Linnæus University
Faculty of Technology
Per-Anders Svensson
Ana Strikić

Description of the package `Lab19`

Introduction

As a part of the examination of the course is a project, where the computer program *Mathematica* is to be used to solve a number of exercises in cryptography as well as coding theory. For this purpose, a *Mathematica* package, `Lab19`, has been developed. This package contains a number of functions specially designed for solving the exercises. The package is loosely based on the notebook constructed by W. Trappe and L. C. Washington in their book *Introduction to Cryptography with Coding Theory*, Pearson Education, 2006.

Getting started

Below is a guide in how to load the package `Lab19` into *Mathematica*.

1. Go to the course room on MyMoodle and download the file `Template.nb` to you computer. This file is a so-called *notebook*. This is where you can execute commands and perform calculations.

2. Double click on `Template.nb` to open it in *Mathematica*. After a few seconds a window, similar to the one pictured in Figure 1 on the following page, should show up.

3. Place the cursor in the cell where you can read

   `Import["http://homepage.lnu.se/staff/psvmsi/Lab19.html","Package"]`

   and press Shift+Enter. The contents of the package `Lab19` will then be loaded into *Mathematica*.

4. Now you are ready to use the *Mathematica* and the package `Lab19`!

5. Besides the notebook `Template.nb`, you may have more then one notebook open at the same time: Select File>New>Notebook(.nb) in the menu or type Ctrl + N.

Remark. Whenever you place the cursor between two cells a horizontal line shows up, equipped with the tiny icon of a plus sign (+) to its left. If you click on this plus sign, a pop-up menu shows up (see Figure 2 on page 3), giving you the opportunity to choose another input format such as Plain text, which is recommended to use for the explaining texts of your solutions in the computer project.

Description of the Package

All predefined commands or functions in *Mathematica* always has a capital initial letter, such as in for instance `Sin[`x`]` for computing $\sin x$ or in `Prime[`n`]` for returning the nth prime

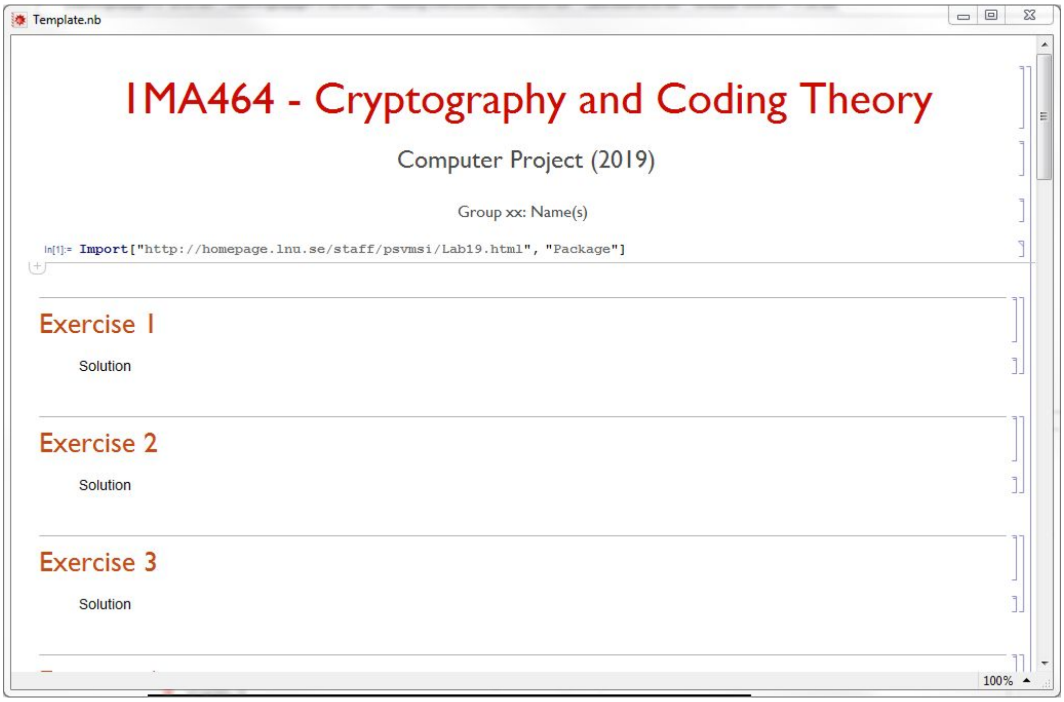

Figure 1: A first view of the template file

number. When it comes to commands in the package `Lab19`, they will always have a small initial letter. ***Mathematica*** is very finical about capital and small letters. Because of this, *always be careful with distinguishing capital and small letters, when typing your commands!* Moreover, when you want ***Mathematica*** to execute your command, finish the line you have been typing by pressing SHIFT+ENTER (or ENTER on the numerical keypad, if available).

List of Commands in the Package `Lab19`

Given below is a short description of each command defined in the package, listed in alphabetic order. It is possible to obtain a clickable list of this commands, by typing

```
?Lab19`*
```

followed by SHIFT+ENTER. This is of course under the assumption that the package has been loaded into ***Mathematica*** according to the instructions above.

`affinecrypt[`*txt*`, `*a*`, `*b*`]` encrypts the text *txt*, by using the affine mapping $E(x) = ax + b \mod 26$.

`allshifts[`*txt*`]` returns a list of all possible shifts modulo 26 of the text *txt*.

`bigrams[`*txt*`]` gives a list of the most common bigrams of the text *txt*.

`carroll` returns a ciphertext (RSA).

`choose[`*txt*`, `*a*`, `*b*`]` picks out the letters on the places $ak + b$, for $k = 0, 1, 2, \ldots$, in the text *txt*.

`clarke[`*n*`]` returns a Vigenère encrypted ciphertext, assigned for laboration group n.

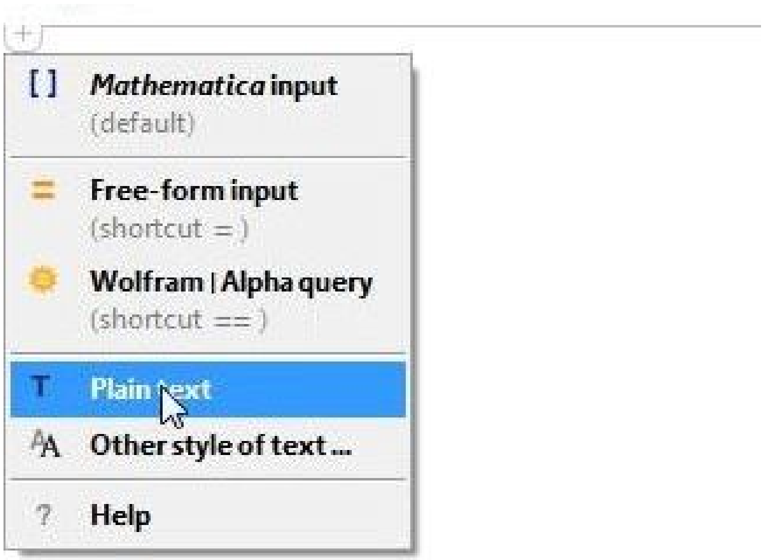

Figure 2: Choosing how to enter input

`coinc[`*txt*`,` *n*`]` returns the numbers of positions in which the text *txt* coincides with itself when translated n steps.

`convert[`*txt*`]` removes blanks, digits, and punctuation marks, and converts capitals to small letters of the text *txt*.

`doubledfreq[`*txt*`]` yields a list of the most frequent digrams in *txt*, in which both of letters are equal.

`frequency[`*txt*`]` performs a frequency analysis of the letters in the text *txt*.

`fromblocks[`$\boldsymbol{v}$`,` *m*`]` converts the vector $\boldsymbol{v}$ to a string. Each element of $\boldsymbol{v}$ corresponds to a block of m letters.

`getaffineciphertext[`*n*`]` returns a ciphertext of laboration group n, encrypted by an affine cipher.

`getelgamalciphertexts[`*n*`]` returns two ciphertexts obtained by ElGamal encryption, specifically for laboration group n.

`getelgamalpublickey[`*n*`]` returns the public key of a ElGamal cryptosystem assigned for laboration group n.

`gethillciphertext[`*n*`]` returns a ciphertext assigned for laboration group n, that has been encrypted by the Hill cipher.

`getinformationbits[`*k*`]` lists all possible binary words of length k.

`getmatrices[`*n*`,` *k*`,` *m*`]` returns a linear $[n, k]$-code assigned for laboration group m, by means of a generating matrix and its corresponding parity check matrix.

`getrsakeys` returns certain keys for an RSA-cryptosystem.

`getrsapublickey[`*m*`]` returns the public key for an RSA-cryptosystem that is assigned for laboration group m.

`haddock` shows an example of a ciphertext (affine cipher).

`ham[`*k*`]` returns a generating matrix and a parity check matrix for a Hamming code with k information bits.

`hammingweight[`*wrd*`]` computes the Hamming weight of the binary word *wrd*, which can be given either as a string or as a vector.

`hill[`*txt*`, `*mat*`]` encrypts the text *txt* using the Hill cipher, with the matrix *mat* as a key. This matrix has to be quadratic.

`randomerror[`*codewordlist*`]` emulates random single bit errors among the words in *codewordlist*.

`sawyer[`*n*`]` yields a ciphertext for an exercise on affine ciphers for laboration group n.

`secretaffine[`*txt*`, `*n*`]` encrypts the text *txt* with the "secret" affine cipher that has been assigned for laboration group n.

`secrethill[`*txt*`, `*n*`]` encrypts the text *txt* with the "secret" Hill cipher that has been assigned for laboration group n.

`secretrsaciphertext[`*n*`, `*e*`]` returns an RSA encrypted ciphertext, where (n, e) has been used as the open key.

`shift[`*txt*`, `*k*`]` encrypts the text *txt* using the shift mapping $E(x) = x + k \mod 26$.

`shiftcipher` returns a ciphertext for an exercise on shift ciphers.

`subciphertext[`*n*`]` returns a ciphertext, assigned for laboration group n, of an ordinary substitution cipher.

`subst[`*txt*`, `*key*`]` encrypts the text *txt* with an ordinary substitution cipher. The key is derived from the string *key*, which must be of length 26 and contains all the letters of the alphabet, in some order.

`text2ascii[`*txt*`]` creates a list of binary vectors of length 7 from the letters of *txt*. Each vector is the binary ASCII code of the corresponding letter in *txt*.;

`toblocks[`*txt*`, `*m*`]` converts the string *txt* to a vector, in which each element corresponds to a block of m letters.

`tptext[`*n*] returns a ciphertext for laboration group n (transposition cipher).

`trigrams[`*txt*`]` Gives a list of the most common trigrams of the text *txt*.

`txt2vec[`*txt*`]` converts the text *txt* to a vector of elements in $\mathbb{Z}_{26}$.

`vec2txt[`$\boldsymbol{v}$`]` converts a vector $\boldsymbol{v}$ of elements in $\mathbb{Z}_{26}$ to a string of letters.

`vec2word[`$\boldsymbol{v}$`]` converts the binary vector $\boldsymbol{v}$ to a binary word, represented as a string.

`vigenere[`*txt*`, `$\boldsymbol{v}$`]` uses the Vigenère cipher to encrypt the text *txt*. The key is decided by the vector $\boldsymbol{v}$.

`vigex` shows an example of a ciphertext (Vigenère cipher).

`word2vec[`*wrd*`]` converts the binary word *wrd* (represented as a string) to a binary vector.

`wrd1, wrd2, wrd3, wrd4, wrd5` Examples of binary words of length 16.

Predefined Commands

Besides the commands defined in `Lab19`, many of the predefined commands of *Mathematica* will of course also be useful. For a detailed description of those commands, one can consult the built-in help function of *Mathematica*, that can be found in the menu Help (choose Documentation Center in this menu), or alternatively, by pressing SHIFT+F1 on the keyboard. You can also type any (predefined) command you want to know more about, and then press F1.

Some Examples

Throughout all examples that follows, it is assumed that the package `Lab19` has been loaded, according to the instructions given in the section *Getting started* above. In other words, after the opening the template file in *Mathematica*, execute

```
In[1]:=  Import["http://homepage.lnu.se/staff/psvmsi/Lab19.html",
           "Package"]
```

followed by SHIFT+ENTER (or ENTER on the numerical keypad). Recall that you should always press SHIFT+ENTER when you want *Mathematica* to execute a command or perform a calculation.[1]

Cryptology

EXAMPLE 1. Let us encrypt the plaintext `caesar`, in the way Julius Caesar himself would have done it. In today's terminology, Caesar used a shift cipher, with $E(x) = x + 3 \mod 26$ as the encryption mapping. Using `Lab19` this encryption can be performed by using the `shift` command:

```
In[2]:=  shift["caesar", 3]

Out[2]=  fdhvdu
```

Note that strings must be surrounded by double prime symbols (").

The decryption mapping to use here is $D(x) = x - 3 \mod 26$, hence the command

```
In[3]:=  shift["fdhvdu", -3]

Out[3]=  caesar
```

restores the plaintext. Note that since $-3 \equiv 23 \pmod{26}$, the command

```
In[4]:=  shift["fdhvdu", 23]

Out[4]=  caesar
```

will restore the plaintext as well. ◊

EXAMPLE 2. In the previous example we encrypted `caesar`, using a shift cipher with $k = 3$ as the key. If we wish to know all possible ciphertexts that can be obtained by encrypting `caesar` using a shift cipher, we can use the `allshifts` command:

[1] You will probably notice that *Mathematica* tries to help you every now and then—when you have typed the first few letters (say three or four) of a command, a menu of commands that begins with the same letters as you have written pops up. From this menu of suggested command you may select the one you wish.

```
In[5]:=  allshifts["caesar"]

         0 - caesar
         1 - dbftbs
         2 - ecguct
         3 - fdhvdu
         4 - geiwev
         5 - hfjxfw
         6 - igkygx
         7 - jhlzhy
         8 - kimaiz
         9 - ljnbja
         10 - mkockb
         11 - nlpdlc
         12 - omqemd
         13 - pnrfne
         14 - qosgof
         15 - rpthpg
         16 - squiqh
         17 - trvjri
         18 - uswksj
         19 - vtxltk
         20 - wuymul
         21 - xvznvm
         22 - ywaown
         23 - zxbpxo
         24 - aycqyp
         25 - bzdrzq
```

We then get a list of all possible shifts modulo 26 of the letters in the plaintext. Using for example the shift $k = 17$ (which means that the encryption mapping will be $E(x) = x + 17 \mod 26$) gives us the ciphertext `trvjri`. ♢

Example 3. If we execute

```
In[6]:=  affinecrypt["affine", 5, 17]

Out[6]=  rqqfel
```

then the affine mapping $E(x) = 5x + 17 \mod 26$ is used to encrypt the plaintext `affine`, using affine cipher. ♢

Example 4. In the package `Lab19` is included an example of a ciphertext that is obtained by means of an affine cipher. To view the ciphertext in all its glory, you simply type

```
In[7]:=  haddock

Out[7]=  orkhtyuknnatkzzcuiywkscreucshnevutkrkuberbtkxbyxbyxty
         wjeenyxowkreyssezykbenalywypneyxtywjkueteywexzcqezqyb
         tkucxwbkxbnautkxoyxokhhekrkxuekxztywryutreherbcyrekwk
         sysyuskiewskxawemgexuewsescrkpne
```

Our objective is to find the plaintext. One way to do this is to make a frequency analysis of the ciphertext. We can then draw conclusions on the nature of the key, based on the fact that frequent (rare) letters of the ciphertext probably will correspond to frequent (rare) letters in ordinary texts, written in English, see Table 2.1 on page 17.

To perform a frequency analysis on the single letters of the ciphertext that is referred to as `haddock`, type

In[8]:= `frequency[haddock]`

Out[8]//TableForm=

a	4
b	9
c	7
d	0
e	24
f	0
g	1
h	5
i	2
j	2
k	22
l	1
m	1
n	8
o	4
p	2
q	2
r	11
s	10
t	11
u	12
v	1
w	12
x	14
y	20
z	6

As we can see, the most common letter in the ciphertext is `e`, just like it is in ordinary English texts. There are two letters that does not occur at all in the ciphertext, namely `d` and `f`. As a first guess, we make the assumption that `e` is mapped to itself by the decryption mapping $D(y) = ay + b$, and that `f` is mapped to `z`. This implies $D(4) = 4$ and $D(5) = 25$ (since `e` $\leftrightarrow$ 4, `f` $\leftrightarrow$ 5, and `z` $\leftrightarrow$ 25). From this we obtain the following system of linear congruence equations:

$$\begin{cases} 4a + b \equiv 4 \pmod{26} \\ 5a + b \equiv 25 \pmod{26}. \end{cases}$$

This system of congruences is easily solved by hand, but we could also be lazy and ask *Mathematica* to do it for us:

In[9]:= `Solve[{4 a + b == 4, 5 a + b == 25}, Modulus -> 26]`

Out[9]= $\{\{a \to 21, b \to 24\}\}$

Thus a possible decryption mapping is $D(y) = 21y + 24$. This is actually an affine mapping since 21 (the coefficient of y) is relatively prime to 26; $\gcd(26, 21) = 1$.

To check if $D(y) = 21y + 24$ really is the correct decryption mapping, we let it operate on the ciphertext by executing

```
In[10]:= affinecrypt[haddock, 21, 24]

Out[10]= graphicallyhaddockisamorecomplexcharacterthantintinhi
         sfeelingsareimmediatelyvisibleinhisfaceheisendowedwit
         haconstantlychangingappearanceandhisrichrepertoireasa
         mimicmakesmanysequencesmemorable
```

Blistering barnacles and thundering typhoons! Our guess was correct![2]

Of course, one should not always expect to obtain the plaintext on the very first try, as we did in this example. Based on the frequency analysis one can make several clever guesses that should be as good candidates as the one we made above. ♢

Example 5. By typing

```
In[11]:= vigex

Out[11]= vptyscgwuzsltktjsukvvpwkywuvpuhqgzxzvatyzvuihhxicvhse
         kqzqlxngmcalvqciwykqnioijqcgjirplioizpxjaxfvptjlrpvts
         jftmmhqgnmioizpndyqrvqduxycbxzxicvhtmkvmsmvfobwlwfwzr
         lxfvptkijvqchxzqvbhctqvhpwkqnpuecqohpkecthdlznmioitji
         cuicoinlbggkiasigktpzvfqvpxrnqcwykkvhbgyckpzirpichpfi
         bdkmxkbpsgfpdtywzqvlppcdmglulkztkekvptzxrimpuhkjmchfr
         esrvrmgzhhxzqvxzvvscxyiucbioiugkdkmeiaihkvumrvrungioi
         jqcgjivpkdkiioinjsdrztzwkjmdbxgwbdmxygadbvtgndyxygxjy
         tfumdmitqvdtmqkvvvrkjmalrxvpdmxygbghrjoqhzmfpiialvqbw
         lvvplioijqcgjiugkdkiifmrvqgtmhzxygztjizxmszmxpiavvjgy
         jlrtgadtirrxapgrvqduwigyjpvvvppaxygltjsugzglwkqztalvf
         iihwfvppamkkaxkievqrhpkqbwlsikochpzpewpgyeihlavuinalr
         vbwlgfoxglwjkwcpwcqahsijuwioiicxesmtcbxvrjucroejowhae
         lfqdhrukupnikticzqzuaxvrjctavajqutjsevzdspvflxmjvtmcj
         iftlxzxftbxvrsgbllievptvvziqchprplioiigaivvvflpaerpli
         omjhttemsktxaczummwpfkbtkxfckwpimgpxnlvtkdttigahpsevp
         tjsdrztzwzqvxzxygvrhpcglavwja
```

we obtain a ciphertext obtained by a Vigenère cipher. (Actually the plaintext is about coding theory, and is picked from Ron M. Roth, *Introduction to Coding Theory*, Cambridge University Press, 2006).

To break the cipher, we first want to find a likely value of its period p. One way to do this is to search for letter sequences in the ciphertext that is repeated every now and then. Such a sequence of letters could be due to that a certain letter sequence in the plaintext has been encrypted several times by the same part of the key of the cipher. If we execute

```
In[12]:= trigrams[vigex]
```

[2] The text is taken from Benoît Peeters, *Tintin and the World of Hergé. An Illustrated History.* Carlsen Copenhagen, 1988, and describes the comic character Captain Haddock.

Out[12]//TableForm=

ioi	10
xyg	6
vpt	6
zqv	4
tjs	4
pli	4
lio	4
bwl	4
xzx	3
xvr	3
vvv	3
vvp	3
rvq	3
rpl	3
qcg	3
oij	3
kjm	3
kdk	3
jqc	3
ijq	3

we obtain a list of the most common trigrams of the text `vigex`. To find out where the most common sequence `ioi` shows up in the ciphertext, we can execute

In[13]:= `StringPosition[vigex, "ioi"]`

Out[13]= $\{\{75, 77\}, \{87, 89\}, \{117, 119\}, \{207, 209\}, \{345, 347\}, \{369, 371\}, \{384, 386\}, \{483, 485\}, \{663, 665\}, \{777, 779\}\}$

which yields a list of all positions in the ciphertext, in which the trigram `ioi` begins and ends. We can see that the ten occurrences of `ioi` start at the positions 75, 87, 117, 207, 345, 369, 384, 483, 663, and 777. If we look closely at these numbers, we see that they all are divisible by 3, which we may confirm by running the command

In[14]:= `Mod[{75, 87, 117, 207, 345, 369, 384, 483, 663, 777}, 3]`

Out[14]= $\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$

In general, `Mod[`L`, `n`]`, where L is a list of integers and n is an integer, will compute the remainders modulo n of all the elements in L.

Since all remainders are the same above, the period might be 3. But we would have reached the same result also if the period is a multiple of 3. Let us therefore also check the possible remainders modulo $2 \cdot 3 = 6$:

In[15]:= `Mod[{75, 87, 117, 207, 345, 369, 384, 483, 663, 777}, 6]`

Out[15]= $\{3, 3, 3, 3, 3, 3, 0, 3, 3, 3\}$

We obtain the remainder 3 in almost all cases (the only exception is the seventh element 384). Thus if we take any two elements a and b in the list (except for 384), then their difference $a - b$ will be divisible by 6. This implies that the period may in fact be 6 instead av 3 (if the period was 3, then probably the list above would have contained about the same number of 0's and 3's). Hence $p = 6$ could be a better guess than $p = 3$. But by the same argument as above, it might as well be a multiple of 6; let us check the remainders when dividing by $2 \cdot 6 = 12$:

```
In[16]:= Mod[{75, 87, 117, 207, 345, 369, 384, 483, 663, 777}, 12]
```

Out[16]= $\{3, 3, 9, 3, 9, 9, 0, 3, 3, 9\}$

Here we obtain a list where the possible remainders 0, 3, and 9 are more equally distributed. Therefore it is not so likely that the period is 12. It seems like if $p = 6$ is a much more better guess.[3]

Another method for finding the period p, is to count the number of coincidences, when the ciphertext is compared, letter by letter, with a copy of itself that is displaced by a certain number of places. By typing

```
In[17]:= coinc[vigex, 2]
```

Out[17]= 42

we find that there are 42 pairs of matching letters in the ciphertext, if the displacement is 2 (i.e., if the first letter of the ciphertext is compared with the third, the second letter with the fourth, the third with the fifth, and so on).

The number of matching letters when the displacement is 3 is calculated in the same manner by

```
In[18]:= coinc[vigex, 3]
```

Out[18]= 33

A list of all coincidences for all possible displacements between 2 and 9 letters (say), is obtained by executing

```
In[19]:= Table[coinc[vigex, i], {i, 2, 9}]
```

Out[19]= $\{42, 33, 46, 40, 81, 42, 36, 25\}$

This commands creates a list, whose elements from the left to the right are the result of `coinc[vigex, i]` when i runs through the integers $2, 3, \ldots, 9$. The fifth element 81 in the list thus corresponds to `coinc[vigex, 6]`, and we see that this element is essentially larger then all the others. This is yet another argument for that the period is $p = 6$.

So let us assume that $p = 6$ from now on. This means that every sixth letter of the plaintext have been encrypted in the same way. Especially, this is the case for the letters on the places $1, 7, 13, 19, \ldots$. By executing the command

```
In[20]:= s1 = choose[vigex, 6, 1];
```

we pick exactly the letters on the positions $6k + 1$ in the ciphertext, where $k = 0, 1, 2, \ldots$. To be able to refer to this string later on, we have labeled it `s1`. The semicolon (`;`) prevents *Mathematica* from printing the string `s1` on the screen. We are not really that very interested in how the string looks like; we are more interested in how often different letters occurs in it. Therefore we make a frequency analysis of `s1`:

```
In[20]:= frequency[s1]
```

[3] To be even more sure, repeat these investigations for the two next to the most common trigrams `xyg` and `vpt` on your own!

Out[20]//TableForm=

a	1
b	0
c	10
d	1
e	2
f	6
g	23
h	2
i	4
j	4
k	11
l	0
m	0
n	4
o	6
p	13
q	19
r	3
s	1
t	8
u	8
v	17
w	2
x	1
y	1
z	0

By the Vigenère encryption algorithm, we know that `s1` is the result of a shift cipher applied on a plaintext, that have about the same letter frequencies as in ordinary English texts. Therefore the letter `g` in the frequency table is of interest, since it could correspond to the frequent letter `e` of ordinary texts. This guess will also match the frequency of other letters in the plaintext with the frequency of letters in English quite well.

To map `e` onto `g` with a shift cipher, we should use the mapping $E_1(x) = x + 2$ (since `e` $\leftrightarrow 4$, `g` $\leftrightarrow 6$, and $E_1(4) \equiv 4 + 2 \equiv 6 \pmod{26}$).

We will now investigate the letters on the positions $6k + 2$, $6k + 3$, $6k + 4$, $6k + 5$, and $6k + 6$, respectively, in a similar way. As an exercise, conclude that $E_2(x) = x + 8$, $E_3(x) = x + 15$, $E_4(x) = x + 7$, $E_5(x) = x + 4$, and $E_6(x) = x + 17$ are likely to be the five remaining shift mappings. Together with $E_1(x) = x + 2$ these six mappings yield the vector $(2, 8, 15, 7, 4, 17)$ of elements in $\mathbb{Z}_{26}$. The corresponding keyword is found by the command

In[21]:= `vec2txt[{2, 8, 15, 7, 4, 17}]`

Out[21]= `cipher`

If a certain vector $\boldsymbol{v}$ of elements in $\mathbb{Z}_{26}$ is used to encrypt a plaintext, the vector we should use for decryption is $-\boldsymbol{v} \mod 26$. This means that we in this particular will restore the message if we execute the command

In[22]:= `vigenere[vigex, -{2, 8, 15, 7, 4, 17}]`

Out[22]=
```
theroleofsourcecodingistwofoldfirstitservesasatransla
torbetweentheoutputofthesourceandtheinputtothechannel
```

```
forexampletheinformationthatistransmittedfromthesourc
etothedestinationmayconsistofanalogsignalswhilethecha
nnelmayexpecttoreceivedigitalinputinsuchacaseananalog
todigitalconversionwillberequiredatthestageandthenaba
ckconversationisrequiredatthedecodingstagesecondlythe
sourceencodermaycompresstheoutputofthesourceforthepur
poseofeconomizingonthelengthofthetransmissionattheoth
erendthesourcedecoderdecompressthereceivedsignalorseq
uencesomeapplicationsrequirethatthedecoderrestorethed
atasothatitisidenticaltotheorignalinwhichcasewesaytha
tthecompressionislosslessotherapplicationssuchasmosta
udioandimagetransmissionsallowsomecontrolleddifferenc
eordistortionbetweentheoriginalandtherestoreddataandt
hisflexibilityisexploitedtoachievehighercompressionth
ecompressionisthencalledlossy
```

As we can see, the plaintext is restored. ◊

Example 6. There is a command in `Lab19` to encrypt messages by a general substitution cipher. The key is here represented by a string of length 26, in which each letter of the alphabet occurs exactly once. The order in which the letters occur in this string, determines how the letters of the plaintext are substituted, in order to obtain the corresponding ciphertext. If we for instance use

```
In[23]:= key = "zxcvbnmasdfghjklqwertyuiop";
```

as a key, it means that each `a` in the plaintext will be replaced by `z`, each `b` by `x`, any `c` will not be changed, `d` will be changed to `v`, and so on, and finally `z` will be replaced by `p`. To encrypt a certain plaintext using this key, one executes

```
In[24]:= subst["anexampleofasubstitutioncipher", key]

Out[24]= zjbizhlgbknzetxersrskjcslabw
```

We can decrypt a ciphertext in the same way, once we know how the decryption key looks like. ◊

Example 7. We are about to encrypt `characters` by a Hill cipher using the matrix

$$A = \begin{pmatrix} 4 & 11 & 2 \\ 1 & 12 & 23 \\ 2 & 1 & 9 \end{pmatrix}$$

as the key. Since A is a 3×3-matrix, the text `characters` has to be divided into blocks of three letters. But `characters` is ten letters long, so this is not possible; we have to add two "junk letters" to the end of the text. The command `hill` in the package `Lab19` does this automatically (using `x` as a junk letter). The only thing we need to to is do input the matrix into *Mathematica*, which we do by typing

```
In[25]:= A = {{4, 11, 2}, {1, 12, 23}, {2, 1, 9}}
```

Out[25]= $\{\{4, 11, 2\}, \{1, 12, 23\}, \{2, 1, 9\}\}$

To be on the safe side we confirm that A actually can be used as a key for a Hill cipher, which means that we must check if $\gcd(\det A, 26) = 1$, i.e., that the determinant of A and 26 are relatively prime. In *Mathematica* this is verified by the command

```
In[26]:= GCD[Det[A], 26]
```

```
Out[26]= 1
```

The command `Det` computes the determinant of a (quadratic) matrix, and `GCD` computes the greatest common divisor of two (or more) integers.

We see that it is possible to use A as the key. Encryption is done by

```
In[27]:= hill["characters", A]
```

```
Out[27]= hilulayqnhrg
```

To decrypt a ciphertext, we need to know the inverse of A modulo 26, since this matrix is the decryption key. The matrix inverse is computed in *Mathematica* as

```
In[28]:= B = Inverse[A, Modulus -> 26]
```

```
Out[28]= {{19,19,5},{15,20,12},{23,8,15}}
```

Encryption is the done by

```
In[29]:= hill["hilulayqnhrg", B]
```

```
Out[29]= charactersxx
```

and as we can see, the plaintext is restored, if we disregard the two junk letters at end of the word.

The notion of a matrices in *Mathematica* could be a little bit crabbed to read, especially if the matrices have many rows and columns. A more readable representation is obtained by the `MatrixForm` command. For instance

```
In[30]:= MatrixForm[B]
```

Out[30]//MatrixForm=

$$\begin{pmatrix} 19 & 19 & 5 \\ 15 & 20 & 12 \\ 23 & 8 & 15 \end{pmatrix}$$

presents the matrix B above in a more acquainted form.[4] ◊

Example 8. Suppose the public key of an RSA cryptosystem is $(n, e) = (4031, 415)$, where $n = 29 \cdot 139$ is the product of the two primes 29 and 139. Let us encrypt the message `theweatherisnice`. We then need to split the plaintext into block of a suitable length m. Since $n = 4031$ in our case, we choose $m = 2$ (since the way we have agreed to encode blocks of letters within this course will never produce a number exceeding n, by this choice of m).

The command `toblocks[`*txt*`, `*m*`]` converts a plaintext to a vector with integer elements, where each such element corresponds to a block of m letters. For this example, we execute

```
In[31]:= plaintext = toblocks["theweatherisnice", 2]
```

[4]Remark: You should not use `MatrixForm` in calculations! For instance, if you want to compute the product of two matrices A and B you write `A.B` in *Mathematica*. Executing `MatrixForm[A].MatrixForm[B]` will just get *Mathematica* confused.

Out[31]= $\{2008, 523, 501, 2008, 518, 919, 1409, 305\}$

Here 2008 corresponds to the text block `th`, 523 corresponds to `ew`, and so on. If needed, `toblocks` adds junk letters to end of the plaintext, in order to make the all blocks of the same size.

A command for RSA encryption is not implemented in `Lab19`, so we need to define it on our own. We do this by declaring

```
In[32]:= rsa[x_, e_, n_] := PowerMod[x, e, n]
```

Note that we in the left-hand side are using underscores after the variable names. The reason for this is to tell *Mathematica* that the variables `x`, `e`, and `n` should be treated symbolically, which means that they can be replaced by other variables or expressions, when the command `rsa` is executed. It also means that if there are any previously assigned values to any of these variables, then they will be ignored in the definition of `rsa`. The above assignment means that `rsa[`x`, `e`, `n`]` will compute $x^e \mod n$ whenever it is executed.

Now we can use `rsa` in the same way as any predefined command. For instance, encrypting the first block of letters in the vector `plaintext`, that we obtained from the `toblocks` command above, we write

```
In[33]:= rsa[2008, 415, 4031]
```

Out[33]= 1035

Hence $2008^{415} \equiv 1035 \pmod{4031}$.

It would be convenient if we could apply `rsa` on every element of the vector `plaintext` at the same time, instead of one by one. The predefined command `Map[`*fkn*`, `*list*`]` can help us to achieve this. This command returns a list in which the function *fkn* has been applied on every element in *list* (which is... well, a list). For example

`Map[f, {a, b, c}]`,

will apply the function `f` on each one of the elements in the list `{a, b, c}`, yielding the result

`{f[a], f[b], f[c]}`.

The function `rsa[`x`, `e`, `n`]` that we have defined above, is however a little bit more complicated, since it takes the value of three variables (x, e, and n) as input. We can however "freeze" the variables e and n to 415 and 4031 respectively, by writing

`rsa[#, 415, 4031]&`

Mathematica will interpret this as a function of one single variable, represented by the symbol `#`. Thus the whole ciphertext will obtained by

```
In[34]:= ciphertext = Map[rsa[#, 415, 4031]&, plaintext]
```

Out[34]= $\{1035, 523, 362, 1035, 2742, 2865, 853, 583\}$

To decrypt, we must use d in the secret key. Since d is the multiplicative inverse of e modulo $(p-1)(q-1)$ (where p and q are the primes such that $n = pq$), we have

```
In[35]:= d = PowerMod[415, -1, (29 - 1)(139 - 1)]
```

Out[35]= 3175

Hence the private key is the triple $(p, q, d) = (29, 139, 3175)$. Decryption is now done in the same manner as encryption:

```
In[36]:= msg = Map[rsa[#, d, 4031]&, ciphertext]
```

Out[36]= $\{2008, 523, 501, 2008, 518, 919, 1409, 305\}$

The command `fromblocks[`$\boldsymbol{v}$`, `m`]`, converts a vector $\boldsymbol{v}$ in which every element corresponds to a block of m letters, to a readable text. Applying this in our example (recall that $m = 2$), yields

```
In[37]:= fromblocks[msg, 2]

Out[37]= theweatherisnice
```

It may happen that junk letters occurs at the end of the message, although it did not happen in this example. ♢

Coding Theory

Example 9. Let us define a generating matrix for a systematic linear $[6, 3]$-code.

```
In[38]:= G = {{1, 0, 0, 1, 1, 0},
              {0, 1, 0, 0, 1, 1},
              {0, 0, 1, 1, 0, 1}};
            MatrixForm[G]
```

Out[39]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

To encode `011` $\in \mathbb{B}^3$ to a codeword in $\mathbb{B}^6$ we write

```
In[40]:= codeword = Mod[{0, 1, 1}.G, 2]
```

Out[40]= $\{0, 1, 1, 1, 1, 0\}$

In *Mathematica*, multiplication of matrices is denoted by a dot (`.`). The command `Mod` is used to ensure that the matrix multiplication is performed modulo 2.

Suppose now that an error occurs, so that the third bit of `codeword` is changed to a zero:

```
In[41]:= codeword[[3]] = 0;
         codeword
```

Out[41]= $\{0, 1, 0, 1, 1, 0\}$

We check, by using the parity check matrix of this code, if this error can be detected. The parity check matrix for this code is given by

```
In[42]:= H = {{1, 1, 0}, {0, 1, 1}, {1, 0, 1},
              {1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
         MatrixForm[H]
```

Out[43]//MatrixForm=

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

To check the modified codeword we execute

```
In[44]:= Mod[codeword.H, 2]
```

```
Out[44]= {1,0,1}
```

and find that the syndrome does not equal the zero vector. Therefore the word we checked cannot be a codeword. Note that the syndrome coincides with the third row of the parity checked matrix H, and that it was exactly the third bit of the original codeword that was changed. $\Diamond$

Example 10. We want to find all codewords that is generated by the matrix G from the previous example. Since it is a $[6, 3]$-code, one way to do this is to multiply each word in $\mathbb{B}^3$ by G. The result will then be those words in $\mathbb{B}^6$ that belongs to the code.

The command `getinformationbits[`k`]` yields a list of all words in $\mathbb{B}^k$. For $k = 3$ we obtain

```
In[45]:= words = getinformationbits[3]
```

```
Out[45]= {{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1},
          {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

By using the `Map` command, we can let G act on every element in this list:

```
In[46]:= allcodewords = Map[Mod[#.G, 2]&, words]
```

```
Out[46]= {{0, 0, 0, 0, 0, 0}, {0, 0, 1, 1, 0, 1},
          {0, 1, 0, 0, 1, 1}, {0, 1, 1, 1, 1, 0},
          {1, 0, 0, 1, 1, 0}, {1, 0, 1, 0, 1, 1},
          {1, 1, 0, 1, 0, 1}, {1, 1, 1, 0, 0, 0}}
```

resulting in the set of all codewords.

There is a predefined command in *Mathematica* for computing Hamming distances between two words. For instance, the Hamming distance between the second and the fifth codeword of `allcodewords` is obtained by typing

```
In[47]:= HammingDistance[allcodewords[[2]], allcodewords[[5]]]
```

```
Out[47]= 4
```

The function `HammingDistance` also takes strings as input. To demonstrate this, let us convert all the binary vectors of `allcodewords` to strings using the command `vec2word` along with the `Map` command as follows.

```
In[48]:= ko = Map[vec2word, allcodewords]
```

```
Out[48]= {000000, 001101, 010011, 011110,
          100110, 101011, 110101, 111000}
```

We compute the Hamming distance between the third and the sixth codeword:

```
In[49]:= HammingDistance[ko[[3]], ko[[6]]]
```

```
Out[49]= 3
```

In `Lab19` a command for computing Hamming weights is defined. If we write

```
In[50]:= weights = Table[hammingweight[ko[[i]]], {i, 2, 8}]
```

Out[50]= $\{3, 3, 4, 3, 4, 4, 3\}$

we obtain a list of all Hamming weights of the codewords `ko[[i]]`, where i runs through the integers $2, 3, \ldots, 8$. In other words, we calculate the Hamming weights of all non-zero codewords. We see that the minimal Hamming weight is 3. This can also be confirmed in *Mathematica* if we write

In[51]:= `Min[weights]`

Out[51]= 3

since the command `Min[`*list*`]` returns the smallest element in *list*. Therefore we conclude that the code can correct all single-bit errors and detect all double-bit errors.

The command `hammingweight` can also take a vector as input; the command

In[52]:= `hammingweight[{0, 0, 1, 1, 0}]`

Out[52]= 2

demonstrates this. $\Diamond$

Example 11. In this final example we will demonstrate some commands in *Mathematica* that may be useful when constructing a cyclic code generated by a polynomial.

We start by defining a polynomial:

In[53]:=
```
Clear[x];
pol = x^10 + x^9 + x^7 + x^5 + x^4 + x^2 + x + 1;
```

The command `Clear[x]` is not necessary, but we use it here to be sure that all earlier assignments of `x`, if any, are deleted.[5]

A polynomial can be factored modulo 2:

In[54]:= `factors = FactorList[pol, Modulus -> 2]`

Out[54]= $\{\{1, 1\}, \{1 + x, 2\}, \{1 + x + x^2, 1\}, \{1 + x + x^3, 2\}\}$

We get as a result a list in which each element is in itself a list. Each one of these lists has two elements; the first element being a factor of the polynomial `pol`, the second element being the power of this factor, in the factorization of `pol`. The list $\{1, 1\}$ corresponds to the trivial constant factor $1^1 = 1$. If we write the output above, using ordinary mathematical notations, we have

$$1 + x + x^2 + x^4 + x^5 + x^7 + x^9 + x^{10} = (1 + x)^2(1 + x + x^2)(1 + x + x^3)^2$$

We refer to the third element in the list of factors like this:

In[55]:= `factors[[3]]`

Out[55]= $\{1 + x + x^2, 1\}$

The factor $1 + x + x^2$ itself is obtained by executing

In[56]:= `factors[[3, 1]]`

[5]Otherwise, if `x` has some kind of value, say 2 for instance, due to some previous calculations, then *Mathematica* would not regard `x` an indeterminate (which is our intention). Instead it would assign the value $2^{10} + 2^9 + 2^7 + 2^5 + 2^4 + 2^2 + 2 + 1 = 1719$ to `pol`.

Out[56]= $1 + x + x^2$

We can also multiply polynomials modulo 2 in *Mathematica*. Suppose, for instance, that we would like to compute

$$f(x) = (1 + x)(1 + x + x^2)(1 + x + x^3)^2$$

modulo 2. Note that all the factors in the right-hand side are factors of `pol`, so one way to compute the product $f(x)$ would be to write

In[57]:=
```
f = PolynomialMod[factors[[2, 1]] *
        factors[[3, 1]] * factors[[4, 1]]^2, 2]
```

Out[57]= $1 + x^2 + x^3 + x^5 + x^6 + x^9$

Here `PolynomialMod[`*poly*`, 2]` reduces all the coefficients of the polynomial *poly* modulo 2.

When constructing generating matrices of cyclic codes, the coefficients of the generating polynomials are essential. To obtain a list of all the coefficients of the polynomial f above, with respect to the variable x, we type

In[58]:=
```
coeff = CoefficientList[f, x]
```

Out[58]= $\{1, 0, 1, 1, 0, 1, 1, 0, 0, 1\}$

The element in the list is sorted in such a way, that the first element is the constant coefficient of f, while the last element is the leading coefficient.

It is possible to reverse a list:

In[59]:=
```
ffeoc = Reverse[coeff]
```

Out[59]= {1, 0, 0, 1, 1, 0, 1, 1, 0, 1}

and to count the number of elements in a list:

In[60]:=
```
Length[ffeoc]
```

Out[60]= 10

We can also make an existing list longer by padding extra zeros to it from the right. By executing

In[61]:=
```
zeroffeoc = PadRight[ffeoc, 15]
```

Out[61]= $\{1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0\}$

we pad zeros from the right to obtain a list of 15 elements, instead of 10.

The elements in a list can be shifted a certain number of steps in one or the other direction. For instance, a shift to the right with 3 steps of the list `zeroffeoc` is the result of executing

In[62]:=
```
RotateRight[zeroffeoc, 3]
```

Out[62]= $\{0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0\}$

The commands we have demonstrated above can be used to construct a 4×15 matrix, in which the ith row is the list `zeroffeoc` shifted i steps to the right:

In[63]:=
```
matrix = Table[RotateRight[zeroffeoc, i], {i, 4}];
MatrixForm[matrix]
```

Out[64]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Instead of `{i, 4}` we may as well write `{i, 1, 4}` in the `Table` command above, since they both mean that the variable `i` should run through the integers 1, 2, 3, 4. If we change `{i, 4}` to `{i, 0, 4}`, then `i` runs through the integers 0, 1, 2, 3, 4, and we obtain a 5×15 matrix instead. $\diamondsuit$