Universidade do Minho

Mestrado em Engenharia Informática

Engenharia de segurança

Ficha de exercício 1

Grupo 1

Rui Carlos Azevedo Carvalho - PG47633
Daniel Barbosa Miranda - PG47123
Ana Luísa Lira Tomé Carneiro - PG46983

Parte I: Criptografia – conceitos básicos

Pergunta P1.1

Segundo os princípios de Kerckhoff um sistema criptográfico não deve precisar de secretismo, podendo, inclusive, ser roubado por um inimigo sem que este cause problemas ao obter o sistema [1].

Desta forma, um sistema criptográfico deve utilizar uma chave para garantir que a informação cifrada por esta permaneça segura, ou seja, o algoritmo que é utilizado para cifrar os dados pode ser revelado de forma pública e mesmo que seja replicado ou utilizado por outro sistema este não deverá ser possível obter os dados cifrado pelo sistema original, a não ser que obtenha a chave que foi utilizada. Assim sendo, é apenas necessário manter segura a chave utilizada para cifrar a informação.

De facto, a utilização de uma chave para manter a informação indecifrável vai ao encontro do princípio de Kerckhoff indicado anteriormente, na medida em que não pertence ao sistema criptográfico em si, sendo apenas um parâmetro que lhe é passado.

Por fim, a utilização de secretismo como fator de segurança cria uma oportunidade de ataque, pois basta que alguém o obtenha para poder ter acesso aos dados cifrados por este, ou seja, a não utilização destas técnicas torna o sistema mais seguro, pois diminui várias possibilidades de ataque ao sistema.

Parte II: Exemplos de Cifras Clássicas

Pergunta P1.1

O programa inicial apresenta as 3 opções ao utilizador. Caso o utilizador escolha a primeira, ou seja, a opção que lhe permite cifrar texto, é lhe pedido que introduza valores sequenciais que correspondem às chaves das várias cifras de césar a aplicar, como se pode verificar no seguinte excerto de código:

```
def processCipher():
    key = ""
    text = input ('Text: ')

    while(key != '-1'):
        print('\nIndique a chave da próxima cifra de césar a ser aplicada.')
        print('Escreva -1 para acabar e obter o resultado da cifra')

        key = input('\nChave: ')

        if int(key) >= 0:
            text = Caesar(key = int(key)).encipher(text)

        elif key != '-1':
            print('Chave inválida')
```

```
    print('Ciphertext: ' + text + '\n')
    return text
```

De facto, a segunda opção processa-se de forma similar mas enquanto que na anterior utiliza-se a função "encipher" desta cifra de forma sequencial, nesta utiliza-se a função "decipher".

```
def processDecipher():

    #Input do ciphertext que se pretente decifrar.
    ciphertext = input ('Ciphertext: ')
    key = ""

    #Para terminar o processo de decifragem a chave que se
    # deve introduzir é -1.
    while(key != '-1'):
        print('\nIndique a chave da próxima cifra de césar a ser aplicada: ')
        print('Escreva -1 para acabar e obter o resultado da cifra.')

        key = input('\nChave: ')

        #Verifica se a chave é positiva e aplica a funcao de decipher
        #da cifra  de cesar com a chave.
        #Guarda o ciphertext resultante.
        if int(key) >= 0:
            ciphertext = Caesar(key = int(key)).decipher(ciphertext)

        elif key != '-1':
            print('Chave inválida')

    #Imprime no ecra o texto original resultante de aplicar a funcao de
    #decifra com varias chaves.
    print('Original text: ' + ciphertext + '\n')

    return ciphertext
```

A última funcionalidade do sistema consiste em criar um ataque de força bruta capaz de quebrar a aplicação de várias cifras de césar. De facto, a utilização de várias cifras de césar é o mesmo que aplicar uma só, sendo a chave desta a soma das várias cifras de césar que foram aplicadas de forma sequencial. Assim, o ataque de força bruta é exatamente igual aquele efetuado na experiência 1.2, ou seja, percorre-se as chaves de 0 a 26 (correspondentes a todas as deslocações possíveis no alfabeto) e realiza-se um *fitness score*. Este *fitness score* utiliza um ficheiro que possui a frequência de ocorrência dos vários caracteres numa dada lingua (ou conjuntos de caracteres) e compara com o texto resultante de aplicar a cifra com cada uma das chaves. Desta forma obtém-se um valor que quanto mais alto for maior será a probabilidade da chave associada ser a chave utilizada para cifrar o texto original [2].

Desta forma, ao utilizar o código da experiência 1.2 que calcula o fitness score (que também se encontra no repositório) e uma adaptação da função break_cesar que

percorre todas as chaves e obtém a que apresenta o maior valor obtém a seguinte função:

```
def break_caesar():

    stats_file = input('Statistic file: ')
    ctext = input('Ciphertext: ')
    if exists(stats_file):

        start_time = time.time()
        #Utiliza o modulo ngram_score
        fitness = ngram_score(stats_file)

        # Remove os espaços e coloca o ciphertext em uppercase
        ctext = re.sub('[^A-Z]','',ctext.upper())

        #Testa todas as chaves e guarda os
        # resultados do fitness score de cada uma delas
        scores = []
        for i in range(26):
            scores.append((fitness.score(Caesar(i).decipher(ctext)),i))

        #Obtem a chave com maior fitness score
        max_key = max(scores)

        #Imprime no ecrã a chave que obteve
        #melhor resultado e o resultado da decifragem.
        print('Chave com melhor resultado = ' + str(max_key[1]) + ':')
        print(Caesar(max_key[1]).decipher(ctext))

        end_time = time.time()

        print('Tempo decorrido: ' + str(end_time - start_time))

    else:
        print('Path inválido.')
```

Nesta é possível verificar que o ficheiro cifrado e o ficheiro que apresenta as frequências das letras ou conjuntos de letras sao dados como input e que é calculado o valor de fitness para cada chave. Por fim obtém-se a chave que apresenta o maior valor e aplicar a função de "decipher" para obter o texto original. Também é apresentado o tempo que demorou a executar este ataque.

De seguida realizou-se um teste em que se cifrou um texto aplicando 1 a 5 cifras com valores aleatórios para as suas chaves. O resultado da aplicação das várias cifras pode ser verificado dentro da pasta test em que o ficheiro ciphertext-2.txt corresponde ao resultado de aplicar duas cifras aleatórias sequenciais ao texto original.

O resultado de aplicar a função "break_cesar" a cada um destes está presente na seguinte tabela, cujos resultado podem ser verificados na pasta test, em que o ficheiro

text-2.txt corresponde ao resultado de aplicar o ataque de força bruta ao ciphertext mencionado acima. Como o texto original estava em inglês, utilizou-se um ficheiro que possui a frequência de conjuntos de 4 caracteres na língua inglesa (fonte).

Nº cifras aplicadas	Obteve o cleantext correto?	Tempo (s)
1	Sim	0.34
2	Sim	0.35
3	Sim	0.36
4	Sim	0.34
5	Sim	0.34

Figure 1: Tempos dos ataques de força bruta

Tal como se pode verificar o tempo é exatamente o mesmo para cada uma delas, sendo que em todos os casos o programa foi capaz de obter o *cleartext* correto. Assim sendo, conclui-se que aplicar várias cifras de césar em sequência produz o mesmo efeito que aplicar apenas uma, ou seja, não há um ganho em termos de segurança, pois não exige um esforço computacional maior para realizar um ataque de força bruta na aplicação sequencial de cifras de césar.

O código encontra-se de forma completa no seguinte link.

Pergunta 2.1

Numa cifra por substituição mono-alfabética cada unidade do *plaintext* é substituída por outra (pode ser pela própria também) unidade no *ciphertext* correspondente, sendo portanto este mapeamento de substituição descrito como uma função bijetiva.

Plaintext	D	A	T	A
Ciphertext	O	X	A	O

Figure 2: Substituição mono-alfabética do *plaintext* DATA

Tomando o exemplo acima, como forma de facilitar o entendimento do problema, o *ciphertext* `OXAO` nunca poderia ser decifrado como o *plaintext* `DATA`, uma vez que a letra `D` é mapeada para `O`, a letra `A` para `X`, a letra `T` para `A` e por último o mapeamento que se torna o problema, a letra `A`, novamente, para `O`. Ora, isto é um problema, pois o `O` já estava atribuído a `D` e `X` a `A`, quebrando o ponto de injetividade, e portanto bijetividade.

Em suma, não podemos ter uma letra no *plaintext* mapeada para duas no *ciphertext*, daí a situação em questão ser impossível.

Pergunta 3.1

O código desenvolvido foi obtido na hiperligação `https://github.com/serengil/crypto/blob/master/python/classical/hill.py`.

Neste tipo de cifra cada bloco de letras é substituído por uma outra unidade, recorrendo ao uso de uma matriz chave que serve tanto para cifrar como para decifrar. O código utilizado recorre ao uso das bibliotecas do Python, String para conversão de letras para números, `Numpy`, a fim de proceder à criação das matrizes no contexto da questão e `Sympy` para inversão de matrizes.

```
import string
import numpy as np
from sympy import Matrix #inverse key
```

Cifrar

```
#raw_message = "act"
raw_message = "attack is to night"
print("raw message: ",raw_message)

message = []

key = np.array([
        [3, 10, 20],
        [20, 9, 17],
        [9, 4, 17]
]) #3x3
```

O primeiro passo consiste em fornecer como input o *plaintext* que se deseja cifrar assim como a palavra chave sobre a qual se vai cifrar, sendo que no caso do código utilizado o *plaintext* já está inicialmente definido, assim como já existe um array correspondente a uma palavra chave.

```
key_rows = key.shape[0]
key_columns = key.shape[1]

if key_rows != key_columns:
        raise Exception('key must be square matrix!')
```

Caso fosse dada uma palavra chave em vez da matriz, para ser criada a matriz quadrada (chave), seria necessário que o tamanho da palavra fosse um quadrado perfeito, para fins que serão posteriormente mencionados. Este processo de verificação está representado acima e é possível denotar o uso da função `shape` do `Numpy` para obter o número de linhas e colunas e, caso estes diferissem, seria levantada uma exceção, uma vez que não se trataria de um quadrado perfeito.

```
module = 26 #english alphabet
```

O algoritmo de cifra desenvolvido opera sobre o uso do alfabeto inglês (latino), sendo portanto, como no excerto, o modulo equivalente a 26 correspondente ao comprimento do alfabeto.

Na tabela acima, temos a representação cada letra e o número pelo qual será substituída, tal como será explicado de seguida.

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
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

Figure 3: Substituição alfabética

```
for i in range(0, len(raw_message)):
        current_letter = raw_message[i:i+1].lower()
        if current_letter != ' ':
                letter_index = letterToNumber(current_letter)
                message.append(letter_index)
```

Neste excerto, o *plaintext* é percorrido de forma a que cada letra seja convertida para minúscula, usando a função `lower` do Python. Seguidamente, se essa letra não for um espaço, é transformada num número e este é adicionado ao *array* `message`, previamente inicializado. Quando se termina de percorrer o *plaintext*, teremos o *array* `message` com todos os seus valores númericos correspondentes.

```
def letterToNumber(letter):
        return string.ascii_lowercase.index(letter)
```

A transformação de letras para números mencionada, equivalente à tabela da figura 2 é feita recorrendo à função acima, que utiliza o alfabeto ASCII, para letras minúsculas, sendo compatível com o alfabeto do algoritmo em questão.

```
message = np.array(message)
message_length = message.shape[0]
print("message: ",message)

#transform message array to matrix
message.resize(int(message_length/key_rows), key_rows)
```

No excerto acima, o *array* `message` é convertido num *array* `Numpy`, sendo guardado o valor do seu tamanho. De seguida, esse array é transformado numa matriz cujo número de linhas deverá corresponder ao tamanho guardado a dividir pelo número de colunas da matriz chave e o seu número de colunas deverá ser igual ao da matriz chave.

$$C(K, P) = (K * P) \ mod \ module$$

Por conseguinte, procede-se à aplicação da cifra na qual, para obter a matriz do *ciphertext* multiplicamos a matriz chave pela matriz do *plaintext* e aplica-se o seu modulo face ao valor *module*, sendo este neste caso 26, sendo este processo simplificado pela fórmula acima.

```
encryption = np.matmul(message, key)
encryption = np.remainder(encryption, module)
print("encrypted text: \n",encryption)
```

O equivalente a essa fórmula, no código selecionado, é o excerto acima em que primeiro efetuamos a multiplicação da matriz `message` com a matriz chave, `key`. De seguida, com o resultado dessa multiplicação aplicamos o módulo, tendo sida obtida a matriz do *ciphertext*.

Decifrar

$P(K, C) = (K^{-1} * C)\ mod\ module$

A forma como funciona este processo é semelhante à anterior, sendo apenas necessário inverter a matriz chave, multiplicar pela matriz do *ciphertext* e por fim efetuar o modulo pelo valor *module* (26, neste caso) obtendo assim, de volta, o *plaintext*, tal como demonstrado na fórmula. Esta inversão é o motivo pelo qual, tal como mencionado anteriormente, a matriz chave deverá ser quadrada.

```
print("finding inverse key")
inverse_key = Matrix(key).inv_mod(module)
inverse_key = np.array(inverse_key) #sympy to numpy
inverse_key = inverse_key.astype(float)
print("inverse key: ",inverse_key)
```

Assim sendo, é necessário primeiro inverter a matriz chave sobre o `module` e, para tal, converte-se numa matriz da biblioteca `Sympy` e utiliza-se a função `inv_mod` segundo o *module*, sendo apenas tornar essa matriz em numa matriz `Numpy` novamente e colocar os seus valores em *floats*.

```
print("validating inverse key. key times inverse key must be idendity matrix")
check = np.matmul(key, inverse_key)
check = np.remainder(check, module)
print("key times inverse key: ",check)
print("it is ",np.allclose(check, np.eye(3)))
```

O excerto de código acima, visa apenas averiguar se a matriz foi realmente invertida de forma correta, sendo que se começa por multiplicar a matriz chave pela sua inversa, e de seguida aplicar o seu módulo sobre o valor *module*. Se de facto tiver sido invertida corretamente, a matriz resultante deste processo dará uma matriz identidade, das mesmas dimensões da matriz chave.

```
print("decryption:")
decryption = np.matmul(encryption, inverse_key)
decryption = np.remainder(decryption, module).flatten()
print("decryption: ",decryption)
```

Já obtida a inversa da matriz chave basta multiplicar a mesma pela matriz do *ciphertext* e, após isso, aplicar o módulo *module* e temos finalmente a matriz do *plaintext*.

```
decrypted_message = ""
for i in range(0, len(decryption)):
        letter_num = int(decryption[i])
        letter = numberToLetter(decryption[i])
        decrypted_message = decrypted_message + letter

print("decrypted message: ",decrypted_message)
```

Por fim, tal como se pode observar acima, o processo termina por iterar sobre a matriz do *plaintext* e, utilizando a função abaixo demonstrada, converte-se cada número na respetiva letra da tabela da figura 2, armazenado cada uma no array `decrypted_message`, sendo que no final, ao imprimirmos esse array teremos o *plaintext*.

```
def numberToLetter(number):
        return chr(int(number) + 97)
```

É importante mencionar que aquando do final do processo de cifra, o autor do código optou por não converter para texto a matriz do *ciphertext*. Caso o desejasse fazer, à semelhança de como foi feito no final do processo de decifra, bastaria iterar sobre essa matriz convertendo cada número para letra e armazenando num array.

Pergunta 4.1

A cifra *One time Pad* (OTP) funciona de forma semelhante à cifra de *Vignère*, ou seja, utiliza uma chave que intercala múltiplas cifras de César tornando todo o processo de cifragem e decifragem menos suscetíveis a ataques. Contudo, a cifra OTP utiliza uma chave não reutilizável gerada aleatoriamente de tamanho igual ou maior que a mensagem a enviar ao contrário da cifra de *Vignère* que utiliza uma chave de tamanho aleatório. Caso a chave de *Vignére* fosse muito mais pequena que a mensagem a enviar então seria possível encontrar padrões no processo de decifragem tornando o método menos robusto. A cifra OTP consegue desta forma mitigar um dos principais problemas relacionados com a cifra de *Vignére*, contudo esta característica introduziu outros problemas ao método OTP.

Um dos problemas está na distribuição da chave. Como a chave tem de ter tamanho igual ou maior que a mensagem a enviar, todo o processo de distribuição da chave do emissor até ao recetor é dificultado e condicionado pelo tamanho da mesma. Além disso, quanto maior for a chave mais lento é o processo de cifragem e decifragem. Outra das características do método OTP que condiciona a distribuição da chave é esta não ser reutilizável. Desta forma, sempre que o emissor queira enviar uma nova mensagem ao recetor é necessário manter uma ligação segura entre as duas entidades e distribuir novamente a chave. Este problema é, contudo, inerente a todos os métodos que utilizam chaves simétricas no processo de cifragem e decifragem.

Outros dos problemas característicos deste método é o da geração da chave. O método OTP requer que a chave tenha um tamanho condicionado pelo tamanho da mensagem a enviar, desta forma é necessário saber em antemão todas as mensagens que vão ser enviadas para o recetor antes de fazer a geração e distribuição da chave, tornando a chave maior quanto maior forem as mensagens a enviar. Finalmente, o grande problema da geração da chave encontra-se no facto de esta ser criada usando um gerador aleatório de valores. Por muito bom que seja o gerador a ser utilizado, este nunca gera valores completamente aleatórios sendo possível a existência de um padrão na própria geração dos valores. Quando maior a mensagem, maior será a chave a gerar e por isso maior a probabilidade de encontrar padrões nos valores a serem gerados.

Concluindo, apesar do método OTP resolver grandes problemas da cifra de *Vignère*, o método introduz também certos problemas que podem ser mitigados caso a cifra OTP seja usada na cifragem e decifragem de mensagens curtas.

Pergunta 5.1

As cifras por transposição são métodos que, em vez de manipular os caracteres da mensagem, manipulam a posição desses caracteres através de permutações. A cifra por transposição dupla é um método que intercala duas permutações distintas para cifrar a mensagem a enviar, de forma a tornar este método mais forte. De seguida apresenta-se um exemplo com o funcionamento deste método.

A cifra por transposição dupla é iniciada com a mensagem a enviar e duas chaves que vão servir para fazer as permutações entre os caracteres da mensagem. Neste caso a mensagem a enviar será ESTE EXEMPLO MOSTRA A TRANSPOSICAO DUPLA, utilizando a chave MINHO para a primeira permutação e a chave ENGSEG para a segunda permutação. Em primeiro lugar vamos numerar cada caractere das chaves a utilizar de acordo com a sua posição no abecedário. Desta forma, o caractere que aparecer primeiro no abecedário terá o valor de 1, o segundo a aparecer terá o valor de 2 e assim sucessivamente. Caso a mesma letra apareça mais do que uma vez na chave então atribui-se de forma aleatória a posição entre esses caracteres, tal como está demonstrado nas figuras seguintes.

3	2	4	1	5
M	I	N	H	O

(a) Posição caracteres MINHO

1	5	3	6	2	4
E	N	G	S	E	G

(b) Posição caracteres ENGSEG

Figure 4: Posição dos caracteres das chaves

O passo seguinte será fazer a primeira permutação com a chave MINHO. Para isso, criamos uma matriz onde cada coluna representa um caracter da chave e colocamos a mensagem a enviar segundo as linhas da matriz, tal como está na figura abaixo.

3	2	4	1	5
M	I	N	H	O
E	S	T	E	
E	X	E	M	P
L	O		M	O
S	T	R	A	
A		T	R	A
N	S	P	O	S
I	C	A	O	
D	U	P	L	A

Figure 5: Matriz da primeira permutação

De seguida, ordenamos as colunas considerando a numeração atribuída à chave e escrevemos o resultado final segundo as colunas ordenadas, isto é, os caracteres EMMAROOL que estão escritos na coluna 1 formam a primeira porção da mensagem

resultante desta primeira permutação. Desta forma, desorganizamos as posições dos caracteres da mensagem original, resultando em:

EMMAROOLSXOT SCUEELSANIDTE RTPAP PO AS A

A segunda permutação funciona de forma semelhante à primeira, contudo iniciamos a matriz com o resultado da permutação anterior e a chave passa a ser ENGSEG. Podemos ver que a matriz é formada por 42 células, logo haverá 2 células que vão ficar bloqueadas pois não vão ter nenhum caractere. A matriz resultante desta permutação encontra-se na figura abaixo e a mensagem que resulta da segunda permutação, ou seja, mensagem cifrada (*ciphertext*) é:

EO LTAAROEITOMSCA OTEDP MLSSEPSAXUNRPA

1	5	3	6	2	4
E	N	G	S	E	G
E	M	M	A	R	O
O	L	S	X	O	T
	S	C	U	E	E
L	S	A	N	I	D
T	E		R	T	P
A	P		P	O	
A	S		A		

Figure 6: Matriz da segunda permutação

De forma a desencriptar a mensagem cifrada, precisamos de ter acesso a ambas as chaves. De seguida, fazemos o caminho inverso, isto é, começando na chave ENGSEG escrevemos a mensagem cifrada pelas colunas de forma ordenada e tendo em conta quais foram as células que foram bloqueadas. O resultado é lido sobre as linhas (figura 5) e será escrito nas colunas da chave MINHO de forma ordenada. A mensagem final, pode ser lida segundo as linhas tal como está na figura 4.

Em suma, os passos para implementar o funcionamento da **cifragem** do método apresentado são:

1. Numerar cada caractere das chaves de acordo com a sua posição no abecedário.
2. Fazer a permutação
   (a) Criar uma matriz onde é escrita a mensagem segundo as linhas
   (b) Ler a mensagem segundo as colunas pela ordem determinada no passo 1
3. Realizar o passo 2 utilizando a chave da segunda permutação e a mensagem final da primeira permutação

Para a **decifragem**, os passos a implementar são:

1. Numerar cada caractere das chaves de acordo com a sua posição no abecedário.

2. Fazer a permutação com a segunda chave

   (a) Criar uma matriz onde é escrita a mensagem a decifrar segundo as colunas pela ordem definida no passo 1

   (b) Ler a mensagem segundo as linhas da matriz

3. Realizar o passo 2 utilizando a chave da primeira permutação e a mensagem da permutação anterior.

De seguida apresenta-se a implementação da cifra de transposição dupla em Python. Começamos por apresentar um menu ao utilizador, para que este consiga escolher se pretende cifrar ou decifrar uma mensagem.

```
# Opções do menu do utilizador
menu = {
    0: 'SAIR',
    1: 'Cifragem',
    2: 'Decifragem',}

# Menu para determinar se o utilizador pertende
# cifrar ou  decifrar uma mensagem
while True:
    for key in menu.keys():
        print(key, '--', menu[key] )
    option = int(input('=>  '))
    if option == 1:
        encrypt()
    elif option == 2:
        decrypt()
    elif option == 0:
        exit()
    else:
        print('Opção Inválida.')
```

Caso o utilizador pretenda cifrar uma mensagem então é invocada a função *encrypt* que irá perguntar ao utilizador qual a mensagem a cifrar e as chaves a serem usadas nas duas permutações para assim conseguir fazer a cifragem da mensagem.

```
def encrypt():
    msg = input('Mensagem a cifrar:  ').upper()
    keyOne = input('Permutação 1:  ').upper()
    keySec = input('Permutação 2:  ').upper()

    lenMsg = len(msg)
    lenOne = len(keyOne)
    lenSec = len(keySec)

    # PASSO 1:
    # Numerar cada caracter das chaves
```

```
    # de acordo com a sua posição no abecedário
    valueOne = orderKey(keyOne)
    valueSec = orderKey(keySec)

    # PASSO 2:
    # Fazer a primeira permutação
    # utilizando a primeira chave e a mensagem a cifrar
    cipherOne = buildCiphertext(lenOne, lenMsg, valueOne, msg)
    # PASSO 3:
    # Fazer a segunda permutação
    # utilizando a segunda chave e a mensagem da 1º permutação
    cipherSec = buildCiphertext(lenSec, lenMsg, valueSec, cipherOne)

    # Mensagem cifrada final
    print()
    print('====> ', cipherSec, ' <====')
    print()
```

De forma a conseguir determinar a ordem das colunas segundo as chaves apresentadas foi criada a função *orderKey* que numera posição dos caracteres na chave recebendo como argumento a chave a ser utilizada.

```
def orderKey(key):
    aux = []
    # Ciclo que percorre os caracteres da chave e insere num array
    # a sua posição no abecedário. Exemplo: E encontra-se na posição 4
    i=0
    for char in key:
        pos = string.ascii_uppercase.index(char)
        aux.insert(i,pos)
        i = i+1
    # Devolve uma ordem apartir do valor das posições de cada caracter.
    # O menor valor fica em 1º lugar e assim sucessivamente
    value = stats.rankdata(aux, method='ordinal')

    return value
```

A função que realiza uma permutação para cifrar mensagens *buildCiphertext* recebe o comprimento da chave e da mensagem, a ordem das colunas (passo 1) e a mensagem a cifrar.

```
def buildCiphertext(lenKey, lenMsg, value, msg):
    finalStr = ""
    column = []
    matrix = []

    # PASSO a:
    # Ciclo que vai escrever pelas linhas da matriz
```

```
    # a mensagem a ser cifrada
    for i in range(lenMsg):
        if i%lenKey == 0:
            # substring que será escrita em cada linha
            sub = msg[i:i+lenKey]
            lst = []
            for j in sub:
                lst.append(j)
            # insere linha com a mensagem na matriz
            matrix.append(lst)

    # PASSO b:
    # Ciclo que guarda num array a mensagem pela ordem
    # das colunas (passo 1).A coluna com numero 1 conterá
    # a 1ª porção da mensagem e assim sucessivamente
    for n in range(1,lenKey+1):
        index = 0
        for x in value:
            # Condição que determina a ordem pela qual
            # guardamos a mensagem
            if x==n:
                ind = 0
                for row in matrix:
                    # Condição que determina se a célula da matriz
                    # a ser lida contém caracteres da mensagem
                    if ind*lenKey+index+1<=lenMsg:
                        # Guarada a mensagem num array designado column
                        column.append(row[index])
                    ind = ind+1
            index = index+1

    # Ciclo que transforma o array que contém a mensagem numa string
    for elem in column:
        finalStr += elem

    return finalStr
```

Caso o utilizador pretenda decifrar uma mensagem então é invocada a função *decrypt* que irá perguntar ao utilizador qual a mensagem a decifrar e as chaves a serem usadas nas duas permutações para assim conseguir fazer a decifragem da mensagem.

```
def decrypt():
    msg = input('Mensagem a decifrar:  ').upper()
    keyOne = input('Permutação 1:  ').upper()
    keySec = input('Permutação 2:  ').upper()

    lenMsg = len(msg)
    lenOne = len(keyOne)
```

```
    lenSec = len(keySec)

    # PASSO 1:
    # Numerar cada caracter das chaves
    # de acordo com a sua posição no abecedário
    valueOne = orderKey(keyOne)
    valueSec = orderKey(keySec)

    # PASSO 2:
    # Fazer a primeira permutação
    # utilizando a segunda chave e a mensagem a decifrar
    decipherOne = buildPlaintext(lenSec, lenMsg, valueSec, msg)
    # PASSO 3:
    # Fazer a primeira permutação
    # utilizando a primeira chave e a mensagem da 1º permutação
    decipherSec = buildPlaintext(lenOne, lenMsg, valueOne, decipherOne)

    print()
    print('====> ', decipherSec, ' <====')
    print()
```

A função que realiza uma permutação para decifrar mensagens *buildPlaintext* recebe o comprimento da chave e da mensagem, a ordem das colunas (passo 1) e a mensagem a cifrar.

```
def buildPlaintext(lenKey, lenMsg, value, msg):
    finalStr = ""
    column = []
    matrix = []

    # Determina o número de linhas da matriz a criar
    rows = (lenMsg//lenKey) + 1 if lenMsg%lenKey else lenMsg//lenKey

    # Ciclo que cria uma matriz
    for i in range(0,rows):
        col = []
        for j in range(0,lenKey):
            # Bloqueia com um '*' as células que não vão conter
            # caracteres da mensagem
            col.append('') if i*lenKey+j+1<=lenMsg else col.append('*')
        matrix.append(col)

    # PASSO a:
    # Ciclo que escreve na matriz a mensagem segundo as
    # colunas de forma ordenada. A 1º porção da menagem
    # irá para a coluna com valor 1 e assim sucessivamente
    elem = 0
    for index in range(1, lenKey+1):
```

```
        col = 0
        for v in value:
            # Condição que determina a ordem pela qual
            # escrevemos a mensagem
            if v==index:
                # Determina a porção da mensagem que irá ser alocada
                # à coluna tendo em conta as células bloqueadas com '*'
                numberElem = rows-1 if '*' in [r[col] for r in matrix] else rows
                sub = msg[elem:elem+numberElem]
                elem = elem+numberElem
                for line in range(0,rows):
                    # Escreve na matriz a substring
                    # calculada na coluna respetiva
                    if matrix[line][col] != '*':
                        matrix[line][col] = sub[line]
            col = col +1

    # PASSO b:
    # Guarda no array column a mensagem lida segundo as linhas da matriz
    for i in range(0,rows):
        for j in range(0,lenKey):
            # Condição que ignora as células que estejam bloqueadas
            if matrix[i][j] != '*':
                column.insert(i*lenKey+j,matrix[i][j])

    # Transforma o array column numa string
    for char in column:
        finalStr += char

    return finalStr
```

Finalmente, a cifra de transposição dupla em Python encontra-se implementada na íntegra no seguinte link do repositório do Github.

Bibliography

[1] "Kerckhoffs' principles from La cryptographie militaire " [online]. Disponível em https://www.petitcolas.net/kerckhoffs/index.html [Acedido em março 2022].

[2] "Cryptanalysis of the Caesar Cipher" [online]. Disponível em http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-caesar-cipher/ [Acedido em março 2022].