Laboratory 2

Mos Daniela, 935

The Bellaso (Vigenère) cipher

The purpose of this application is to present the implementation of the Bellaso (Vigenère) cipher. It is a polyalphabetic substitution cipher, where the ciphertext is obtained by addition modulo n of the repeated keyword, where n is the length of the alphabet.

The constants for this application are defined below: the alphabet, consisting of all (uppercase) letters from A to Z and space, saved as a map, the length of the alphabet (n = 27, for this application), and the keys, saved in a separate list, again, to be easier to find them by index.

```
<<Constants>>=
alphabet = {
    ' ': 0, 'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5, 'F': 6, 'G': 7,
    'H': 8, 'I': 9, 'J': 10, 'K': 11, 'L': 12, 'M': 13, 'N': 14,
    'O': 15, 'P': 16, 'Q': 17, 'R': 18, 'S': 19, 'T': 20, 'U': 21,
    'V': 22, 'W': 23, 'X': 24, 'Y': 25, 'Z': 26,
}
alphabet_len = len(alphabet.keys())
alphabet_keys = list(alphabet.keys())

@
```

Mathematical aspects

The set of plaintext characters (P), the set of ciphertext characters (C) and the set of possible keys (K) are the same, then we can write $P = C = K$. The keyspace is $K = \{k = (k_1, ..., k_m) | k_i \in Z_n, i = \overline{1, n}\} = (Z_n)^m$, where m is the size of the key. Now having the keyspace and the encryption and decryption functions
$e_k(x_1, x_2, ..., x_m) = (x_1 + k_1, x_2 + k_2, ..., x_m + k_m)$ mod n, $\forall k = (k_1, k_2, ..., k_m)$ key $\in K$ and $(x_1, x_2, ..., x_m) \in (Z_n)^m$,
$d_k(y_1, y_2, ..., y_m) = (y_1 - k_1, y_2 - k_2, ..., y_m - k_m)$ mod n, $\forall k = (k_1, k_2, ..., k_m)$ key $\in K$ and $(y_1, y_2, ..., y_m) \in (Z_n)^m$,
we find the underlying group $(K, *) = ((Z_n)^m, +)$. To verify this, we check the

group axioms:

1. $\forall a, b \in (Z_n)^m$, then $a, b \in (Z_n)^m$
   a (mod n) + b (mod n) = a + b (mod n), then a + b $\in (Z_n)^m$

2. Associativity: $\forall a, b, c \in (Z_n)^m$, then $(a + b) + c = a + (b + c) \in (Z_n)^m$
   We know that addition modulo n is associative in $((Z_n)^m, +)$

3. Identity element: $\exists e \in (Z_n)^m$ such that $a + e = e + a = a, a \in (Z_n)^m$
   We know that e = 0 is the identity element for addition, then
   a + 0 (mod n) = 0 + a (mod n) = a (mod n) holds for any $a \in (Z_n)^m$

4. Inverse element: for each element a $\in (Z_n)^m$, there exists $a^{-1} \in (Z_n)^m$ such that $a + a^{-1} = a^{-1} + a = e$
   Having e = 0 identity element, the inverse of a for addition is a^{-1} =-a:
   a + (-a) (mod n) = (-a) + a (mod n) = 0 (mod n) = 0 holds for any $a \in (Z_n)^m$

From 1, 2, 3, 4, we conclude that $((Z_n)^m$, +) is a group. Knowing that addition is also commutative, then $((Z_n)^m$, +) is an abelian group. Now we can prove that decryption with k equals encryption with k^{-1} and viceversa.

$d_k(x) = x - k$ (mod n) $= x + (-k)$ (mod n) $= x + k^{-1}$ (mod n) $= e_{k^{-1}}(x)$ and
$e_k(x) = x + k$ (mod n) $= x - (-k)$ (mod n) $= x - k^{-1}$ (mod n) $= d_{k^{-1}}(x)$

Also, we can prove that $e_{k_1 * k_2}(x) = (e_{k_1} \circ e_{k_2})(x), \forall x \in P^m$:
$e_{k_1 * k_2}(x) = x + (k_1 + k_2)$ (mod n) $= (x + k_2) + k_1$ (mod n) $= (e_{k_1}(e_{k_2}(x)) = (e_{k_1} \circ e_{k_2})(x)$ and the inverse holds.

Encryption

This algorithm takes as input a message in plaintext and a keyword. The steps of the encryption mechanism are as follows:

1. Split the plaintext into blocks of the same length as the keyword. The last block will only contain as many letters from the keyword as they are left in the block.

2. Change the symbols into their numerical values associated in the defined alphabet and add them modulo n.

3. The result is transformed back into a symbol from the alphabet and the ciphertext is the concatenation of these blocks.

Implementation

```
<<Encrypt>>=
def encrypt(text, key):
    cipher = ""
    for x in range(0, len(text), len(key)):
        chunk = text[x: x + len(key)]
        for letter, key_letter in zip(chunk, key):
            new_letter = (alphabet[letter] + alphabet[key_letter]) \
            % alphabet_len
            cipher += alphabet_keys[new_letter]
    return cipher

@
```

Decryption

To decrypt the ciphertext, the mechanism is similar, but instead of adding the numerical values of the keyword to each block, subtract them.

Implementation

```
<<Decrypt>>=
def decrypt(cipher, key):
    plaintext = ""
    for x in range(0, len(cipher), len(key)):
        chunk = cipher[x: x + len(key)]
        for letter, key_letter in zip(chunk, key):
            old_letter = (alphabet[letter] - alphabet[key_letter]) \
            % alphabet_len
            plaintext += alphabet_keys[old_letter]
    return plaintext

@
```

Example

To take an example, we encrypt the plaintext: THIS IS AN EXAMPLE
using the keyword: THE KEY

THIS IS| AN EXA|MPLE

20 8 9 19 0 9 19 | 0 1 14 0 5 24 1 | 13 16 12 5
THE KEY|THE KEY|THE
20 8 5 0 11 5 25 | 20 8 5 0 11 5 25 | 20 8 5 0

By adding the values, we obtain:
40 16 14 19 11 14 44 | 20 9 19 0 16 29 26 | 33 24 17 5

By computing modulo 27:
13 16 14 19 11 14 17 | 20 9 19 0 16 2 26 | 6 24 17 5

And converting back to letters, we obtain the ciphertext:
MPNSKNQTIS PBZFXQE

To decrypt, the process is similar:
MPNSKNQ|TIS PBZ|FXQE
13 16 14 19 11 14 17 | 20 9 19 0 16 2 26 | 6 24 17 5
THE KEY|THE KEY|THE
20 8 5 0 11 5 25 | 20 8 5 0 11 5 25 | 20 8 5 0

Subtract and wrap-around if necessary:
20 8 9 19 9 19 | 0 1 14 0 5 24 1 | 17 16 12 5

And convert back to letters:
THIS IS AN EXAMPLE

Running the application

The application can be run from the command line and the following arguments can be passed: mode, -k, keyword, input file. The mode can be either -e (encrypt) or -d (decrypt).
The input file and keyword must not contain any other characters than the ones in the alphabet (only spaces and A to Z characters, uppercase or lowercase).
The arguments need to be passed in the specified order. For example, the following command will encrypt the contents of input.txt with the keyword "key":

```
py lab2.py -e -k key input.txt
```

Testing

Having the group $((Z_n)^m, +)$, the order of a group element, denoted ord(k), is the value λ such that $k^\lambda = k + k + ... + k = 0, e = 0$ being the identity element of the group. Then knowing the key, we can get the plaintext after successively encrypting it for a number of ord(k) times, which is called the period of the key.

The order of the key in this case is
$ord(k) = ord(k_1, ..., k_m) = lcm(\frac{n}{gcd(k_1,n)}, ..., \frac{n}{gcd(k_m,n)})$, where $k = (k_1, ..., k_m) \in K$ the key used.
Since this cipher can be reduced to at most m Caesar ciphers (the letters in the keyword can repeat), we need to compute the lowest common multiple of the orders of each letter of the key. These $ord(k_i), i = \overline{1, m}$ represent the smallest number of repetitions needed to obtain the initial letter by shifting it with the same value.

The results of the encryption and decryption are only printed.

```
<<test>>=
from functools import reduce
<<Constants>>
<<File Handling>>
<<Encrypt>>
<<Decrypt>>

def gcd(a, b):
    while b != 0:
        aux = b
        b = a % b
        a = aux
    return abs(a)

def test():
    message = "All work and no play makes Jack a dull boy".upper()
    keyword = "long key".upper()

    order = []
    for letter in keyword:
        order.append(alphabet_len // gcd(alphabet[letter], alphabet_len))

    repeat = reduce(lambda a, b: a * b // gcd(a, b), order)

    print("Testing")

    def recursive_test(times, keyword, plain):
        if times > 0:
            recursive_test(times - 1, keyword, encrypt(plain, keyword))
        else:
            if plain == message:
                print(f"The order of they key is ord(k)={repeat}")
            else:
                print(f"The order of they key is not ord(k)={repeat}")
            print(f"Obtained:\n{plain}")

    recursive_test(repeat, keyword, message)

    print("Testing encryption function")
    ciphertext = encrypt(message, keyword)
    print(ciphertext)

    print("Testing decryption function")
    plaintext = decrypt(ciphertext, keyword)
    print(plaintext)

test()
@
```

Below are the implementations of the file input/ output operations. Once the encryption or decryption is done, the results are printed on the screen and saved into files.

```
<<File Handling>>=
def read_file(filename='input.txt'):
    with open(filename, 'r') as f:
        text = f.read()
    return text.strip("\n").upper()


def write_file(text, filename='output.txt'):
    with open(filename, 'w') as f:
        f.write(text)

@
```

Putting everything together:

```
<<*>>=
import sys

<<Constants>>
<<File Handling>>
<<Encrypt>>
<<Decrypt>>

if __name__ == "__main__":
    if len(sys.argv) < 5:
        print("Not enough arguments")
    else:
        option = sys.argv[1]
        key_option = sys.argv[2]
        key = sys.argv[3].upper()
        filename = sys.argv[4]
        try:
            if key_option == "-k":
                text = read_file(filename)
                if option == "-e":
                    cipher = encrypt(text, key)
                    print(cipher)
                    write_file(cipher, "encrypted.txt")
                elif option == "-d":
                    plaintext = decrypt(text, key)
                    print(plaintext)
                    write_file(plaintext, "decrypted.txt")
                else:
                    print("Wrong command format")
            else:
                print("Please respect the following format when "
                      "specifying the key: -k \"your key\"")
        except Exception:
                print("Please make sure that the input file "
                      "exists or that it respects the specified alphabet")

@
```