TDT4113 (PLAB2) Project 3: Encryption

This document describes the third project in PLAB.

- The task must be solved individually
- The task should be solved with object-oriented code written in Python
- The deadline for the assignment is one week. That is, your implementation will be uploaded to BLACKBOARD
  - latest at 8am, Wednesday, February 10, 2021
  - and demonstrated before 4pm at the same day.

NB! Read the full assignment text before beginning the implementation. The functionalities of the system are described step by step, but you may want to see the "whole package" before choosing the design. Remember that you must use an OO-design to get the project approved.

1 Background to the task

In this project, you will implement cryptography algorithms to encrypt a text and to decrypt the text again. In addition, you should implement a "brute force" method to try to break / hack the encryption.

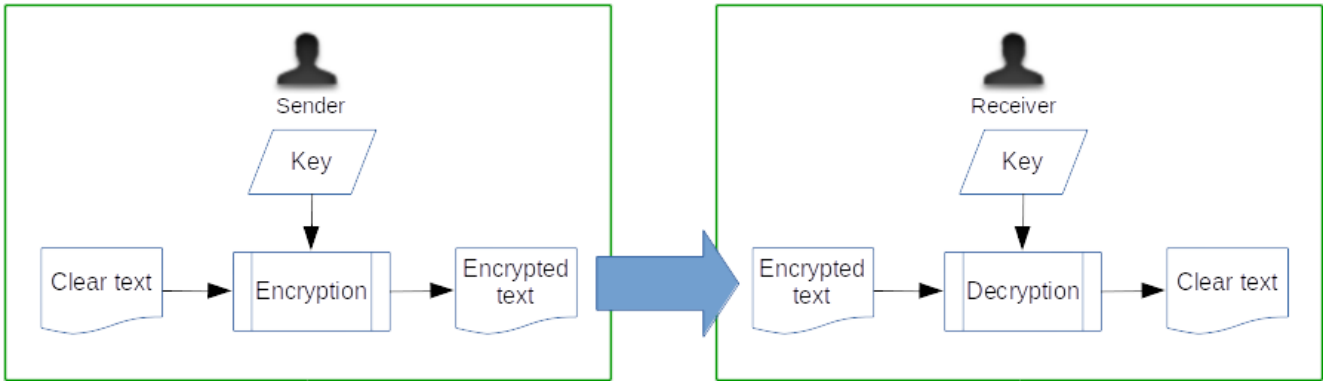

Figure 1: Simplified schematic description of cryptography

Encryption at a schematic level is shown in Figure 1. A sender and a receiver want to communicate over a non-secure line of communication (for example, by email). The two agree on some encryption keys as well as an algorithm to use the keys to encrypt/decrypt the message. The sender takes the original message in clear text, sends it through the encryption algorithm with its key and gets the encrypted text. Receiver receives the encrypted text and uses its key to decrypt. The result is that the receiver receives the message back in clear text, without the directly readable message being exposed on the communication line.

We will look at a number of different encryption algorithms of varying difficulty. The algorithms are based on a set of valid characters the text may contain (i.e., the alphabet). In this

description, I will use the alphabet of the 26 letters in English, all written in capital letters. In this way we can read the characters and recognize them easily. This also makes the examples shorter to fit a single line in this document. But in your implementation, you should use a larger alphabet, namely all characters with ASCII values, from 32 (space) to 126 (the character "~"). The encryption keys are what enable the sender and receiver to communicate. As we shall see later, the sender and receiver will not necessarily have the same key, but the two keys correspond to each other. How this is done depends on the encryption method in use. If a third party gets the encryption keys and knows which algorithm is in use, it can break the encryption, which we will return to later.

2 Suggested code structure

You are going to implement multiple ciphers, so it is useful to define a superclass `Cipher` that has the implementation of the various encryption algorithms as subclasses. The `Cipher` class should keep information shared by all ciphers, such as the alphabet. It is also useful to explicitly represent that the alphabet has a total of 95 characters, so all mod operations must be done with modulo 95 in their implementation (while I use modulo 26 in this document). In addition, it is useful to add dummy versions of the required methods here, such as `encode` and `decode`. In addition, you should create the `verify` method, which gets clear text, encodes the clear text, decodes the encrypted text, and verifies that the decoded text is identical to the initial clear text. It is also possible to create a method to generate encryption and decryption keys for the sender and receiver.

We have two or three "persons" involved in this task: one who sends the message, one who receives it, and possibly one who tries to hack the message. Therefore, you should implement the `Person` class. An instance of this class has a key (access to the value of `self.key`) and a cipher algorithm (instance of one of the subclasses of `Cipher`) that it can work with. The class therefore needs the methods `set_key`, `get_key`, and `operate_cipher`. Create the subclasses `Sender` and `Receiver`, and customize the subclasses use `self.operate_cipher()` so that the sender encrypts text and the receiver decrypts text. A `Hacker` class is also useful to create; this will be discussed in more detail later. To better understand how to fill these classes with content, it is better to read this entire document first, so that you get a complete picture of the functionalities to be implemented. For your own part, a UML diagram can also be useful, but this is not required to pass the task.

In this project you can freely use the help routines in the file `cypto-util.py`. Before you use anything from it, it is important that you read the code and understand how it works so that it does not give you errors in your program.

Implementation - Part 1: Implement the `Cipher`, `Person`, `Sender`, and `Receiver` classes.

3 Various ciphers

3.1 Caesar

The first to be implemented is the so-called `Caesar` cipher. This cipher uses a secret key given as an integer, for example 2. With this key, the encryption procedure can be illustrated using a table in Table 1 (which is specific to the key we chose).

The first line are all symbols of the alphabet, followed by a line of numbers (starting at zero, increasing by one for each symbol). These numbers represent the symbols. To code a symbol, such as "P", we look up what numerical value it has. Here is the answer that "P" gives the value 15. Then we add the value of the key, and get the encoded value, given in the third line.

Table 1: Caesar cipher with key=2

Clear text	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
numeric val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Coded val.	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1
Coded text	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B

Since we chose key 2, the answer is $15 + 2 = 17$. Finally, we look up which letter this number corresponds to in the fourth line, and find that for the value 17 corresponds to the letter "R". So "P" is encrypted to "R". If we want to encrypt the word "PYTHON", the answer is with key 2 "RAVJQP".

To decode a message, we can go from the bottom line upwards in the same table. Say that we have received the encrypted text "EQFG". To decrypt this string, we start with the symbol "E" in the line of encoded text. This symbol corresponds to coded value 4, which gives the numeric value for clear text 2, and should therefore be "translated" into the letter "C". Furthermore, "Q" becomes "O", "F" becomes "D", and "G" becomes "E" so that "EQFG" is decrypted to "CODE". If we look at the encryption table we see that the encoded value is calculated as the original number value plus the key value for the symbols from "A" through "X". When we get to "Y", which has number value 24, it is more difficult, because the number value $24 + 2 = 26$ does not correspond to any letter in our alphabet. So instead of just adding the key value as we move from the original number value to the coded number value, the operation is to take the sum mod 26 - as there are 26 symbols in the alphabet. In your implementation, you should use a larger alphabet of size 95. So you should use $\mod 95$ instead.

Decryption is basically the same as the encryption, only differing that we select a paired key. Here is the encryption table with the paired key=24, which turns out to work:

Table 2: Caesar cipher with key=24

Clear text	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
numeric val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Coded val.	24	25	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Coded text	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X

Suppose we want to encrypt the text "EQFG", that is, encrypt again the same text that we got as the coded version of the word "CODE". We now get "E" $\rightarrow$ "C", "Q" $\rightarrow$ "O", "F" $\rightarrow$ "D", and "G" $\rightarrow$ "E", so we're back to the same word as we started with, and we see that encryption first with key 2, then with key 24 gives us the clear text back. In other words, decryption is the same procedure as encryption but with key 24. The reason why this works is that if you have a symbol with the number value x, add $\mod 26$ (first encryption) and then add $\mod 26$ we will return where we started:

$$(x + 2 \mod 26) + 24 \mod 26 = (x + 26) \mod 26 = x.$$

In general, for a 26-symbol alphabet, if a text is encoded with key n, $1 \leq n \leq 25$, we can decrypt it by doing the same operation but with key $26 - n$. You can of course also use -2 as a key in decryption because $x + 2 + (-2) \mod 26 = x$, and $-2 \mod 26 = 24$ holds well. However, it is common to use positive numbers as keys. So we go for 24 instead of -2 in this description.

Implementation - Part 2:
Make the class `Caesar` as a subclass of `Cipher`. The dummy methods from the superclass (`encode`, `decode`, generate keys, etc.) must be implemented for the `Caesar` cipher. Verify that it works by using `verify()`. Test the entire infrastructure by letting a sender and a receiver share keys. Let the sender encrypt the message and provide encrypted text to the receiver, and let the receiver decrypt the message.

3.2 Multiplication cipher

The `Caesar` cipher is a simple operation where we take each symbol's numeric value and add a number. The multiplication cipher, which is now described, is much the same, but as its name says, the addition is replaced by a multiplication. The Multiplication Cipher table with key=3 is shown in Table 3. Note that here we also do the operation $\mod 26$. For example "C", which has the value 2, is not translated to the value $10 \cdot 3 = 30$, but to $10 \cdot 3 \mod 26 = 4$, i.e. the letter "E":

Table 3: Multiplication cipher with key=3

Clear text	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
numeric val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Coded val.	0	3	6	9	12	15	18	21	24	1	4	7	10	13	16	19	22	25	2	5	8	11	14	17	20	23
Coded text	A	D	G	J	M	P	S	V	Y	B	E	H	K	N	Q	T	W	Z	C	F	I	L	O	R	U	X

If we use this table to encrypt the message "CODE", we will get "EQJM". To decrypt a message, it is natural to think that we take the numeric value of a letter and divide by the key value, since we used it during encryption. Unfortunately, it does not work. For example, "E" has a value 4, but if we divide by the key value 3, we do not get an integer. Instead, we will look for a new key value that can be used for decryption and that matches the key used for encryption. If we call the new key m, we require it to first times a number x with the original key n (modulo 26) and then with the new key m (modulo 26), then it will give us back x, i.e.

$$(x \cdot n \mod 26) \cdot m \mod 26 = x$$

for all $x \in \{0, \ldots, 25\}$. This can also be written as $n \cdot m \mod 26 = 1$, and the value of m is called modular inverse to n. It is a bit complex to find which value is modular inverse to a key, but you can freely use `modular_inverse` in the `cypto-util.py` file.

It turns out that if you encrypt with key $n = 3$, you can decrypt by multiplying by key $m = 9$. This pair fits the definition because $n \cdot m \mod 26 = 9 \cdot 3 \mod 26 = 27 \mod 26 = 1$. The corresponding the encryption table looks like this:

Table 4: Multiplication cipher with key=9

Clear text	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
numeric val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Coded val.	0	9	18	1	10	19	2	11	20	3	12	21	4	13	22	5	14	23	6	15	24	7	16	25	8	17
Coded text	A	J	S	B	K	T	C	L	U	D	M	V	E	N	W	F	O	X	G	P	Y	H	Q	Z	I	R

This works because we get "E" → "C", "Q" → "O", "J" → "D", and "M" → "E". Thus, decryption is the same procedure as the encryption, except that in the decryption we use key=9 instead of key=3 in the encryption.

Multiplication cipher works only for some keys. The requirement for a key n to work is that for two numbers x_1 and x_2, $n \cdot x_1 \mod 26 = n \cdot x_2 \mod 26$ is only possible if $x_1 \mod 26 = x_2 \mod 26$. In Table 5 you see what happens when we use key=2:

Table 5: Multiplication cipher with key=2

Clear text	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
numeric val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Coded val.	0	2	4	6	8	10	12	14	16	18	20	22	24	0	2	4	6	8	10	12	14	16	18	20	22	24
Coded text	A	C	E	G	I	K	M	O	Q	S	U	W	Y	A	C	E	G	I	K	M	O	Q	S	U	W	Y

Since both "A" and "N" are encrpyted to "A" in encoded text, the encoding is not unique and it is not possible to unambiguously decrypt the message. The problem is that since "A" has numeric value 0 and "N" has numeric value 13, they both give the same encrypted numeric value ($2 \cdot 0 \mod 26 = 0$ for "A", while the symbol "N" also gives $2 \cdot 13 \mod 26 = 26 \mod 26 = 0$). Thus, when the encrypted text contains the letter "A", we do not know whether it is encrypted from "A" or "N", and therefore the decoding is not unique. In fact a key n works only if it has a modular inverse, i.e., there is an m such that $n \cdot m \mod 26 = 1$.

Implementation - Part 3:
Implement the cipher in the Multiplicative class, as a subclass of `Cipher`. In particular, make sure the method that generates keys behaves correctly.

3.3 Affine cipher

One problem with the multiplication cipher is that it always encodes an "A" as an "A". That is because the number value of "A" is 0, the encoded value is $0 \cdot n = 0$ with any valid key n). The issue can be overcome by using a combination of multiplication cipher and Caesar cipher, called Affine cipher. It uses a tuple of two integers as the key, $n = (n_1, n_2)$, where n_1 is used for the Multiplication cipher and n_2 for the Caesar cipher. Here you see the result of encoding with key $n = (3, 2)$. We first use Multiplication cipher with key 3, then Caesar with key 2.

Table 6: Affine cipher with key=2

Clear text	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
numeric val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Coded val.	2	5	8	11	14	17	20	23	0	3	6	9	12	15	18	21	24	1	4	7	10	13	16	19	22	25
Coded text	C	F	I	L	O	R	U	X	A	D	G	J	M	P	S	V	y	B	E	H	K	N	Q	T	W	Z

When decoding, it is important to do the operations in the opposite order, that is, first Caesar with key 24 (which decodes the use of key $n_2 = 2$) and then multiplication cipher with key 9 (which decodes the use of key $n_1 = 3$).

Implementation - Part 4:
Implement the cipher in the `Affine` class, as a subclass of `Cipher`.

4 "Unbreakable" cipher

It is reasonably okay to hack these ciphers, for example, by looking at how often each letter occurs in the cipher text, and comparing it to what is common in natural language[1].

In order to make the code more difficult to break, the so-called "unbreakable" cipher was found in the 18th century. It has a key-word instead of a single key-value. For example, let's use the key-word "PIZZA" and let's see how we can encode the message "HEMMELIGHET". The

[1]See for example `https://en.wikipedia.org/wiki/Letter_frequency`.

first thing we do is print the clear text and translate it into numeric values. Then we repeat the keyword as many times as we need to get up to the length of the message (in the example we need a little over 2 times PIZZA), and translate the letters to their numeric values. The number values are added (mod 26), which gives the row of coded number values. These numbers are converted into letters, giving the coded text "WMLLEAQFGEI". Notice that there are three E's in the clear text and they are encrypted differently: the first becomes "M" and the other two remain "E". That is, a hacker cannot recover the clear text with one-to-one symbol mapping.

Table 7: "Unbreakable" cipher with keyword=PIZZA

Clear text	H	E	M	M	E	L	I	G	H	E	T
numeric val.	7	4	12	12	4	11	8	6	7	4	19
Keyword	P	I	Z	Z	A	P	I	Z	Z	A	P
Key-value	15	8	25	25	0	15	8	25	25	0	15
Coded val.	22	12	11	11	4	0	16	5	6	4	8
Coded text	W	M	L	L	E	A	Q	F	G	E	I

To decrypt the code, we need another key-word that matches what we used to create the encrypted text. Denote e_i the value of the ith symbol in the encryption key-word, and d_i the value of the ith symbol in the decryption key-word. They fulfill d_i =(alphabet_size - e_i) mod alphabet_size. In the description we use alphabet of size 26. So the encryption key-word "PIZZA" (values=[15,8,25,25,0]) corresponds to decryption key-word "LSBBA" (values=[11,18,1,1,0]). Again note that you should instead use alphabet of size 95 in your implementation.

Implementation - Part 5:
Implement the cipher in the `Unbreakable` class, as a subclass of `Cipher`.

4.1 RSA

All the ciphers we have described so far have the weakness that the sender and receiver must match their keys, and if one of the keys becomes known to a third party, the encrypted message can easily be hacked. This is not the case for the RSA cipher. For RSA encryption, the receiver announces which key to be used in encryption. This is almost harmless because it is difficult (computationally an NP-hard task) to find the decryption key that matches it. There is solid mathematical guarantee based on prime numbers, but here we skip the lengthy details, while focusing on the encryption and decryption procedure.

4.1.1 Encryption keys

The first step is to find encryption keys. Receiver does the following:

1. Generate two random prime numbers p and q. To implement this job, you can use the `generate_random_prime` function in the `cypto-util.py` file if desired. The method takes b (number of bits) as input, and generates a prime number that is between $2b$ and $2b+1$. You can run the function to generate p and q, but since it is required that $p \neq q$, so you may need to run the function more than twice and ensure two different prime numbers are obtained.

2. Define $n = p \cdot q$ and $\phi = (p-1)(q-1)$.

3. Select e as a random integer greater than 2 and less than ϕ.
   Here `random.randint`(3, ϕ - 1) can be used.

4. Set d as modular inverse to e with respect to ϕ, i.e., d is chosen such that $d \cdot e = 1 \mod \phi$.
   Remember that the modular inverse is in the `crypto-util.py` file.

5. After the keys have been generated:

   - (n, e) is the key that the sender should use to encrypt messages to be sent to the receiver. The encryption key can be freely published without breaking the cipher (easily).
   - (n, d) is secret and retained by the receiver to decrypt the messages.

4.1.2 Encryption of an integer message

To look at what happens when the sender encrypt a message, let's first assume that a positive integer t ($0 \leq t < n$) needs to be sent. This is encrypted with $c = te \mod n$. This calculation is done efficiently with Python command `power(t, e, n)`. Receiver decrypts the message by calculating $t' \leftarrow cd \mod n$. It can be shown that the decoding is unique and correct, i.e. $t' = t$ (details bypassed).

4.1.3 Encryption of text messages

The next challenge is to "translate" a text message to integer t, so that we can use the encryption method for integers described above. We do this by taking the text, translating it into a byte string, and then re-encoding the byte string to an (unsigned) int. Suppose we have the text string "CODE". The ASCII values for the letters are 67 ("C"), 79 ("O"), 68 ("D") and 69 ("E"). We use bit string length 8 since the ASCII alphabet is 256 characters long and $\log_2 256 = 8$. Then the binary values in string for these four numbers are "01000011", "01001111", "01000100" and "01000101" respectively. The overall byte string representation for "CODE" can be found by putting the individual strings together, i.e., "01000011010011110100010001000101". This is the binary representation of the number 1129268293, which is thus the number we use for "CODE". We can send this number through the integer encryption as indicated above.

If the message is very long, this integer may be too large and invalid because $t < n = p \cdot q$ is required for the encoding of integer t. Therefore, instead of taking the whole message as a single number, we can take blocks of a fixed length l, and let each block define an integer. For example, if we want to specify "CODE" with block length $l = 2$ it will give us the blocks "CO" and "DE". In general, we get a list of message parts that can be individually encoded into integers which we can then run through the encryption. Typically, one would choose $l \leq b/4$, where b was the number of bits used to generate p and q. A common choice is $b = 1024$ and $l = 256$, but for faster execution of the code, for example, $b = 8$, $l = 1$ can be used in the project.

The `blocks_from_text` function in `cypto-util.py` does a lot of this work for you. The function take a text as input and returns the list of integers ready for encryption. Similarly, the receiver will need to reassemble the (decrypted) integers into a text message again. Here you can use `text_from_blocks`, which are also located in `cypto-util.py`.

Implementation - Part 6: Implement the cipher in the class `RSA`, as a subclass of `Cipher`.

4.2 Hacking of cipher

All the ciphers we have seen except RSA can easily be hacked. For example, if we know that the sender and receiver are using the `Caesar`-cipher, we can try with key $n = 0, 1, 2, \ldots$ and check

what makes the best sense. Each candidate key generates a suggested clear text that a hacker can validate, for example by looking at each word in the candidate text and seeing if it is an "approved" word. This is brute-force hacking, and it works surprisingly well for most ciphers we have gone through.

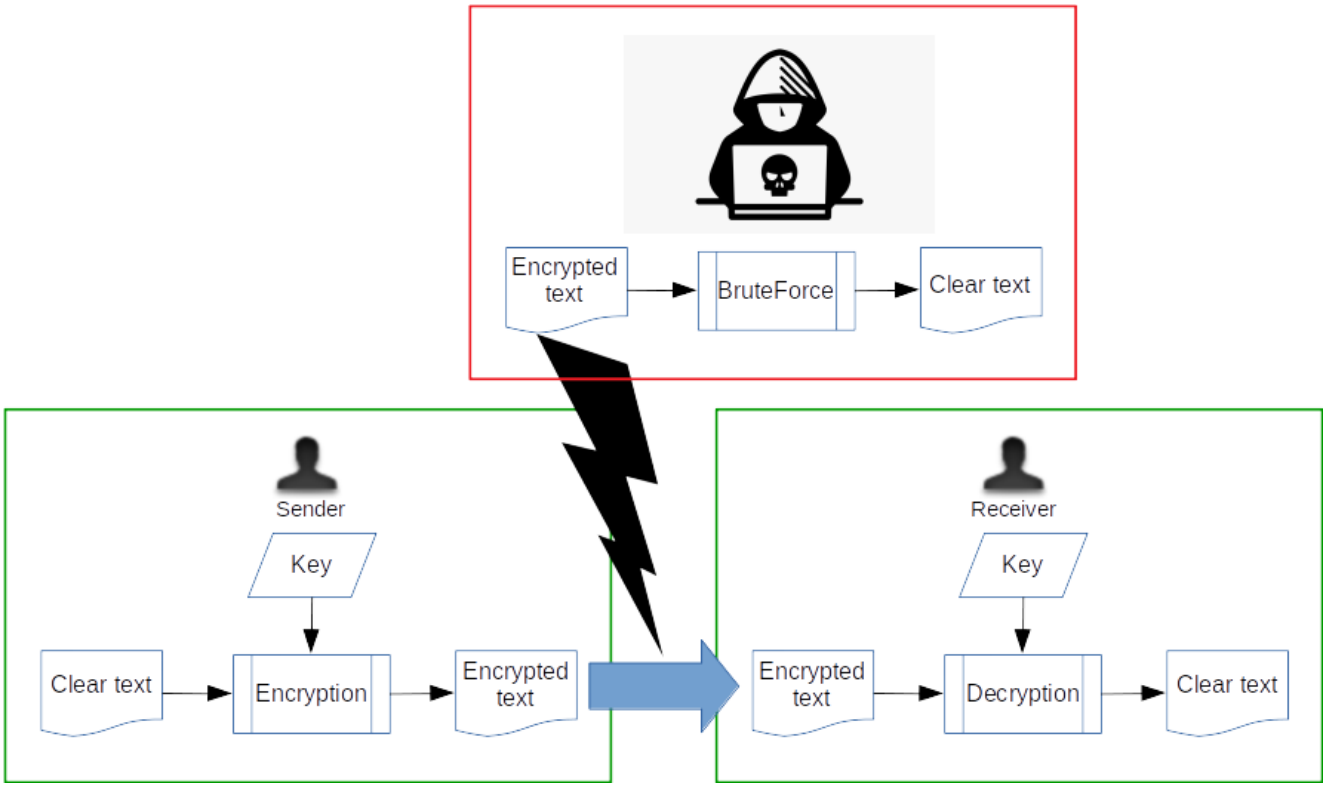

Figure 2: The hacker gets hold of the encrypted text and can hack the message in a brute-force manner by trying all the keys.

Implement a hacker class that can hack encrypted text in a brute-force manner. It will evaluate the various alternative keys by looking at whether the words generated as clear text are English words. You can use a list of English words in the file `english_words.txt` located in BLACKBOARD. All ciphers described here except RSA should be hacked by your hacker. For the sake of simplicity, it is smart to extend the implementation of each cipher so that it can tell itself the possible keys. For example, this is defined as range(0, alphabet_size) for `Caesar`.

As candidate keys for the `Unbreakable` cipher, you can use all the words in `english_words.txt`.

Implementation - Part 7:
Implement the `Hacker` class, which can either be a subclass of `Person` or of `Receiver`; Choose what you think is most natural for your code. The class should have a method that takes an encrpyted English text and a cipher as input, and returns the best possible decoding of the text after testing candidate keys until it is satisfied. Your hacker should be able to hack all described ciphers except RSA.

5 What are required to pass the task

To pass this task you must:

- Solve all the required part assignments
- You must do the work alone and have it approved by the deadline.

- The system must be implemented with object-oriented Python.
- Pylint score 8.0 or higher.

— o —