II Encryption

KTH/EECS:IX1500 - Discrete Mathematics, v1.4
Niharika Gauraha, niharika@kth.se
Notebook by Göran Andersson

1. Implementation and Assessment

The course includes two-three project tasks on a total of 4 hp. The projects are assessed in writing and orally. They will be given a summary grade in scale A-F. The assessment includes the number of solved project tasks and the quality. The first two projects are mandatory for grade C-E. The third project task is optional and targets higher grades A-B.

In this project task you will work in a group of two and solve a mathematical task, write a report (Swedish or English) in *Mathematica* and prepare a short oral presentation on your solution to the task (Swedish or English).

Carefully read the following information so that you know which rules apply and what is expected of you.

1.1 Report

The report should be written in Mathematica and contain

- title and authors of the report
- email@kth.se
- a summary containing the results of resolved parts
- separate sections for each part containing e.g.
  - mathematical formulas and equations
  - a brief discussion
  - explanatory diagrams
  - your conclusions.
- separate code section (do not mix code and text with conclusions and results)

The report will normally be uploaded one-two days prior to the examination (see 1.5 below).

1.2 Oral Presentation

You should prepare an oral presentation of your solution to the task. The presentation will be carried out with your own laptop and *should not take more than five minutes*, effective time. A computer projector will be available at the presentation. Please carefully consider your presentation, what is important, in what order and how it will be illustrated. Practice the presentation in advance and make sure to meet the time frame. One of you in the group (of two) will be selected randomly for the oral presentation and the other part will be selected for questioning.

1.3 Rules

The task is considered *individually* and *assumes that you have full knowledge of all the material you are presenting*. In order to be approved, then you have solved the task and be able to explain the entire task and solution.

To account for the task that you do not have solved is considered cheating. It is also cheating to copy the solution or part of a solution from another. If two solutions are presented as (partially) copies they are rejected both.

If the solution contains parts, e.g. background material, which you do not have produced, you must clearly indicate this and indicate the source.

Suspicion of cheating or misleading can be reported to the Disciplinary Board.

1.4 Examination

- The examination is carried out according to the booking in canvas.
- The booking is done according to information in Canvas Projektuppgifter/Projekt 2.
- Please notice that this project should be reported in **groups of two students**.
- The report (including code section) should be uploaded according to the information in Canvas.
- **Notice that the report shall always be uploaded in time even if you cannot attend at the oral presentation.** In case of illness, etc., then the oral presentation may be made during project 3. If you are unable to attend, you must in advance inform the teacher (in writing).
- If you ignore the oral presentation or the uploading of the report without contacting the teacher (in writing) you will have to wait for the next course for a new project exam.

2. Preparations

2.1 Study

2.2 Guidance

2.2.1 Caesar Cipher (★)

A very famous (and simple) encryption scheme is the Caesar cipher which is built on a cyclic alphabet shift (3 steps). Here's an example using English block letters.

```
ClearAll["`*"]
```

```
shift = 3;
```

```
alphabet = CharacterRange["A", "Z"]
```

```
substitution = RotateRight[alphabet, shift]
```

```
TableForm[{alphabet, substitution}, TableSpacing → {1, 1}]
```

The substitutions for encryption and decryption are:

```
encrypt = Thread[alphabet → substitution]
```

```
decrypt = Thread[substitution → alphabet]
```

Now, the encryption goes:

```
message = "SENDMOREMONEY";

cipher = StringReplace[message, encrypt]
```

The decryption is the other way around.

```
StringReplace[cipher, decrypt]
```

This is of course insecure today and we need more sophisticated methods.

2.2.2 Hill Cipher (★)

A more complicated shift is provided by a shift matrix. Here's an example that is using the 26 English block letters. We introduce a message matrix $\mathbb{M}$, a key matrix $\mathbb{K}$. We compute the operations mod 26. The encryption goes

$$\mathbb{C} \equiv_{26} \mathbb{K} \cdot \mathbb{M}$$

where $\mathbb{C}$ is the cipher matrix. The decryption goes

$$\mathbb{M} \equiv_{26} \mathbb{K}^{-1} \cdot \mathbb{C}$$

The key matrix has to be invertible mod 26. This means that the determinant $|\mathbb{K}| \in \mathbb{Z}_{26}$ has to be invertible, i.e. gcd(|𝕂|, 26) = 1. The the inverse can be computed by

$$\mathbb{K} \cdot \mathbb{x} = \mathbb{1} \cdot \mathbb{y} \Leftrightarrow \mathbb{K}^{-1} \cdot \mathbb{K} \cdot \mathbb{x} = \mathbb{K}^{-1} \cdot \mathbb{1} \cdot \mathbb{y} \Leftrightarrow \mathbb{1} \cdot \mathbb{x} = \mathbb{K}^{-1} \cdot \mathbb{y}$$

$$\therefore (\mathbb{K} \mid \mathbb{1}) \sim \ldots \sim \left(\mathbb{1} \,\middle|\, \mathbb{K}^{-1}\right)$$

where the operations are performed mod 26 (see reduced row echelon form in linear algebra course).

▼ Let's pick a 3×3 key matrix for simplicity.

$$\mathbb{K} = \begin{pmatrix} 19 & 1 & 17 \\ 9 & 8 & 9 \\ 8 & 14 & 1 \end{pmatrix}$$

We can easily find the determinant here.

$$|\mathbb{K}| = \begin{vmatrix} 19 & 1 & 17 \\ 9 & 8 & 9 \\ 8 & 14 & 1 \end{vmatrix} = 19 \begin{vmatrix} 8 & 9 \\ 14 & 1 \end{vmatrix} - 1 \begin{vmatrix} 9 & 9 \\ 8 & 1 \end{vmatrix} + 17 \begin{vmatrix} 9 & 8 \\ 8 & 14 \end{vmatrix} = 19\,(-118) - (-63) + 17 \times 62 = -1125 \equiv_{26} 19$$

The Euclidean algorithm goes:

$$26 = 1 \times 19 + 7,\ \ 19 = 3 \times 7 - 2,\ 7 = 3 \times 2 + \underline{1},\ 2 = 2 \times 1 + 0$$

Since 26 and 19 are co-primes (relatively prime) the inverse to 19 mod 26 can be found. This also mean that the inverse matrix exists mod 26. In order to find $\mathbb{K}^{-1}$ mod 26 we start with the matrix $(\mathbb{K} \mid \mathbb{1})$ and find the row reduced Echelon form $\left(\mathbb{1} \,\middle|\, \mathbb{K}^{-1}\right)$. All operations are performed modulo 26.

$$(\mathbb{K} \mid \mathbb{1}) = \left(\begin{array}{ccc|ccc} 19 & 1 & 17 & 1 & 0 & 0 \\ 9 & 8 & 9 & 0 & 1 & 0 \\ 8 & 14 & 1 & 0 & 0 & 1 \end{array}\right) \sim \langle r_1 := r_1 - 2\,r_2 \rangle \sim \left(\begin{array}{ccc|ccc} 1 & 11 & 25 & 1 & 24 & 0 \\ 9 & 8 & 9 & 0 & 1 & 0 \\ 8 & 14 & 1 & 0 & 0 & 1 \end{array}\right)$$

The first calculation is done as

$$r_1 := r_1 - 2\,r_2 = (19,\ 1,\ 17,\ 1,\ 0,\ 0) - 2\,(9,\ 8,\ 9,\ 0,\ 1,\ 0) = (1,\ -15,\ -1,\ 1,\ -2,\ 0) \equiv_{26} (1,\ 11,\ 25,\ 1,\ 24,\ 0)$$

Notice that our goal is to create the identity matrix to the left. We continue as

$$\sim \langle r_2 := r_2 - 9\,r_1,\ r_3 := r_3 - 8\,r_1 \rangle \sim \left(\begin{array}{ccc|ccc} 1 & 11 & 25 & 1 & 24 & 0 \\ 0 & 13 & 18 & 17 & 19 & 0 \\ 0 & 4 & 9 & 18 & 16 & 1 \end{array}\right) \sim \langle r_2 := r_2 - 3\,r_3 \rangle \sim \left(\begin{array}{ccc|ccc} 1 & 11 & 25 & 1 & 24 & 0 \\ 9 & 8 & 9 & 15 & 23 & 23 \\ 8 & 14 & 1 & 18 & 16 & 1 \end{array}\right)$$

$$\sim \langle r_3 := r_3 - 4\,r_2 \rangle \sim \left(\begin{array}{ccc|ccc} 1 & 11 & 25 & 1 & 24 & 0 \\ 0 & 1 & 17 & 15 & 23 & 23 \\ 0 & 0 & 19 & 10 & 2 & 13 \end{array}\right) \sim \left\langle r_3 := 19^{-1}\, r_3 \equiv_{26} 11\, r_3 \right\rangle \sim \left(\begin{array}{ccc|ccc} 1 & 11 & 25 & 1 & 24 & 0 \\ 0 & 1 & 17 & 15 & 23 & 23 \\ 0 & 0 & 1 & 6 & 22 & 13 \end{array}\right)$$

$$\sim \langle r_1 := r_1 + r_3,\ r_2 := r_2 + 9\,r_3 \rangle \sim \left(\begin{array}{ccc|ccc} 1 & 11 & 0 & 7 & 20 & 13 \\ 0 & 1 & 0 & 17 & 13 & 10 \\ 0 & 0 & 1 & 6 & 22 & 13 \end{array}\right) \sim \langle r_1 := r_1 - 11\,r_2 \rangle \sim \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 2 & 7 & 7 \\ 0 & 1 & 0 & 17 & 13 & 10 \\ 0 & 0 & 1 & 6 & 22 & 13 \end{array}\right) = \left(\mathbb{1} \,\middle|\, \mathbb{K}^{-1}\right)$$

$$\therefore \mathbb{K}^{-1} \equiv_{26} \begin{pmatrix} 2 & 7 & 7 \\ 17 & 13 & 10 \\ 6 & 22 & 13 \end{pmatrix}$$

The message “DISCRETEMATHEMATICS” is divided into columns of length three, appended with Z and converted into numbers in $\mathbb{Z}_{26}$.

$$\mathbb{M} = \begin{pmatrix} 3 & 2 & 19 & 0 & 4 & 19 & 18 \\ 8 & 17 & 4 & 19 & 12 & 8 & 25 \\ 18 & 4 & 12 & 7 & 0 & 2 & 25 \end{pmatrix}$$

The cipher matrix is simply computed by

$$\mathbb{C} = \mathbb{K} \cdot \mathbb{M} = \begin{pmatrix} 19 & 1 & 17 \\ 9 & 8 & 9 \\ 8 & 14 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 2 & 19 & 0 & 4 & 19 & 18 \\ 8 & 17 & 4 & 19 & 12 & 8 & 25 \\ 18 & 4 & 12 & 7 & 0 & 2 & 25 \end{pmatrix} \equiv_{26} \begin{pmatrix} 7 & 19 & 23 & 8 & 10 & 13 & 12 \\ 19 & 8 & 25 & 7 & 2 & 19 & 15 \\ 24 & 24 & 12 & 13 & 18 & 6 & 25 \end{pmatrix}$$

So the cipher is then “HTYTIYXZMIHNKCSNTGMPZ”.

The decryption goes as

$$\mathbb{M} = \mathbb{K}^{-1} \cdot \mathbb{C} = \begin{pmatrix} 2 & 7 & 7 \\ 17 & 13 & 10 \\ 6 & 22 & 13 \end{pmatrix} \cdot \begin{pmatrix} 7 & 19 & 23 & 8 & 10 & 13 & 12 \\ 19 & 8 & 25 & 7 & 2 & 19 & 15 \\ 24 & 24 & 12 & 13 & 18 & 6 & 25 \end{pmatrix} \equiv_{26} \begin{pmatrix} 3 & 2 & 19 & 0 & 4 & 19 & 18 \\ 8 & 17 & 4 & 19 & 12 & 8 & 25 \\ 18 & 4 & 12 & 7 & 0 & 2 & 25 \end{pmatrix}$$

which corresponds to “DISCRETEMATHEMATICSZZ”.

Mathematica. We pick a slightly larger example, since we have computer support. The key is chosen to be a 5×5 matrix. This means that the column length should be 5 in the message matrix.

```
ClearAll["`*"]
```

We transform the message into $\mathbb{Z}_{26}$.

```
m = 26; A = 65;
message = "THISISASECRETTHATCANNOTBEREVEALEDFORANYONEBUTYOU";
message26 = ToCharacterCode[message] - A

n = 5; (* column length *)

pad = ToCharacterCode["Z"][[1]] - A

𝕄 = Transpose@Partition[
    Mod[message26, m],
    n, n, 1, pad];
MatrixForm[𝕄]
```

Next, we create an invertible matrix,

```
𝕂 = RandomInteger[{0, m - 1}, {n, n}];
While[GCD[Det[𝕂], m] ≠ 1,
  𝕂 = RandomInteger[{0, m - 1}, {n, n}];
 ];
MatrixForm[𝕂]
```

```
MatrixForm[
 𝕂1 = Inverse[𝕂, Modulus → m]
]
```

Now, the encryption goes:

```
MatrixForm[
 ℂ = Mod[𝕂.𝕄, m]
]
```

```
cipher = FromCharacterCode[Flatten[Transpose[ℂ]] + A]
```

The decryption use the inverse key $\mathbb{K}^{-1}$.

```
ℂr = Transpose@Partition[
     ToCharacterCode[cipher] - A,
     n];
MatrixForm[ℂr]
```

```
MatrixForm[
 𝕄r = Mod[𝕂1.ℂr, m]
]
```

```
receivedmessage = FromCharacterCode[Flatten@Transpose[𝕄r] + A]
```

The Hill Cipher (1929) is of course also out of date and we need more sophisticated methods.

2.2.3 Euler's Totient Theorem

According to Fermat's Little Theorem we know that $a^{p-1} = 1$ in the field $\mathbb{Z}_p$. Euler did extend this result to the ring $\mathbb{Z}_n$.

$a \in \mathbb{Z}_n,\ (a,\ n) = 1 \Rightarrow a^{\phi(n)} \equiv_n 1$

$\phi(n)$ can also be denoted by curly phi $\varphi(n)$.

In order to prove the theorem we can notice that for if a and x_i are invertible then $x_i \to a\, x_i$ is a bijection. Now, if $x_1, \ldots, x_{\phi(n)}$ are all invertible elements then $a\, x_1, \ldots, a\, x_{\phi(n)}$ is a permutation mod n, i.e. with the same product.

$$\prod_{i=1}^{\phi(n)} (a\, x_i) \equiv_n \prod_{i=1}^{\phi(n)} x_i \Leftrightarrow a^{\phi(n)} \prod_{i=1}^{\phi(n)} x_i \equiv_n \prod_{i=1}^{\phi(n)} x_i \Leftrightarrow a^{\phi(n)} \equiv_n 1$$

▼ Example

What is $14^{792\,002}$ mod 1 356 531 ?

Solution:

$a = 14 = 2 \times 7,\ n = 1\,356\,531 = 3 \times 11^2 \times 37 \times 101 \Rightarrow (a,\ n) = 1$

$$\phi(1\,356\,531) = 3 \times 11^2 \times 37 \times 101 \times \left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{11}\right)\left(1 - \frac{1}{37}\right)\left(1 - \frac{1}{101}\right) = 11 \times 2 \times 10 \times 36 \times 100 = 792\,000$$

Euler's Totient Theorem gives

$14^{792\,002} = 14^{\varphi(n)+2} \equiv_n 14^2 = 196$

Mathematica

```
PowerMod[14, 792002, 1356531]
```

2.2.4 RSA

RSA encryption, which is an asymmetric method, was described 1977 by Ron **R**ivest, Adi **S**hamir and Len **A**dleman. The method take advantage of Euler's Totient Theorem. Its security is built on that it is time consuming to factor numbers with large prime factors only.

▼ Consider the following example about the time to factor two consecutive (not too) large integers.

```
n = 38! (* easy to factor *)

{t1, f1} = AbsoluteTiming[FactorInteger[n]]

{t2, f1} = AbsoluteTiming[FactorInteger[n + 1]]

t2 / t1
```

We can see that the second number $38! + 1$ takes many times longer to factor than $38!$.

▼ Let's say Alice and Bob want to exchange messages. They both follow the steps 1, 2, 3 and 4 below.

1. Choose two large prime numbers p, q (size $\gtrsim 2^{2024}$ or even 2^{4096}) and build $n = p\,q$.
2. Now, choose an encryption key e (size a few digits) which is relatively prime to $\phi = (p - 1)\,(q - 1)$.
3. Finally, determine an inverse d to e in the ring $\mathbb{Z}_\phi$, $d\,e \equiv_\phi 1$. The inverse d is the decryption key.
4. **Publish** $(n,\ e)$ with an encryption description, but keep $(d,\ p,\ q)$ **secret**.

Let's say Alice wants to send a message to Bob.

- $m_{\text{Alice}} < n_{\text{Bob}}$
- $c_{\text{Alice}} \equiv_{n_{\text{Bob}}} m_{\text{Alice}}^{e_{\text{Bob}}}$

Bob decrypts the cipher by

- $m_{\text{Alice}} \equiv_{n_{\text{Bob}}} c_{\text{Alice}}^{d_{\text{Bob}}}$

Explanation:

Since d is the inverse to e in the ring $\mathbb{Z}_\phi$ there is an integer k (Bézout's Identity) so that

$e\,d \equiv_\phi 1 \Leftrightarrow e\,d = 1 + k\,\phi$

i.e. using Euler's Totient Theorem $(*)$

$$c^d = m^{e\,d} = m^{1+k\,\phi} = m\left(m^{\phi}\right)^k \overset{(*)}{\equiv}_n m\,(1)^k \equiv_n m$$

That is the clear text is revealed by computing $c^d \bmod(n)$.

Notice that you have to have a message $m < n$ in order to avoid ambiguity.

RSA is usually only used in the beginning of an encryption session in order to transfer a key for a symmetric encryption method, e.g. for advanced encryption standard (AES). The reason is that a symmetric method is in general faster.

Mathematica

Once again let's say Alice wants to send a message to Bob. We create primes (small scale trial with 15-20 figures) och and key according to the description above.

```
ClearAll["`*"]
```

```
pBob = NextPrime[RandomInteger[{10^15, 10^20}]]
```

```
qBob = NextPrime[RandomInteger[{10^15, 10^20}]]
```

```
nBob = pBob qBob
```

```
ϕBob = (pBob - 1) (qBob - 1)
```

```
rnd := RandomInteger[{10^3, 10^4}]
```

```
While[GCD[eBob = rnd, ϕBob] ≠ 1];
eBob
```

```
dBob = PowerMod[eBob, -1, ϕBob]
```

```
messageAlice = "discrete math";
```

```
ascii = ToCharacterCode[messageAlice]
```

Now we create a message as a single number $m_{\text{Alice}} < n_{\text{Bob}}$ by coding the ascii numbers in a position system with the base 256 (or another suitable base).

```
B = 256;
B^Range[StringLength[messageAlice]]-1
```

We take advantage of the scalar product to create the message $c_0\,B^0 + c_1\,B^1 + c_2\,B^2 + \ldots$

```
mAlice = ToCharacterCode[messageAlice] . B^Range[StringLength[messageAlice]]-1
```

The encryption is done by Alice:

```
cAlice = PowerMod[mAlice, eBob, nBob]
```

The cipher c_{Alice} is sent to Bob. The decryption is done by Bob:

```
mFromAlice = PowerMod[cAlice, dBob, nBob]
```

This number contains the message text which is revealed by extracting the ascii characters one by one module B.

```
firstcharacter = Mod[mFromAlice, B]
```

Here's a loop fir the rest.

```
q = mFromAlice; ascii = {};
While[q ≠ 0,
  AppendTo[ascii, Mod[q, B]];
  q = Quotient[q, B]
 ];
ascii
```

```
messageFromAlice = FromCharacterCode[ascii]
```

2.2.5 About Primes

Large primes (> 1000 bits) are usually created by statistical methods. According to Gauss' prime-counting function $\pi(n)$ the number of primes less or equal than n is approximately

$$\pi(n) \approx \frac{n}{\ln(n)}$$

At the end of the 19th century it was proved that the error approach 0 as $n \to \infty$. We can use this formula to estimate the probability of hitting a prime in with an odd random number in $[n/2, n) = \left[2^b, 2^{b+1}\right)$, where n consists of b bits.

$$p = \frac{\#\text{primes}}{\#\text{odd numbers}} = \frac{\pi(n) - \pi\left(\frac{n}{2}\right)}{\frac{1}{2}\left(n - \frac{n}{2}\right)} \approx 2\,\frac{n/\ln(n) - \left(\frac{n}{2}\right)/\ln\left(\frac{n}{2}\right)}{n - \frac{n}{2}} = \frac{2}{b\ln(2)}\left(\frac{b-1}{b+1}\right)$$

If we try k times the probability **not** hitting a prime is $(1-p)^k$. Then the probability of hitting **at least one** prime is $p_k = 1 - (1-p)^k$. Now let's make $k = b$ and comparing with $\lim_{x\to\infty}\left(1 + \frac{a}{x}\right)^x = e^a$ we get

$$p_b = 1 - \left(1 - \frac{2}{b\ln(2)}\left(\frac{b-1}{b+1}\right)\right)^b \to 1 - e^{-2/\ln(2)} \approx 0.944$$

so if we try $b = 100$ times for a 100-bits prime we have 94 × % probability of hitting at least one prime. If we try 3 b times the probability is $1 - e^{-3\times 2/\ln(2)} \approx 0.9998$. Now, the found prime candidates are tested statistically. The probability of that the number is composite decreases fast with the number of tested factors.

In Mathematica the correct $\pi(x)$, $x \lesssim 10^{15}$ can be computed with `PrimePi`.

```
PrimePi[25]
```

```
PrimePi[10^6]
```

★★★ Let's generate 250 odd numbers with 250 bits and extract a prime number. A well-know method is the Miller–Rabin test which tests a number and responds true with an error of 4^{-k}.

```
nlist = 2 RandomInteger[{2^248, 2^249}, 250] - 1;
Short[nlist, 5]
```

Here's a code that implements the primal test. The parameter k is a quality parameter.
If you `Compile` the method (e.g. for C) you have to deal with machine size integers, typically in the range $\pm\left(2^{63} - 1\right)$. However, larger integers can be handled by the Chinese remainder theorem (see

Böiers, example 44, ch. 6.6).

```
MillerRabin[k_][n_] := Module[
  {d = n - 1, r = 0, a, x, i, result = True},
  While[Mod[d, 2] == 0, d /= 2; r++];
  Do[
   a = RandomInteger[{2, n - 2}];
   x = PowerMod[a, d, n];
   If[x ≠ 1,
    For[i = 0, i < r, i++,
     If[x == n - 1, Continue[]];
     x = Mod[x x, n]
    ];
    If[x ≠ n - 1, result = False]
   ],
   {k}
  ];
  result
 ]

(* simple test *)
{MillerRabin[5][1001], MillerRabin[5][1009]}

(* the 250 bits number *)
plist = Select[nlist, MillerRabin[20]]

(* compare with built-in PrimeQ *)
Select[nlist, PrimeQ]
```

2.2.6 Using RSA for Authentication

In RSA it's not just Alice that can send a message to Bob. Anyone that access Bob's public keys can sen an encrypted message. So how can Bob know that the message is from Alice?

A rather straight forward way of doing this is that Alice also encrypt the message with her secret key d_{Alice}. Bob will later decrypt, using Alice's public key.

Let's say Alice wants to send a message to Bob.

- $m_{\text{Alice}} < \min(n_{\text{Alice}}, n_{\text{Bob}})$
- start with the key that belongs to $\min(n_{\text{Alice}}, n_{\text{Bob}})$
  - $n_{\text{Alice}} < n_{\text{Bob}} \Rightarrow c_0 \equiv_{n_{\text{Alice}}} m_{\text{Alice}}^{d_{\text{Alice}}},\ c_{\text{Alice}} \equiv_{n_{\text{Bob}}} c_0^{e_{\text{Bob}}}$
  - $n_{\text{Bob}} < n_{\text{Alice}} \Rightarrow c_0 \equiv_{n_{\text{Bob}}} m_{\text{Alice}}^{e_{\text{Bob}}},\ c_{\text{Alice}} \equiv_{n_{\text{Alice}}} c_0^{d_{\text{Alice}}}$

Bob decrypts the cipher by

- start with the key that belongs to $\max(n_{\text{Alice}}, n_{\text{Bob}})$
  - $n_{\text{Bob}} > n_{\text{Alice}} \Rightarrow c_1 \equiv_{n_{\text{Bob}}} c_{\text{Alice}}^{d_{\text{Bob}}},\ m_{\text{Alice}} \equiv_{n_{\text{Alice}}} c_1^{e_{\text{Alice}}}$
  - $n_{\text{Alice}} > n_{\text{Bob}} \Rightarrow c_1 \equiv_{n_{\text{Alice}}} c_{\text{Alice}}^{e_{\text{Alice}}},\ m_{\text{Alice}} \equiv_{n_{\text{Bob}}} c_1^{d_{\text{Bob}}}$

Usually the authentication is not done on the full message, but rather a message digest or a cryptographic checksum, e.g. secure hash algorithm (SHA).

2.2.7 Diffie-Hellman Key Exchange (★)

This key exchange algorithm was published by Whitfield Diffie and Martin Hellman 1976. The algorithm shows how to exchange a secret key (e.g. for a symmetric encryption algorithm.

Public keys: Alice and Bob agree on a large prime p and a generator g for $\mathbb{Z}_p$.

- Alice picks a random number x in $\mathbb{Z}_p$ and computes $X \equiv_p g^x$. She sends X to Bob.
- Bob picks a random number y in $\mathbb{Z}_p$ and computes $Y \equiv_p g^y$. He sends Y to Alice.
- Alice computes $K_A \equiv_p Y^x$
- Bob computes $K_B \equiv_p X^y$

Now Alice and Bob have the same key. Notice that this exchange method is sensible for a man-in-the-middle attack (since Bob is not authenticated).

Explanation:

$K_A \equiv_p Y^x \equiv_p (g^y)^x \equiv_p (g^x)^y \equiv_p X^y \equiv_p K_B$

Mathematica

Example with not too large numbers.

```
ClearAll["`*"]
```

```
p = NextPrime[RandomInteger[10^25]]
```

```
g = PrimitiveRoot[p, 1000]
```

```
x = RandomInteger[{2, p - 1}]
```

```
X = PowerMod[g, x, p]
```

```
y = RandomInteger[{2, p - 1}]
```

```
Y = PowerMod[g, y, p]
```

```
KA = PowerMod[Y, x, p]
```

```
KB = PowerMod[X, y, p]
```

```
KA == KB
```

2.2.8 RSA in Mathematica (★★★)

Here's the previous encryption (without Alice's authentication) with built-in commands. We will now use professional number sizes.

```
ClearAll["`*"]
```

In this example we RSA (default) and 2048 bits (default) as a public modulus (n), which is the minimum recommended key size today. The public exponent (e) is by default the prime number $65\,537 = 2^{16} + 1$.

```
GenerateAsymmetricKeyPair[Method -> Association["Cipher" -> "RSA", "KeySize" -> 2048,
```

```
"PublicExponent" -> 65537]]
```

The keys for Bob are:

```
keysBobRSA = GenerateAsymmetricKeyPair[]
```

```
publicKeyBob = keysBobRSA["PublicKey"]
```

```
privateKeyBob = keysBobRSA["PrivateKey"]
```

Now, Alice get the public keys for Bob and encrypt her message.

```
messageAlice = "discrete math";
```

```
cipherObjectAliceRSA = Encrypt[publicKeyBob, messageAlice]
```

Now, Bob decrypt using his private key.

```
Decrypt[privateKeyBob, cipherObjectAliceRSA]
```

In order to exchange information outside Mathematica we need to use a standard formats.

```
Normal[publicKeyBob]
```

```
eBob = publicKeyBob["PublicExponent"]
```

```
nBob = publicKeyBob["PublicModulus"]
```

Alice creates a file with the encrypted bytes.

```
cipherBytesAliceRSA = Normal[cipherObjectAliceRSA["Data"]]
```

```
file = FileNameJoin[{NotebookDirectory[], "Alice.bin"}]
```

```
BinaryWrite[file, cipherBytesAliceRSA]; Close[file]
```

Now, Bob reads the file from Alice and decrypts its contents.

```
cipherBytesFromAliceRSA = BinaryReadList[file]
```

```
object = EncryptedObject[
  Association["Data" → ByteArray[cipherBytesFromAliceRSA], "OriginalForm" → String]]
```

The decryption goes

```
Decrypt[privateKeyBob, object]
```

Instead of a binary file we can use a string format, e.g. Base64. This is easy to include in a text message. The `InputForm` of a byte array reveals the string.

```
InputForm[cipherObjectAliceRSA["Data"]] // StandardForm
```

```
extractString[a_ByteArray] := Module[
  {str, s, e},
  str = ToString[InputForm[a]];
  {{s, s}, {e, e}} = StringPosition[str, "\""];
  StringTake[str, {s + 1, e - 1}]
 ]
```

```
cipherStringAliceRSA = extractString[cipherObjectAliceRSA["Data"]]

SetDirectory[NotebookDirectory[]]

Export["Alice.txt", cipherStringAliceRSA]

cipherString = Import["Alice.txt"]

object = EncryptedObject[
  Association["Data" → ByteArray[cipherString], "OriginalForm" → String]]

Decrypt[privateKeyBob, object]
```

2.2.9 A Complete File Example (★★★)

In this example we use a professional approach. Assume that Alice wants to transfer a file to Bob. In order to do that we want to use the symmetric encryption method AES256. The symmetric key is transferred to Bob with RSA4096. Alice also computes a cryptographic checksum with SHA512 for to secure the file and for authentication.

Encryption by Alice

1. Alice use the message file m_0 to compute a digest $h_0 = h(m_0)$ using SHA512. This is used to secure the file contents from modification.
2. Alice encrypts the digest with her private or secret decryption key d_A and get $h_A = d_A(h_0)$. This is done to authenticate Alice.
3. Alice generates a symmetric key k_0 and initialization vector v_0 for AES256.
4. Alice encrypts the file and ends up with $c_A = k_0(m_0)$.
5. Alice encrypts the key $\{k_0, v_0\}$ with Bob's public encryption key e_B and ends up with $k_A = e_B(\{k_0, v_0\})$.
6. Finally, the information $\{h_A, k_A, c_A\}$ is transferred to Bob.

After receiving the information the next phase takes place:

Decryption by Bob

7. Bob use his private key to decrypt the file symmetric key $k_1 = d_B(k_A)$.
8. Bob decrypts the file by $m_1 = k_1(c_A)$.
9. Bob computes a digest using SHA512, $h_1 = h(m_1)$.
10. Bob decrypts the received digest with Alice's public key by $h_2 = e_A(h_A)$.
11. If $h_2 = h_1$ the file is accepted.

Mathematica

```
ClearAll["`*"]

dir = SetDirectory[NotebookDirectory[]]
```

0. Alice

Alice creates her RSA keys.

```
keysAlice = GenerateAsymmetricKeyPair[Method →
    Association["Cipher" → "RSA", "KeySize" → 4096, "PublicExponent" → 65 537]];
eA = keysAlice["PublicKey"];
dA = keysAlice["PrivateKey"];
```

0. Bob

Bob creates his RSA keys.

```
keysBob = GenerateAsymmetricKeyPair[Method →
    Association["Cipher" → "RSA", "KeySize" → 4096, "PublicExponent" → 65 537]];
eB = keysBob["PublicKey"];
dB = keysBob["PrivateKey"];
```

1. Alice

Alice save the clear text and computes a digest.

```
m0 = WikipediaData["Moon"];
StringLength[m0]
```

```
StringTake[m0, {1, 250}] (* view part of the message *)
```

```
Export["cleartext.txt", m0];
h0 = FileHash["cleartext.txt", "SHA512"]
```

2. Alice

Alice encrypts the digest with her private key.

```
object = Encrypt[dA, h0];
hABytes = Normal[object["Data"]];
Length[hABytes]
```

3. Alice

Alice generates a symmetric key. Notice that the mode of operation is CBC by default.

```
v0 = RandomInteger[255, 16]
```

```
k0 = GenerateSymmetricKey[
  Method → Association["Cipher" → "AES256", "InitializationVector" → ByteArray[v0]]]
```

4. Alice

Alice encrypts the message.

```
object = Encrypt[k0, m0];
cABytes = Normal[object["Data"]];
Short[cABytes]
```

5. Alice

Alice encrypts the symmetric key and the initialization vector.

```
object = Encrypt[eB, ByteArray[Join[Normal[k0["Key"]], v0]]];
kABytes = Normal[object["Data"]];
Length[kABytes]
```

6. Alice

Save the file:

```
file = FileNameJoin[{dir, "Alice.bin"}];
stream = OpenWrite[file, BinaryFormat → True];
BinaryWrite[stream, hABytes];
BinaryWrite[stream, kABytes];
BinaryWrite[stream, cABytes];
Close[stream];

Clear[m0, h0, hABytes, kABytes, cABytes, object]
```

7. Bob

Bob reads the file content, identify the symmetric key and decrypts it with his secret key.

```
block = 512; (* bytes *)
file = FileNameJoin[{dir, "Alice.bin"}];
stream = OpenRead[file, BinaryFormat → True];
hABytes = BinaryReadList[stream, "Byte", block];
kABytes = BinaryReadList[stream, "Byte", block];
cABytes = BinaryReadList[stream];
Close[stream];

object = EncryptedObject[Association[
     Association["Data" → ByteArray[kABytes], "OriginalForm" → ByteArray]]];
k1v1 = Normal[Decrypt[dB, object]]

k1 = SymmetricKey[Association["Cipher" → "AES256",
    "BlockMode" → "CBC", "Key" → ByteArray[k1v1[[1 ;; 32]]],
    "InitializationVector" → ByteArray[k1v1[[33 ;; 48]]]]]
```

8. Bob

The decryption of the message file goes:

```
object =
  EncryptedObject[Association["Data" → ByteArray[cABytes], "OriginalForm" → String]];
m1 = Decrypt[k1, object];
StringTake[m1, {1, 250}]
```

9. Bob

```
Export["bobstext.txt", m1];
h1 = FileHash["bobstext.txt", "SHA512"]
```

10. Bob

```
object = EncryptedObject[
   Association["Data" → ByteArray[hABytes], "OriginalForm" → Expression]];
h2 = Decrypt[eA, object]
```

11 Bob

Is this the correct message?

```
h2 == h1
```

2.3 Finding a Suitable Model

2.3.1 Linear Function

A straight line is a simple model that is useful to the predictions. Example: assume that our model is $y = a\,x + b$ is correct and we have data in the interval $0 \le x \le 10$.

```
ClearAll["`*"]
```

```
𝕩 = {0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5,
   4., 4.5, 5., 5.5, 6., 6.5, 7., 7.5, 8., 8.5, 9., 9.5, 10.};
𝕪 = {1.06, 1.73, 2.26, 1.84, 2.35, 2.72, 2.86, 3.43, 3.36,
   4.01, 3.72, 4.9, 5.03, 5.51, 6.08, 5.91, 6.63, 7., 7.26, 7.14, 8.2};
```

```
data = Transpose[{𝕩, 𝕪}]
```

```
dataplot = ListPlot[data, AxesLabel → {x, y}, PlotStyle → Red]
```

```
dataplot = ListPlot[data, AxesLabel → {m[x], y},
  PlotStyle → Red, Ticks → None, BaseStyle → {FontSize → 12}]
```

We can now fit a straight line to the data using the method of least squares.

```
solution = FindFit[data, a x + b, {a, b}, x]
```

```
y[x_] = a x + b /. solution
```

```
modelplot = Plot[y[x], {x, 0, 10}, AxesLabel → {x, y}];
Show[dataplot, modelplot, ImageSize → 250]
```

We can see that our model can be used for prediction, for example $y(15)$.

```
y[15]
```

The error that the prediction has depends on the distribution of data and number of measuring points. You'll learn more about this in mathematical statistics.

That you can fit a straight line visually is a strong indication of a stable model.

2.3.2 Exponential Function

The straight line can be used even if the model is not linear. As you might know the exponential gives a straight line in a semi-log plot (lin-log plot) in which the x-axis is linear and the y-axis is logarithmic. The scale on the y-axis is chosen using the inverse function $y = e^x \Leftrightarrow \ln(y) = x$.

$y = c\, e^{a\,x} \Leftrightarrow \ln(y) = \ln(c) + k\,x \Leftrightarrow \ln(y) = a\,x + b \Leftrightarrow y = e^{a\,x+b}$

$y = c\, 10^{k\,x} \Leftrightarrow \log_{10}(y) = \log_{10}(c) + a\,x \Leftrightarrow \log_{10}(y) = a\,x + b \Leftrightarrow y = 10^{a\,x+b}$

In other words, if we compute the logarithm of y-data we get a straight line.

Example:

```
ClearAll["`*"]
```

```
𝕩 = {0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5,
   4., 4.5, 5., 5.5, 6., 6.5, 7., 7.5, 8., 8.5, 9., 9.5, 10.};
𝕪 = {2.88, 5.65, 9.61, 6.32, 10.5, 15.2, 17.5, 30.9, 28.8, 55.1,
   41.1, 134., 152., 248., 437., 368., 759., 1090., 1430., 1260., 3660.};
```

```
data = Transpose[{𝕩, 𝕪}]
```

```
dataplot = ListLogPlot[data, AxesLabel → {x, y}, PlotStyle → Red]
```

```
ListPlot[Transpose[{𝕩, Log10[𝕪]}], AxesLabel → {HoldForm[x], HoldForm[Log10[y]]}]
```

★ **If data fits a straight line in a log(y)-lin(x) plot, then the model is an exponential.**

```
data2 = Transpose[{𝕩, Log[𝕪]}]
```

```
solution = FindFit[data2, a x + b, {a, b}, x]
```

```
y[x_] = ℯ^(a x+b) /. solution
```

```
modelplot = LogPlot[y[x], {x, 0, 15}, AxesLabel → {x, y}];
Show[modelplot, dataplot]
```

We can see that our model can be used for prediction, for example $y(15)$.

```
y[15]
```

Note: an exponential function a^x (Swe: exponentialfunktion) is a function where the exponential is the independent variable.

The other way around: Assume that our model $e^{c\,y+d} = x$ is the model in focus.

$$e^{c\,y+d} = x \Leftrightarrow c\,y + d = \ln(x) \Leftrightarrow y = \frac{1}{c}\ln(x) - \frac{d}{c} \Leftrightarrow y = a\ln(x) + b$$

In other words, if we compute the logarithm of x-data we get a straight line.

```
ClearAll["`*"]
```

```
𝕩 = {2.88, 5.65, 9.61, 6.32, 10.5, 15.2, 17.5, 30.9, 28.8, 55.1,
   41.1, 134., 152., 248., 437., 368., 759., 1090., 1430., 1260., 3660.};
𝕪 = {0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5,
   4., 4.5, 5., 5.5, 6., 6.5, 7., 7.5, 8., 8.5, 9., 9.5, 10.};
```

```
data = Transpose[{𝕩, 𝕪}]
```

```
dataplot = ListLogLinearPlot[data, AxesLabel → {x, y}, PlotStyle → Red]
```

★ **If data fits a straight line in a lin(y)-log(x) plot, then the model is an logarithmic function.**

```
data2 = Transpose[{Log[𝕩], 𝕪}]

solution = FindFit[data2, a lnx + b, {a, b}, lnx]

y[x_] = a Log[x] + b /. solution

modelplot = LogLinearPlot[y[x], {x, 1, 10^5}, AxesLabel → {x, y}];
Show[modelplot, dataplot]
```

We can see that our model can be used for prediction, for example $y(10^5)$.

```
y[10^5]
```

2.3.3 Power Function

Now, assume our model is $y = c\,x^a$.

$y = c\,x^a \Leftrightarrow \ln(y) = \ln(c) + a\ln(x) \Leftrightarrow \ln(y) = a\ln(x) + b \Leftrightarrow y = e^b\,x^a$

That is, if data compute the logarithm for both x and y we get a straight line.

Example:

```
ClearAll["`*"]

𝕩 = {1., 1.6, 2.7, 4.5, 7.4, 12., 20., 33., 55., 90., 150.,
   245., 400., 650., 1100., 1800., 3000., 5000., 8000., 13000., 22000.};
𝕪 = {2.88, 5.65, 9.61, 6.32, 10.5, 15.2, 17.5, 30.9, 28.8, 55.1,
   41.1, 134., 152., 248., 437., 368., 759., 1090., 1430., 1260., 3660.};

data = Transpose[{𝕩, 𝕪}]

dataplot = ListLogLogPlot[data, AxesLabel → {x, y}, PlotStyle → Red]
```

★ **If data fits a straight line in a log(y)-log(x) plot, then the model is a power function.**

```
data2 = Transpose[{Log[𝕩], Log[𝕪]}]

solution = FindFit[data2, a lnx + b, {a, b}, lnx]

y[x_] = e^b x^a /. solution

modelplot = LogLogPlot[y[x], {x, 1, 10^6}, AxesLabel → {x, y}];
Show[modelplot, dataplot]
```

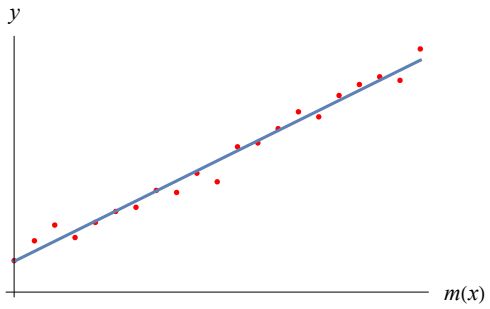
We can see that our model can be used for prediction, for example $y(10^5)$.

```
y[10^6]
```

Note: a power function x^a (Swe: potensfunktion) is a function where the base is the independent variable.

2.3.4 General Function

We can use the idea of a straight line in a general function case. Let's say we have a model function $m(x)$ and we want to test this against data. Then we compute $g(𝕩)$ and compare this with 𝕪. If we can fit a straight line we have a good model.



Example:

```
ClearAll["`*"]
```

```
𝕩 = {0., 0.25, 0.51, 0.77, 0.99, 1.31, 1.48, 1.72, 2.05, 2.22,
   2.44, 2.87, 3.09, 3.12, 3.45, 3.68, 4.08, 4.04, 4.28, 4.98, 4.89};
𝕪 = {0.233, 0.369, 0.599, 0.78, 0.567, 0.553, 0.42, 0.65, 1.28,
   1.43, 1.69, 1.75, 1.52, 1.45, 1.47, 1.62, 2.34, 2.3, 2.53, 2.6, 2.53};
```

```
data = Transpose[{𝕩, 𝕪}];
```

```
dataplot = ListPlot[data, AxesLabel → {x, y}, PlotStyle → Red]
```

Assume that our model is $y = m(x) = \frac{x}{2} e^{-\frac{x}{2}} \sin(\pi x) + \frac{x}{2}$. We compute $m(\mathbb{x})$ and compare with given $\mathbb{y}$. If this plot fits a straight line we have a good model function.

```
m[x_] := x/2 ℯ^(-x/2) Sin[π x] + x/2
```

```
dataplotlin = ListPlot[Transpose[{m[𝕩], 𝕪}],
  AxesLabel → {HoldForm[m[x]], HoldForm[y]}, PlotStyle → Red]
```

Next, we compute the straight line parameters.

```
sol = FindFit[Transpose[{m[𝕩], 𝕪}], a mx + b, {a, b}, mx]
```

How is the line fit looking?

```
modelplotlin = Plot[a mx + b /. sol, {mx, 0, 4}];
Show[dataplotlin, modelplotlin]
```

The adjusted model is then:

```
f[x_] = Expand[a m[x] + b /. sol]
```

The model function and the original data:

```
modelplot = Plot[f[x], {x, 0, 8}, AxesLabel → {x, y}];
Show[modelplot, dataplot]
```

```
(* interpolation *)
f[3.85]
```

```
(* predicition *)
f[8]
```

3. Mathematical Task

Required for grade C-E. Keep in mind that it is the mathematical method on the task that is interesting to consider, not the answer. Furthermore, solve the given task, not a variant or extension.

- Use KTH Canvas to discuss Mathematica code.
- A general hint is to start in a small scale and test your methods with not too large numbers.
- Take advantage of that you are using a mathematical programming language. Don't "invent the wheel". However, you are not supposed to use the built-in encryption methods, described in sections 2.2.8 and 2.2.9.

For Pass Grade: RSA cipher

Professor Alice is sending a message to the student Bob according to the procedure in section 2.2.4. You are supposed to crack the message. When translating to ASCII, you can assume the base 256.

```
nBob = 126 456 119 090 476 383 371 855 906 671 054 993 650 778 797 793 018 127;
eBob = 7937;
```

```
cipher = {71 813 256 693 940 924 296 894 077 934 214 561 172 810 879 712 474 411,
  9 448 822 287 828 090 646 994 864 850 737 396 938 193 829 207 476 291,
  88 668 970 435 389 288 697 377 439 396 925 326 741 948 237 682 465 270,
  86 506 877 126 882 849 406 016 686 638 047 102 838 609 248 170 576 618,
  16 709 897 999 784 737 136 957 685 475 437 549 241 701 090 506 782 283,
  112 082 150 953 644 879 808 862 406 205 324 790 087 623 126 644 040 573,
  101 300 870 021 945 928 543 132 671 557 050 279 918 096 489 651 239 300,
  32 937 734 818 698 596 498 554 567 892 462 717 857 351 635 451 752 837,
  103 250 795 649 561 696 933 993 996 191 026 658 588 156 558 009 063 626,
  9 944 688 399 741 552 477 615 864 010 579 036 184 245 783 411 883 057,
  119 023 583 366 882 743 798 043 931 890 543 936 842 945 498 718 476 068,
  80 592 157 601 474 287 498 990 443 067 778 705 256 100 803 395 677 817,
  102 380 508 653 117 028 882 619 903 124 827 386 257 126 674 349 361 274,
  29 811 966 446 563 529 123 471 275 226 007 901 312 118 773 446 042 793,
  92 762 330 649 448 230 399 110 375 463 713 210 616 117 461 806 861 915,
  52 785 580 108 219 931 044 518 308 758 110 100 269 607 232 524 031 605,
  96 630 768 452 430 661 169 900 035 905 564 353 166 088 443 035 700 946,
  104 675 165 348 205 433 706 999 623 683 285 417 639 543 643 952 502 324,
  40 951 632 727 878 687 548 912 007 343 839 372 258 522 783 062 745 255,
  3 439 648 578 841 960 331 931 477 586 254 936 252 926 807 061 184 128,
  92 627 296 356 479 225 180 584 868 594 165 589 614 134 261 562 166 537,
  17 702 026 915 107 984 931 197 975 326 852 130 481 340 232 863 388 490,
  35 046 202 376 732 485 019 333 999 169 687 110 582 305 137 751 148 612,
  77 294 680 692 381 954 105 730 803 435 472 597 801 358 963 832 333 113,
  58 483 888 921 987 241 464 318 109 604 079 587 034 521 404 720 554 634,
  36 276 638 436 400 152 414 964 124 035 009 391 520 390 748 123 243 684,
  51 639 523 466 890 776 909 441 678 913 110 130 114 797 820 309 131 676,
  88 728 872 239 148 972 759 884 018 820 709 080 618 086 999 507 011 767,
  45 676 147 252 256 101 340 528 647 372 987 947 783 315 245 082 701 701,
  23 720 650 117 296 688 653 687 823 869 949 231 140 410 366 974 406 435,
  116 873 909 796 842 028 543 216 809 278 057 888 647 421 675 552 833 624,
  48 366 928 605 018 172 969 920 839 968 881 382 332 820 063 246 862 564,
  35 425 491 594 411 738 404 916 586 785 616 696 655 411 948 001 887 947,
  40 450 505 118 769 549 506 412 191 479 348 341 185 611 602 935 328 569,
  107 418 270 783 831 708 663 699 380 219 027 152 916 779 513 788 697 702,
  101 200 673 359 310 801 145 084 267 798 164 209 444 861 857 835 311 695,
  65 616 489 296 627 251 359 500 608 540 483 019 164 860 755 372 512 518,
  11 847 413 450 576 524 199 351 895 472 796 724 862 275 584 777 010 578,
  2 731 217 915 540 071 371 661 447 436 484 606 877 270 200 777 923 464,
  10 599 418 784 042 349 226 543 806 726 994 624 123 223 235 946 860 821}
```

- The numbers is the list seems to be of the same size. Why?

For Higher Grade: RSA decryption analysis

Write a method `RSAcrack[cipher,n,e]` that will crack a standard RSA cipher and delivers clear text from the string cipher. When you are finished with your method, you should investigate how long it will take to crack the cipher of the English text “GO DOWN DEEP ENOUGH INTO ANYTHING AND YOU WILL FIND MATHEMATICS.” for different sizes on your public key n (100-200 bits). Visualize your results in a proper graph. It is very important that you study the section

2.3 in the instructions. Your graph should lead you to a model where you can predict how long it would take to crack a cipher if n is 1024, 2048 bits or 4096 bits

- Motivate your selected mathematical model!