Algorithmic complexity and graphs: simple cryptographic examples

29 septembre 2022

Cryptography

- We will study some cryptography algorithms, that will provide first examples of algorithmic complexities.
- Please note that this section is <u>not</u> intended to be a cryptography course, but rather a course to focus on some mathematical aspects of the involved algorithms.
- The practical implementation of a real cryptosystem contains way more than the core mathematical principles but this is outside the scope of this course.

First example (Chiffrage par substitution)

- We want to be able to **cipher a text** by **permutating** the letters of the alphaet.

Chiffrage par substitution

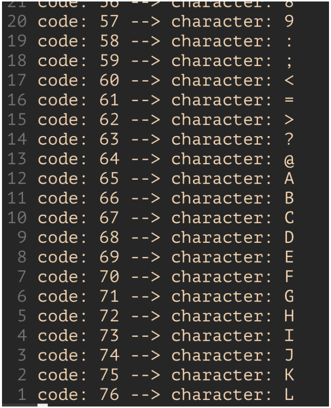


Figure – Unicode codes. We will work with messages containing only uppercase letters.

First example (Chiffrage par substitution)

- We want to be able to **cipher a text** by **permutating** the letters of the alphaet.

$$A \mapsto F, \quad B \mapsto P,$$

- $$C \mapsto A, \quad D \mapsto \dots$$

Figure – Example permutation

Ciphering

Exercice 1 : First ciphering example

- **cd .code/crypto_intro**
- Please modify the file **crypto_intro/cipher_1.py** so that the function *cipher_1(s)* produces a random key and ciphers the text s, which is a string.
- "cipher" means "chiffrer" in french

Breaking the code : *known-plaintext attack, attaque à texte clair connu*

Exercice 1 : First ciphering example part II

- Please modify the file **crypto_intro/decipher_1.py** in order to attempt to find the key from a **coded message** and an **extract**.

Breaking the code : *known-plaintext attack, attaque à texte clair connu*

Exercice 1 : First ciphering example part II

- Please modify the file **crypto_intro/decipher_1.py** in order to attempt to find the key from a **coded message** and an **extract**.
- Is it working ?

Breaking the code : *known-plaintext attack, attaque à texte clair connu*

Exercice 1 : First ciphering example part II

- Please modify the file **crypto_intro/decipher_1.py** in order to attempt to find the key from a **coded message** and an **extract**
- Is it working ?
- Why is it taking such a long time ?

Number of permutations

- How many keys are possible ?

Number of permutations

- How many keys are possible ?
- Let us count them.

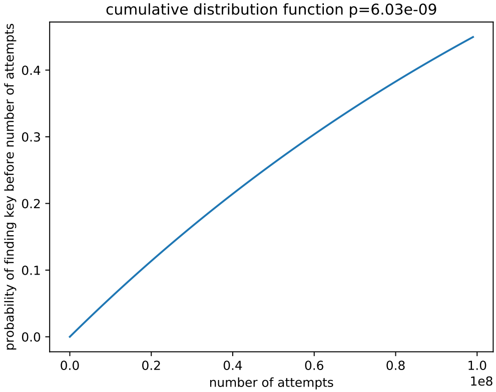
Number of permutations

- How many keys are possible ?
- $26! = 403291461126605635584000000$
- It is the number of **permutations.**

Exercice 2 : How many keys would actually stop the program ? What is the probability that we have found a key that stops the program at trial n ?

Geometric distribution

Many keys could stop the program : all the keys that give the known text.



Necessary time

Exercice 3 : Please evaluate the time that would be necessary on your machine to evaluate all possible keys.

Necessary time

- I need 3.6 milliseconds to try 100 keys.

Necessary time

- I need 3.6 milliseconds to try 100 keys.
- So I need 0.036 millisecond to try 1 key.

Necessary time

- I need 3.6 milliseconds to try 100 keys.
- So I need 0.036 millisecond to try 1 key.
- Which means $\simeq 1.45 \times 10^{25}$ seconds for 26! permutations.

Necessary time

- I need 3.6 milliseconds to try 100 keys.
- So I need 0.036 millisecond to try 1 key.
- Which means $\simeq 1.45 \times 10^{25}$ seconds for 26! permutations.
- Or $\simeq 4.6 \times 10^{17}$ years.

Necessary time

On my machine, the necessary time in order to have a 40% probabilty of stopping the program is around 30 minutes, using the geometric law.

$$P(\text{number of attemps} \leq 10^8) \simeq 40\% \tag{1}$$

First example

However, what would be a shortcoming of this method ?

First example

However, what would be a shortcoming of this method ?
It is vulnearble to **statistical attacks**.

Second example

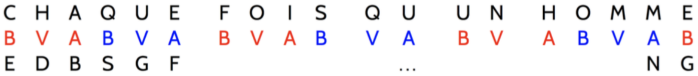
- Let us do another example
- 

Figure – Second ciphering method

Second example

Exercice 4 : Please modify the file **crypto_intro/cipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way

Second example : Breaking the code

- Please modify the file **crypto_intro/decipher_2.py** in order to attempt to find the key from a **coded message** and an **extract** (same attack type as before : *known plain text)* .

Second example

- Please modify the file **crypto_intro/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 50 characters. For which key sizes does the algorithm break the code ?

Second example

- Please modify the file **crypto_intro/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?
- What is the number ok keys that are to be tried, as a function of the size of the key ?

Second example

- Please modify the file **crypto_intro/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?
- What is the number of keys that are to be tried, as a function of the size of the key ? $26^{\text{key size}}$

Second example

- Please modify the file **crypto_intro/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?
- What is the number of keys that are to be tried, as a function of the size of the key ? $26^{\text{key size}}$
- So probably you won't be able to break the code for $k \geq 7$ or so.

The problem of complexity

- Complexity is key for security problems. However, it is important in most other fields as well and could prevent a program from working well.

Private and public keys

- Before diving into complexity we will study a more complex cryptosystem
- It will allow us to study a more complex algorithm

Private and public keys

- Before diving into complexity we will study a more complex cryptosystem
- **RSA** is based on a Public-key system

Private and public keys

- Before diving into complexity we will study a more complex cryptosystem
- **RSA** is based on a Public-key system
- As opposed to symmetric key algorithms

Symmetric key algorithm

- In the first examples we saw, the same key is used to cipher and to decipher the message

Symmetric key algorithm

- In the first examples we saw, the same key is used to cipher and to decipher the message
- This is called a **symmetric key algorithm**

Symmetric key algorithm

- In the first examples we saw, the same key is used to cipher and to decipher the message
- This is called a symmetric key algorithm
- However would there be an advantage of using **two** keys ?

Public keys and private keys

- **Public key** : used to cipher a text
- **Private key** : used to decipher a text

Public keys and private keys

- **Public key** : used to cipher a text
- **Private key** : used to decipher a text
- There is no need to transmit the private key on the network.
- Whereas in a symmetric context, one needs a secure canal to transmit the key.

Asymmetric cryptosystem

How many keys do we need to generate for each case to enable n persons to communicate ?

Asymmetric cryptosystem

How many keys do we need to generate for each case ?

- Symmetric : each subset of 2 persons must have 1 key.
- Asymmetric : each person must have 1 public key and 1 private key.

Asymmetric cryptosystem

How many keys do we need to generate for each case ?

- Symmetric : each subset of 2 persons must have 1 key : $\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$.
- Asymmetric : each person must have 1 public key and 1 private key : $2n$.

(If generating a key is very long, or if n is very large, this could be a significant advantage, however this does not determine the choice between symmetric and asymmetric in most cases)

Examples :

- Symmetric : AES
- Asymmetric : RSA, ssh, sftp

Examples :

- Symmetric : AES
- Asymmetric : RSA, ssh, sftp
- We will study a simplification of RSA. In real applications, the method is not implemented this way. RSA is sometimes even used to cipher as AES key, that is used to cipher the message.
- Also we won't mention block ciphering.

RSA

- RSA is based on **modular exponentiation**.
- Let M be a message to cipher. We assume M is an integer. Let C be the code (also an integer)

RSA

- RSA is based on **modular exponentiation**.
- Let M be a message to cipher. We assume M is an integer. Let C be the code (also an integer)
- We work **modulo an integer** n (hence the name **modular** exponentiation)
- $17 \equiv 1 \mod 4$
- $25 \equiv 0 \mod 5$

RSA and modular exponentiation

- We work **modulo an integer** n (hence the term **modular** exponentiation)
- $17 \equiv 1 \mod 4$
- $25 \equiv 0 \mod 5$
- Advanced notion : This means that instead of working with the **ring** of **integers** $\mathbb{Z}$ (anneau des entiers relatifs) we work in the **quotient ring** $\mathbb{Z}/n\mathbb{Z}$ (anneau quotient).

RSA

- RSA is based on **modular exponentiation**.
- M : message to cipher. C : code.
- **Public key** : (n, a)
- **Private key** : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$

RSA

- RSA is based on **modular exponentiation**.
- M : message to cipher. C : code.
- **Public key** : (n, a)
- **Private key** : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$
- Let D be de **deciphered** message
- $D \equiv C^b \mod n$

RSA

- M : message to cipher. C : code.
- **Public key** : (n, a)
- **Private key** : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$
- Let D be de **deciphered** message
- $D \equiv C^b \mod n$
- In order for the algorithm to work, we must have $D \equiv M \mod n$.

RSA

- M : message to cipher. C : code.
- Public key : (n, a), Private key : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$
- Let D be de **deciphered** message
- $D \equiv C^b \mod n$
- In order for the algorithm to work, we must have $D \equiv M \mod n$.
- Which means : $M^{ab} \equiv M \mod n$

RSA

- M : message to cipher. C : code.
- Public key : (n, a), Private key : b
- $M^{ab} \equiv M \mod n$
- The construction of n, a, and b comes from **number theory** (Fermat theorem, Gauss theorem)

RSA : construction of the keys

- Choose p and q prime numbers
- $n = pq$
- $\phi = (p-1)(q-1)$
- Choose a coprime with ϕ (entiers premiers entre eux)
- Choose b inverse of a modulo ϕ, which means

$$ab \equiv 1 \mod \phi \tag{2}$$

Setting up a RSA system

Exercice 5 : Building RSA I : choosing keys

- **cd** to the **./rsa** directory
- Please modify **rsa_functions.py** so that when calling **generate_rsa_keys()** from **cipher_rsa.py**, a public key and a private key are created and saved.
- You can change the prime numbers used.

Setting up a RSA system

Exercice 5 : Building RSA II : ciphering the text

- Please uncomment the end of **cipher_rsa.py** and modify **rsa_functions.py** so that when calling **cipher_rsa()** from **cipher_rsa.py**, a public key and a private key are created and the text stored in texts is coded and stored in **./crypted_messages**.

Setting up a RSA system

Exercice 5 : Building RSA III : checking that the system works by deciphering the text

- Please modify **rsa_functions.py** so that when calling **decipher_rsa()** from **decipher_known_rsa.py**, the generated public key private key are used to decipher the crypted text.

Attacking RSA

Exercice 5 : Trying to break RSA

- Modify **rsa_functions.py** so that when calling **find_private_key()** from **decipher_unknown_rsa.py the secret private key is found** from the public key and used to decipher the crypted message.
- The function to edit is **primary_decomposition.py**.

Attacking RSA

Exercice 6 : Trying to break RSA

- For which key index does the attack start to take a long time (several minutes) ?
- What happens if we write
  pow(coded_index, b, mod=n)
  instead of
  coded_indexb % phi** ?

Conclusion on RSA

It is extremely hard to break RSA if n is sufficiently large, because you need to find the decomposition of n in **prime numbers.** This is another important example of a algorithmic that is too **complex** to be solved.
In real applications p and q have several hunders of numbers and **randomly generated** (pseudo random number generation).