2 & -2 & 1 & 1 & 2 & -2 & 0 & 3 & 1 & -3 & 2 & -2 & -1 & -1 & 2 \\
-2 & 0 & -3 & -2 & -2 & 1 & 1 & -1 & 0 & -2 & -1 & 1 & -3 & 2 & -2 & -1 & -1 & 2 & 2 & -2 & 0 & 3
\end{pmatrix}$$

```
sage: LM.row(9)
```

$$(-1,\ 1,\ 0,\ -1,\ 0,\ 0,\ -1,\ 0,\ 0,\ 0,\ 1,\ -1,\ 0,\ 0,\ 0,\ -1,\ 0,\ 0,\ 1,\ 1,\ -1,\ -1)$$

Exercise 5

Encrypt and decrypt the message "Fully secure systems do not exist today and they will not exist in the future" by means of the ITRU cryptosystem with $f = 557, g = 806, r = 17$ and $q = 13844041$.

```
sage: s='
   Fullysecuresystemsdonotexisttodayandtheywillnotexistinthefuture
   '
```

```
sage: r=17
```

```
sage: p=1000
```

```
sage: f=557
```

```
sage: g=806
```

```
sage: q=13844041
```

```
sage: m=[ord(k) for k in s]
```

```
sage: Fp=(1/f)%p
```

```
sage: Fq=(1/f)%q
```

```
sage: h=(p*Fq*g)%q
```

```
sage: e=[((r*h)+m[i])%q for i in [0..len(m)-1]]
```

```
sage: matrix(9,7,e)
```

$$\begin{pmatrix} 13272199 & 13272246 & 13272237 & 13272237 & 13272250 & 13272244 & 13272230 \\ 13272228 & 13272246 & 13272243 & 13272230 & 13272244 & 13272250 & 13272244 \\ 13272245 & 13272230 & 13272238 & 13272244 & 13272229 & 13272240 & 13272239 \\ 13272240 & 13272245 & 13272230 & 13272249 & 13272234 & 13272244 & 13272245 \\ 13272245 & 13272240 & 13272229 & 13272226 & 13272250 & 13272226 & 13272239 \\ 13272229 & 13272245 & 13272233 & 13272230 & 13272250 & 13272248 & 13272234 \\ 13272237 & 13272237 & 13272239 & 13272240 & 13272245 & 13272230 & 13272249 \\ 13272234 & 13272244 & 13272245 & 13272234 & 13272239 & 13272245 & 13272233 \\ 13272230 & 13272231 & 13272246 & 13272245 & 13272246 & 13272243 & 13272230 \end{pmatrix}$$

```
sage: a=[(f*e[i])%q for i in [0..len(e)-1]]
```

```
sage: C=[(Fp*a[l])%p for l in [0..len(a)-1]]
```

```
sage: D=[chr(k) for k in C]
```

```
sage: ''.join(D)
```

```
Fullysecuresystemsdonotexisttodayandtheywillnotexistinthefuture
```

11.8 Shamir's secret sharing

Exercise 1

Use Shamir's secret sharing to share $s = 19$ over $\mathbb{F}_{79}$ such that 4 users can reconstruct the secret. Generate at least 6 shares.

```
sage: P.<x>=PolynomialRing(GF(79))
```

```
sage: s=19
```

```
sage: cfs=[s,1,2,3]
```

```
sage: F=P(cfs)
```

```
sage: F
```

$$3x^3 + 2x^2 + x + 19$$

```
sage: "Shares:"
```

Shares:

```
sage: [(k,F(x=k)) for k in [1..6]]
```

$$[(1, 25), (2, 53), (3, 42), (4, 10), (5, 54), (6, 34)]$$

Exercise 2

Shamir's secret sharing with $p = 31$. We know the following shares and we also know that 3 parties can reconstruct the secret

$$S_1 = 16, \quad S_4 = 16,$$
$$S_2 = 5, \quad S_5 = 7,$$
$$S_3 = 5, \quad S_6 = 9.$$

Recover the secret by using the shares S_1, S_2, S_3 and S_4, S_5, S_6.

Using S_1, S_2, S_3 we obtain:

```
sage: M=matrix.vandermonde([1,2,3],ring=GF(31))
```

```
sage: M
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix}$$

```
sage: v=vector(GF(31),[16,5,5])
```

```
sage: v
```

(16, 5, 5)

```
sage: sol=M.solve_right(v)
```

```
sage: sol
```

(7, 19, 21)

```
sage: "Secret:␣",sol[0]
```

(Secret: ,7)

Using S_4, S_5, S_6 we obtain:

```
sage: M=matrix.vandermonde([4,5,6],ring=GF(31))
```

```
sage: M
```

$$\begin{pmatrix} 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 6 & 5 \end{pmatrix}$$

```
sage: v=vector(GF(31),[16,7,9])
```

```
sage: v
```

(16, 7, 9)

```
sage: sol=M.solve_right(v)
```

```
sage: sol
```

(7, 19, 21)

```
sage: "Secret:␣",sol[0]
```

(Secret: ,7)

11.9 Knapsack cryptosystems

Exercise 1

Generate a superincreasing sequence $\{w_1, w_2, \ldots, w_6\}$ such that $w_i \leq 2^i - 2^{i-1}$.

```
sage: w=[1]
```

```
sage: for k in [2..6]:
....:     x0=Set([sum(w)+1..2^k-2^(k-1)]).random_element()
....:     w.append(x0)
```

```
sage: w
```

[1, 2, 4, 8, 16, 32]

Exercise 2

We have the superincreasing sequence $\{1, 2, 4, 10, 20, 40\}$. Pack a knapsack weighing 53.

```
sage: K=[1, 2, 4, 10, 20, 40]
```

```
sage: w=53
```

```
sage: sol=[]
```

```
sage: for k in [1..len(K)]:
....:     if w>=K[-k]:
....:         sol.append(K[-k])
....:         w=w-K[-k]
```

```
sage: sol,sum(sol)
```

$$([40, 10, 2, 1], 53)$$

Exercise 3

Given a superincreasing sequence $S = \{2, 3, 7, 14, 30, 57, 120, 251\}$. Transform S into a general knapsack with $r = 41$ and $p = 491$. Encrypt the message $m = 10110111$.

```
sage: S=[2,3,7,14,30,57,120,251]
```

```
sage: p=491
```

```
sage: r=41
```

```
sage: m=[1,0,1,1,0,1,1,1]
```

```
sage: Sr=[(k*r)%p for k in S]
```

```
sage: Sr
```

$$[82, 123, 287, 83, 248, 373, 10, 471]$$

```
sage: M=[m[k]*Sr[k] for k in [0..len(m)-1]]
```

```
sage: sum(M)
```

$$1306$$

Exercise 4

We have the general knapsack $\{82, 123, 287, 83, 248, 373, 10, 471\}$ and a ciphertext is 548. Apply the LLL-algorithm to recover the original message.

```
sage: GK=[82,123,287,83,248,373,10,471]
```

```
sage: m=548
```

```
sage: Kmat=matrix (ZZ,9,1,GK+[-m])
```

```
sage: A = identity_matrix(ZZ,8)
```

```
sage: B = matrix(ZZ,8,1)
```

```
sage: A = A.augment(B)
```

```
sage: A.transpose( ).augment(Kmat)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 82 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 123 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 287 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 83 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 248 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 373 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 471 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -548 \end{pmatrix}$$

```
sage: KLLL=(A.transpose()).augment(Kmat).LLL()
```

```
sage: KLLL
```

$$\begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & 1 & -1 & 1 & -1 \\ 1 & 0 & 0 & 1 & 0 & 0 & -1 & 2 & 1 \\ 0 & 1 & 0 & 0 & -2 & 1 & 0 & 0 & 0 \\ -1 & -2 & -1 & 0 & 0 & 0 & -1 & -1 & 0 \\ 1 & 0 & 2 & 0 & 0 & 1 & -1 & -1 & 0 \end{pmatrix}$$

```
sage: KLLL.row(2)[0:-1]
```

$$(1,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0)$$

Exercise 4

Alice and Bob use the Chor-Rivest cryptosystem with $p = 13, h = 4, f(x) = x^4 + 3x^2 + 12x + 2$ and $g(x) = x, d = 3118$. The private permutation is given by $\pi = [1, 5, 6, 9, 0, 10, 11, 3, 2, 7, 8, 4, 12]$. Determine $(c_0, c_1, \ldots, c_{p-1})$ and encrypt the message $M = (1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0)$.

```
sage: p=13
```

```
sage: h=4
```

```
sage: F.<q>=GF(p^h)
```

```
sage: PF.<x>=PolynomialRing(GF(p))
```

```
sage: f=x^4 + 3*x^2 + 12*x + 2
```

```
sage: g=F.multiplicative_generator()
```

```
sage: a=[discrete_log(q+i,g) for i in [0..p-1]]
```

```
sage: d=3118
```

```
sage: perm=[1, 5, 6, 9, 0, 10, 11, 3, 2, 7, 8, 4, 12]
```

```
sage: c=[(a[i]+d)%(p^h-1) for i in perm]
```

```
sage: M=[1,0,0,0,1,0,1,0,0,0,1,0,0]
```

```
sage: C = sum(M[i]*c[i] for i in [0..p-1])
```

```
sage: C
```

24416

Exercise 4

Decrypt the ciphertext obtained in the previous exercise.

```
sage: p=13
```

```
sage: h=4
```

```
sage: F.<q>=GF(p^h)
```

```
sage: PF.<x>=PolynomialRing(GF(p))
```

```
sage: f=x^4 + 3*x^2 + 12*x + 2
```

```
sage: g=F.multiplicative_generator()
```

```
sage: a=[discrete_log(q+i,g) for i in [0..p-1]]
```

```
sage: d=3118
```

```
sage: perminv=[4, 0, 8, 7, 11, 1, 2, 9, 10, 3, 5, 6, 12]
```

```
sage: C=24416
```

```
sage: r=(C-h*d)%(p^h-1)
```

```
sage: s=g^r
```

```
sage: S=[-k[0] for k in (PF(s.polynomial())+f).roots()]
```

```
sage: M0=[0 for _ in [0..p-1]]
```

```
sage: for k in S:
....:     M0[perminv[k]]=1
```

```
sage: M0
```

$$[1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0]$$

Bibliography

[1] M. R. K. Ariffin and N. A. Abu. AA_{β}-cryptosystem: A chaos based public key cryptosystem. *International Journal of Cryptology Research*, 1(2):149–163, 2009.

[2] S. R. Blackburn. The discrete logarithm problem modulo one: cryptanalysing the Ariffin-Abu cryptosystem. *J. Math. Cryptol.*, 4(2):193–198, 2010.

[3] G.R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.

[4] R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Math. Comp.*, 36(154):627–630, 1981.

[5] B. Chor and R. L. Rivest. A knapsack type public key cryptosystem based on arithmetic in finite fields (preliminary draft). In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 54–65, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[6] G.I. Davida. *Chosen Signature Cryptanalysis of the RSA (MIT) Public Key Cryptosystem*. TR-CS. Department of Electrical and Computer Science, College of Engineering and Applied Science, University of Wisc., 1982.

[7] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in cryptology (Santa Barbara, Calif., 1984)*, volume 196 of *Lecture Notes in Comput. Sci.*, pages 10–18. Springer, Berlin, 1985.

[8] P. Gaborit, J. Ohler, and P. Solé. CTRU, a polynomial analogue of NTRU. Technical Report RR-4621, INRIA, November 2002.

[9] J. N. Gaithuru, M. Salleh, and I. Mohamad. Itru: Ntru-based cryptosystem using ring of integers. *International Journal of Innovative Computing*, 7(1):33–38, 2017.

[10] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. The Clarendon Press, Oxford University Press, New York, fifth edition, 1979.

[11] J. Hoffstein, J. Pipher, and J. H. Silverman. Ntru: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[12] I. Ilić and S. S. Magliveras. Weak discrete logarithms in non-abelian groups. *J. Combin. Math. Combin. Comput.*, 74:3–11, 2010.

[13] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982.

[14] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Ann. of Math. (2)*, 126(3):649–673, 1987.

[15] R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24(5):525–530, Sep. 1978.

[16] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology—EUROCRYPT '99 (Prague)*, volume 1592 of *Lecture Notes in Comput. Sci.*, pages 223–238. Springer, Berlin, 1999.

[17] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Inform. Theory*, IT-24(1):106–110, 1978.

[18] J. M. Pollard. A Monte Carlo method for factorization. *Nordisk Tidskr. Informationsbehandling (BIT)*, 15(3):331–334, 1975.

[19] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[20] A. Shamir. A polynomial-time algorithm for breaking the basic merkle - hellman cryptosystem. *IEEE Transactions on Information Theory*, 30(5):699–704, Sep. 1984.

[21] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[22] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, USA, 6th edition, 2013.

[23] W. A. Stein et al. *Sage Mathematics Software, version* `8.9`. The Sage Development Team, 2019. `http://www.sagemath.org`.

[24] M. J. Wiener. Cryptanalysis of short rsa secret exponents. *IEEE Transactions on Information Theory*, 36(3):553–558, May 1990.