Linnæus University
Faculty of Technology
Per-Anders Svensson
Björn Lindenberg

Computer Project in Cryptography and Coding Theory

2021

Guidelines - Read Carefully!

This computer project is divided into three categories: classical ciphers, modern ciphers, and coding theory. The exercises of the project are to be solved by using the *Mathematica* package `Lab21`, that is described in the document *Description of the package `Lab21`*. The general outlines for the computer project is as follows:

- Team-work is allowed, with a maximum of two (2) persons in each group. You can sign up for group on MyMoodle.
- To each laboration group a unique group number will be assigned. ***You have to recall this number, since some of the exercises cannot be solved without it!***
- The laboration report should be saved as a *Mathematica* notebook file. Use the template file available in the Computer Project folder on MyMoodle, but give it another name, for example `Group3.nb` (if 3 is your group number). Send the complete report via the hand-in tool in MyMoodle.
- The notebook shall contain the different commands that have been executed and programs that have been written to solve each one of the exercises, along with ***careful motivations for the solutions***. Those motivations should describe how you have reasoned to solve the exercises.
- The grade of the computer project is either U (Fail; "Underkänd") or G (Pass; "Godkänd"). For the group to obtain the grade G on the project, at least 75 % of the exercises on the project must be solved correctly.
- Questions can be asked in the discussion forum Ask questions about the computer project on MyMoodle, or by using the Slack channel #1ma464-crypthography-and-coding-theory.
- The last date to hand in the report is **June 7, 2021**. Then the last date for doing supplementary corrections, if needed, is **June 17, 2021**.
- Deferment of sending in the report could be allowed in exceptional cases, but it is then required that you request for this, *well before the deadline*. Reports that are handed in too late, without any request for deferment, will ***not*** be corrected until August.

- A second chance to hand in the report is before **August 23, 2021**, with a deadline for supplementary corrections on **August 30, 2021**. Reports that are handed after this date will *not be corrected at all*, unless a request for deferment has been made, well before the deadline.
- And finally: Do not wait too long starting with the computer project. The tasks of the project cover more or less the whole course. Try to keep about the same pace as lectures of the course.

Good luck!

Principles

If the plaintext is written in English, we will encode the 26 letters of the English alphabet with the elements in $\mathbb{Z}_{26} = \{0, 1, 2, \ldots, 25\}$ according to the following table:

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12
n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

We do not distinguish between capitals and small letters, so for example `s` and `S` will both be encoded as 18. All spaces and punctuation marks, such as points, commas, colons, semicolons, exclamation marks, question marks, etc., will be excluded before encryption. This also holds for any digit. The package `Lab21` contains a command `convert` that will do this work for you

As an example, executing

```
convert["This message contains 9 words, and it ends here!"]
```

will yield

```
thismessagecontainswordsanditendshere
```

as the output.

In some cryptosystems (typically RSA and ElGamal), the plaintext is divided into blocks consisting of n letters each. If the numbers of letters in the plaintext is not a multiple of n, some so-called "junk letters" are added to the last block, so that each block will contain exactly n letters. Then each letter is encoded according to the scheme

$$\texttt{a} \leftrightarrow 01,\ \texttt{b} \leftrightarrow 02,\ \ldots,\ \texttt{y} \leftrightarrow 25,\ \texttt{z} \leftrightarrow 26.$$

For example, to divide the plaintext `cryptology` into four-letter blocks, we must add some junk letters to the last block. Using `x` as such a junk letter, we obtain

```
cryp tolo gyxx
```

We now encode each block, letter by letter, to obtain

$$3182416,\ 20151215,\ 7252424.$$

In `Lab21`, there are commands `fromblocks` and `toblocks` for jumping back an forth between the letter blocks and their number representations.

Exercises

Classical ciphers

Exercise 1. If you execute the command

```
shiftcipher
```

you will obtain a ciphertext of a shift cipher. Find the *encryption mapping* that has been used to yield this ciphertext.

Exercise 2. If you execute the command

`sawyer[`*n*`]`

where n is your group number, you will get an excerpt from the famous novel *The Adventures of Tom Sawyer* by Mark Twain (1832–1910) that has been encrypted, using an affine cipher. Find the plaintext, based on a frequency analysis on the ciphertext.

Exercise 3. The package `Lab21` contains a number of secret affine ciphers, one for each laboration group. The purpose of this exercise is that you shall break the cipher belonging to your group. By typing

`secretaffine[`*txt*`, `*n*`]`

where n is your group number, you will encrypt *txt* with the secret affine cipher that has been assigned to your laboration group. (Recall that all strings must be surrounded by double prime symbols (`"`). This means that *Mathematica* for instance will interpret `"text"` as the string "text", but `text` as a variable that has the name *text*.)

(a) Find the affine mapping that has been used for encryption, by making a chosen plaintext attack, i.e., choose your own plaintexts and encrypt them (using the `secretaffine` command), and use the resulting ciphertexts to derive the key.

(b) When you have found the secret encryption mapping, compute the corresponding decryption mapping, and decrypt a certain ciphertext, that has been chosen especially for your laboration group. To obtain the ciphertext of your group, execute

`getaffineciphertext[`*n*`]`

where n is the number of your group.

Exercise 4. The command

`clarke[`*n*`]`

returns a Vigenère encrypted excerpt of the science fiction novel *Rendezvous with Rama*, written by Arthur C. Clarke (1917-2008). Here n denotes the number of your laboration group. Find the period of the cipher, the keyword, and restore the plaintext.

Exercise 5. By executing the command

`subciphertext[`*n*`]`

where n is your group number, you will be given a cipher obtained by a substitution cipher. The objective with this exercise is to restore the plaintext.

To simplify the work, here come some advices and hints:

- Do a frequency analysis of single letters in the ciphertext. Compare the result to the frequency table for the English alphabet, see e.g., Table 1.1 on page 14 in the textbook, or by searching for information on the Internet.
- It could also be fruitful to hunt for two-letter combinations, so called bigrams, in the ciphertext. Frequent bigrams in the ciphertext are likely to correspond to frequent bigrams in English, such as `th`, `he`, `an`, `in`, `er`, `on`. If you type

  ```
  bigrams[txt]
  ```

  you will obtain a list of the most frequent bigrams of the text *txt*, along with their frequencies.
- In some bigrams both letters can be the same. You can perform a frequency analysis on these kinds of bigrams only, by using the command

  ```
  doubledfreq[txt],
  ```

  which will yield a list of the most common doubled letter bigrams in the text *txt*, along with their frequencies. Among the most frequent bigrams of this kind in English we find `ll`, `tt`, `ee`, `ss`, `oo`.
- Three-letter combinations, or trigrams, might also be useful to look for. Among the most common trigrams of English we find `the`, `ing`, `and`, `her`, `ere`. The command

  ```
  trigrams[txt]
  ```

  behaves in the same way as `bigrams` and `doubledfreq`.
- The predefined command `StringReplace` of *Mathematica* can be used to replace one or more letters by other letters in a string. For instance, if we wish to replace each `o` by `T` in the string `byhookorbycrook`, then we execute

  ```
  StringReplace["byhookorbycrook", "o" -> "T"]
  ```

  This operation yields

  ```
  byhTTkTrbycrTTk
  ```

  One may also replace two or more letters at the same time. Say that we in the examples above not only want to replace every `o` by a `T`, but also each `k` by `E`, and each `r` by `A`. Then we run the command

  ```
  StringReplace["byhookorbycrook",
    {"o" -> "T", "k" -> "E", "r" -> "A"}]
  ```

  in order to get

  ```
  byhTTETAbycATTE
  ```

as the result. You can use `StringReplace` to replace certain letters of the ciphertext by the letters that you believe they correspond to in the plaintext. To make it easier to distinguish replaced letters from the letters not yet replaced, use CAPITALS for the replaced letters, just like we have done in the example above, where it easily can be seen that `A`, `E`, and `T` are replaced letters, while `b`, `c`, `h`, and `y` are still the same as in the original string.

- The plaintext is about cryptology, which means that it probably contains certain words that are related to this subject, such as . . . ? ☺

Exercise 6. Uncle Scrooge McDuck wants to earn more money by buying stocks of the company *Crooked Dealings Inc.* Since he consider himself to be short of money, he does not want to buy more than 30 stocks, and therefore he sends the message `buymethirtystocks` to his stockbroker. To prevent his arch-enemy John D. Rockerduck to find out about his investments, he encrypts the message using a Hill cipher, using his favorite matrix

$$A = \begin{pmatrix} 25 & 24 & 25 & 2 & 9 \\ 3 & 5 & 0 & 17 & 2 \\ 5 & 7 & 15 & 8 & 19 \\ 21 & 18 & 19 & 5 & 13 \\ 5 & 3 & 8 & 15 & 19 \end{pmatrix}$$

as the key. Unfortunately an error occurs during the electronic transmission of the ciphertext, causing the 6th letter of the ciphertext to become the letter `h`. The stockbroker decrypts the received message, unaware of this error. How many stocks will the stockbroker buy for uncle Scrooge?

Tip: To conveniently replace a character at a certain position in a string, you can use the *Mathematica* command `StringReplacePart`. If you for instance want to change the nth character of the string *text* to the letter `s`, write

```
StringReplacePart[text, "s", {n, n}]
```

Exercise 7. The package `Lab21` contains a number of secret keys for a Hill cipher—one for each laboration group. Every such key is thus a quadratic matrix of order m, for some $m \geq 2$, with elements in $\mathbb{Z}_{26}$. The purpose of this exercise is to find the key of your group.

(a) If you execute

```
secrethill[txt, n]
```

where n is your group number, you will perform a Hill encryption of the text *txt* with the secret key assigned to your group. Use this command to perform a chosen plaintext attack on the cipher.

(b) When you have found the correct matrix for encryption, find the corresponding decryption matrix and decrypt the certain ciphertext of your group. You can reach this ciphertext by using the command

```
gethillciphertext[n]
```

where n is the number of your laboration group.

Exercise 8. A transposition cipher permutes the letters of the plaintext. Such a cipher can be described as a Hill cipher, if one chooses the encryption matrix to be a so-called *permutation matrix.* A permutation matrix is a matrix in which every row and every column of the matrix contains exactly one 1, while all the remaining elements are 0, such as for example the 4×4 matrix

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

In `Lab21` there is command that returns a ciphertext obtained from a such transposition cipher. Execute

```
tptext[n]
```

where n is your group number, to get the ciphertext for your group. The plaintext is a statement in English, that was first divided into nine-letter blocks. Then each such block was then encrypted by one and the same permutation matrix of type 9×9. Find this matrix!

Modern cryptosystems

Exercise 9. Start by reading Example 8 in the document *Description of the package Lab21*, where it described how RSA can be implemented in *Mathematica.*[1]

Consider an RSA-cryptosystem, where the public key is given by

$$(n, e) = (9\,776\,755\,521\,585\,602\,051,\ 2\,711\,130\,174\,055\,970\,503).$$

The prime factors of n are $p = 8\,219\,357$ and $q = 1\,189\,479\,362\,143$. If you execute the command

```
{p, q, e} = getrsakeys;
```

in *Mathematica*, the variables p, q, and e will assigned the values of these key parameters. Here e is the so-called encryption exponent.

(a) Determine the corresponding decryption exponent d.

(b) The package `Lab21` contains a command `carroll` that will return the ciphertext (represented as a sequence of numbers), obtained when the first few lines of a poem by Lewis Carroll (1832–1898) is encrypted, using RSA with the given public key (n, e) of this exercise. Decrypt it! (When using the command `fromblock` to convert the result of your encryption into text, use 9 as the block size.)

Exercise 10. When constructing a public key for an RSA-cryptosystem, it is not enough to choose the primes p and q to be large. One also have to take properties of the numbers $p-1$ and $q-1$ into consideration: If one in the prime factorizations of

$$p - 1 = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_t^{\alpha_t} \qquad \text{and} \qquad q - 1 = q_1^{\beta_1} q_2^{\beta_2} \dots q_s^{\beta_s}$$

[1] The method used in that example can (after some modifications) also be applied to the situation of Exercise 12, where you will work with the ElGamal cryptosystem.

can find an integer M such that $p_i \leq M$ for all i, but at the same time $q_j \gg M$ (q_j is much larger than M) for at least one j, then a factor of n is likely to be found by Pollard's $p-1$ method, see the lecture notes.

(a) Write a function `pollard[`n`, `B`]` that tries to factor an integer n, using Pollard's $p-1$ method with at most B iterations. You can construct your own functions in *Mathematica* by using the `Module` command. This command is defined as

`Module[{`$x, y, \dots$`}, `*expr*`]`,

where $\{x, y, \dots\}$ is a finite list of *local variables* (i.e., variables that are defined only within the scope of the `Module` command and thereby do not affect any globally defined variables) and where *expr* are the expression that is to be executed within the module. This expression may consist of several different *Mathematica* commands, separated from each other by a semicolon (`;`). A pseudo-code for the `pollard` command may look something like this:

```
pollard[n_, B_] := Module[{$a$, $g$, $i$},
    $a \leftarrow 2$; (* use $a = 2$ as the initial value *)
    $i \leftarrow 2$; (* $i$ is a counter *)
    while $i < B$ and a factor of $n$ has not been found,
        $a \leftarrow a^i \mod n$; (* compute $a^{i!} \mod n$ recursively *)
        $g \leftarrow \gcd(a-1, n)$;
        if $1 < g < n$ then STOP and return $g$ as a factor;
        $i \leftarrow i + 1$ (* increase the counter *)
    end while;
```

Note that we are using three local variables a, g, and i, where a denotes the initial value ($a = 2$), i is a counter, and g computes the greatest common divisor of two integers in each round. Useful *Mathematica* commands to use are

- `PowerMod[`x`, `y`, `m`]` computes $x^y \mod m$
- `GCD[`s`, `t`]` computes $\gcd(s, t)$; the greatest common divisor of s and t
- `While[`*test*`, `*body*`]` executes *body* until *test* fails
- `Return[`*expr*`]` returns the result of the expression *expr*
- `Break[]` can be used to exit a `While`-loop
- `If[`*cond*`, `*expr*`]` executes *expr* if the condition *cond* is true. If c_1 and c_2 are conditions then c_1 `&&` c_2 will be the condition that both c_1 and c_2 are true (hence `&&` is interpreted as the logical AND operator).

Try to factor each one of the three integers $n_1 = 69527$, $n_2 = 864109$, and $n_3 = 655051$ using your `pollard` function. Do you manage to factor all of them? If you fail, no matter how you choose the bound B, what could be the reason for failure?

(b) When executing the command

`{n, e} = getrsapublickey[`m`]`

where m is your group number, (n, e) is assigned to a public key for an RSA cryptosystem. Use your `pollard` function from part (a) to find the prime factors p and q of n. How large must you choose the bound B in order to succeed in finding a factor?

(c) The command

```
secretrsaciphertext[n, e]
```

returns a ciphertext, that is encrypted using RSA and the public key (n, e). Use the same values of n and e as in exercise (b), and find the plaintext, which is a quotation of Albert Einstein (1879-1955).

After finding the numbers that represents the plaintext, convert them to text blocks by applying the `fromblocks` command of `Lab21`. Use 9 as the block size.

Exercise 11. A method for computing discrete logarithms is *Shanks's algorithm* (also known as the *Baby Step, Giant Step Algorithm*). We give a short description of this algorithm below.

Given a prime p, a primitive root g modulo p, and an element $h \in \mathbb{Z}_p^*$, we want to compute the discrete logarithm $x = L_g(h)$, or in other words, to find an x such that $g^x \equiv h \pmod{p}$ and $0 \le x \le p-2$. In Shanks's algorithm, we start by selecting an integer n such that $n^2 > p$. For example $n = \lceil \sqrt{p} \rceil$ will do.[2] According to the Division Algorithm, we may write

$$x = qn + r,$$

for some integers q and r, where $0 \le r < n$. Since $p < n^2$, must have $0 \le q < n$ as well (since $q \ge n$ would imply that $x = qn + r \ge qn \ge n^2 > p$, which does not agree with $0 \le x \le p-2$).

The congruence equation $g^x \equiv h \pmod{p}$ can now be written as

$$g^{qn+r} \equiv h \pmod{p} \iff g^{qn}g^r \equiv h \pmod{p}.$$

Multiplying both sides of this equation by the multiplicative inverse of g^{qn} modulo p, yields

$$g^r \equiv g^{-qn}h \pmod{p}, \tag{1}$$

where we know that $0 \le r < n$ and $0 \le q < n$.

Therefore we may find x, by looking for r and q that fulfill (1) and then compute $x = qn + r$. In Shanks's algorithm we do so by first computing $g^r \bmod p$ for all $r = 0, 1, 2, \ldots, n-1$ and store them in a list B (this is the so-called "baby step list"). Then we also compute the elements $g^{-qn}h \bmod p$ for $q = 0, 1, 2, \ldots, n-1$ and collect them in another list G (the "giant step list"). By the above reasoning, the lists B and G must have an element in common, and from this common element we may find the q and r we are looking for.

Implement Shanks's algorithm in *Mathematica* to solve the discrete logarithm problems

$$13^x \equiv 14 \pmod{993169} \qquad \text{and} \qquad 456789^x \equiv 987654 \pmod{1020431}.$$

[2]By $\lceil a \rceil$ we mean the smallest integer that is larger than or equal to a, so for example $\lceil 3.14 \rceil = 4$, $\lceil \sqrt{17} \rceil = 5$, and $\lceil 7 \rceil = 7$.

Some of the *Mathematica* commands that were mentioned in Exercise 10 should be useful here as well. Other useful commands (depending on how you want to implement the algorithm) might be

- `Ceiling[`x`]` computes $\lceil x \rceil$
- `Sqrt[`x`]` computes $\sqrt{x}$
- `Table[`*expr*`, {`i`, `a`, `b`}]` computes *expr* when i runs from a to b and store all the results in a list
- `For[`*start*`, `*test*`, `*incr*`, `*body*`]` first executes *start*, then repeatedly executes *body* and *incr* until *test* becomes `False`
- `MemberQ[`*list*`, `*element*`]` returns `True` if *element* is a member of *list*, otherwise `False`
- `Intersection[`*list1*`, `*list2*`]` lists all common elements of the two lists *list1* and *list2*
- `Position[`*element*`, `*list*`]` returns a list of positions at which *element* occur in *list*.

Exercise 12. Each laboration group has been assigned a public key of an ElGamal cryptosystem in `Lab21`. It can be reached by entering

```
{p, g, h} = getelgamalpublickey[n]
```

where n is the group number. By doing so, the variables `p`, `g`, and `h` will be assigned the values (p, g, h) of the public key; here p is an eight-digit prime, g is a primitive root modulo p, and h is an element in $\mathbb{Z}_p^*$.

(a) Implement in *Mathematica* a command `elgamalencrypt` that encrypts a message m using the ElGamal algorithm, by returning the ordered pair (r, t), where

$$r = g^k \mod p \qquad \text{and} \qquad t = h^k m \mod p.$$

Here the integer k is to be chosen randomly in the interval $2 \leq k \leq p - 2$.

Hint: Start from the following pseudo-code:

```
elgamalencrypt[m_, p_, g_, h_] :=
  Module[{k, r, t},
    generate the random number $k$ ;
    compute $r$ ;
    compute $t$ ;
    return $(r, t)$
  ]
```

To generate the random number k, you can use the built-in random number generator of *Mathematica*. To do this, you must first initiate the generator by executing the command

```
SeedRandom[];
```

Then, to generate a random integer s such that $x \le s \le y$, you give the command

```
s = RandomInteger[{x, y}]
```

Use your function `elgamalencrypt` to encrypt the plaintext

```
whataboutsecondbreakfast
```

and the command `toblocks` to divide the plaintext into blocks of three letters each, before encryption. The `Map` command can be used to let `elgamalencrypt` act on each one of these blocks simultaneously.

(b) Use your implementation of Shanks's algorithm in Exercise 11 to find the private key a.

(c) Implement a command

```
elgamaldecrypt[{r, t}, p, a]
```

to decrypt a ciphertext (r, t). This means that your command should compute

$$m = tr^{-a} \mod p.$$

The ciphertext is represented by a sequence of ordered pairs

$$c = ((r_1, t_1), (r_2, t_2), \ldots, (r_n, t_n));$$

one pair for each block of the plaintext. To decrypt them all at the same time, use the command `Map` in the same way as before, i.e.,

```
Map[elgamaldecrypt[#, p, a]&, c]
```

Write

```
{c1, c2} = getelgamalciphertexts[n]
```

in order to assign to `c1` and `c2` two different ciphertexts that has been encrypted by the public key of laboration group n. Decrypt them with your `elgamaldecrypt` command, and use `fromblocks` with block size 3 to convert the numbers representing the plaintext into letters.

(d) Answer the question stated in the plaintext that you found in part (c).

Coding theory

Exercise 13. By executing

```
{G, H} = ham[4];
```

in *Mathematica*, you will define G and H as a generating matrix and a parity-check matrix, respectively, for the Hamming code Ham(4). Note that this code is a linear $[15, 11]$-code, since $15 = 2^4 - 1$ and $11 = 2^4 - 4 - 1$).

(a) Use G to encode the following words in $\mathbb{B}^{11}$:

`00101001101`, `11101011101`, and `00100000101`

(b) Check, by using H, if any of the words

`110011100011000`, `010110011001100`, or `101010101010101`

in $\mathbb{B}^{15}$ is a codeword. If a word is not a codeword, correct it to the nearest one, with respect to the Hamming distance.

Exercise 14. To each laboration group a linear $[n,k]$-code is assigned, within the package `Lab21`. A generating matrix G and a parity-check matrix H for the linear $[n,k]$-code of your group, is obtained by writing

```
{G, H} = getmatrices[n, k, m];
```

where m is your group number.

(a) Let C be the the linear $[16,9]$-code that you will obtain when you execute `getmatrices` with your group number. Compute the minimum distance $d(C)$ of your code and use this to determine the error detecting and correcting capabilities of your code.

(b) In `Lab21` there are five words of length 16 defined, and you can reach each one of them by the names given in Table 1. All of the words in that table are defined as vectors in `Lab21`, so you do not need to convert them to vectors (using the `word2vec` command) in order to be able to multiply them with a matrix.

Name	Word
`wrd1`	`1111111111111111`
`wrd2`	`0000000011111111`
`wrd3`	`0000111100001111`
`wrd4`	`0011001100110011`
`wrd5`	`0101010101010101`

Table 1: Shortcuts to some words (for Exercise 14).

Check if any of these five words is a codeword in your code C. If a certain word is not a codeword, can it be corrected to a codeword, or in other words, is the nearest codeword unique?

Exercise 15. In `Lab21` there is a command `text2ascii` that takes as input a string (consisting of letters in the small English alphabet) and returns a list containing the ASCII codes of the letters in the string, expressed as binary vectors of length 7. For example

```
text2ascii["word"]
```

will return the list

$$\{\{1,1,1,0,1,1,1\},\{1,1,0,1,1,1,1\},\{1,1,1,0,0,1,0\},\{1,1,0,0,1,0,0\}\},$$

since the letters of `word` have in turn the ASCII codes 119, 111, 114, and 100, which written binary are 1110111, 1101111, 1110010, and 1100100, respectively. In this exercise we will emulate encoding and error correction of a sequence of such binary words.

(a) Create a linear $[n, 7]$-code C by choosing a generator matrix G of type $7 \times n$ (preferably in standard form), and its corresponding parity-check matrix H of type $n \times (n-7)$. You may choose n in any way you wish, but make sure that the resulting code *is able to correct all single-bit errors*! Also motivate why your code really fulfills this property.

(b) Create a *Mathematica* command that takes as input a message, represented as string of letters in the small English alphabet, converts this string to a list of binary vectors of length 7 (using `text2ascii`), and then encodes each one of these vectors in this list to codewords of length n, using the generator matrix G that you chose in (a). (Thus the output should be a list of codewords of length n; one codeword for each letter in the message.)

(c) Apply the `Lab21` command `randomerrors` on the list of codewords you obtained in (b). This command causes single-bit errors to occur in some of the codewords in the list.[3] Write a *Mathematica* program that takes as input this modified list of words, and corrects[4] all the erroneous words, by using the parity-check matrix H that you chose in (a). The program should return a list of the erroneous words along with the corresponding codewords to which they have been corrected.

Exercise 16. In this exercise you will construct a cyclic $[n, k]$-code. The values of n and k depend on the number of your laboration group, and are picked from Table 2.

Group	n	k	Group	n	k	Group	n	k	Group	n	k
1	36	15	11	49	24	21	28	10	31	56	26
2	44	31	12	51	32	22	30	11	32	39	14
3	54	34	13	57	36	23	35	16	33	69	35
4	72	52	14	65	48	24	45	19	34	78	48
5	84	69	15	98	68	25	66	31	35	30	16
6	99	52	16	93	68	26	70	32	36	91	48
7	21	12	17	63	46	27	73	46	37	89	55
8	23	12	18	60	44	28	90	33	38	80	50
9	31	16	19	42	29	29	75	48	39	62	41
10	39	14	20	40	25	30	77	43	40	50	29

Table 2: Parameters for cyclic codes (Exercise 16).

(a) Use *Mathematica* to find a generating polynomial $g(x)$ and a parity-check polynomial $h(x)$ for the code of yours.

(b) Construct a *Mathematica* command `cyclicmatrices[`*g*`, `*h*`]`, that takes as its input the polynomials $g = g(x)$ and $h = h(x)$ that you chose in exercise (a), and returns the corresponding generating matrix G and parity-check matrix H of the code. Use the `Module` command, according to

```
cyclicmatrices[g_, h_] :=
  Module[{local variables}, commands];
```

[3] At least one of the words will be erroneous.
[4] by syndrome decoding

where `g` and `h` denote the two polynomials $g(x)$ and $h(x)$, respectively.