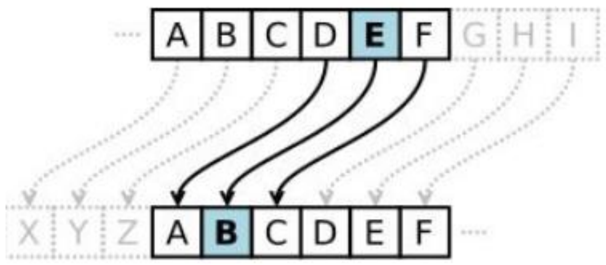
Lesson 4: Cryptographic tools/Cryptographic tools Pt. 1 (Accounts)

Historically speaking, one of the earliest examples of cryptography occurred with the Caesar Cipher. Caesar's Cipher is one of the earliest and simplest methods of cryptography. It is simply a type of substitution cipher, that is, each letter of a given text is replaced by a letter with a fixed number of positions at the bottom of the alphabet.
For example, with a shift of 1, A would be replaced by B, B would become C, and so on. The method seems to take its name from Julius Caesar, who apparently used it to communicate with his officials.

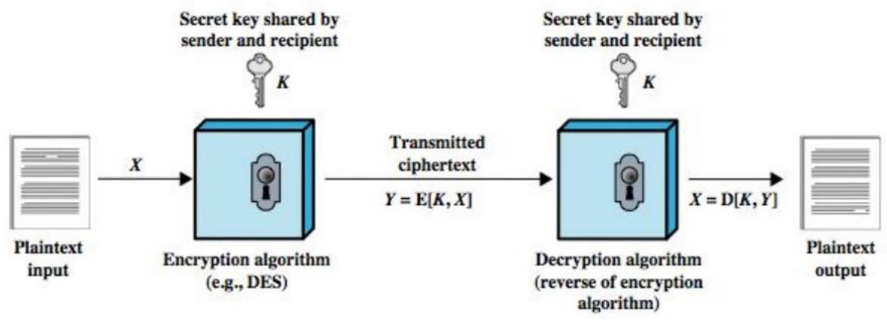
To encrypt a given text we therefore need an integer value, known as a *shift*, which indicates the number of positions at which each letter of the text has been shifted down.
Encryption can be represented using modular arithmetic, first turning letters into numbers, according to the scheme A = 0, B = 1,..., Z = 25.
This shift is decided a priori and all possible combinations are shifted.



Cryptography/encryption uses complex mathematical algorithms and digital keys to encrypt data. An encryption algorithm (cipher) and an encryption key encode data into a cipher text. Once the ciphertext is transmitted to the recipient, the same or a different key (cipher) is used to decode the ciphertext into its original value. The main problem is people who can sneak into the transmission and steal data that would not be confidential to them.
Encryption keys are the secret of data encryption. They are essentially codes and function like physical keys: only the right key unlocks the encrypted data. The generation of encryption keys can be done manually or with software that scrambles the data with an algorithm and creates an encryption key.
The two most distinct encryption methods are symmetric and asymmetric.

1) Symmetric cryptography

Symmetric key cryptography, also called private key cryptography, uses a single key to encrypt and decrypt data. To achieve secure communications, the sender and receiver must have the same key. The key provides an unbroken layer of encryption from beginning to end, using the same key for encryption and decryption keys. The single key can take the form of a password, a code or a randomly generated string of numbers. Popular examples of symmetric encryption are AES, DES and Triple DES.


Secret key shared by sender and recipient
K
Secret key shared by sender and recipient
K
X
Transmitted ciphertext
Y = E[K, X]
X = D[K, Y]
Plaintext input
Encryption algorithm (e.g., DES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

2) Asymmetric cryptography

Asymmetric key cryptography, also known as public key cryptography, uses two different keys: a public key to encrypt and a private key to decrypt. Asymmetric encryption offers increased security through verification of the origin of the data and non-repudiation (the author cannot dispute the authorship). However, it slows down the transmission process, network speed and machine performance. A popular example of asymmetric encryption is RSA.

In symmetric cryptography, there are threats/threats:
- Cryptanalysis
  - It is based on the nature of the algorithm
  - Plus some knowledge of the characteristics of plain text/plaintext
  - Even some plain-text-text sample pairs.
  - Leverages algorithm features to infer a plaintext or specific key
- Brute force attack/bruteforce attack
  - Try as many keys as possible on a cipher text until an intelligible plaintext translation is obtained

Exhaustive key search/exhaustive key search, or *brute force search*, is the basic technique of trying every possible key in turn until the correct one is identified. To identify the correct key, it may be necessary to have a plaintext and the corresponding ciphertext, or, if the plaintext has some recognizable feature, the ciphertext alone may be sufficient. Exhaustive key search can be performed on any cipher, and sometimes a weakness in the cipher's key program can help improve the efficiency of an exhaustive key search attack.

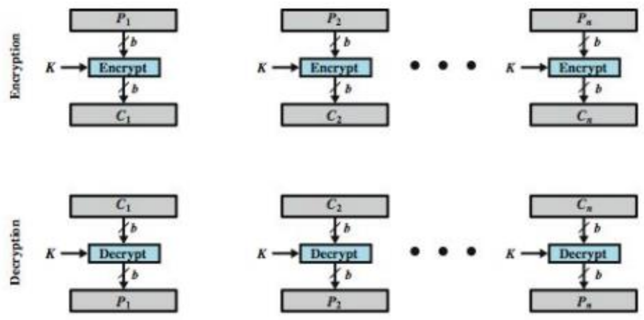
There are four popular encryption algorithms, DES/Triple-DES, AES, RSA.
- DES is the old "data encryption standard" from the 1970s. Its key size is too short for proper security (56 effective bits, this can be brute forced, as was demonstrated more than a decade ago). Also, DES uses 64-bit blocks with a 56-bit key, which raises some potential problems when encrypting several gigabytes of data with the same key (a gigabyte is not that big nowadays).
- 3DES is a trick for reusing DES implementations by cascading three DES instances (with separate keys) and using two/three unique keys. It is much more secure, but also much slower, especially in software
- AES is the successor to DES as the standard symmetric encryption algorithm for U.S. federal organizations (and as a standard for almost everyone else as well). AES accepts 128-, 192-, or 256-bit keys (128 bits is already very unbreakable), uses 128-bit blocks (so there are no problems), and is efficient in both software and hardware. It was selected through an open competition involving hundreds of cryptographers over several years. Basically, you can't get any better.

So, when in doubt, use AES. However, there still remains many widespread DES, in its Triple-DES variant.

Note that a block cipher is a box that encrypts "blocks" (128-bit data blocks with AES). When encrypting a "message" that can be longer than 128 bits, the message must be split into blocks and the way the split is done is called the mode of operation or "chaining." The naive mode (simple split) is called ECB and presents problems. The correct use of a block code is not simple and is more important than selecting between, for example, AES or 3DES.
It is precisely blocks that we are talking about when you have to cipher, specifically we have block ciphers/ ciphers and stream ciphers/ ciphers.

Block ciphers encrypt data in blocks of a predetermined length, while stream ciphers do not and encrypt plaintext one byte at a time. The two encryption approaches, therefore, vary widely in terms of implementation and use cases.
Block ciphers convert plaintext data into cipher text in fixed block sizes. The block size generally depends on the encryption scheme and is usually in eighths (64- or 128-bit blocks). If the length of the plaintext is not a multiple of 8, the encryption scheme uses padding to guarantee complete blocks. For example, to perform 128-bit encryption on a 150-bit plaintext, the encryption scheme provides two blocks, one with 128 bits and one with the remaining 22 bits. Redundant bits are added to the last block to make the entire block equal to the ciphertext block size of the encryption scheme.

They use symmetric keys and algorithms to perform data encryption and decryption, but they also require an initialization vector (IV) to function. An initialization vector is a pseudorandom or random sequence of characters used to encrypt the first character block of the plaintext block. The resulting ciphertext for the first block of characters serves as the initialization vector for subsequent blocks. Therefore, the symmetric cipher produces a unique block of ciphertext for each iteration, while the IV is transmitted along with the symmetric key and does not require encryption.
Examples of algorithms that use them: AES/DES

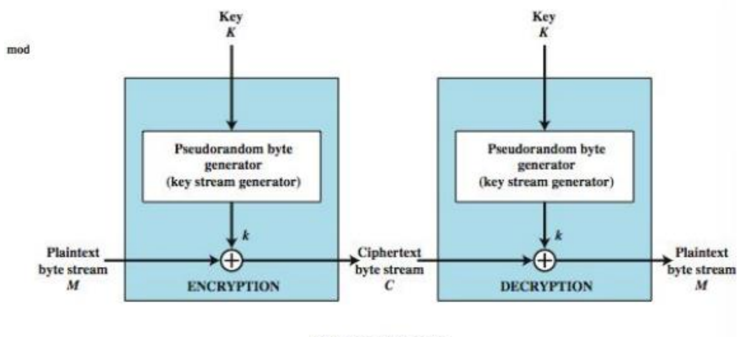


(a) Block cipher encryption (electronic codebook mod

A stream cipher encrypts a continuous string of binary digits by applying time-varying transformations to the plaintext data. Therefore, this type of encryption works bit by bit, using streams of keys to generate ciphertext for plaintext messages of arbitrary length. The cipher combines a key (128/256 bits) and a nonce digit (64-128 bits) to produce the key stream, a pseudorandom number subjected to XOR with the plaintext to produce the ciphertext. While the key and nonce can be reused, the key stream must be unique for each encryption iteration to ensure security. Stream ciphers achieve this by using feedback shift registers to generate a unique nonce (number used only once) to create the key stream.

Encryption schemes using stream ciphers are less likely to propagate system-level errors, since an error in the translation of one bit generally does not affect the entire block of plaintext. Stream ciphers also occur in a linear and continuous manner, making them easier and faster to implement. On the other hand, stream ciphers lack diffusion, as each plaintext digit is mapped to a ciphertext output. They also do not validate authenticity, making them vulnerable to advertisements. If hackers breach the encryption algorithm, they can enter or modify the encrypted message without detection. Stream ciphers are mainly used to encrypt data in applications where the amount of plaintext cannot be determined and in low-latency use cases.

Examples of algorithms using this principle: RC4 (another algorithm widely used in cryptography)



(b) Stream encryption

Let us analyze the following example:

- Substitution cipher: the alphabet is shifted by one set of characters. Clearly it is very basic and not secure. An example is precisely Caesar's cipher.
  For example, having the text
  "QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD"
  The solution is to try all possible shifts/all alphabet combinations, through cryptanalysis and brute force.
  The solution is: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."

XOR is also often used in cryptographic algorithms, an operation that between two numbers:

- Returns 1 when these are different
- Returns 0 when these are equal
- It is represented by the symbol "^"

Examples:

enc_message = clear_message ^ key
clear_message = enc_message ^ key
key = clear_message ^ enc_message

The XOR has some properties to consider:

- It is commutative
- It is associative
- All XORs with 0 result in the other term: *a ^ 0 = a*
- All XORs with the same term give zero: *a ^ a = 0*

XOR is used between a key and a message:

- Often len(key) << len(message)
- "Let's repeat the key" in the message

Example:

clear_message = "THIS IS A MESSAGE"
key = "YOU"

T	H	I	S		I	S		A		M	E	S	S	A	G	E
84	72	73	83	32	73	83	32	65	32	77	69	83	83	65	71	69

T	H	I	S		I	S		A		M	E	S	S	A	G	E
Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O

Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O
89	79	85	89	79	85	89	79	85	89	79	85	89	79	85	89	79

The XOR between two integers is the result of the XOR of their binary representation:

msg	84	72	73	83	32	73	83	32	65	32	77	69	83	83	65	71	69
key	89	79	85	89	79	85	89	79	85	89	79	85	89	79	85	89	79
enc	13	7	28	10	111	28	10	111	20	121	2	16	10	28	20	30	10

- 84 = 1010100
- 89 = 1011001
- 13 = 0001101

Interestingly, if:

- The key is the same size as the message
- The key is kept secret and generated in a truly random way

Then the XOR cipher is certainly impossible to crack. This is known as a one-time pad. However, a simple XOR should not be used in production because the key length must be too long to be practical.

XOR is an inexpensive way to encrypt data with a password. There are two main types:

- Single-byte XOR encryption
    - Single-byte XOR encryption is trivial to force, since there are only 255 key combinations to try.
- Multi-byte XOR encryption
    - Multi-byte XOR encryption becomes exponentially more difficult the longer the key, but if the encrypted text is long enough, character frequency analysis is a viable method for finding the key. Character frequency analysis involves dividing the ciphertext into groups based on the number of characters in the key. These groups are then forced using the idea that some letters appear more frequently in the English alphabet than others.

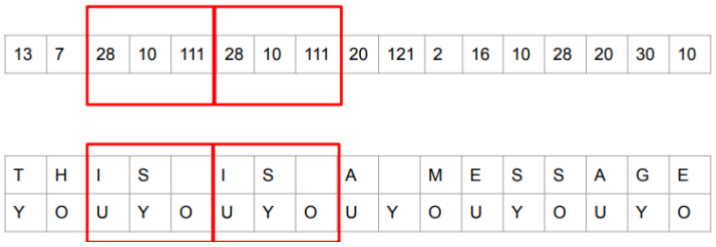
The Kasiski method is a cryptanalytic method for attacking the *Vigenère cipher.*
It is the simplest of the polyalphabetic ciphers. It is based on the use of a verse to control the alternation of substitution alphabets. The method can be considered a generalization of Caesar's cipher; instead of always shifting the letter to be ciphered by the same number of places, it is shifted by a variable but repeated number of places, determined on the basis of a keyword, to be agreed upon between sender and receiver, and to be written repeatedly under the message, character by character; the key was also called a worm, for the reason that, being generally much shorter than the message) and ciphers similar to it.

The cipher text is obtained by shifting the clear letter by a fixed number of characters, equal to the ordinal number of the corresponding letter of the worm. In fact, an arithmetic sum is performed between the ordinal of the clear (A = 0, B = 1, C = 2...) and the ordinal of the worm; if the last letter, Z, is passed, it starts again from A, according to the logic of finite arithmetic.
The advantage over monoalphabetic ciphers (such as Caesar's cipher or those by substitution of letters for symbols/other letters) is obvious: the text is encrypted with n cipher alphabets. In this way, the same letter is encrypted (if repeated consecutively) n times; thus, this makes cryptanalysis of the ciphertext more complex.

Returning to the Kasiski method, Major Kasiski (its creator) noted that often in a Vigénère cryptogram one can see identical sequences of characters placed at a certain distance from each other; this distance may, with some probability, correspond to the length of the key, or a multiple thereof.

Generally the same letter with the Vigénère cipher is ciphered differently in its various occurrences, as befits polyalphabetic ciphers, but if two letters of the plaintext are placed at a distance equal to the length of the key (or a multiple thereof), this causes them to be ciphered in the same way.

By identifying all the repeated sequences (which is frequently the case in a long text), it can almost certainly be deduced that the key length is the greatest common divisor among the distances between repeated sequences, or at most a multiple thereof. Next, an example.
Knowing the length n of the key allows the encrypted message to be traced back to n interleaved messages encrypted with an easily decipherable Caesar cipher.



13	7	28	10	111	28	10	111	20	121	2	16	10	28	20	30	10

T	H	I	S		I	S		A		M	E	S	S	A	G	E
Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O

Exercises Lesson 4

1) Julius: Text, help and solution

Text

Julius,Q2Flc2FyCg==

World of Cryptography is like that Unsolved Rubik's Cube, given to a child who has no idea about it. A new combination at every turn.
Can you solve this one, with weird name?
ciphertext: fYZ7ipGIjFtsXpNLbHdPbXdaam1PS1c5lQ==

Aid:

1) The challenge description contains two "secret" messages. Both end with ==. Does this recall you an encoding strategy?
2) This is a Caesar cipher. A brute force is enough to break it.

Solution

```
# in the first line of the description contains a hint→ Julius,Q2Flc2FyCg==
# since it ends with ==, the first hypothesis is that this is a base64 encoding
# let's decode it!
import base64
enc_b64 = 'Q2Flc2FyCg=='
#we define a function. It might be helpful in future
def base64tostring(text):
   return base64.b64decode(text).decode('utf-8', errors="ignore")

print(f "Decoding=\t{base64tostring(enc_b64)}")
# the hints says "caesar," and we know a famous cipher with this name. We can apply the reverse to the
# ciphered text given in description
```

```
puzzle = 'fYZ7ipGIjFtsXpNLbHdPbXdaam1PS1c5lQ=='
print("The length of the puzzle is:\t", len(puzzle))
# the length is a multiple of 4, and the alphabet seems too regular (no punctuation).
# we can think that this is a base64 encoded string
puzzle_dec = base64tostring(puzzle)
print("Decoded puzzle:", puzzle_dec)


# now we have a lot of no sense characters. We can try to apply the caesar cipher
def caesar_cracker(text, from_ = -30, to_=+30):
  for i in range(from_, to_): #keys [-30, 30]
    #decode
    curr_step = ' '.join([chr(ord(c) + i) for c in text])
    #print
    print(f"Step={i}\t{curr_step}")
caesar_cracker(puzzle_dec)
## we look for some readable flags. We find one at step -24
#FLAG: ecCTF3T_7U_BRU73?!
```

<u>Other solution</u>

```
import base64
ciphertext = 'fYZ7ipGIjFtsXpNLbHdPbXdaam1PS1c5lQ=='
#We can see the string is a base64 one, so we define a function to decode it

def base64decode(text):
  return str(base64.b64decode(text).decode('ascii', 'ignore'))
#Fundamental to put "str" and "ignore" to read all the characters without having problems

decoded=base64decode(ciphertext)
print(decoded)
# As of now, we have this kind of string (with the b that stays for binary, as visible):
# b'}\x86{\x8a\x91\x88\x8c[l^\x93KlwOmwZjmOKW9\x95' (25 characters in total)
# Let's try to implement a bruteforce algorithm to decrypt it, making a shift of 24 to get the right one
flag = ' '
for i in range(len(decoded)):
  code = ord(decoded[i]) - 24
  flag += chr(code)
print(flag)
```

A possible general implementation of the *Caesar cipher* is as follows, taking a generic text and a custom shift to be inserted; it distinguishes between uppercase and lowercase letters by making a shift equal to all the letters of the alphabet starting with the character in that specific position.

```
def caesarCipher(plainText, shift):
  cipherText = ""
  for i in range(len(plainText)):
    if plainText[i].isupper():
            cipherText += chr((ord(plainText[i]) + shift - 65) % 26 + 65)
#65 is the ASCII value of 'A' and 26 is the number of letters in the alphabet
        else:
            cipherText += chr((ord(plainText[i]) + shift - 97) % 26 + 97)
#97 is the ASCII value of 'a' and 26 is the number of letters in the alphabet
    return cipherText
```

Decoding does not change with respect to this function, since it literally subtracts the shift made:

```
def caesarCipherDecoder(cipherText, shift):
    plainText = ""
    for i in range(len(cipherText)):
        if cipherText [i].isupper():
            plainText += chr((ord(cipherText [i]) - shift - 65) % 26 + 65)
#65 is the ASCII value of 'A' and 26 is the number of letters in the alphabet
        else:
            plainText += chr((ord(cipherText [i]) - shift - 97) % 26 + 97)
#97 is the ASCII value of 'a' and 26 is the number of letters in the alphabet
    return cipherText
```

2) I Agree: Text, aid and solution

Text

Crack the cipher: vhixoieemksktorywzvhxzijqni
Your clue is, "caesar is everything. But he took it to the next level."

Help:

1) You might try with a Caesar ... but it is not enough.
   The challenge description tells that this is "the next level of Caesar cipher." What is it? Have a look on Google, if you find the cipher, you solve the exercise.

Solution

```
# The description says: "caesar is everything."
# it is a caesar cipher, and we need to crack it as in the previous exercise

puzzle = 'vhixoieemksktorywzvhxzijqni'

# Specifying the steps of the for loop, where \t means "tab"
def caesar_cracker(text, from_ = -30, to_=+30):
    for i in range(from_, to_): #keys [-30, 30]
        # decode
        curr_step = ' '.join([chr(ord(c) + i) for c in text])
        # print
        print(f"Step={i}\t{curr_step}")

caesar_cracker(puzzle)
# I don't see any proper flag. We need to find another way
#The description says that it is the "next level" of caesar.
# After some investigation, we can find that the evolution of a caesar cipher is the vigenere cipher.
# However, the vigenere also requires a key.
# Google can help us! There are some online bruteforce services for these kinds of ciphers.
# https://www.guballa.de/vigenere-solver
# The flag is reached: theforceisstrongwiththisone. The key is "caesar" ... as the hint suggested
```

The following is a possible implementation made by me in Python, interesting at the code level and for understanding Vigénère's reasoning:

```
#The hint "to the next level" says everything
#cause it refers to the Vigénère cipher, which is a Caesar cipher that uses an entire keyword
text='vhixoieemksktorywzvhxzijqni'

#The key word is "caesar" and we define an algorithm to use the Vigénère cipher
def vigenere(text, key):
   key = key * (len(text) // len(key) + 1) #repeat the key word and floor "//" division between text/text+1
   return ' '.join([chr((ord(text[i]) - ord(key[i])) % 26 + ord('a')) for i in range(len(text))])
   #the algorithm is the same as the Caesar cipher but we use the key word instead of a single letter
   #so, we make a shift subtracting from the key word from the text modulo 26 to keep the letters in the
   #alphabet and adding the ascii code of 'a' to get the right letter; we do this for every letter in the text
print (vigenere(text, 'caesar'))
#the result is "theforceisstrongwiththisone"
```

A general coding implementation of Vigenere that resembles that of Caesar Cipher is as follows:

```
#Vigenere Cipher function
def vigenereCipher(plainText, key):
   cipherText = ""
   for i in range(len(plainText)):
      if plainText[i].isupper():
         cipherText += chr((ord(plainText[i]) + ord(key[i % len(key)]) - 65) % 26 + 65) #here we modulo also
the key length and this is the uppercase implementation
      else:
         cipherText += chr((ord(plainText[i]) + ord(key[i % len(key)]) - 97) % 26 + 97) #here we modulo also
the key length and this is the lowercase implementation
   return cipherText
```

Decoding, as before, literally subtracts the position of the length of the key realized shift according to the uppercase/lowercase letter:

```
def vigenereCipherDecoder(cipherText, key):
   plainText = ""
   for i in range(len(cipherText)):
      if cipherText[i].isupper():
         plainText += chr((ord(cipherText[i]) - ord(key[i % len(key)]) - 65) % 26 + 65) #65 is the ASCII value
of 'A' and 26 is the number of letters in the alphabet
      else:
         plainText += chr((ord(cipherText[i]) - ord(key[i % len(key)]) - 97) % 26 + 97) #97 is the ASCII value
of 'a' and 26 is the number of letters in the alphabet
```

```
    return plainText
```

3) Alphabet Soup: Text, help and solutions.

Text:
(A file "encrypted.txt" is given with the following text):
MKXU IDKMI DM BDASKMI NLU XCPJNDICFQ! K VDMGUC KW PDT GKG NLKB HP LFMG DC TBUG PDTC CUBDTCXUB. K'Q BTCU MDV PDT VFMN F WAFI BD LUCU KN KB WAFI GDKMINLKBHPLFMGKBQDCUWTMNLFMFMDMAKMUNDDA

Aid:

1) Okay, in this exercise we need to do some educated guesses, a.k.a., cryptoanalysis.
   You might want to: i) get the frequency of the characters and ii) map it somehow (remember that this is an English text).
2) Here we report some mappings:
   K -> i
   F -> a
   P -> y

Solution

```
# we only have an encrypted message
puzzle = "MKXU IDKMI DM BDASKMI NLU XCPJNDICFQ! K VDMGUC KW PDT GKG NLKB HP LFMG DC TBUG PDTC CUBDTCXUB. K'Q BTCU MDV PDT VFMN F WAFI BD LUCU KN KB WAFI GDKMINLKBHPLFMGKBQDCUWTMNLFMFMDMAKMUNDDA"

# since I do not have any clue, I try to use the cryptanalysis strategy
# in the ciphers, in general, each letter is associated with a given alphabet
# we can try to find associations.

#the first step is to see the frequency of each character with this function
chr2freq = {}
For c in puzzle:
    if c not in chr2freq:
        chr2freq[c] = 1
    else:
        chr2freq[c] += 1

# sort the dictionary by value in descending order and store it in a list of tuples (key, value)
sorted_x = sorted(chr2freq.items(), key=lambda kv: kv[1], reverse = True)
#lambda is used to create an anonymous function, in this case, it is used to sort the dictionary by value
#taking each time the first element as seen by kv
print(sorted_x)
```

```
#This here prints: [(' ', 30), ('D', 17), ('M', 15), ('K', 14), ('U', 11), ('B', 10), ('C', 10), ('F', 9), ('N', 8), ('I', 7),
('L', #7), ('G', 7), ('T', 7), ('P', 6), ('A', 5), ('W', 4), ('X', 3), ('Q', 3), ('V', 3), ('H', 2), ('S', 1), ('J', 1), ('!', 1), ('.',
1)]

# hypothesis 1: the text is english written
# we can find online what are the most used english characters
# e.g., http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html
# we see that the 'K' is "alone," and we can think to
# K = I
voc = {'K': 'i'}
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle) #this is a list comprehension that substitutes the
character in the string with the one in the guess dictionary
print(voc, '\n' ,dec)
# then there is an 'i'Q', which is an M
voc['Q'] = 'm'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# F -> 'a'
voc['F'] = 'a'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

## the semilast word contains four letters, and the third character
# is an 'a'. This word could be flagged
voc['W'] = 'f'
voc['A'] = 'l'
voc['I'] = 'g'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)
# not a lot of info...
# however, there is a word with GiG ... G must be a 'D'
voc['G'] = 'd'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# then there is a sentence with "if PDT did"
# PDT could be "you," a likely word with letters not used yet
voc['P'] = 'y'
voc['D'] = 'o'
voc['T'] = 'u'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# slightly better. The second word is goiMg
# M->n
voc['M'] = 'n'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
```

```
print(voc, '\n' ,dec)

# back on the second sentence
# if you did NLiB Hy ... seems "if you did this by"
voc['N'] = 't'
voc['L'] = 'h'
voc['B'] = 's'
voc['H'] = 'b'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

#the fourth word must be "solving"
voc['S'] = 'v'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)


#fifth word is "the"
voc['U'] = 'and'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

#then, "I wonder if you ..."
voc['V'] = 'w'
voc['C'] = 'r'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

#ready to conclude. we can see the flag ... but let's finish the job
#niXe -> nice
voc['X'] = 'c'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)


#the last word of the first sentence is cryptogram
voc['J'] = 'p'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)
## we did it
# nice going on solving the cryptogram!
# i wonder if you did this by hand or used your resources.
# i'm sure now you want a flag so here it is
# flag doingthisbyhandismorefunthananonlinetool
```

Wanting to do things that are not manual, tools widely used by online challenges in replacement ciphers (and, for this challenge, by the rest of the Web) are the following:
https://www.quipqiup.com/

https://www.guballa.de/substitution-solver
By literally entering the text, the first time the flag and the translated text come up; therefore, such resolution tools are used.

4) DES Solver: Text, help and solution.
(*useful implementation/visual exercise; quite complex and more than the exams themselves*)

More solution references at the link: https://ctftime.org/task/5405

Text

Larry is working on an encryption algorithm based on DES.
He has not worked out all the kinks yet, but he thinks it works.
Your job is to confirm that you can decrypt a message, given the algorithm and parameters used.
His system works as follows:

1. *Choose a plaintext that is divisible into 12bit 'blocks'*
2. *Choose a key at least 8bits in length*
3. *For each block from i=0 while i<N perform the following operations.*
4. *Repeat the following operations on block i, from r=0 while r<R*
5. *Divide the block into 2 6bit sections Lr,Rr*
6. *Using Rr, "expand" the value from 6bits to 8bits.*
   Do this by remapping the values using their index, e.g.
   1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6
7. *XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back to the beginning if necessary.*
8. *Divide the result into 2 4bit sections S1, S2*
9. *Calculate the 2 3bit values using the two "S boxes" below, using S1 and S2 as input respectively.*

   S101234567
   010101000111001110011100O
   1001100110010000111101011

   S201234567
   010000011010111100101101O
   110101100011111001000110O

10. *Concatenate the results of the S-boxes into 1 6bit value*
11. *XOR the result with Lr*
12. *Use Rr as Lr and your altered Rr (result of previous step) as Rr for any further computation on block i*
13. *increment r*

He has encrypted a message using Key="Mu", and R=2. See if you can decipher it into plaintext.
Submit your result to Larry in the format Gigem{plaintext}.
Binary of ciphertext: 01100101 00100010 10001100 01011000 00010001 10000101

Aid:

1) This is a tough one! You need to carefully reverse the process presented in the exercise description.

Here a couple of functions written for you:

```
def rule9b0(b):
    #get indexed
    row = int(b[0])
    col = int(b[1:], 2)

    matrix = [['101','010','001','110','011','100','111','000'],
        ['001',  '100','110','010','000','111','101',’011']
    ]

    return matrix[row][col]

def rule9b1(b):
    #get indexed
    row = int(b[0])
    col = int(b[1:], 2)

    matrix = [['100','000','110','101','111','001','011','010'],
        ['101','011','000','111','110','010','001','100']]

    return matrix[row][col]
```

2) This is my encryption algorithm.

```
def rule9b0(b):
   #get indexed
   row = int(b[0])
   col = int(b[1:], 2)

   matrix = [['101','010', '001', '110', '011', '100', '111', '000'],
      ['001','100','110','010','000','111','101','011']]
   return matrix[row][col]

def rule9b1(b):
   #get indexed
   row = int(b[0])
   col = int(b[1:], 2)

   matrix = [['100','000','110','101','111','001','011','010'],
      ['101','011','000','111','110','010','001','100']]

   return matrix[row][col]

# we need to convert the text into bits
def string2binary(text):
   return ' ' '.join(f"{ord(c):08b}" for c in text)
```

```
def binary2string(text):
    return ' ' '.join(f"{ord(c):08b}" for c in text)

def splitblock(block):
    Lr = block[:6]
    Rr = block[6:]
    return Lr, Rr

def expand_miniblock(b):
    return b[0] + b[1] + b[3] + b[2] + b[3] + b[2] + b[4] + b[5]

def xor(a, b):
    res = int(a, 2) ^ int(b, 2)

    return f"{res:08b}"

def encrypt(text, key, R):
    text_encr = ' '

    #Rule 1.
    text_bin = string2binary(text)
    if (len(text_bin) % 12 != 0):
        raise Exception(f'Rule 1 not respected.')

    #Rule 2
    key_bin = string2binary(key)
    if (len(text_bin) < 8):
        raise Exception('Rule 2 not respected')

    #rule3
    #we have some blocks ...
    for bnum in range(len(text_bin) // 12):
        i = bnum

        #define the block
        from_ = 0 + 12*bnum
        to_ = 12 * (bnum + 1)
        block = text_bin[from_:to_]

        #rule4
        for r in range(R):
            #rule5
            Lr, Rr = splitblock(block)

            #rule6
            Rr_expanded = expand_miniblock(Rr)
            #Rule7
            curr_key = key_bin[(i* R + r) : ((i* R + r)+8)]
            Rr_exp_xor_key = xor(Rr_expanded, curr_key)
            #Rule8
            Rr_exp_xor_key_0 = Rr_exp_xor_key[:4]
            Rr_exp_xor_key_1 = Rr_exp_xor_key[4:]

            #Rule9
            Rr_exp_xor_key_0_conv = rule9b0(Rr_exp_xor_key_0)
            Rr_exp_xor_key_1_conv = rule9b1(Rr_exp_xor_key_1)

            #Rule10
            Rr_sboxes = Rr_exp_xor_key_0_conv + Rr_exp_xor_key_1_conv
            if len(Rr_sboxes) != 6:
                raise Exception("Error on Rule 10")

            #Rule11
            Rr_alt = xor(Lr, Rr_sboxes)[2:]

            #Rule12
            block = Rr + Rr_alt

            #Rule13
            #end of step

        #append the result.
        text_encr += block

    return text_encr
```

Solution

```
#the description give us a simplified version of DES.
#the first idea is to reconstruct the process

#Rule 1. Choose a plaintext that is divisible into 12bit 'blocks'
#Rule 2. Choose a key at least 8bits in length.
#Rule 3. For each block from i=0 while i<N perform the following operations.
#Rule 4. Repeat the following operations on block i, from r=0 while r<R
#Rule 5. Divide the block into 2 6bit sections Lr,Rr
#Rule 6. Using Rr, "expand" the value from 6bits to 8bits.
# Do this by remapping the values using their index, e.g..
# 1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6
#Rule 7. XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back to the
# beginning if necessary.
#Rule 8. Divide the result into 2 4bit sections S1, S2
#Rule 9. Calculate the 2 3bit values using the two "S boxes" below, using S1 and S2 as input respectively.
   #
```

```
            #          S101234567
            #          0101010001110011100111000
            #          1001100110010000111101011
   #
            #          S201234567
            #          0100000110101111001011010
            #          1101011000111110010001100
#Rule10. Concatenate the results of the S-boxes into 1 6bit value
#Rule11. XOR the result with Lr
#Rule12. Use Rr as Lr and your altered Rr (result of previous step) as Rr for any further computation on
block i
#Rule13 increment r
# Let's find the other two 3-bit values in a matrix for the S1 and S2 block of rule 9
def rule9b0(b):
   # get indexed
   row = int(b[0])
   col = int(b[1:], 2)
   matrix = [['101','010', '001', '110', '011', '100', '111', '000'],
      ['001','100','110','010','000','111','101','011']]
   return matrix[row][col]

def rule9b1(b):
   # get indexed
   row = int(b[0])
   col = int(b[1:], 2)
   matrix = [['100','000','110','101','111','001','011','010'],
      ['101','011','000','111','110','010','001','100']]
   return matrix[row][col]

# We need to convert the text into bits (8 binary bits, hence the "08b" in the code)
def string2binary(text):
   return ' ' '.join(f"{ord(c):08b}" for c in text)
# Similarly, we convert the binary to the string (similar reasoning with the same code, but in reverse)
def binary2string(text):
   return ' ' '.join(f"{ord(c):08b}" for c in text)

# Made for rule 5 (Divide the block into 2 6bit sections Lr,Rr)
# To do that, we just iterate in row the blocks, iterating up until 6 for the first one
# and, for the second one, starting from the 6th block all the way to the end
def splitblock(block):
   Lr = block[:6]
   Rr = block[6:]
   return Lr, Rr

# Made for rule 6
# Using Rr, "expand" the value from 6bits to 8bits. Do this by remapping the values using their index, e.g.
# 1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6. In the code we use this exact reasoning
def expand_miniblock(b):
    return b[0] + b[1] + b[3] + b[2] + b[3] + b[2] + b[4] + b[5]

# XOR function already defined for rule 7→ XOR the result of this with 8bits of the Key beginning with
# Key[iR+r] and wrapping back to the beginning if necessary.

 # To accomplish the function, we take "a" and "b" in binary and return the result XORed in string form
def xor(a, b):
    res = int(a, 2) ^ int(b, 2)
    return f"{res:08b}"




# The whole function of encryption (the decryption follows up right ahead this one); R stays for
"rounds",
# so it does around a binary
def encrypt(text, key, R):
    text_encr = ' '

#Rule 1→ Choose a plaintext that is divisible into 12bit 'blocks' (so we use the modulo, as done with
    # even numbers usually)
    text_bin = string2binary(text)
    if (len(text_bin) % 12 != 0):
        raise Exception(f'Rule 1 not respected.')

    #Rule 2→ Choose a key at least 8 bits in length (so just checking the length with "len")
    key_bin = string2binary(key)
    if (len(text_bin) < 8):
        raise Exception('Rule 2 not respected')

    # Rule 3→ For each block from i=0 while i<N perform the following operations.
    # What we are doing here with the double slash (//) is the "floor" division, so
    # divides the first number by the second number and rounds the result down to the nearest integer
    # (or whole number). We make this to have 12-bit divisible blocks
    for bnum in range(len(text_bin) // 12):
        i = bnum
# define the block, with a start index for a 12-bit block and the end is similar but adding one to the sum
        from_ = 0 + 12*bnum
        to_ = 12 * (bnum + 1)
        block = text_bin[from_:to_] # So the block will iterate all the way in the text
            # Rule 4→ Repeat the following operations on block i, from r=0 while r<R

  for r in range(R):
            # Rule 5→ Divide the block into 2 6bit sections Lr,Rr
            Lr, Rr = splitblock(block)

            # Rule 6→ Using Rr, "expand" the value from 6bits to 8bits. Do this by remapping the values using
            # their index - e.g., 1 2 3 4 5 6 -> 1 2 4 3 5 6
            Rr_expanded = expand_miniblock(Rr)

            # Rule 7→ XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back
            # the beginning if necessary.
            curr_key = key_bin[(i* R + r) : ((i* R + r)+8)]
            Rr_exp_xor_key = xor(Rr_expanded, curr_key)

            # Rule 8→ Divide the result into two 4-bit sections S1, S2
            Rr_exp_xor_key_0 = Rr_exp_xor_key[:4]
            Rr_exp_xor_key_1 = Rr_exp_xor_key[4:]

            # Rule9→ Calculate the two 3-bit values using the two "S boxes" below, using S1 and S2 as input
            #respectively.
            Rr_exp_xor_key_0_conv = rule9b0(Rr_exp_xor_key_0)
            Rr_exp_xor_key_1_conv = rule9b1(Rr_exp_xor_key_1)

            #Rule10→ Concatenate the results of the S-boxes into one 6bit value
            Rr_sboxes = Rr_exp_xor_key_0_conv + Rr_exp_xor_key_1_conv

      if len(Rr_sboxes) != 6:
             raise Exception("Error on Rule 10")

            #Rule11→ XOR the result with Lr
            Rr_alt = xor(Lr, Rr_sboxes)[2:]

            #Rule12→ Use Rr as Lr and your altered Rr (result of previous step) as Rr for any further
            #computation on block i
            block = Rr + Rr_alt
            #end of step - Rule13→ Increment R (already incremented because of the for lop nature)
        #append the result.
        text_encr += block
    return text_encr

# solution
def decrypt(text, key, R):
    text_dec = ' '

    #the text is already in the bit format.
    #we only need to convert the key
    text_bin = text
    key_bin = string2binary(key)
    if (len(text_bin) < 8):
        raise Exception('Rule 2 not respected')

    #like the previous cycle, we need to iterate over the blocks.
    #since the blocks are independent between each other, we can use the
    #same order of the encryption algorithm
    for bnum in range(len(text_bin) // 12):
        i = bnum

        #define the block
        from_ = 0 + 12*bnum
        to_ = 12 * (bnum + 1)
        block = text_bin[from_:to_]

        #we now need to reverse the rule4 loop
        #since this time the results obtained in a specific round affect the
        #next one, we use the reverse order
        #our goal is to retrieve the original Lr and Rr
        for r in range(R-1, -1, -1):
            #in the first round we obtain the components
            # block = Lr + Rr
            Rr, Rr_alt = splitblock(block)

            #to reverse Rule11 we can use the xor properties
            #e.g.: A xor B = C, C xor B = A
            #however, we need to have one of the components at least (A or B) since
            # we have C
            # N.B. we have something useful. Which is Rr.
            # We know half of the info! we can easily obtain Rr_sboxes
            # compute from rule6 to rule10, where Lr is not involved at all
            # Rule 6
            # Using Rr, "expand" the value from 6bits to 8bits. Do this by remapping the values using their
            # index, e.g., 1 2 3 4 5 6 -> 1 2 4 3 5 6
            Rr_expanded = expand_miniblock(Rr)
            #Rule 7→ XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping
back
            # the beginning if necessary.
            curr_key = key_bin[(i* R + r) : ((i* R + r)+8)]
            Rr_exp_xor_key = xor(Rr_expanded, curr_key)
            #Rule 8→ Divide the result into 2 4bit sections S1, S2
            Rr_exp_xor_key_0 = Rr_exp_xor_key[:4]
            Rr_exp_xor_key_1 = Rr_exp_xor_key[4:]
            #Rule9→ Calculate the 2 3bit values using the two "S boxes" below, using S1 and S2 as input
            #respectively.
            Rr_exp_xor_key_0_conv = rule9b0(Rr_exp_xor_key_0)
            Rr_exp_xor_key_1_conv = rule9b1(Rr_exp_xor_key_1)
            Rr_sboxes = Rr_exp_xor_key_0_conv + Rr_exp_xor_key_1_conv
            #Rule10→ Concatenate the results of the S-boxes into 1 6bit value
            if len(Rr_sboxes) != 6:
             raise Exception("Error on Rule 10")

            #we can finally obtain Lr
            Lr = xor(Rr_alt, Rr_sboxes)
            Lr = Lr[2:]
```

```
            block = Lr + Rr

        #obtain the new block
        new_block = Lr + Rr
        #print(new_block)
        # raise Exception('# DEBUG: ')

        #append the result.
        text_dec += new_block

    #Convert from 8digit bits into the integer, and then
    #in the ascii representation
    res = ' '
    for i in range(len(text_dec) // 8):
        res += chr(int(text_dec[(i * 8): ((i+1) * 8)] ,2))
    print(res)

#print(encrypt('abc', 'Mu', 2))
puzzle = "0110010100100010100011000101100000001000110000101"
key_ex = 'Mu'
R_ex = 2
#decrypt(puzzle, key_ex, R_ex)
#flag: Min0n!
decrypt(encrypt('Min0n!', 'Mu', 2), 'Mu', 2) #driver code example
```

Lesson 5: Cryptographic tools/Cryptographic tools Pt. 2 (Accounts)

Another type of threat that exists to data is the lack of <u>message/message authentication</u>. In this case, the user is unsure of the author of the message. Message authentication can be provided by cryptographic techniques using secret keys as in the case of encryption.
Requirements for an authentication mechanism:

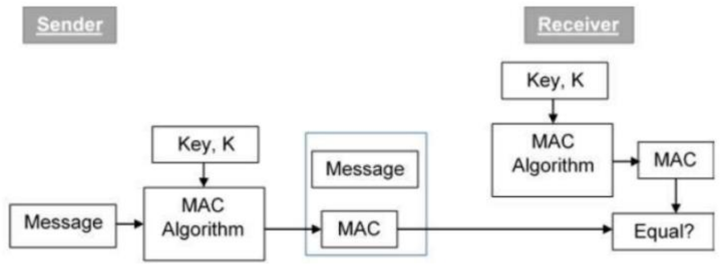
- Disclosure: Means releasing the contents of the message to someone who does not have an appropriate cryptographic key.
- Traffic analysis: Determining the traffic pattern through connection duration and frequency of connections between different parties.
- Deception: Adding out-of-context messages from a fraudulent source into a communication network. This leads to distrust between the communicating parties and can also cause the loss of critical data.
- Content editing: Editing the content of a message. This includes entering new information or deleting/changing existing information.
- Changing the sequence: Changing the order of messages between parties. This includes inserting, deleting, and reordering messages.
- Timing modification: Includes replay and delay of messages sent between parties. This also stops session tracking.
- Source rejection: When the source denies being the originator of a message.
- Destination rejection: When the recipient of the message denies receipt.

In general:

- Protects against active attacks
- Verifies the authenticity of the received message
  - Unaltered content
  - From an authentic source
  - Timely and in the correct sequence
- Can use conventional encryption
  - Only the sender and the receiver have the necessary key

- Or a separate authentication mechanism
  - Adding the authentication tag to the plaintext message

The MAC algorithm is a symmetric key cryptographic technique for providing message authentication. To establish the MAC process, the sender and receiver share a symmetric key K.
In essence, the MAC is an encrypted checksum generated on the underlying message and sent along with a message to ensure its authentication.
The process of using MAC for authentication is illustrated in the following image.



Let us now try to understand the whole process in detail:

- The sender uses a publicly known MAC algorithm, enters the message and the secret key K, and produces a MAC value.
- Like hashing, the MAC function compresses an arbitrarily long input into an output of fixed length. The main difference between hash and MAC is that MAC uses the secret key during compression.
- The sender forwards the message along with the MAC. It is assumed here that the message is sent in plaintext, since this is to provide authentication of the message origin and not confidentiality. If confidentiality is required, the message must be encrypted.
- When it receives the message and MAC, the receiver enters the received message and the shared secret key K into the MAC algorithm and computes the MAC value again.
- The receiver now verifies the equality between the MAC just computed and the MAC received from the sender. If they match, the receiver accepts the message and makes sure it was sent by the intended sender.
- If the calculated MAC does not match the one sent by the sender, the recipient cannot determine whether it is the message that has been altered or whether it is the origin that has been forged. Ultimately, the recipient confidently assumes that the message is not authentic.

Limitations of MAC

There are two main limitations of MAC, both due to the symmetrical nature of its operation.

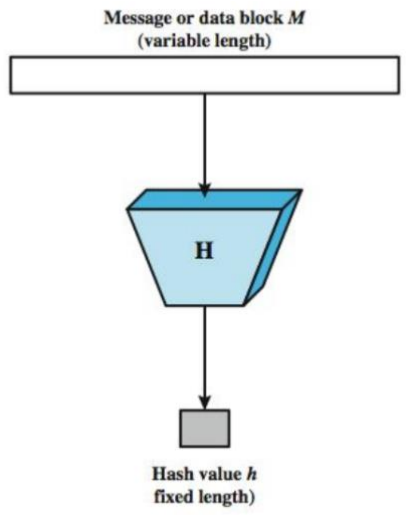
1) Establishing a shared secret.

- It can provide message authentication between predetermined legitimate users who have a shared key.
- This requires the creation of a shared secret before the MAC is used.

2) Inability to provide non-repudiation

- Nonrepudiation is the assurance that the originator of a message cannot negate previously sent messages and commitments or actions.
- The MAC technique does not provide a non-repudiation service. If the sender and receiver are involved in a dispute over the origin of the message, MACs cannot provide proof that the message was actually sent by the sender.
- Although no third party can calculate the MAC, the sender could deny having sent the message and claim that the recipient forged it, since it is impossible to determine which of the two parties calculated the MAC.

Similarly, secure hash algorithms, also known as SHA (Secure Hash Algorithms), are a family of cryptographic functions designed to ensure the security of data. They work by transforming data with a hash function: an algorithm that consists of bitwise (bit by bit) operations, modulo addition and compression functions.



The hash function then produces a fixed-size string that looks nothing like the original. These algorithms are designed to be one-way functions, which means that once transformed into their respective hash values, it is virtually impossible to transform them back into the original data. Some algorithms of interest are SHA-1, SHA-2, and SHA-3, each of which was subsequently designed with increasingly strong encryption in response to hacker attacks. SHA-0, for example, is now obsolete because of widely exposed vulnerabilities.
A common application of SHA is password encryption, as the server side only needs to keep track of the hash value of a specific user, rather than the actual password. This is useful in case a malicious user breaks into the database, as they will only find the hash functions and not the actual passwords, so if they were to enter the hash value as a password, the hash function would convert it to another string and thus deny access.

In addition, SHAs exhibit the snowball effect, whereby changing very few encrypted letters causes a large change in the output; or conversely, drastically different strings produce similar hash values. This effect causes the hash values to provide no information about the input string, such as its original length. In addition, SHAs are also used to detect data tampering by attackers: if a text file is slightly modified, and in a barely perceptible way, the hash value of the modified file will be different from the hash value of the original file, and the tampering will be quite obvious.

The properties of a hash function are as follows:

- Applied to data of any size
- H produces a fixed-length result.
- $H(x)$ is relatively easy to calculate for any given x.
- Unidirectional property
    - It is computationally inefficient to find x such that $H(x) = h$
- Weak collision resistance
    - (given x) it is computationally impossible to find y ≠ x such that $H(y) = H(x)$
- Strong resistance to collisions
    - Computationally impossible to find any pair (x, y) such that $H(x) = H(y)$

- Two attack approaches
    - Cryptanalysis
        - Exploiting logical weakness in algorithms.
- Brute force attack
    - Test many inputs
    - Force proportional to the size of the hash code
- Most widely used SHA hash algorithm.
    - SHA-1 provides 160-bit hashes.

○ Newer SHA-256, SHA-384, SHA-512 offer improved size and security

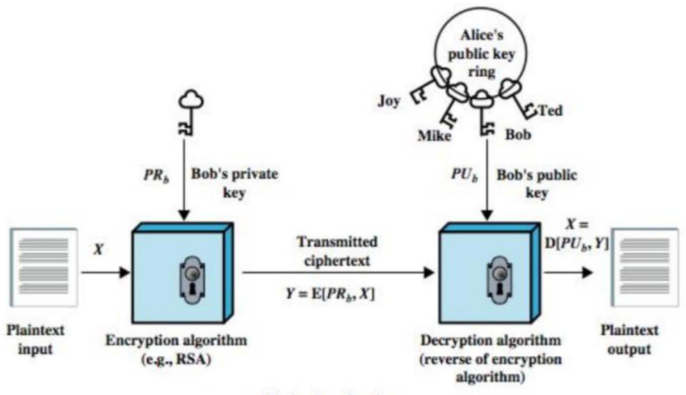
Asymmetric is a form of cryptosystem in which encryption and decryption are performed using different keys: the public key (known to all) and the private key (secret key). This is known as public-key cryptography.

Characteristics of the public encryption key:

- Public key cryptography is important because the decryption key cannot be determined by knowing only the cryptographic algorithm and the encryption key.
- One of the two keys (public and private) can be used for encryption and the other for decryption.
- With the public key encryption system, public keys can be freely shared, allowing users an easy and convenient way to encrypt content and verify digital signatures, while private keys can be kept secret, ensuring that only the owners of private keys can decrypt content and create digital signatures.
- The most widely used public key cryptosystem is RSA (Rivest-Shamir-Adleman). The difficulty of finding the prime factors of a compound number is the backbone of RSA.

Example:

The public keys of each user are in the Public Key Register. If B wants to send a confidential message to C, then B encrypts the message using C's public key. When C receives the message from B, C can decrypt it using its own private key. When C receives the message from B, it can decrypt it using its own private key. No other recipient besides C can decrypt the message because only C knows its private key.



(b) Authentication

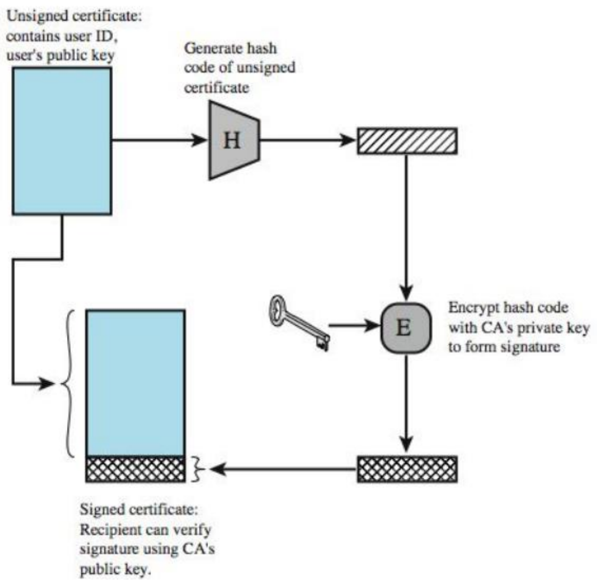
Components of public key cryptography:

- Plaintext / plaintext:
    - It is the readable or understandable message. This message is given as input to the encryption algorithm.
- Ciphertext / ciphertext:
    - Encrypted text is produced as the output of the encryption algorithm. This message is simply not understandable.
- Cryptographic algorithm:
    - The encryption algorithm is used to convert plaintext to ciphertext.
- Decipherment algorithm:
    - It accepts the ciphertext and the corresponding key (private key or public key) as input and produces the original plaintext.
- Public and private key:

- One private key (secret key) or public key (known to all) is used for encryption and the other for decryption.

Features:

1. Computationally simple creation of key pairs
2. The sender, who knows the public key, can encrypt messages with computational ease to encrypt messages
3. The recipient, who knows the private key, can decrypt the text to decipher the ciphertext
4. Computationally infeasible for the adversary to determine the private key from the public key
5. Computationally infeasible for the adversary to retrieve the original message
6. Useful if one of the two keys can be used for each role


Unsigned certificate: contains user ID, user's public key
Generate hash code of unsigned certificate
H
E
Encrypt hash code with CA's private key to form signature
Signed certificate: Recipient can verify signature using CA's public key.

Weaknesses of public key cryptography:

- Public key cryptography is vulnerable to Brute-force attacks.
- This algorithm also fails when the user loses his private key, then Public Key Cryptography becomes the most vulnerable algorithm.
- Public key cryptography is also weak against man-in-the-middle attacks. In this attack, a third party can disrupt public-key communication and thus change the public keys.
- If the user's private key used to create the certificate at a higher level in the Public Key Infrastructure (PKI) server hierarchy is compromised or accidentally disclosed, a "man-in-the-middle attack" is also possible, making any subordinate certificate completely insecure. This is also the weakness of public key cryptography.

Applications of public key cryptography:

- Cryptography/decryption:
  Confidentiality can be achieved by using public key encryption. In this case, the plaintext is encrypted using the recipient's public key. This ensures that no one outside the recipient's private key can decipher the ciphertext.
- Digital signature:
  A digital signature is used to authenticate the sender. In this case, the sender encrypts the plaintext using his or her own private key. This step ensures authentication of the sender because the recipient can decrypt the cipher text only with the sender's public key.
- Key exchange:
  This algorithm can be used for both key management and secure data transmission.

<u>Random numbers</u> are useful for a variety of purposes, such as generating data encryption keys, simulating and modeling complex phenomena, and selecting random samples from larger data sets. They have also been used aesthetically, for example in literature and music, and are of course increasingly popular in games and betting. When talking about single numbers, a random number is a number drawn from a set of possible values, each of which is equally likely, that is, a uniform

distribution. When talking about a sequence of random numbers, each number drawn must be statistically independent of the others.

With the advent of computers, programmers recognized the need to introduce randomness into a computer program. However, surprising as it may seem, it is difficult for a computer to do anything by chance. A computer follows its instructions blindly and is therefore completely predictable. There are two main approaches to generating random numbers using a computer:
Pseudo-random number generators (PRNGs) and true random number generators (TRNGs). These approaches have very different characteristics and each has its pros and cons.

In essence, PRNGs are algorithms that use mathematical formulas or simply precomputed tables to produce sequences of numbers that appear random.
The basic difference between PRNGs and TRNGs is easy to understand if you compare computer-generated random numbers with the rolls of a die. Since PRNGs generate random numbers using mathematical formulas or precomputed lists, using a PRNG corresponds to rolling a die many times and writing down the results. Each time you ask for the roll of a die, you get the next one in the list. In effect, the numbers look random, but they are actually predetermined. TRNGs work by having a computer actually roll the die or, more commonly, use some other physical phenomenon that is easier to relate to a computer than a die.

PRNGs are efficient, that is, they can produce many numbers in a short time, and deterministic, that is, a given sequence of numbers can be reproduced at a later time if the starting point of the sequence is known. Efficiency is a positive feature if the application needs many numbers, while determinism is useful if the same sequence of numbers needs to be reproduced at a later time. PRNGs are typically periodic, which means that the sequence will repeat. Although periodicity is almost never a desirable feature, modern PRNGs have such a long period that it can be ignored for most practical purposes. These characteristics make PRNGs suitable for applications where many numbers are required and where it is useful that the same sequence can be reproduced easily. Popular examples of such applications are simulation and modeling applications. PRNGs are not suitable for applications where it is important that the numbers be truly unpredictable, such as data encryption and gambling.

Compared with PRNGs, TRNGs extract randomness from physical phenomena and introduce it into a computer. One can imagine that it is a die connected to a computer, but usually a physical phenomenon is used that is easier to connect to a computer than a die. The physical phenomenon can be very simple, such as small variations in mouse movements or in the time between keystrokes. In practice, however, you have to be careful about the source you choose. For example, it can be complicated to use keystrokes in this way because keystrokes are often buffered by the computer operating system, which means that several keystrokes are collected before being sent to the program waiting for them. To the program waiting for the keystrokes, it will appear that the keystrokes were pressed almost simultaneously and perhaps there is not much randomness.

Exercises Lesson 5

1) Crack the Hash: Text, help and solution

Text

It's happened again! Some of our beloved friends at linkedin.com forgot to salt their password hashes. Due to a rather interesting SQL injection issue, a hacker group has published the following MD5 hash online! Can you crack it?

Hash: `365d38c60c4e98ca5ca6dbc02d396e53`
Hint. Use a md5 cracker

Help:

1) The description suggests that the MD5 has been already cracked and that you can find the solution online. Write on your research engine (e.g., Google) "MD5 cracker." You can find several web pages where you insert an MD5, and they give you a solution.

Solution

Very simple, literally we have an MD5 hash and so we translate it.

We have the following MD5 hash:
365d38c60c4e98ca5ca6dbc02d396e53

The description suggests us to use a tool.
https://crackstation.net/
the password is password12345

Normally, the MD5 cryptographic algorithm is not always reversible.
You can encrypt a word in MD5, but you cannot create the reverse function to decrypt an MD5 hash in the plaintext (at least, not of all of them). This is the property of a hash function. Therefore, unfortunately, it is also not possible to go through a universal Python implementation (it is also possible to make an algorithm, but it may never work, since it touches to search all possible combinations and, randomly by brute force, you might get there; however, it may have a runtime *as* long *as the life of the universe*).
A possible totally bruteforce implementation tries all possible combinations and works in some limited cases, as I report below:

```
import hashlib #library used for MD5/SHA, etc.
def md5reversehashing(text):
  for i in range(0, 100000000): # 100000000 is the number of possible combinations
    hash_object = hashlib.md5(str(i).encode()) # convert the number to a string and encode it
    if hash_object.hexdigest() == text: # if the hash is equal to the text
      print(i) # print the number
 break
#Driver Code
md5reversehashing('e10adc3949ba59abbe56e057f20f883e') # 123456
#md5reversehashing('365d38c60c4e98ca5ca6dbc02d396e53') # This one coming from the challenges is not working
```

2) Ready XOR Not: Text, help and solution.

Text

original date: "El Psy Congroo"
encrypted data: "IFhiPhZNYi0KWiUcCls="
encrypted flag: "I3gDKVh1Lh4EVyMDBFo="

The flag is a composition of two names (two animals (?)).

Aid:

1) You can notice the following:
   - The encrypted strings are in base64. Decode them!
   - All the strings have the same length
   - The challenge is named "xor" ... this should suggest you something
2) To obtain the flag, you need to decrypt it.
   Since we have an example of plaintext and its encrypted counterparts, you can use the "xor" property to find the key: remember, you need to first represent the characters as "ascii numbers," xor them, and reconvert it into ascii char.

Solution

```
import base64

original_data = "El Psy Congroo"
encrypted_data = "IFhiPhZNYi0KWiUcCls="
encrypted_flag = "I3gDKVh1Lh4EVyMDBFo="
# we can note that all the strings have the same length
# since we have an example of encryption, and we know that this is a xor,
# we can simply try to obtain the key in the example, and apply it to the
# crypted flag

# Usual function to decode base64 strings
def base64tostring(text):
  return base64.b64decode(text).decode('utf-8', errors="ignore")

#decode the encryption from base64
enc_data= base64tostring(encrypted_data)
enc_flag= base64tostring(encrypted_flag)
#we know apply the xor to obtain the key
# So, we take the position of each character of "x" and "y" and then, iterating,
# using the zipper() function, which returns a zipper object (an iterator of tuples where the first item in
each
#passed iterator is paired together), and then the second item in each passed iterator are paired
# together etc. Given the ordinary XOR logic, we take the key via the XOR between raw and encoded
data
key = ' '.join([chr(ord(x) ^ ord(y)) for x, y in zipper(original_data, enc_data)])

print('key:\t',key)
#this seems a reasonable key
```

```
flag = ' '.join([chr(ord(x) ^ ord(y)) for x, y in zipper(enc_flag, key)])
print("Flag:\t", flag)
#flag: FLAG=Alpacaman
```

3) Top: Text, Aids and Solution

Text

Perfectly secure. That is for sure! Or can I break it and reveal my secret?
We are given an encryption script and a file which is encrypted with it

We are precisely given two files: a Python file "top.py," which we show below, and an encrypted file with that, specifically "top_secret" with no extension. Below is, precisely, *top.py*.

```
import random
import sys
import time
# Let's import the main modules, especially "time" to take the current system time, "random"
# to use the time for the seed of the randomness choice and then "sys", in order to
# open in writeback mode and writing the bytes with algorithm here

cur_time = str(time.time()).encode('ASCII')
# So, the idea is taking the current time and encode the string in ASCII
random.seed(cur_time) # then "plant" the seed for the randomness

msg = input('Your message: ').encode('ASCII') # The input, also, is in ASCII
key = [random.randrange(256) for _ in msg]
# We put the random range up until all the 256 characters of ASCII Encoding, looping
# in all of the message

# What we are doing here is creating a zipper function, so we apply the XOR function
# and then XORing the entire message + the current time with the XOR sign (0x88) of the current
# time (in length) + the key itself; we can see we repeat the same data X number of times; that's the
# vulnerability
c = [m ^ k for (m,k) in zipper(msg + cur_time, key + [0x88]*len(cur_time))]

# In the end, we just write the file entirely in the buffer
with open(sys.argv[0], "wb") as f:
  f.write(bytes(c))
```

The file "top_secret" consists of the following text:

```
f¸,¹ÚfœX'^ß÷" qDµ8{"tI±y†-Vu³Þ[H° Å†ýã" ¿ô'oVŽ¥©-¹½¹"¿¹±¹""¦°¿º°¿½º
```

Aid:

1) We are given a script that generated the encrypted file. The challenge here is to spot potential vulnerabilities. The first thing to do is to debug it; we can find the following steps:
    - Set a seed with a time.

- Get an input string
- Define a key for the string
- Encryption: the final output is the concatenation of both encrypted message and encrypted time!

2) The time characters are encrypted with some fixed values (0x88). Since we are talking about a xor operation, we can retrieve the original time. To know the length, get a time variable and see how many characters we do have.
Then, once we have the "plain" time, we ca regenerate the original sequence of characters that compose the key ... and apply it to the encrypted text. We finally have the flag.

<u>Solution</u>

```
#the encryption is composed by a xor between a character and a key.
#the message is the concatenation of msg + cur_time
#the key is the concatenation of the list of keys + list of 0x88

####--------RESOLUTION -----------------
#we know that |msg| = |key|, and |cur_time| = |[0x88]|
#we can use the xor property to retrieve the cur_time of the execution

#useful to use "rb" to read in binary and open correctly the file
#otherwise, it won't work (infact, as a no extension file, it is considered binary)
With open("top_secret", "rb") as f:
   secret = f.read()

#let's try to think about the algorithm
#there's a message which is encoded in ascii,
#a key selected randomly using the current time as the seed
#and a xor function between the message and the key
#and 0x88, multiplied X times with the length of the current time

#we can see that the time length is added many times
#so we can xor it by subtracting from the 'secret' string the 'cur_time' length, given how many times is
#repeated, also knowing the XOR function has a vulnerability because of it
sec_time = secret[-len(cur_time):]
plain_time = ' '.join([chr(m ^ k) for (m, k) in zipper(sec_time, [0x88]*len(cur_time))])
print(f "Plain time:\t{plain_time}") #what we printed seems a correct datatime format

# we now leverage on the pseudonumber vulnerabilities
# the algorithm set a seed, so it is not random the generator.
# So, we plant the seed as the plain_time encoded in ASCII in order to correctly read it
random.seed(plain_time.encode("ASCII"))

# get the keys, so we iterate on each key into the secret string subtracting the current time
# and then reapplying the given XOR function to the new plain text, printing it
keys_secret = [random.randrange(256) for _ in secret[:-len(cur_time)]]
plain_text = ' '.join([chr(m ^ k) for (m, k) in zipper(secret[:-len(cur_time)]], keys_secret)])
print(plain_text)
```

```
#flag reached
# Here is your flag: 34C3_otp_top_pto_pot_tpo_opt_wh0_car3s
Print:
75
Plain time          :1513719133.8728752
Here is your flag: 34C3_otp_top_pto_pot_tpo_opt_wh0_car3s
```

4) Repeated XOR: Text and Solution

Text

There is a secret passcode hidden in the robot's "history of cryptography" module. But it's encrypted! Here it is, hex-encoded: encrypted.txt. Can you find the hidden passcode?

Hint:
Like the title suggests, this is repeating-key XOR. You should try to find the length of the key - it's probably around 10 bytes long, maybe a little less or a little more.

```
#Follow the following procedure
STEP 1: Key length identification
#1.1 set the key length to test
#1.2 shift the secret string by key_length
#1.3 count the number of characters that are the same in the same position
     Between the original secret and its shifted version
#1.4 take the highest frequency over different key length by repeating 1.1 - 1.3
#STEP 2. Cryptoanalysis
```

Paired with this, as mentioned by the text, is the file "encrypted.txt" with a long text in hexadecimal (e.g. of some characters: 2AE6BD0B6ECE21162AF4B20C6CC2 and so on).

Solution

```
#the goal of this challenge is to leverage on the xor weaknesses.

#we can read the file first
#the file is in hex encoded; it could be good to bring it in a proper form
#we know that FF is 256, i.e., we can represent the text in a decimal format,
#where each number can be encoded in ascii
With open("encrypted.txt", 'r') as file:
   secret_hex = file.read()

def hex2dec(text):
   res = []
   for i in range(len(text)//2): # we take each couple of bytes and then convert into "int" in 16 bytes form
      #get the current pair of hex
      curr = text[i*2:(i+1)*2]
      #convert to int the current paired two bytes string form and then express it in sixteen bytes each;
      #this way, we can convert the 16-bit form into the 10-bit form required (hex to decimal)
```

```
        res.append(int(curr, 16))
    return res

secret = hex2dec(secret_hex)
#STEP 1: Key length identification
#shift string -> it allows us the comparison
# To make a shift, we can split the text starting from the key length and then iterate to the end
# summing the iteration from the beginning up until the key length
def shift(text, key_length):
    return text[key_length:] + text[:key_length]
#freq counter
#we compare the original sentence with its shifted version
#we count the amount of same characters in the same position
def freq_counter(s1, s2):
    freq = sum([1 for (x, y) in zipper(s1, s2) if x == y])
# Note we sum one using a zipper between s1 and s2 only if in the same position (so, to iterate on both)
    return freq

#test over different lengths.
#the hints suggests us that the length is about 8. So, we look between [5, +15]
for kl in range(5, 16):
    print(f "Length:\t{kl}\tFreq:\t{freq_counter(secret, shift(secret, kl))}")

#the highest value is with length = 8.

#STEP 2: Cryptoanalysis
#split the corpus into 8-chars' lengths.
def splitter(text, key_length):
    res = []
    for i in range(key_length):
        res.append(text[i::key_length]) # we take the i-th char of each block
# then, we return a string split according to the parameter "key_length" passed
    return res

secret_ = splitter(secret, 8) # At this point, we split the string

#we need to define a method that show us the k-th most frequent character in
#a given string
from collections import Counter
def k_char(text, k):
    #use the counter→ Counter is an unordered collection where elements are stored as Dict keys and
their # count as dict value. Counter elements count can be positive, zero or negative integers. However,
there is
# no restriction on its keys and values. Although values are intended to be numbers, but we can store
# other objects too.
    freq = Counter(text) # We find the frequency of the text passed, counting how much it appears

# Here, we order the data collection like a list and then return a sorted list, using an iterable (the items
# of the frequency count), a function (made shortened thanks to lambda, which sets the "key" parameter)
#making it appear in reverse order
   ordered = sorted(freq.items(), key=lambda x: x[1], reverse=True)
   return ordered[k][0] # we can return the ordered elements, started from the first (this is like *ordered)

## we can now see the top N frequent words
#print(k_char(secret))

#we now work on the Cryptoanalysis, based on each column of the matrix M[secret//8 X 8]
#we can first assume that the most common character in each column is the space ' '.
key_sec = [k_char(secret_[0], 0), k_char(secret_[1], 0), k_char(secret_[2], 0), k_char(secret_[3], 0),
   [k_char(secret_[4], 0), k_char(secret_[5], 0), k_char(secret_[6], 0), k_char(secret_[7], 0)]

# What we are doing here is seeing the frequency of each of the eight characters of the secret
# given the position and then having a list of the frequency of each one

#xor the key: to do that, we use the kth character in the list and then XOR it with the space character
real_key = [k ^ ord(' ') for k in key_sec]

#decode the secret
real_message = ' '
# Enumerate() method adds a counter to an iterable and returns it in a form of enumerating object. This
# enumerated object can then be used directly for loops or converted into a list of tuples using the list()
#method. Syntax→ enumerate(iterable, start=0)→ so, it iterates on secret
for i, c in enumerate(secret):
   key_pos = i % 8
# we then apply the XOR function as seen with "real_key" but this time XORing the key in each position
   real_message+= chr(c ^ real_key[key_pos])

print(real_message)

#your flag is: 8eb31c92334eac8f6dacfbaaa5e40294a31e66e0
```

Other solution (written by me)
You can also compare a good writeup (complete with extended explanation of even the full logical reasoning) at the link: https://ehsan.dev/pico2014/cryptography/repeated_xor.html

```
#we do have a txt file and we're gonna read it
With open('encrypted.txt', 'r') as f:
   data = f.read()
   data=data.replace(' ', '').replace('\n', '') #removing the \n from the end of the line and converting to ascii

#we do know the file is hex encoded, so we decode its bytes into
#decimal format and then we convert it to ascii
def hex_to_dec(text):
   result=[]
```

```
    #we take each pair of bytes and convert it to decimal (2), looping into hex (16)
    for i in range(0, len(text), 2):
        #taking each pair of bytes in the text and converting it to decimal
        current=" ".join(text[i:i+2])
        #appending the result considering each pair corresponds to a byte in hex
        result.append(int(current, 16))
    return result

decoded_data=hex_to_dec(data) #we decode the data (when printed, it seems like a map of integers)

#We have a repeated xor problem here, so we have to make a frequency
#analysis in order to understand which characters are the most frequent ones
#and then we can guess the key

#First, we have to guess the key length; infact, if the key is shorter than the message,
#the key will repeat itself many times in order to cover the whole message.
#So, after converting to integers, we duplicate the key and xor
#the message with the duplicated key.


#Step 1 - Key length identification (we know it's probably 10 bytes long)
#Here, we do need to make an educate guess using the frequency analysis and a shift function,
#just to play out with some different lengths
def shift(text, n):
    return text[n:] + text[:n] #we shift the text by n bytes and sum them together

def count_same(a, b): #we count the same bytes in the text and simply return it
    count=0
    For x, y in zipper(a,b):
        if x == y:
            count+=1
    return count

def guess_key_length(text, key_length):
    #what we are doing here is making a range based on the key length
    #then counting the same bytes in the text and making a shift of "n" bytes
    #in order to try all of the possible combinations
    #we return the key length with the highest count of same bytes
    #the higher the count, the more likely the key length is correct
    return max(range(1, key_length), key=lambda n: count_same(text, shift(text, n)))

print(guess_key_length(decoded_data, 10)) #we print the key length, which is 8
#Up until here, we completed step (1)

#Step 2 - Cryptoanalysis
#We do know the key length, so we can guess the XOR is made with 8 bytes block in mind
#Remember that the key is repeated every 8 bytes within the text,
#so the idea is to take the most frequent characters every 8 bytes based on the key length
```

```
from collections import Counter #we import this in order to count the most frequent characters

#Given it should be English text, the most frequent characters would be
#e, t, a, o, i, n, s, h (in this order), thinking also ' ' (the space) should be the most frequent one

#Because XOR is its own inverse, we can find the key by XORing cipher text and known plain text
#(given a character, in whatever column it appears, given the fixed key length, the XOR always gives the
#same result). Thus we can find the key if we know the most common character in english and the most
#common character in the nth column.

def most_frequent_chars(text, key_length):
  #we split the text into blocks of 8 bytes
  #then we count the most frequent characters in each block
  #and we return the result (i::key_length represents every element of text in indexes from i to
key_length)
  return [Counter(text[i::key_length]).most_common(1)[0][0] for i in range(key_length)]
  #we return the most frequent character (hence the (1)) in each block (given the key length)
key_secure=most_frequent_chars(decoded_data, 8)
print(key_secure) #we print the map of the most frequent characters in the text, split by 8

#now, we need to find the key, which is the XOR of the most frequent characters and the plain text itself
real_key = [k ^ ord(' ') for k in key_secure]



#decode the secret and print the flag
for i in range(len(decoded_data)):
  decoded_data[i] ^= real_key[i % len(real_key)] #we xor the data with the key given the 8 bytes key
split

print(bytes(decoded_data).decode('ascii'))
#we print the final result (given it is the flag and all of the other text) as ascii
```

Lesson 6: User authentication/User authentication (Accounts)

User authentication verifies the identity of someone attempting to access a network or computing resource by authorizing a human-to-machine transfer of credentials during interactions on a network to confirm the authenticity of the user. The term is contrasted with automatic authentication, which is an automated authentication method that does not require user input.
It helps ensure that only authorized users can access a system, preventing unauthorized users from logging in and potentially damaging systems, stealing information, or causing other problems. Almost all interactions between humans and computers, with the exception of guest accounts and those that log in automatically, involve user authentication. Authorizes access over wired and wireless networks to allow access to networked systems and resources and the Internet.

User authentication is a simple process and consists of two steps:

1) Identification. Users must prove who they are.
2) Verification. Users must prove that they are who they say they are and must prove that they are authorized to do what they are trying to do.

User authentication can be as simple as requiring the user to type in a unique identifier, such as a user ID, along with a password to access a system. But it can also be more complex, such as requiring the user to provide information about physical objects or the environment or even to perform actions, such as placing a finger on a fingerprint reader.
An example of authentication:

- User real name: Alice Toklas
- User ID: ABTOKLAS
- Password: A.df1618hJb

This information is stored in a system and only Alice can access it with these credentials. However, if not well protected, attackers can use this information anyway.

There are 4 ways to authenticate a user's identity, for example based on the individual:

- Data it knows - e.g., password, PIN.
- Data it holds - e.g., key, token, smartcard
- Data showing who he/she is (static biometrics) - e.g. fingerprint, retina
- Data demonstrating a characteristic of its own (dynamic biometrics)-e.g., voice, signature

They can be used together or in combination and can all provide authentication to the user and all have problems. A classic login method is the classic username and password, compared by the system and then compared to verify they match one's own. Through the user's ID, their privileges are verified.

Some examples of password attacks and vulnerabilities:

- *Keystroke timing analysis, in which the* timing at which keys are pressed is read and the extent to which they are used is analyzed in milliseconds. Similarly, one can carp about the movement of keys from a physical point of view.

- *URL Hijacking/Typosquatting*, starting with the fact that it can happen that an attacker physically accesses a machine, perhaps using an invisible cable and somehow modifying user information. In this way, we send an email with a fake link; by doing so, going to the page has the correct URL, but the page has a fake link and would not be recognizable at all. Similarly, we use *tabnagging*, so we exploit old tabs to insert malicious links or *UI redressing/iFrame overlay*, to insert malicious links on buttons/graphical elements.

- *Offline dictionary attack*: the attacker has the hash of the target password and tries to obtain it, based on common passwords or known information about the target. An attempt should be made to protect this information, possibly by hiding it in a safe place.

- *Specific account* attack: the attacker targets a specific account and tries to guess the correct password, based on common passwords or known information about the target. As a countermeasure, *lockout* mechanisms can be introduced, so for example, leaving a user excluded after a certain number of login attempts.

- *Popular password guessing*: attacker tests popular passwords
  against a wide range of accounts, knowing that users tend to choose simple, short passwords or enter sensitive information (loved ones, friends, etc.). As a countermeasure, we try not to use

common passwords (possibly even auto-generated, so using a mechanism that tries to create of a chosen length with or without special characters, already predetermined) or avoiding using sub-sequences/patterns of useful characters.

- *Workstation hijacking/hijacking*: The attacker waits for a workstation to be unattended in order to take possession of it. As a countermeasure, in addition to automatic log-out mechanisms, they look for abnormal behavior (strange log-ins, leaving the user out after a certain time, etc.).

- *Exploiting user mistakes/exploiting user errors*: Users tend to write down passwords For example, post-it notes near the protected device and tend to have devices with preconfigured passwords. As a countermeasure, efforts should be made to educate users (urging them to adopt behaviors such that they do not leak useful information to the outside world) and use combined authentication mechanisms (e.g., passwords and tokens).

- *Exploiting multiple password uses*: Users tend to use the same password (or similar passwords) in several systems. If an attacker guesses the password, he can damage many systems. As a countermeasure, we also try here to train users not to engage in vulnerable behavior (choosing different and perhaps self-generated passwords, for example) and prohibit password reuse across systems.

- *Password spraying attack*, whereby the hacker attempts to authenticate using the same password on several accounts before switching to another password. Password spraying is more effective because most Web site users set simple passwords, and the technique does not violate blocking policies because it uses multiple accounts. Attackers orchestrate password spraying primarily on Web sites where administrators set a standard default password for new users and unregistered accounts.

- *Electronic monitoring*: If the password is communicated over a network, the attacker uses *sniffing* (listens in on the network, tries to steal information from packets on the network to steal the password). As a countermeasure, it tries to use secure/short-circuit-protected or physically good channels as well.

- *Session hijacking*, then automatic access to a machine as they are already authenticated; automatic log-off mechanisms. *Multi-factor authentication* can also be used for this. If enabled on your account, a potential hacker can only send a request for access to your account to the second factor (normally, this is a message/notification/physical device). Hackers will probably not have access to your mobile device or fingerprint, which means they will be locked out of your account.

- *RDP Hacking (Remote Desktop Protocol*), which is easily achieved by manipulating the network, especially if it is not secure or by exploiting holes at the software update level.

Useful sites and directions on passwords

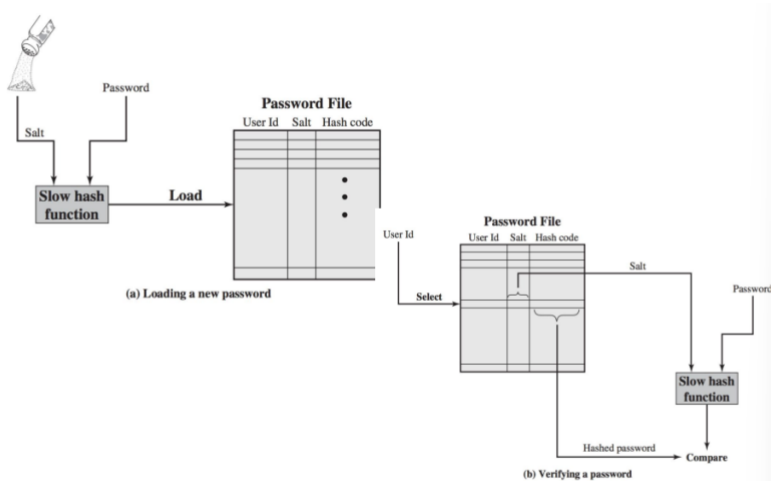
- Have I Been Pwned: https://haveibeenpwned.com/
- The most common passwords (classic, *password* or *123456*): https://blog.dashlane.com/ten-most-common-passwords/
    - Knowing all the common passwords, we try a bruteforce

- Use a *password manager* (Bitwarden which I personally recommend) to save them all. Don't use local files, Chrome to save them or similar applications, external files or a notebook to write them down; it's all very inconvenient and not very scalable
- Passwords are never saved in plain text; in this way, the sysadmin/system administrator will not know/see the password no matter how much they manage. Similarly, passwords are not often changed; whatever computation you have computed, you should reuse it for future computations.
- There are attacks and challenges that exploit the most common passwords, through so-called "rockyou" files, which contain precisely the most common ones

One mechanism used are hashed *passwords*, so the user creates a new password and it is combined with a *salt* (random data used as an additional input to a *one-way function*, so a function that is easy to compute on any input but difficult to invert on random input; it allows a hash function to be performed on data, password or passphrase (set of alphanumeric words/strings used for authentication)). The ID, the password with the hash and the salt are saved on a file (password file) and these hash functions are designated to be slow. Because of the *salt, for* example a random number, it complicates the fact that the password can be stolen.

Hashes, when used for security, must be slow. A cryptographic hash function used for hashing passwords must be slow to compute because a quickly computed algorithm could make brute force attacks more feasible, especially with the rapidly evolving power of modern hardware. This can be achieved by making the hash computation slow by using many internal iterations or by making it memory intensive.

A slow cryptographic hash function hinders this process but does not stop it, since the speed of hash computation is of interest to both intended and malicious users. It is important to achieve a good balance between speed and usability of hashing functions. A well-intentioned user will not have a noticeable impact on performance when attempting a single valid login.



We use a salt for three main reasons:
1) Prevent duplicate passwords (if two users use the same password, different jumps can produce different passwords)
2) Increase the difficulty of dictionary attacks (if the salt has b bits, then the factor will be 2^b
3) Impossible to tell if the person uses the same password in multiple systems

Some common *password cracking* attacks are as follows:
- Dictionary attacks→ Try each word in a dictionary on a hash of a password file, trying all common passwords, and if there are no matches, we try every possible modification (numbers, punctuation). They are computationally expensive

- Rainbow table attack→ It is a password cracking method that uses a special table (a "rainbow table") to crack password hashes in a database. Applications do not store passwords in plain text, but encrypt them using hashes. After the user enters his or her password to log in, it is converted into a hash and the result is compared with the hashes stored on the server to look for a match. If it matches, the user is authenticated and can log in to the application. The rainbow table refers to a precompiled table that contains the hash value of the password for each salt used during the authentication process. If hackers have access to the list of password hashes, they can crack all passwords very quickly with a rainbow table.

In fact, users might choose either short passwords (easily guessed, and systems usually reject short passwords) or unguessable passwords (thus, attackers use a list of similar passwords, taking an hour on the fastest systems, with only one success to steal the data).
Similarly, using various methods to gain access (credit cards, magnetic cards, SIM cards, smart cards, etc.) or biological characteristics (signature, retina, voice, etc.).

As for the use of *memory cards*, they store but do not process data.
Normally they are magnetic stripe cards, e.g. bank card with an electronic memory card, used alone for physical access, It can also be used with password/PIN for computer use.
The disadvantages of memory cards are:
○ Need a special reader (which increases the cost of the security solution).
security solution)
○ Problems with token loss (we can't trust users).
○ User dissatisfaction (not totally approved by users)

Authentications can be locally or remotely, and of course, doing it over a network can lead to various problems of interception, replay (reuse), etc. Generally, challenge-response is used:

- The user submits the identity
- The host responds with:
  - a random number r (also called nonce)
  - A hash function h
  - A function f
- The user calculates $f(r, h(P))$ and resends
- The host compares the user's value with its own calculated value, if it matches the user authenticates
- Protects against a range of attacks

There can be several: fingerprint scanning, voice authentication, via face, OTPs (temporary/One-Time *Passcodes*, unique and tied to a single account).