

Lab 02: Classical Ciphers

In this lab students will get an understanding of some common classical cryptography schemes, and learn the Python code that will bring all the topics together. Specifically, students will gain cryptographic knowledge about Substitution Ciphers, Caesar Cipher, Vigenère Cipher, Column Transposition, and Affine Cipher.

Lab 02 Outcomes:

- Explore the basics of encryption schemes
- Explore the use of historical ciphers and their cryptanalysis
- Gain an understanding of why it is critical to use well-established encryption algorithms

Instructor Note:

Perform an understanding practice of given different ciphers codes and after execution write line wise code description. Lastly perform Home- Graded Lab Tasks to assess your understanding level.

1) Solved Lab Activities

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>25</i>	<i>Medium</i>	<i>CLO-6</i>
<i>2</i>	<i>30</i>	<i>Medium</i>	<i>CLO-6</i>
<i>3</i>	<i>35</i>	<i>Medium</i>	<i>CLO-6</i>
<i>4</i>	<i>30</i>	<i>High</i>	<i>CLO-6</i>
<i>5</i>	<i>30</i>	<i>High</i>	<i>CLO-6</i>
<i>6</i>	<i>30</i>	<i>High</i>	<i>CLO-6</i>
<i>Home- Graded Lab Tasks</i>		<i>High</i>	<i>CLO-6</i>

Practice -Lab Activity 1: Substitution Ciphers

The substitution cipher simply substitutes one letter in the alphabet for another based upon a cryptovariable. The substitution involves shifting positions in the alphabet. This includes the Caesar cipher and ROT-13, which will be covered shortly. Examine the following example:

Plaintext: WE HOLD THESE TRUTHS TO BE SELF-EVIDENT, THAT ALL MEN ARE CREATED EQUAL.

Ciphertext: ZH KROG WKHVH WUXWKV WR EH VHOI-HYLGHQW, WKDW DOO PHQ DUH FUHDWHG HTXDO.

The Python syntax to both encrypt and decrypt a substitution cipher is presented next. This example shows the use of

```
key = 'abcdefghijklmnopqrstuvwxyz'

def enc_substitution(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result

def dec_substitution(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
            i = (key.index(l) - n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result

origtext = 'We hold these truths to be self-evident, that all men are created equal.'
ciphertext = enc_substitution(13, origtext)

plaintext = dec_substitution(13, ciphertext)

print("Original Text:", origtext)
print("Ciphertext:", ciphertext)
print("Decrypted Text:", plaintext)
```

CODE OUTPUT:

```
Original Text: We hold these truths to be self-evident, that all men are created equal.
Ciphertext: jr ubey gurer gehguf gb or fry-svirqrag, gung nyy zra ner perngrq rdhn y.
Decrypted Text: we hold these truths to be self-evident, that all men are created equal.
```

seful link to understand Try Except in python.

https://www.w3schools.com/python/python_try_except.asp

```
try:
    print(x)
except:
    print("An exception occurred")
```

```
An exception occurred
```

Practice -Lab Activity 2: Transposition Ciphers:

A simple example for a transposition cipher is columnar transposition cipher where each character in the plain text is written horizontally with specified alphabet width. The cipher is written vertically, which creates an entirely different cipher text. Consider the plain text hello world, and let us apply the simple columnar transposition technique as shown below:

h	e	l	l
o	w	o	r
l	d		

The plain text characters are placed horizontally and the cipher text is created with vertical format as: holewdlo lr. Now, the receiver has to use the same table to decrypt the cipher text to plain text.

Code

The following program code demonstrates the basic implementation of columnar transposition technique:

```
def split_len(seq, length):
    return [seq[i:i + length] for i in range(0, len(seq), length)]

def encode(key, plaintext):
    order = {
        int(val): num for num, val in enumerate(key)
    }
```

```

ciphertext = ''
for index in sorted(order.keys()):
    for part in split_len(plaintext, len(key)):
        try:
            ciphertext += part[order[index]]
        except IndexError:
            pass
    return ciphertext

print(encode('3214', 'HELLO'))

```

Code Output

LEHOL

Explanation

- Using the function `split_len()`, we can split the plain text characters, which can be placed in columnar or row format.
- `encode` method helps to create cipher text with key specifying the number of columns and prints the cipher text by reading characters through each column.
-

Practice -Lab Activity 3: Caesar Cipher

The Caesar cipher is one of the oldest recorded ciphers. De Vita Caesarum, Divus Iulius (“The Lives of the Caesars, the Deified Julius), commonly known as The Twelve Caesars, was written in approximately 121 CE. In The Twelve Caesars, it states that if someone has a message that they want to keep private, they can do so by changing the order of the letters so that the original word cannot be determined. When the recipient of the message receives it, the reader must substitute the letters so that they shift by four positions.

Simply put, the cipher shifted letters of the alphabet three places forward so that the letter A was replaced with the letter D, the letter B was replaced with E, and so on. Once the end of the alphabet was reached, the letters would start over: \

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
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	C

The Caesar cipher is an example of a mono-alphabet substitution. This type of substitution substitutes one character of the ciphertext from a character in plaintext. Other examples that include this type of substitution are Atbash, Affine, and the ROT-13 cipher. There are many flaws with this type of cipher, the most obvious of which is that the encryption and decryption methods are fixed and require no shared key. This would allow anyone who knew this method to read Caesar’s encrypted messages with ease. Over the years, there have been several variations that include ROT-13, which shifts the letters 13

places instead of 3. We will explore how to encrypt and decrypt Caesar cipher and ROT-13 codes using Python.

For example, given that x is the current letter of the alphabet, the Caesar cipher function adds three for

encryption and subtracts three for decryption. While this could be a variable shift, let's start with the original shift of 3: $\text{Enc}(x) = (x + 3) \% 26$ $\text{Dec}(x) = (x - 3) \% 26$

These functions are the first use of modular arithmetic; there are other ways to get the same result, but this is the cleanest and fastest method. The encryption formula adds 3 to the numeric value of the number. If the value exceeds 26, which is the final position of Z, then the modular arithmetic wraps the value back to the beginning of the alphabet. While it is possible to get the ordinal (ord) of a number and convert it back to ASCII, the use of the key simplifies the alphabet indexing. You will learn how to use the ord() function when we explore the Vigenère cipher in the next section. In the following Python recipe, the enc_caesar function will access a variable index to encrypt the plaintext that is passed in.

```
key = 'abcdefghijklmnopqrstuvwxyz'

def enc_caesar(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result
```

The output of this should result in the following:

```
zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh fuhdwhg htxdo.
```

Decryption

The reverse in this case is straightforward. Instead of adding, we subtract. The decryption would look like the following:

```

key = 'abcdefghijklmnopqrstuvwxyz'

def dec_caesar(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
            i = (key.index(l) - n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result

ciphertext = 'zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh
fuhdwhg htxdo.'

plaintext = dec_caesar(3, ciphertext)

print(plaintext)

```

```

we hold these truths to be self-evident, that all men are created equal.

```

Practice -Lab Activity 4: ROT-13

Now that you understand the Caesar cipher, take a look at the ROT-13 cipher. The unique construction of the ROT-13 cipher allows you to encrypt and decrypt using the same method. The reason for this is that since ROT-13 moves the letter of the alphabet exactly halfway, when you run the process again, the letter goes back to its original value.

To see the code behind the cipher, take a look at the following:

```

key = 'abcdefghijklmnopqrstuvwxyz'
def enc_dec_ROT13(n, plaintext):
    result = "

```

```

for l in plaintext.lower():
    try:
        i = (key.index(l) + n) % 26
        result += key[i]
    except ValueError:
        result += l
return result
plaintext = 'We hold these truths to be self-evident, that all men are created equal.'
ciphertext = enc_dec_ROT13(13, plaintext)
print(ciphertext)
# Decrypt the ciphertext by running the same function with the same shift of 13
plaintext = enc_dec_ROT13(13, ciphertext)
print(plaintext)

```

Code output:

```

jr ubyq gurfr gehguf gb or frys-rivqrag, gung nyy zra ner perngrq rdhny.
we hold these truths to be self-evident, that all men are created equal.

```

Whether we use a Caesar cipher or the ROT-13 variation, brute-forcing an attack would take at most 25 tries, and we could easily decipher the plaintext results when we see a language we understand. This will get more complex as we explore the other historical ciphers; the cryptanalysis requires frequency analysis and language detectors. We will focus on these concepts in upcoming chapters.

Practice -Lab Activity 5: Vigenère Cipher

The Vigenère cipher consists of using several Caesar ciphers in sequence with different shift values. To encipher, a table of alphabets can be used, termed a tabula recta, Vigenère square, or Vigenère table. It consists of the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyword.

Here's an example:

Keyword: DECLARATION

D	E	C	L	A	R	A	T	I	O	N
3	4	2	11	0	17	0	19	8	14	13

Plaintext: We hold these truths to be self-evident, that all men are created equal.

Ciphertext: zi jzlu tamgr wwwehj th js fhph-pvzdxvh, gkev llc mxv oeh gtpakew mehdp.

To create a numeric key such as the one shown, use the following syntax. You should see the output [3, 4, 2, 11, 0, 17, 0, 19, 8, 14, 13]:

```
def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i].upper()) - 65 # Ensure uppercase
        keyArray.append(keyElement)
    return keyArray

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
print(key)
```

```
[3, 4, 2, 11, 0, 17, 0, 19, 8, 14, 13]:
```

```
def shiftEnc(c, n):
    if c.isalpha():
        return chr(((ord(c) - ord('A') + n) % 26) + ord('A'))
    else:
        return c # Keep non-alphabet characters unchanged

def enc_vigenere(plaintext, key):
    secret = ""
    for i in range(len(plaintext)):
        secret += shiftEnc(plaintext[i], key[i % len(key)])
    return secret

def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i].upper()) - 65 # Ensure uppercase
        keyArray.append(keyElement)
    return keyArray

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
plaintext = 'ALL MEN ARE CREATED EQUAL'
ciphertext = enc_vigenere(plaintext, key)
print(ciphertext)
```

Code Output:

```
DPN MVN IFR GTPAKEW SDXEN
```

When you know the key, such as in this case, you can decrypt the Vigenère cipher with the

following:

```
def shiftDec(c, n):  
    if c.isalpha():  
        c = c.upper()  
        return chr(((ord(c) - ord('A') - n) % 26) + ord('A'))  
    else:  
        return c # Return non-alphabetic characters unchanged  
  
def dec_vigenere(ciphertext, key):  
    plain = ""  
    for i in range(len(ciphertext)):  # Iterate over each character in ciphertext  
        if ciphertext[i].isalpha() else ciphertext[i]  
        plain += shiftDec(ciphertext[i], key[i % len(key)])  
    return plain  
  
def key_vigenere(key):  
    keyArray = []  
    for i in range(0, len(key)):  
        keyElement = ord(key[i].upper()) - 65 # Ensure uppercase  
        keyArray.append(keyElement)  
    return keyArray  
  
secretKey = 'DECLARATION'  
key = key_vigenere(secretKey)  
  
ciphertext = 'DNP RYR OIU OIKORO VZZIV' # You may replace this with the actual ciphertext  
decoded = dec_vigenere(ciphertext, key)  
  
print(decoded)
```

Code Output:

```
AJN RHR GUH KGZAO NLMFR
```

We will perform cryptanalysis by creating a random key that will use the same encryption function, and then we will use frequency analysis to help find the appropriate key. For now, it is more important to understand how the Python code works with this cryptography scheme.

Lab Activity 5: One-Time Pad Function

Now that you have seen how XOR works, it will be easier to understand the one-time pad. OTP takes a random sequence of 0s and 1s as the secret key and will then XOR the key with your plaintext message to produce the ciphertext:

GEN: choose a random key uniformly from $\{0,1\}^{\ell}$ (the set of binary strings of length ℓ)
ENC: given $k \in \{0,1\}^{\ell}$ and $m \in \{0,1\}^{\ell}$ then output is $c := k \oplus m$
DEC: given $k \in \{0,1\}^{\ell}$ and $c \in \{0,1\}^{\ell}$, the output message is $m := k \oplus c$

The output given by the OTP satisfies Claude Shannon's notion of perfect secrecy (see "Shannon's Theorem"). Imagine all possible messages, all possible keys, and all possible ciphertexts. For every message and ciphertext pair, there is one key that causes that message to encrypt to that ciphertext. This is really saying that each key gives you a one-to-one mapping from messages to ciphertexts, and changing the key shuffles the mapping without ever repeating a pair.

The OTP remains unbreakable as long as the key meets the following criteria:

- The key is truly random.
- The key the same length as the encrypted message.
- The key is used only once!

When the key is the same length as the encrypted message, each plaintext letter's subkey is unique, meaning that each plaintext letter could be encrypted to any ciphertext letter with equal probability. This removes the ability to use frequency analysis against the encrypted text to learn anything about the cipher. Brute-forcing the OTP would take an incredible amount of time and would be computationally unfeasible, as the number of keys would equal 26 raised to the power of the total number of letters in the message. In Python 3.6 and later, you will have the option to use the `secrets` module, which will allow you to generate random numbers. The function `secrets.randbelow()` will return random numbers between zero and the argument passed to it:

```
>>> import secrets
>>> secrets.randbelow(10)
3
>>> secrets.randbelow(10)
1
>>> secrets.randbelow(10)
7
```

```
*cipher text.py - C:/Users/PAKISTAN/Desktop/cipher text.py (3.11.5)*
File Edit Format Run Options Window Help
import secrets
|

===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
Traceback (most recent call last):
  File "C:/Users/PAKISTAN/Desktop/cipher text.py", line 1, in <module>
    secrets.randbelow(10)
NameError: name 'secrets' is not defined
>>>

===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
>>> secrets.randbelow(10)
>>> 2
>>> |
```

```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

= RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py
DPN MVN IFR GTPAKEW SDXEN
>>>

===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
AJN RHR GUH KGZAO NLMFR
>>>

===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
Traceback (most recent call last):
  File "C:/Users/PAKISTAN/Desktop/cipher text.py", line 1, in <module>
    secrets.randbelow(10)
NameError: name 'secrets' is not defined
>>>

===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
>>> secrets.randbelow(10)
>>> 2
>>> secrets.randbelow(10)
>>> 5
>>> secrets.randbelow(10)
>>> 6
```

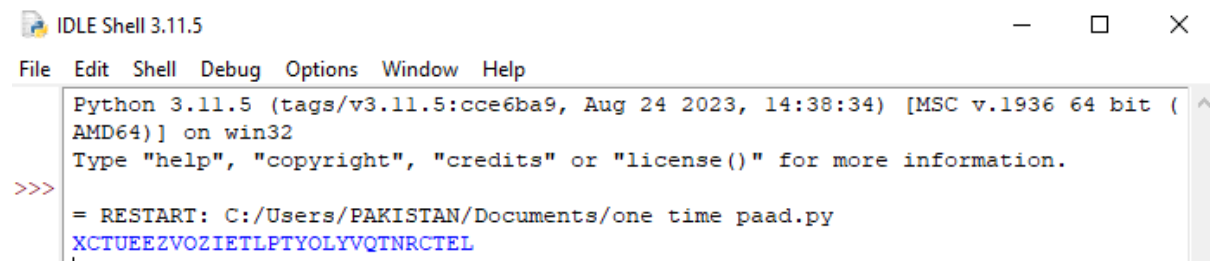
You can generate a key equal to the length of the message using the following in the Python code.

```
import secrets

msg = "helloworldthisistheonetimepad"
key = ""
for i in range(len(msg)):
    key += secrets.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ') # Use uppercase letters
print(key)
```

Code Output:

TWPIRWVUSCGQPYTAHDNDGPIDTKCTB



```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/PAKISTAN/Documents/one time paad.py
XCTUEEZVOZ1ETLPTYOLYVQTNRCTEL
```

One time pad code in python

```
import secrets

def key_generation(length):
    """
    Generates a random binary key of a given length.
    """
    return "".join(secrets.choice('01') for _ in range(length))

def xor_operation(binary_str1, binary_str2):
    """
    Performs bitwise XOR between two binary strings.
    """
    return "".join(str(int(a) ^ int(b)) for a, b in zip(binary_str1, binary_str2))

def encrypt(key, message):
    """
    Encrypts the message using the one-time pad encryption (XOR operation).
    """
    return xor_operation(key, message)

def decrypt(key, ciphertext):
    """
    Decrypts the ciphertext using the one-time pad decryption (XOR operation).
    """
    return xor_operation(key, ciphertext)

# Test the one-time pad algorithm
l = 10 # Length of the binary string (can be set to any desired length)
message = "".join(secrets.choice('01') for _ in range(l)) # Generate a random binary message

# Key generation
```

```

key = key_generation(l)

# Encryption
ciphertext = encrypt(key, message)

# Decryption
decrypted_message = decrypt(key, ciphertext)

# Display the results
print("Message:      ", message)
print("Key:          ", key)
print("Ciphertext:    ", ciphertext)
print("Decrypted Text: ", decrypted_message)

# Ensure the decrypted message matches the original
assert decrypted_message == message, "Decryption failed! The original message and decrypted message don't match."

```

Code Output:

```

Message:      0111101111
Key:          0010010001
Ciphertext:   0101111110
Decrypted Text: 0111101111

```

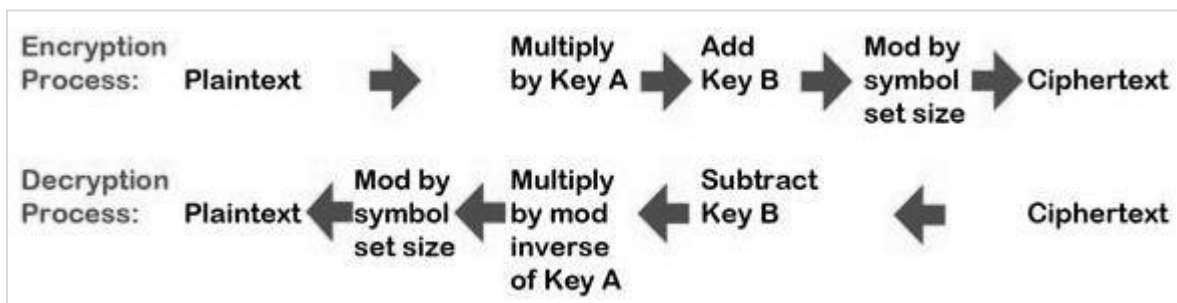
1) Home- Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the practice lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Task 1

Affine Cipher is the combination of Multiplicative Cipher and Caesar Cipher algorithm. The basic implementation of affine cipher is as shown in the image below:

Write python code for above mentioned Affine Cipher with code output.



Task 2

While using Caesar cipher technique, encrypting and decrypting symbols involves converting the values into numbers with a simple basic procedure of addition or subtraction. If multiplication is used to convert to cipher text, it is called a wrap-around situation. Consider the letters and the associated numbers to be used as shown below:

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

The numbers will be used for multiplication procedure and the associated key is 7. The basic formula to be used in such a scenario to generate a multiplicative cipher is as follows:

$$(\text{Alphabet Number} * \text{key}) \bmod (\text{total number of alphabets})$$

Write Python code for multiplicative cipher and also provide how it can be hacked. Your programs should include code output mechanism in any form.

Task 3

CrypTool: CrypTool is one of the most comprehensive open-source cryptography tools. It includes tutorials, simulations, and visualizations of classical and modern cryptographic algorithms. You can analyze ciphers, break them using various techniques (e.g., frequency analysis, brute force), and experiment with encryption and decryption. Practice encrypting plaintext using different ciphers and then try decrypting it.

- Select a cipher like Vigenère from the toolbox.
- Enter the plaintext and the key.
- Run the tool to encrypt the message.
- Try decrypting the ciphertext by breaking it using key guessing or cryptanalysis techniques like frequency analysis.

Perform Cryptanalysis: Perform / Explore CrypTool for cryptanalysis of Lab practice activities of ciphers.

- **Frequency Analysis:** Useful for classical ciphers like Caesar or Vigenère.
- **Known-Plaintext Attack:** Some tools allow you to input known plaintext to derive keys.

Task 4

Perform given tasks for following Transposition Cipher code.

```
def split_len(seq, length):
    return [seq[i:i + length] for i in range(0, len(seq), length)]

def encode(key, plaintext):
    order = {
        int(val): num for num, val in enumerate(key)
    }
    ciphertext = ''
    for index in sorted(order.keys()):
        for part in split_len(plaintext, len(key)):
            try:
                ciphertext += part[order[index]]
            except IndexError:
                pass
    return ciphertext

print(encode('3214', 'HELLO'))
```

1) Handle Different Key Sizes

Modify the encode function to handle cases where the length of the key is not equal to the length of the plaintext. **Task:** Add padding to the plaintext when it is shorter than the key.

2) Decode Function

Create a decode function that reverses the encode process. **Task:** Write a function decode(key, ciphertext) that decipheres the encrypted message and returns the original plaintext.

3) Support for Uppercase and Lowercase Letters

Modify the code to preserve the original case (uppercase and lowercase letters) in the plaintext.

- **Task:** Adjust the encode function to handle both uppercase and lowercase letters, so it doesn't always convert to lowercase.

4) Encrypt Full Sentences with Spaces

- Modify the encode function to handle spaces and punctuation without removing them.
- **Task:** Ensure that spaces and punctuation are preserved and not encrypted when encoding full sentences.

5) Dynamic Key Generation

Automatically generate a random key if the user does not provide one. **Task:** Write a function that generates a random key based on the length of the plaintext.

6) Add a Menu Interface

Create a simple command-line interface where the user can choose to encode or decode a message. **Task:** Write a menu system where the user can input a choice to either encode, decode, or exit.

Note: Please note that from learning perspective try to work on you own implementation so that in Midterm assessments you will be able to solve challenging task yourself.