SageMath Examples from the CrypTool Book

Dr. Doris Behrendt

FrOSCon 2022
Hochschule Bonn-Rhein-Sieg

21. August 2022

Contents

1 What is the CrypTool Book?

- The CrypTool Book (CTB) is a book about cryptography. It's part of the open source project CrypTool (CT).
  Project manager: Prof. Esslinger, University Siegen.
- Title: Learning and experiencing Cryptography with CrypTool and Sagemath.
- Main Webpage of the project: https://www.cryptool.org/en/
- Navigate to the CTB via the tab „Documentation" oder directly: https://www.cryptool.org/en/documentation/ctbook/
- You can find some - not all yet !!! - of the SageMath Examples here: https://www.cryptool.org/en/documentation/ctbook/sagemath
- At the moment (August 21 2022) on our homepage there is still the 12th edition from 2018; new edition 2022 will soon be finished.

  The SageMath examples and screenshots in this slideshow are taken from the coming edition and therefore, when referenced, have no page number.

2 Overview Contents of CTB

1. Security Definitions and Encryption Procedures
2. Paper and Pencil Encryption Methods
3. Prime Numbers
4. Introduction to Elementary Number Theory with Examples
5. The Mathematical Ideas behind Modern Cryptography
6. Hash Functions, Digital Signatures, and PKIs
7. Elliptic Curves
8. Introduction to Bitblock and Bitstream Ciphers
9. Homomorphic Ciphers
10. Survey on Current Academic Results: Solving Discrete Logarithms and Factoring
11. Crypto 202x Perspectives for Long-Term Cryptographic Security
12. Lightweight Introduction to Lattices

- Appendices:
  - SageMath, Jupyter
  - CT1, CT2 etc.
  - openssl
  - ...

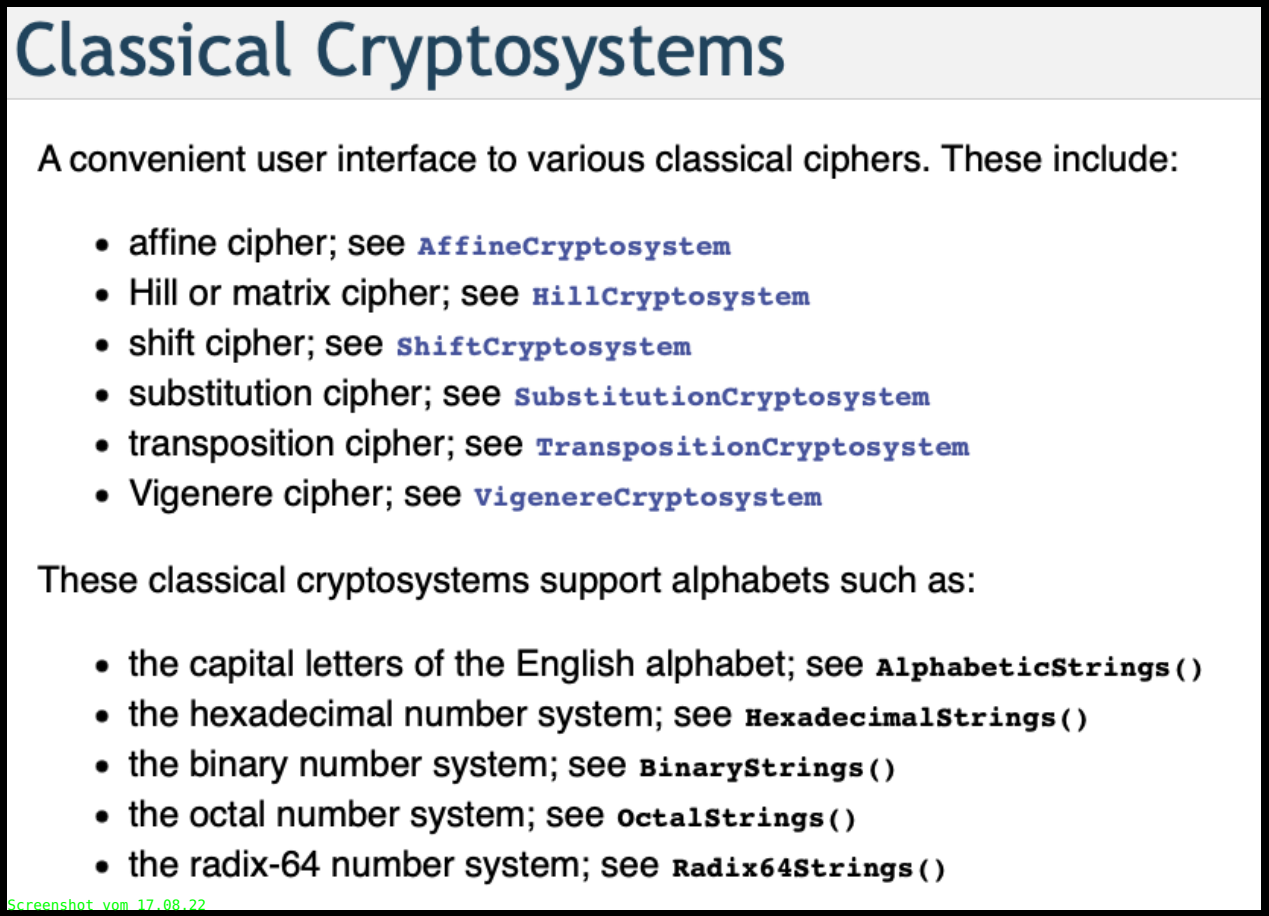
3 Chosen examples for today

Short examples such that you can try to follow on your own machine during the talk.

- classical
- transformation of bit blocks, XOR
- RSA

4 Classical Cryptosystems

https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/classical.html



Classical Cryptosystems

A convenient user interface to various classical ciphers. These include:

- affine cipher; see **AffineCryptosystem**
- Hill or matrix cipher; see **HillCryptosystem**
- shift cipher; see **ShiftCryptosystem**
- substitution cipher; see **SubstitutionCryptosystem**
- transposition cipher; see **TranspositionCryptosystem**
- Vigenere cipher; see **VigenereCryptosystem**

These classical cryptosystems support alphabets such as:

- the capital letters of the English alphabet; see **AlphabeticStrings()**
- the hexadecimal number system; see **HexadecimalStrings()**
- the binary number system; see **BinaryStrings()**
- the octal number system; see **OctalStrings()**
- the radix-64 number system; see **Radix64Strings()**

Screenshot vom 17.08.22

4.1 Substitution Cipher, Variant 1

Suppose we have a message written with „letters“ (German: Buchstaben) from a (finite) set Ω.

For now, let

$$\Omega = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$$

with $|\Omega| = 26$.

```
sage: Buchstaben='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: Omega=[i for i in Buchstaben]
sage: Omega
['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']
sage: len(Omega)
26
```

In a substitution procedure, the letters from Ω become simply put in another order. Mathematically more precise: The encryption is given by a permutation of Ω.

```
sage: k=Permutations(26).random_element()
sage: k
[13, 8, 6, 4, 11, 16, 9, 14, 26, 18, 7, 22, 12, 3, 21, 2, 23, 24, 20, ▸
 ▸5, 19, 10, 1, 25, 17, 15]
```


```
sage: key=[Omega[k(i+1)-1] for i in range(len(Omega))]
#keep in mind that range(len(Omega))=range(26)=0,1,2,...,24,25; k(0) ▸
 ▸is not defined;
#Omega[k(i)] instead of Omega[k(i+1)-1] returns an error!!
sage: key
['M','H','F','D','K','P','I','N','Z','R','G','V','L','C','U','B','W','▸
 ▸X','T','E','S','J','A','Y','Q','O']
```

So „the key“ is the above list. It’s more clear if we use a dictionary instead of a list,
syntax: {key1:value1,key2:value2,...}

```
sage: keydict={Omega[i]:key[i] for i in range(len(Omega))}
sage: keydict
{'A':'M', 'B':'H', 'C':'F', 'D':'D', 'E':'K', 'F':'P', 'G':'I', 'H':'N
 ', 'I':'Z', 'J':'R', 'K':'G', 'L':'V', 'M':'L', 'N':'C', 'O':'U', '
 P':'B', 'Q':'W', 'R':'X', 'S':'T', 'T':'E', 'U':'S', 'V':'J', 'W':'
 A', 'X':'Y', 'Y':'Q', 'Z':'O'}
```

Access the values of a key:

```
sage: keydict['A']
'M'
```

Iterate, then convert to a string without blanks by using join():

```
sage: msg='HALLOWELT'           # msg for message
sage: c=''.join([keydict[i] for i in msg])
sage: c                         # c for ciphertext
'NMVVUAKVE'
```

More compact: define a function ver for encryption (German: Verschl"usselung):

```
sage: def ver(z):       # verschlüssele Zeichenkette/string z
....:     return(''.join([keydict[i] for i in z]))
....:
sage: ver(msg)
'NMVVUAKVE'
```

First problem: blanks

```
sage: m2='HALLO WELT'
sage: ver(m2)
---------------------------------------------------------
...
KeyError: ' '
sage:
```

Second problem: lower case letters

```
sage: m3="HalloWelt"
sage: ver(m3)
---------------------------------------------------------
KeyError                  Traceback (most recent call last)
...
KeyError: 'a'
sage:
```

4.2 String Monoids

Better use free string monoids. In algebra a monoid is a set of „things“ together with a „composition“ that yields associativity. In the case of letters this composition is the concatenation. https://doc.sagemath.org/html/en/reference/monoids/sage/monoids/string_monoid.html

```
sage: Buchstabenx=AlphabeticStrings()
sage: Buchstabenx
Free alphabetic string monoid on A-Z
sage: alph=Buchstabenx.alphabet()
sage: alph
('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')
sage: alph[0]
'A'
sage: type(alph)
<class 'tuple'>
```

Tuples (round brackets) do not allow to overwrite their items - in contrast to lists (square brackets).

```
sage: alph[0]='B'
------------------------------------------------------------
TypeError                 Traceback (most recent call last)
...
TypeError: 'tuple' object does not support item assignment
sage: type(Omega)
<class 'list'>
sage: Omega[0]
'A'
sage: Omega[0]='B'
sage: Omega[0]
'B'
```

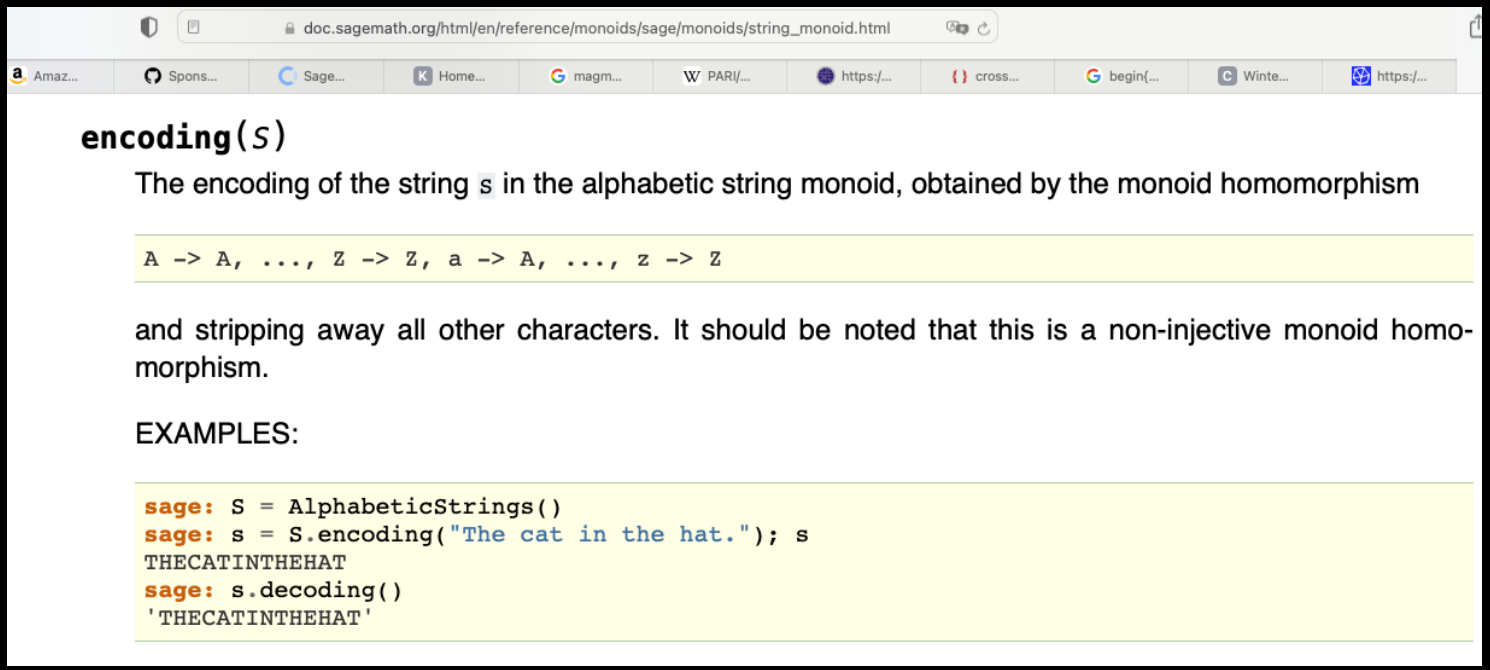
As an alternative to Buchstabenx.alphabet():

```
sage: Buchstabenx.gens()
(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)
sage: type(Buchstabenx.gens())
<class 'tuple'>
sage: type(Buchstabenx.gen(0))
<class 'sage.monoids.string_monoid_element.StringMonoidElement'>
```

Compared with the variant ('A','B',...,'Z') from before, where the characters are strings of length 1, the characters in the above listing are generators of the monoid, that is, algebraically abstract objects and not strings. Calling the python built-in function ord() which returns the decimal ASCII code of an ASCII character results in an error:

```
sage: ord(Buchstabenx.gen(0))
--------------------------------------------
...
TypeError: ord() expected string of length 1, but StringMonoidElement found
sage: ord(Buchstaben[0])
65
sage: ord('A')
65
```

The class AlphabeticStrings has a method encoding(). If we use it, we don't have to care about punctuation, spaces, lowercase letterns etc.:



Really „stripping away all other characters“?

```
sage: P="Hello Wörld!"
sage: msg=Buchstabenx.encoding(P)
------------------------------------------------------------
...
ValueError: 'Ö' is not in list
...
During handling of the above exception, another exception occurred:
...
TypeError: Argument x (= HELLOWÖRLD) is not a valid string.
sage:
```

4.3 Substitution Cipher, Variant 2

Not only for the set of characters but also for the substitution cryptosystem there is „a convenient user interface“ in SageMath:

```
sage: reset()   # deletes all user defined variables
sage: S=SubstitutionCryptosystem(AlphabeticStrings())
sage: key=S.random_key()
sage: key
ZGVBULOKFHTDRASYECNJXWMPIQ
sage: P="Das ist eine geheime Nachricht, die keine Umlaute und scharfe
 ▸ ss enthalten darf!" # this is a secret message with no umlauts
sage: msg=S.encoding(P)
sage: msg
DASISTEINEGEHEIMENACHRICHTDIEKEINEUMLAUTEUNDSCHARFESSENTHALTENDARF
sage: C=S.enciphering(key,msg)
sage: C
BZNFNJUFAUOUKUFRUAZVKCFVKJBFUTUFAUXRDZXJUXABNVKZCLUNNUAJKZDJUABZCL
sage: DC=S.deciphering(key,C)
sage: DC
DASISTEINEGEHEIMENACHRICHTDIEKEINEUMLAUTEUNDSCHARFESSENTHALTENDARF
```

Conventions in the CrypTool Book:

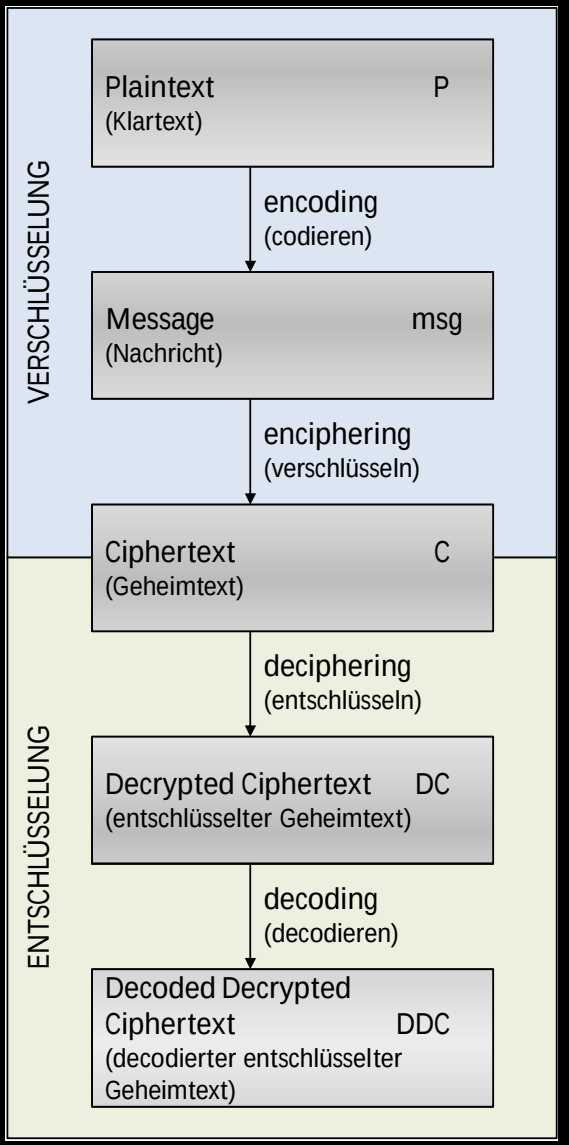
1. encode: $P \longrightarrow msg$
2. encipher: $msg \longrightarrow C$
3. decipher: $C \longrightarrow DC$

   The step from C to DC (with $msg \stackrel{!}{=} DC$) sometimes is also called decryption instead of deciphering, but it avoided because one could think of exhumations.

4. decode: $DC \longrightarrow DDC$

   The step from DC back to $DDC \stackrel{!}{=} P$ is left out. It is necessary e.g. if the msg, C and DC are represented binary.

Comment: When using the term de-/encoding then the mapping from „before“ to „after“ is known. When speaking of encrypting/enciphering/deciphering, the map between „before“ and „after“ is a secret. Often not all information about the mapping is secret, but at least a part of it.



4.4 Special Case of Substitution Cipher: Cyclic Translation

If we don't allow arbitrary permutations, but only cyclic shifting to the right by 3 places (mod 26): Caesar Cipher

Shifting by arbitrary $k < 26$ places to the right (also mod 26): Shift Cipher

```
sage: reset()
sage: S=ShiftCryptosystem(AlphabeticStrings())
sage: key=3
sage: P="Gallia est omnis divisa in partes tres, quarum unam incolunt
 ▸Belgae ..."
sage: msg=S.encoding(P)
sage: msg
GALLIAESTOMNISDIVISAINPARTESTRESQUARUMUNAMINCOLUNTBELGAE
sage: C=S.enciphering(key,msg)
sage: C
JDOOLDHVWRPQLVGLYLVDLQSDUWHVWUHVTXDUXPXQDPLQFROXQWEHOJDH
sage: DC=S.deciphering(key,C)
sage: DC
GALLIAESTOMNISDIVISAINPARTESTRESQUARUMUNAMINCOLUNTBELGAE
```

4.5 The Hill Cipher, Variant 1

First we show how to do it „hands on“:

We start with the bijection between the 26 ASCII letters and the numbers from 0 to 25. This time the letter A is not mapped onto 1 like we did it before.

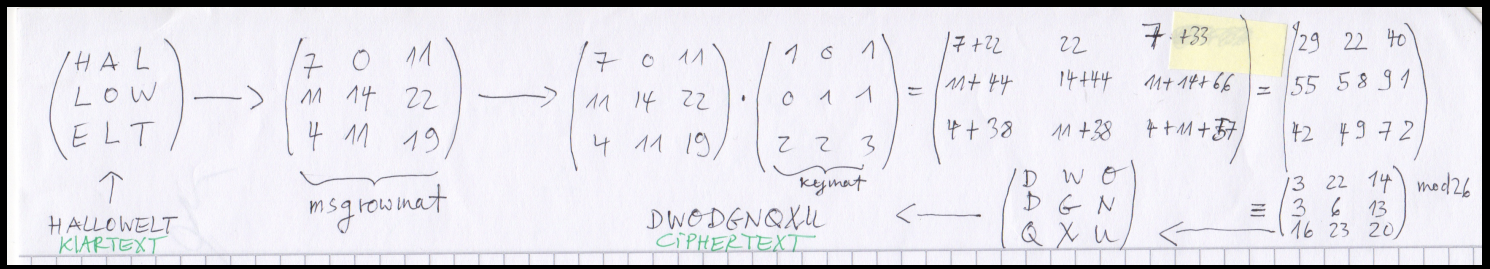
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

```
sage: Buchstaben='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: bij={Buchstaben[i]:i for i in range(len(Buchstaben))}
sage: bij
{'A':0,'B':1,'C':2,'D':3,'E':4,'F':5,'G':6,'H':7,'I':8,'J':9,'K':10,'L
 ':11,'M':12,'N':13,'O':14,'P':15,'Q':16,'R':17,'S':18,'T':19,'U':20
 ,'V':21,'W':22,'X':23,'Y':24,'Z':25}
```

The key is a square matrix $K = \text{keymat}$ with entries from $\mathbb{Z}_{26}$ (that is, mod 26):

```
sage: R=Zmod(26)
sage: keymat=matrix(R,[[1,0,1],[0,1,1],[2,2,3]])
# this is the matrix from the sagemath doc webpage: https://doc.
  sagemath.org/html/en/reference/cryptography/sage/crypto/classical.
  html#sage.crypto.classical.HillCryptosystem
# not every matrix is a good choice, see later
sage: keymat
[1 0 1]
[0 1 1]
[2 2 3]
```

The message - capital letters without ü, ä, SS etc. - is replaced by the via bij mapped numbers and written row by row into a (also 3×3) matrix. The length of the message must be divisible by 3.




```
sage: msg='HALLOWELT'
```

```
sage: len(msg)%3==0
True
sage: msgzahl=[bij[c] for c in msg]
sage: msgzahl           # zahl (German) = number
[7, 0, 11, 11, 14, 22, 4, 11, 19]
sage: msgrowmat=matrix(R,3,msgzahl)
sage: msgrowmat
[ 7  0 11]
[11 14 22]
[ 4 11 19]
```

Now we multiply the key matrix with the message matrix, the message matrix being the left factor. Mind that the examples on Wikipedia (https://en.wikipedia.org/wiki/Hill_cipher) write the message into a matrix column by column and then multiply from the right.

```
sage: Cmat=msgrowmat*keymat
sage: Cmat
[ 3 22 14]
[ 3  6 13]
[16 23 20]
sage: Cmat.list()
[3, 22, 14, 3, 6, 13, 16, 23, 20]
```

Later we want to compare our result with the result that we get using the Hill Cipher from sagemath. For this we need letters again instead of a list of numbers < 26. So we proceed by inverting the dictionary from before and using it to convert Cmat to a string:

```
sage: jib={value:key for (key,value) in bij.items()}
sage: jib
{0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11
 :'L',12:'M',13:'N',14:'O',15:'P',16:'Q',17:'R',18:'S',19:'T',20:'U
 ',21:'V',22:'W',23:'X',24:'Y',25: 'Z'}
sage: C=''.join([jib[i] for i in C.list()])
sage: C
'DWODGNQXU'
```

4.6 Hill Cipher, Variant 2

After this hands on solution now the quite shorter variant from https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/classical.html#sage.crypto.classical.HillCryptosystem:

```
sage: keylen=3
sage: A=AlphabeticStrings()
sage: H=HillCryptosystem(A,keylen)
sage: HKS=H.key_space()
sage: key=HKS([[1,0,1],[0,1,1],[2,2,3]])
sage: key
[1 0 1]
[0 1 1]
[2 2 3]
sage: P="Hallo Welt!"
sage: msg=H.encoding(P)
sage: msg
HALLOWELT
sage: len(msg)%keylen==0
True
sage: C=H.enciphering(key,msg)
sage: C
DWODGNQXU # As expected the ciphertext $C$ from here and from before are the same.
sage:
```

4.7 Inverse Matrix

The way back is only one line:

```
sage: DC=H.deciphering(key,C)
sage: DC
HALLOWELT
```

The deciphering in the „hands on“ case is also quite easy: You just have to compute the inverse matrix of the key matrix (mod 26 of course) and multiply it with Cmat from the right.

```
#sage: R=Zmod(26)       # should still be in namespace
#sage: keymat=matrix(R,[[1,0,1],[0,1,1],[2,2,3]]) # like above
sage: keymat.inverse()  # shorter alternative: ~keymat
[25 24  1]
[24 25  1]
[ 2  2 25]
#sage: Cmat=matrix(R,[[3,22,14],[3,6,13],[16,23,20]])   # like above
sage: DC=Cmat*keymat.inverse()
sage: DC
[ 7  0 11]
[11 14 22]
[ 4 11 19]
```

Now convert the numbers to letters again and the matrix to a string:

```
sage: DCchars=''.join([jib[i] for i in DC.list()])      # jib s.o.
sage: DCchars
'HALLOWELT'
sage:
```

Just for the record: Over $\mathbb{Q}$ this doesn't work!

```
sage: E=matrix(QQ,[[1,0,1],[0,1,1],[2,2,3]])
sage: ~E
[-1 -2  1]
[-2 -1  1]
[ 2  2 -1]
sage: F=matrix(QQ,[[25,24,1],[24,25,1],[2,2,25]])
sage: F*E
[27 26 52]
[26 27 52]
[52 52 79]
```

When choosing the key matrix, one has to make sure that it has an inverse in $\mathrm{Mat}(n \times n, \mathbb{Z}_{26})$. This is the case only of $\det(K) \notin \{0, 2, \ldots, 12, 13, 14, \ldots, 22, 24\}$, that is, the determinant of K has no common divisors with 26 over $\mathbb{Z}$.

```
sage: det(E)
-1
sage: type(keymat)
<class 'sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_float'>
sage: type(E)
<class 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
sage: det(keymat)
25
```

4.8 Attack on the Hill Cipher

You can attack the Hill cipher if you know a pair of corresponding ciphertext and plaintext, but not the key. This is called a „known plaintext attack“ (KPA).

https://www.informatik.hu-berlin.de/de/forschung/gebiete/algorithmenII/Lehre/ws05/krypto1/skript/kap2.pdf:

> Known Plaintext Attack
>
> The opponent is in possession of a number of matching plaintext cryptotext pairs. Experience has shown that this makes it possible to decrypt further cryptotexts or to determine the keys used is made much easier.

We recall the plain message matrix msgrowmat and the cipher text matrix Cmat from before:

```
sage: msgrowmat
[ 7  0 11]
[11 14 22]
[ 4 11 19]
sage: Cmat
[ 3 22 14]
[ 3  6 13]
[16 23 20]
# if you don't have it in your SageMath namespace anymore:
# msgrowmat=matrix(Zmod(26),[[7,0,11],[11,14,22],[4,11,19]]), same ▸
  ▸syntax for Cmat
sage: P=msgrowmat       # short P for plaintext matrix
sage: C=Cmat            # short C for ciphertext matrix
```

The encryption was done by matrix multiplication $C = PA$. If P is invertible over $\mathbb{Z}_{26}$ we can multiply this matrix equation with its inverse P^{-1} from the left and get $P^{-1}C = A$:

```
sage: A=~P*C
sage: A
[1 0 1]
[0 1 1]
[2 2 3]
```

For testing if a message, represented as a capital letter string s, corresponds to an invertible matrix, we have to compute the determinant and check wether it is invertible in the ring $\mathbb{Z}_{26}$. As noted earlier, this is the case if the determinant has no common divisors with 26. And of course the letters of the message must fill a 3×3 matrix, so it has to have 9 letters:

```
sage: def test(s):                  # s a capital letter string
....:     if len(s)!=9:
....:         print('length not =9')
....:     P=matrix(R,3,[bij[c] for c in s])
....:     d=P.det()
....:     if gcd(26,d)!=1:          # gcd greatest common divisor
....:         return(False)
....:     else:
....:         return(True)
....:
sage: test('HALLOWELT')
True
sage: test('NEINDOCHO')
False
```

Then we get an error message:

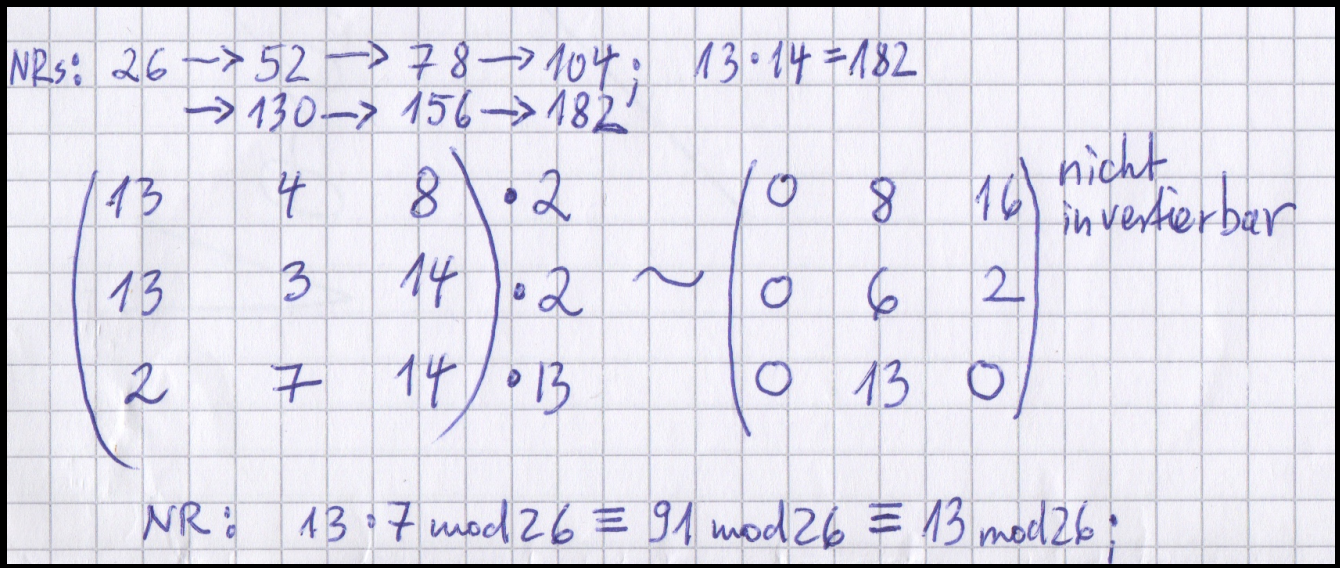
```
sage: reset()   # begin again from the start
sage: Buchstaben='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: bij={Buchstaben[i]:i for i in range(len(Buchstaben))}
sage: R=Zmod(26)
sage: string="NEINDOCHO"
sage: P=matrix(R,3,[bij[c] for c in string])
sage: A=matrix(R,3,[[1,0,1],[0,1,1],[2,2,3]])
# A is supposed to be unknown
sage: C=P*A
# but we need A in order to compute C
sage: C
[ 3 20 15]
[15  5  6]
[ 4  9 25]
# now we forget A again and assume that only C and P are known
#                       und A is what we are searching for
sage: ~P        # Reminder: ~P is the inverse of P
------------------------------------------------------------
TypeError                 Traceback (most recent call last)
...
   5795             except (TypeError, ZeroDivisionError):
...
   1493             if not b.gcd(d).is_one():
...
TypeError: matrix denominator not coprime to modulus
...
ZeroDivisionError: input matrix must be nonsingular
```

The not invertible (number-) matrix P corredponding to the text string "NEINDOCHO" is:

```
sage: P
[13  4  8]
[13  3 14]
[ 2  7 14]
```

(The code for converting the text string into a number matrix was described on page 21 already.)

That this matrix is not invertible over $\mathbb{Z}_{26}$ we can check by hand:



At the end of the second chapter „Paper and Pencil Encryption Methods“ of the CrypTool Book there is a slightly longer SageMath example of a KPA against the Hill Cipher which allows to investigate larger matrices and longer plaintext-ciphertext pairs.

5 XOR encryption

Taken from the CrypTool Book, chapter 8.3. „Bitstream ciphers“:

„The basic method of bitstream encryption is simply denoted by XOR. It interprets plaintexts as sequences of bits. Also the key is a bit sequence, called **key stream**. The encryption algorithm adds the current bit of the plaintext and the current bit of the key stream by XOR.“

```
msg: 01000100011101 ...
key: 10010110100101 ...
     ------------------
  c: 11010010111000 ...
```

5.1 fun with bits

It is possible to use the monoid with zeroes and ones, similar to the 26 capital letter monoid used previously. Instead of AlphabeticStrings() we then use BinaryStrings():

```
sage: S=BinaryStrings()
sage: S
Free binary string monoid
sage: S.encoding('A')
01000001
sage: S.encoding('A',padic=True)
10000010                # 8-Bit ASCII reversed
sage: ord('A')          # decimal ASCII code of capital A
65
sage: bin(65)           # with Pyhton built-in functions we can
'0b1000001'             # also get the binary representation, as a
 string
sage: bin(65)[2:].zfill(8)      # cut 0b on the left, fill with zeroes
 (zero-fill) up to length 8
'01000001'
```

The binary representation as string we can also get with binary() instead of with bin(65), for a list we can use bits(). But here, too, you have to watch out for the data types:

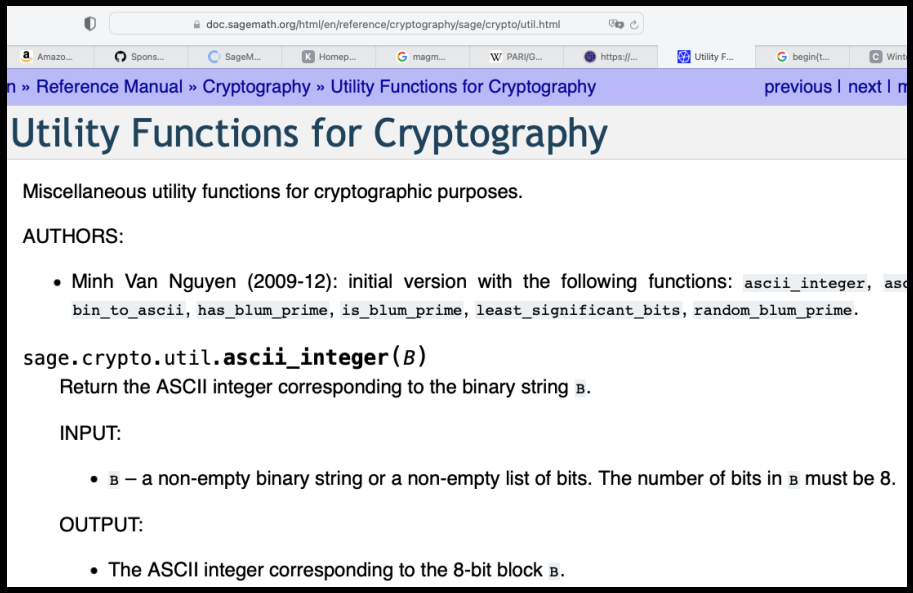
```
sage: 65.binary()
'1000001'                       # Alternative
sage: type(65)
<class 'sage.rings.integer.Integer'>
sage: type(ord('A'))
<class 'int'>
sage: 65.bits()
[1, 0, 0, 0, 0, 0, 1]
sage: ord('A').bits()           # ord('A')=65
-----------------------------------------------------
AttributeError         Traceback (most recent call last)
...
AttributeError: 'int' object has no attribute 'bits'
sage: Integer(ord('A')).bits()
[1, 0, 0, 0, 0, 0, 1]
```

As you can see from the AttributeError above, there is a difference between a SageMath integer and a python integer.

Unfortunately, the way back is not implemented for BinaryStrings():

```
sage: S.decoding(01000001)
------------------------------------------------------------
...
KeyError: 'decoding'
...
AttributeError: 'BinaryStringMonoid_with_category' object has no ▸
 ▸attribute 'decoding'
```

But there is the command ascii_integer(), see https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/util.html:



doc.sagemath.org/html/en/reference/cryptography/sage/crypto/util.html

n » Reference Manual » Cryptography » Utility Functions for Cryptography previous | next | n

Utility Functions for Cryptography

Miscellaneous utility functions for cryptographic purposes.

AUTHORS:

- Minh Van Nguyen (2009-12): initial version with the following functions: `ascii_integer`, `asc` `bin_to_ascii`, `has_blum_prime`, `is_blum_prime`, `least_significant_bits`, `random_blum_prime`.

sage.crypto.util.**ascii_integer**(*B*)

Return the ASCII integer corresponding to the binary string `B`.

INPUT:

- `B` – a non-empty binary string or a non-empty list of bits. The number of bits in `B` must be 8.

OUTPUT:

- The ASCII integer corresponding to the 8-bit block `B`.

```
sage: S=BinaryStrings()
sage: B=S.encoding('A')
sage: B
01000001
sage: ascii_integer(B)
-----------------------------------------------------------
...     # don't forget to import :-)
NameError: name 'ascii_integer' is not defined
sage: from sage.crypto.util import ascii_integer
sage: ascii_integer(B)
65
sage: ascii_integer(01000001)
-----------------------------------------------------------
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
sage: ascii_integer('01000001')
65
sage: type(B)
<class 'sage.monoids.string_monoid_element.StringMonoidElement'>
sage: type(01000001)
<class 'sage.rings.integer.Integer'>
sage: x=01000001
sage: x         # x is an int with base 10, not 2
1000001         # the leading zero is stripped away
```

Other commands from cypto.util that you can/must import:

- ascii_to_bin
- bin_to_ascii

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin('A')
01000001
sage: ascii_to_bin('Hallo, Welt')
010010000110000101101100011011000110111100101100001000000101011101100101
 ▸0101101100011101 00
sage: ascii_to_bin('HalloWelt')
010010000110000101101100011011000110111101010111011001010110110001110100
 ▸00
sage: ascii_to_bin(['H','a','lloWelt'])
010010000110000101101100011011000110111101010111011001010110110001110100
 ▸00
```

Disadvantage: Representation of bit sequences as abstract elements of a monoid, problems with type conversion when processing further. To illustrate this, we compare PLUS and MAL (German „2 mal 3“ is „2 times 3“):

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin('A')
01000001
sage: type(ascii_to_bin('A'))
<class 'sage.monoids.string_monoid_element.StringMonoidElement'>
sage: a=ascii_to_bin('A')
sage: b=ascii_to_bin('B')
sage: a+b                       # PLUS -> ERROR
------------------------------------------------------------
...
AttributeError: 'StringMonoidElement' object has no attribute '_add_'
...
TypeError: unsupported operand parent(s) for +: 'Free binary string monoid' and '▸
 ▸Free binary string monoid'
sage: a*b                       # MAL  -> OK
0100000101000010
sage: a2=bin(ord('A'))[2:]
sage: b2=bin(ord('B'))[2:]
sage: a2+b2                     # PLUS -> OK
'10000011000010'
sage: a2*b2                     # MAL  -> EROOR
------------------------------------------------------------
...
TypeError: can't multiply sequence by non-int of type 'str'
```

The other way around from binary to ASCII:

```
sage: from sage.crypto.util import bin_to_ascii
sage: bin_to_ascii(01000001)           # like this it's a base 10 int
---------------------------------------------------------------
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
sage: bin_to_ascii('01000001')         # string: ok
'A'
sage: bin_to_ascii([0,1,0,0,0,0,0,1])  # list: also ok
'A'
sage: bin_to_ascii(3*[0,1,0,0,0,0,0,1])
'AAA'
```

Another way from binary to ASCII is worked out at the end of the RSA example on page 59.

5.2 Conversion Routines for Bitblocks

In the CrypTool Book, chapter 8.4. „Appendix: Boolean Maps in SageMath“, we define some useful conversion functions. The term bitblock in those examples always refers to a list.

The link https://www.cryptool.org/en/documentation/ctbook/sagemath leads to a file called „bitciphers.sage“. The following code is taken from there:

```
#############################################################
# Sage module 'bitciphers.sage'                             #
# --------------------------------------------------------- #
# Klaus Pommerening (Johannes-Gutenberg-Universitaet Mainz) #
#   2014-Dec-31, last version 2015-Aug-11                   #
# ...
# We refrain from defining a class "Bitblock", avoiding object ▸
 ▸oriented
# overhead and type conversion struggles.
# ...
#############################################################
##### Conversion routines for bitblocks #####################
#############################################################
```

```
def int2bbl(number,dim):                  # example ctd.
  """Converts number to bitblock of length dim via base-2 ▸
   ▸representation."""
  n = number                              # catch input
  b = []                                  # initialize output
  for i in range(0,dim):
    bit = n % 2                           # next base-2 bit
    b = [bit] + b                         # prepend
    n = (n - bit)//2
  return b

def bbl2int(bbl):
  """Converts bitblock to number via base-2 representation."""
  ll = len(bbl)
  nn = 0                                  # initialize output
  for i in range(0,ll):
    nn = nn + bbl[i]*(2**(ll-1-i))        # build base-2 representation
  return nn                               # return base-10 int
```

```
def str2bbl(bitstr):                    # example ctd.
  """Converts bitstring to bitblock."""
  ll = len(bitstr)
  xbl = []
  for k in range(0,ll):
    xbl.append(int(bitstr[k]))
  return xbl

def bbl2str(bbl):
  """Converts bitblock to bitstring."""
  bitstr = ""
  for i in range(0,len(bbl)):
    bitstr += str(bbl[i])
  return bitstr

def txt2bbl(text):
  """Converts ASCII-text to bitblock."""
  ll = len(text)
  xbl = []
  for k in range(0,ll):
    n = ord(text[k])
    by = int2bbl(n,8)
    xbl.extend(by)
  return xbl
```

Test: The link again, then go to bottom of page: https://www.cryptool.org/de/documentation/ctbook/sagemath

```
sage: load('./bitciphers.sage') # download
sage: thebeast=int2bbl(6,66); thebeast
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0▸
 ▸,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0]
sage: type(thebeast)
<class 'list'>
sage: type(thebeast[0])
<class 'sage.rings.integer.Integer'>
sage: number(thebeast[0])
---------------------------------------------------
...
NameError: name 'number' is not defined
sage: txt2bbl('the number of the beast')
[0,1,1,1,0,1,0,0,0,1,1,0,1,0,0,0,0,1,1,0,0,1,0,1,0,0,1,0,0,0,0,0,0,1,1▸
 ▸,0,1,1,1,0,0,1,1,1,0,1,0,1,0,1,1,0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1,0,▸
 ▸0,1,0,1,0,1,1,1,0,0,1,0,0,0,1,0,0,0,0,0,0,1,1,0,1,1,1,1,0,1,1,0,0,1▸
 ▸,1,0,0,0,1,0,0,0,0,0,0,1,1,1,0,1,0,0,0,1,1,0,1,0,0,0,0,1,1,0,0,1,0,▸
 ▸1,0,0,1,0,0,0,0,0,0,1,1,0,0,0,1,0,0,1,1,0,0,1,0,1,0,1,1,0,0,0,0,1,0▸
 ▸,1,1,1,0,0,1,1,0,1,1,1,0,1,0,0]
sage: len(txt2bbl('the number of the beast'))
184
```

5.3 XOR from the CrypTool Book

For encrypting with XOR you can use the following function, taken from the same file as above:

```
##########################################################
##### XOR of bitblocks ###################################
##########################################################
def xor(plain,key):     #input two lists of 0's and 1's
  """Binary addition of bitblocks.
  Crops key if longer than plain.
  Repeats key if shorter than plain.
  """
  lk = len(key)
  lp = len(plain)
  ciph = []
  i = 0
  for k in range(0,lp):
    cbit = (plain[k] + key[i]) % 2
    ciph.append(cbit)
    i += 1
    if i >= lk:
      i = i-lk
  return ciph
```

Here in this example the key is repeated if one has a „very“ long message and a quite „short“ key. In this case it is possible to break the cryptosyste

If the message is shorter than the key and if the key is destroyed after using it, the XOR encryption is an example of perfect secrecy in the sense of Shannon. This is known as the One Time Pad.

If the one-time pad is perfect – why not use it in any case?

- The key management is unwieldy: Key agreement becomes a severe problem since the key is as long as the plaintext and awkwardly to memorize. Thus the communication partners have to agree on the key stream prior to transmitting the message, and store it. Agreeing on a key only just in time needs a secure communication channel – but if there was one why not use it to transmit the plaintext in clear?
- The key management is inappropriate for mass application or multi-party communication because of its complexity that grows with each additional participant.

Screenshot taken from the CrypTool Book, chapter bistream ciphers.

Example of usage of a periodic key:

Example We generate a key stream of period 8 by repeating $k = 10010110$. Each letter is represented by bytes in the ISO character set. Remark: The plaintext here is in German.

```
      D    |   u    |        |   b    |   i    |   s    |
a: 01000100|01110101|00100000|01100010|01101001|01110011|
k: 10010110|10010110|10010110|10010110|10010110|10010110|
   -------- -------- -------- -------- -------- --------
c: 11010010|11100011|10110110|11110100|11111111|11100101|

      t    |        |   d    |   o    |   o    |   f
   01110100|00100000|01100100|01101111|01101111|01100110
   10010110|10010110|10010110|10010110|10010110|10010110
   -------- -------- -------- -------- -------- --------
   11100010|10110110|11110010|11111001|11111001|11110000
```

This encryption is easily done by hand, or by the SageMath example 8.3.1.

Re-encoding the ciphertext bytes in the ISO-8859-1 character set the cryptogram looks like this:

Ò ã ¶ ô œ å â ¶ ò ù ù ð

This might bedazzle laypersons. An expert immediately notes that all characters are from the upper half of the possible 256 bytes. This observation suggests that the plaintext is in natural language, encrypted with a key whose leading bit is 1. If the attacker guesses that the conspicuous character ¶ = 10110110 corresponds to the space character 00100000, she derives the key as the difference 10010110. This breaks the cryptogram.

Screenshot taken from the CrypTool Book, chapter bistream ciphers.

Reproduction of the „strange“ ASCII characters from the screenshot:

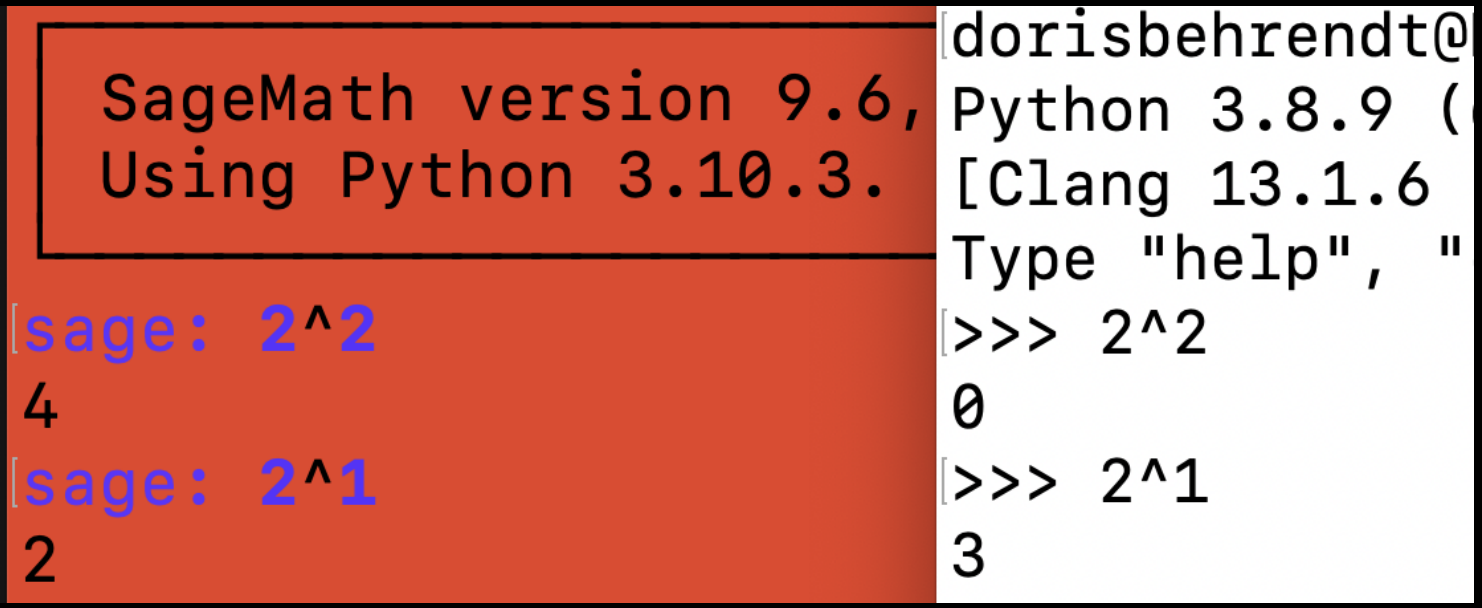
```
sage: load('./bitciphers.sage') # perhaps still in namespace: txt2bbl ▸
 ▸und bbl2str
sage: plaintext="Du bist doof"  # du bist doof = you're dumb
sage: bintext=txt2bbl(plaintext)
sage: binstr=bbl2str(bintext)
sage: binstr
'010001000111010100100000011000100110100101110011011101000010000001100▸
 ▸10001101111011011101100110'
sage: testkey=[1,0,0,1,0,1,1,0]
sage: altkey=[0,0,0,1,0,1,1,0]
sage: keystr=bbl2str(testkey)
sage: altkeystr=bbl2str(altkey)
sage: keystr
'10010110'
sage: altkeystr          # for later comparison a key
'00010110'               # with leading bit 0 instead of 1
sage: ciphertext=xor(bintext,testkey)
sage: ciphstr=bbl2str(ciphertext)
sage: from sage.crypto.util import bin_to_ascii
sage: bin_to_ascii(ciphstr)
'Òãűôßåâűòùùð'
```

```
sage: altciphertext=xor(bintext,altkey)
sage: altciphstr=bbl2str(altciphertext)
sage: bin_to_ascii(altciphstr)
'Rc6t\x7feb6ryyp'
```

Non printable ASCII characters start with an \x followed by two HEX digits. In binary representation they have a zero as leading bit.

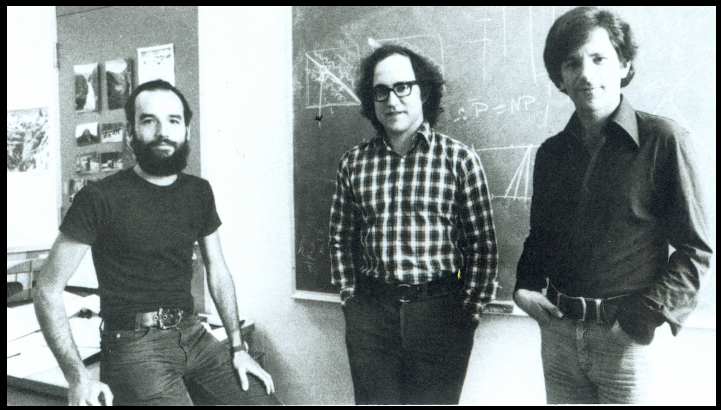
```
sage: x=bin_to_ascii(ciphstr); x
'Òãűôßåâűòùùð'
sage: x[2]
'ű'
sage: ord(x[2])
182             # not in the 7 bit ASCII set since  > 2^7=128
sage: bin(ord(x[2]))
'0b10110110'
sage: y=bin_to_ascii(altciphstr)
sage: y
'Rc6t\x7feb6ryyp'
sage: bin(ord(y[4]))
'0b1111111'
sage: ord(y[4])
127     # the ASCII character 127 stands for DELETE.
sage: y[4]
'\x7f'
```

5.4 XOR with python „only“: the hat ^



I don’t know if (and if it’s possible, how) we could call the simple python XOR from within SageMath. Searching the internet on the fly produced no answers.

6 RSA



https://cryptologicfoundation.org/what-we-do/educate/bytes/this_day_in_history_calendar.html/event/2020/09/20/1600578000/1983-three-inventors-receive-patent-for-encryption-algorithm-rsa/78258
https://www.zib.de/node/2931

6.1 Short walk through theory

1. the person who wants to receive an encrypted message (Bob) chooses two distinct primes p, q and computes their product n,
2. computes the number r of numbers that have no common divisors with n using the Euler totient functiom ϕ or the formula $r = \phi(pq) = (p-1)(q-1)$ for the special case where n is a product of only two different primes,
3. searches an element e of $\mathbb{Z}_r$ which has a multiplicative inverse; this condition is fulfilled if e and r have no common divisors,
4. Bob computes the inverse d of e in $\mathbb{Z}_r$ (that is: $\mod r$)
5. the numbers n and e are published, p, q and d are kept secret by Bob;
6. the person that wants to write a secret message to Bob (Alice) fetches n and e;
7. Alice converts the plaintext to a natural number $msg < n$,
8. computes $C = msg^e$ in $\mathbb{Z}_n$ (also $\mod n$); C is the ciphertext,
9. Alice sends C to Bob
10. Bob computes $DC = C^d$ in $\mathbb{Z}_n$; DC is the deciphered ciphertext, equal to the plaintext, encoded as natural number $< n$

The procedure works under the assumption that is is not possible to factor n „fast“ and therefore the attacker does not know r and then also cannot compute the inverse d of e „fast“.

6.2 Small Example of RSA

```
sage: p,q=random_prime(2^10),random_prime(2^10)
sage: p==q
False
sage: n=p*q
sage: Zn=Zmod(n)
sage: r=euler_phi(n)
sage: Zr=Zmod(r)
sage: e=ZZ.random_element(r)    # e=exponent=key for enciphering
sage: while gcd(e,r)!=1:
....:     e=ZZ.random_element(r)
....:
sage: d=Zr(e)^-1 # d=inverse of e in Zmod(r), key for deciphering
sage: msg=Zn.random_element(); msg
13492
sage: C=msg^e; C        # C ciphertext
113247
sage: n,p,q,r,e,d
(371323, 449, 827, 370048, 75447, 56071)
sage: C^d       # C^d deciphered ciphertext must be = msg
13492
```

6.3 Large Example of RSA

An RSA example with 512 bit primes is contained in chapter 12.10 „Lattices and RSA“ of the CrypTool Book. Here is a slightly altered example:

```
sage: P='6, 66, the number of the beast!'
sage: P_ascii=[ord(x) for x in P]
sage: P_ascii
[54,44,32,54,54,44,32,116,104,101,32,110,117,109,98,101,114,32,111,102
 ,32,116,104,101,32,98,101,97,115,116,33]
sage: P_bin=[bin(x)[2:].zfill(8) for x in P_ascii]
sage: P_bin
['00110110','00101100','00100000','00110110','00110110','00101100','00
 100000','01110100','01101000','01100101','00100000','01101110','011
 10101','01101101','01100010','01100101','01110010','00100000','0110
 1111','01100110','00100000','01110100','01101000','01100101','00100
 000','01100010','01100101','01100001','01110011','01110100','001000
 01']
sage: msg=Integer(''.join(P_bin),2)
sage: msg
95714286764733858046056695692827363370527019483767976502265996337803916
 321
sage: b=512
sage: p=random_prime(2^b-1,lbound=2^(b-1)+2^(b-2))
```

```
sage: q=random_prime(2^b-1,lbound=2^(b-1)+2^(b-2))
sage: p==q
False
sage: p.nbits(),q.nbits()
(512, 512)
sage: n=p*q
sage: n.nbits()
1024
sage: e=2^16+1  # not chosen, but recommended
sage: e
65537
sage: r=(p-1)*(q-1)
sage: r
107810173007506943736824962481404270820994227032408953703927122304982
 ▸95928532806282500656672337232486265296268253175077692437146473839599
 ▸60225675713366269241773276053673456466881428784788854175057119921099
 ▸65416592345227650190817138485152571023849869141852468637073364928679
 ▸48037129287663764457388728530764618700
sage: %time factor(r)   # better don't do it ;-) see last slide
sage: r.nbits()
1024
sage: d=inverse_mod(e,r)
sage: d
```

```
862339911687523307082732099155849868121418187476678786351765213291646924740164828743300261443244284017045805526359108709235278950105306233593087058766394554572179135596933967840935323976021555925191622805465772292490011216034950053451001610756113325732102870781425550339597304477530039854833265870929583605733246479944239379864634999053827420237289087458176767592603875320470692775739284765593366878240641258088657910215163904604130420019954885077413963767503253855341966451537790111763479186103097071601857592938755969799410205040151415410793474190454305304887097766773210186375400521869176814703638044658793598387263216
sage: d<r
True
sage: e*d%r
1
sage: C=power_mod(msg,e,n)
sage: C                          # ciphertext
957142867647338580460566956928273633705270194837679765022659963378039163210
sage: DC=power_mod(C,d,n)        # decrypted ciphertext
sage: DC
6321
sage: DC_bin=bin(DC)[2:]
sage: while len(DC_bin)%8!=0:
....:     DC_bin='0'+DC_bin
....:
```

```
sage: DC_ascii=[chr(int(DC_bin[x*8:8*(x+1)],2)) for x in range(len(▸
 ▸DC_bin)/8)]
sage: StringDC=''.join(DC_ascii)
sage: StringDC
'6, 66, the number of the beast!'
```

6.4 Factoring

And by the way, factoring r, a 1024 bit number, could take years, depending on r ;-)

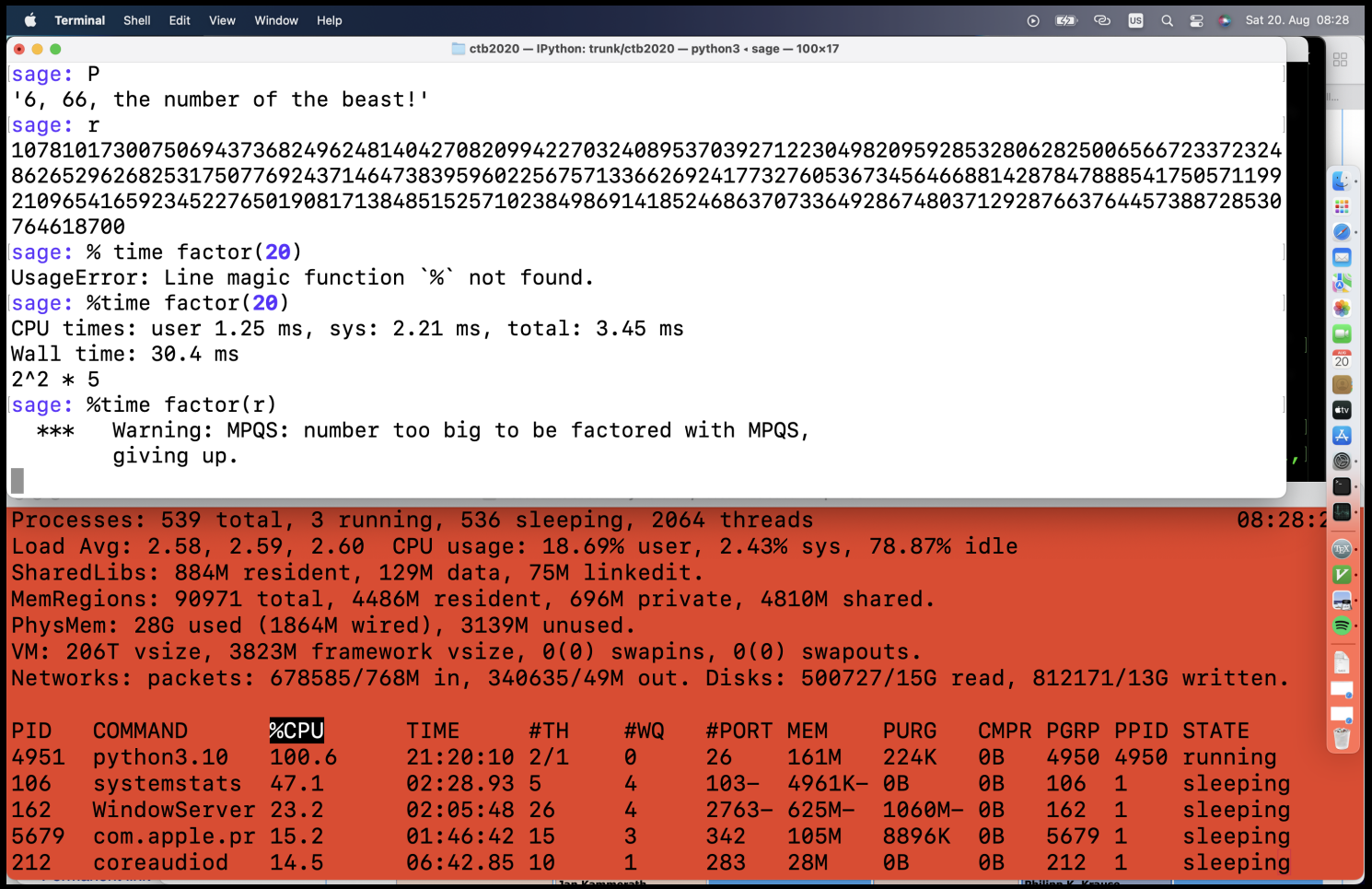
4.12.4 Status regarding factorization of concrete large numbers

An exhaustive overview about the factoring records of composed integers using different methods can be found on the following web pages:

- http://primerecords.dk/consecutive_factorizations.htm
- https://en.wikipedia.org/wiki/Integer_factorization_records
- https://en.wikipedia.org/wiki/RSA_Factoring_Challenge

Screenshot from the CrypTool Book.

Of course, I tried it anyway:



Unfortunately I did not log the exact time till it stopped, but before stopping, my MBP tried hard for about half a day.

But 330 bit is possible on a FroSCon afternoon on my MBP:

```
sage: rsa100
152260502792253336053561837813263742971806811496138068865790849458012296325895289765400035069200613
sage: rsa100.factor(algorithm='qsieve')
379752279369436739228088727554456278545655366381994009469095092088103068373529276146838921489972406
```

It took more than 8 hours, but less than a day ;-)

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro18,2
Chip:	Apple M1 Max
Total Number of Cores:	10 (8 performance and 2 efficiency)
Memory:	32 GB