

# Segurança Computacional - Trabalho 1



Gabriel Cruz Vaz Santos - 200049038

## Descrição do código - Cifrador/Decifrador

A função principal recebe a mensagem e senha digitadas pelo usuário e chama as demais funções responsáveis pelos processos.

```
main.py > ...
1  from cypher import cypher
2  from decypher import decypher
3  from calculateFrequency import calculateFrequency
4
5  if __name__ == "__main__":
6      key = input("digite uma chave cifradora: ")
7      message = input("digite uma mensagem para ser encriptada: ")
8
9      cyphertext, keystream = cypher(message, key)
10     print(f"cyphertext gerado: {cyphertext}")
11
12     originalMessage = decypher(cyphertext, keystream)
13     print(f"mensagem original: {originalMessage}")
```

A função de cifra recebe a mensagem e a senha inseridas pelo usuário. Com isso, chama a função `generateKeystream()`, para que a chave utilizada na cifra possua o mesmo tamanho que a mensagem. Checamos se ele é um caractere dentro do escopo [a-z]. Caso seja, ele é cifrado utilizando o keyStream, se não, ele é ignorado pelo cifrador.

```

generateKey.py > generateKeystream
1 def generateKeystream (message, key):
2     messageSize = len(message)
3     keySize = len(key)
4     if messageSize == keySize:
5         return key
6     elif messageSize > keySize:
7         keyStream = [letter for letter in key]
8         for i in range(len(message) - len(key)):
9             keyStream.append(keyStream[i % len(key)])
10        return "".join(keyStream)
11    else:
12        return key[0: messageSize]

cypher.py > cypher
1 from generateKey import generateKeystream
2 from checkEspecialCharacter import checkEspecialCharacter
3
4 def cypher(message, key):
5     messageSize = len(message)
6     cyphertext = ""
7     try:
8         keystream = generateKeystream(message, key)
9     except:
10        print("Erro ao gerar KeyStream, tente novamente")
11    else:
12        print(f"Keystream gerado: {keystream}")
13        for i in range(messageSize):
14            if checkEspecialCharacter(message[i]):
15                cyphertext += message[i]
16            else:
17                cypherCharacter = (ord(message[i]) + ord(keystream[i]) - 2 * ord('a')) % 26
18                cypherCharacter += ord('a') # shift
19                cyphertext += chr(cypherCharacter)
20        return cyphertext, keystream

checkEspecialCharacter.py > checkEspecialCharacter
1 def checkEspecialCharacter(character):
2     asciiCharacter = ord(character)
3     if asciiCharacter < 97 or asciiCharacter > 122:
4         return True
5     else:
6         return False

```

A função para decifrar recebe o criptograma junto com nossa chave já formatada para possuir o mesmo tamanho que a mensagem e segue o mesmo procedimento que a função de cifra: Caso o caractere esteja dentro do escopo [a-z] ele é decifrado, caso o contrário ele é ignorado pelo decifrador.

Descrição do código - Quebra cifra Vigenere análise de frequência

```
breakCipher.py > ...  
  
1  from KasiskiAttack import KasiskiAttack  
2  
3  ciphertext = input("digite um texto cifrado: ")  
4  commonNgrams = KasiskiAttack.findCommonNgrams(ciphertext)  
5  
6  print("Sequências Repetidas e Suas Posições:")  
7  print(commonNgrams)  
8  
9  
10 keySize = KasiskiAttack.findKeyLength(commonNgrams)  
11 print(f"Tamanho da chave: {keySize}")  
12
```

A classe KasiskiAttack possui todas as funções necessárias para encontrar o comprimento de chave da cifra de Vigenere. Para isso, iremos dividir a mensagem cifrada em ngrams de tamanho N e então encontrarmos padrões de repetições para esses Ngrams. Após encontrados, é feito o cálculo entre a distância dessas repetições. Essas operações são realizadas nas funções `findDistanceBetweenNgrams` e `findCommonNgrams`.

```

KasiskiAttack.py > KasiskiAttack > findPotentialDividers
3 class KasiskiAttack:
4
5     def findDistanceBetweenNgrams(arrayOfPositions):
6         arrayOfPositionsSize = len(arrayOfPositions)
7         arrayOfDistances = []
8         for i in range(arrayOfPositionsSize - 1):
9             distance = arrayOfPositions[i + 1] - arrayOfPositions[i]
10            arrayOfDistances.append(distance)
11            return arrayOfDistances
12
13
14     def findCommonNgrams(ciphertext, ngramSize=3):
15         ngramsPositions = {}
16         ngramsDistances = {}
17         ciphertextSize = len(ciphertext)
18         for i in range(ciphertextSize):
19             ngram = ciphertext[i:i+ ngramSize]
20             if ngram not in ngramsPositions.keys():
21                 ngramsPositions[ngram] = [i]
22             else:
23                 ngramsPositions[ngram].append(i)
24         for key in list(ngramsPositions.keys()):
25             if len(ngramsPositions[key]) > 1:
26                 ngramsDistances[key] = KasiskiAttack.findDistanceBetweenNgrams(ngramsPositions[key])
27         return ngramsDistances
28
29     def findPotentialDividers(number):
30         factors = set()
31         for i in range(1, number):
32             if number % i == 0:
33                 factors.add(i)
34                 factors.add(number//i)
35         return sorted(factors)
36
37     def findKeyLength(ngramsDistancesDict):
38         dividers = []
39         for key in ngramsDistancesDict:
40             for i in range(len(ngramsDistancesDict[key])):
41                 potentialDividers = KasiskiAttack.findPotentialDividers(ngramsDistancesDict[key][i])
42                 for potentialDivider in potentialDividers:
43                     dividers.append(potentialDivider)
44         countedDividers = Counter(dividers).most_common()
45         countedDividersSize = len(countedDividers)
46         if(countedDividersSize > 1):
47             return countedDividers[1][0]
48         elif(countedDividers == 1):
49             return countedDividers[0][0]
50         else:
51             return 0

```

Agora possuímos um dicionário no qual cada chave corresponde a um padrão de repetição encontrado e seu respectivo valor é um array com as posições em que ele é encontrado na mensagem cifrada. Note que esse dicionário possui apenas os padrões que ocorrem mais de uma vez. Após isso, é calculado os múltiplos comum entre cada uma das distâncias através da função `findPotencialDividers()`. Com todos esses dados, a função `findKeyLength()` retorna o múltiplo comum que mais se repete nessa análise. Iremos dividir o criptograma em grupos do mesmo tamanho, sendo este o tamanho da chave encontrada. Note que cada grupo é cifrado utilizando

uma letra da chave, a letra mais frequente em cada grupo nos irá indicar a letra correspondente na chave.

Após isso, temos uma função responsável por reconstruir a chave dados os dados encontrados no processo acima.

```
frequencyAnalysis.py > FrequencyAnalysis > calculateFrequency
1 import collections
2 from checkEspecialCharacter import checkEspecialCharacter
3
4 class FrequencyAnalysis:
5     def calculateFrequency(ciphertextGroup):
6         numberOfNormalCharacters = collections.defaultdict(int)
7         FrequencyOfNormalCharacters = {}
8         ciphertextGroupSize = len(ciphertextGroup)
9         totalOfNormalCharacters = 0
10
11         for i in range(ciphertextGroupSize):
12             if(not checkEspecialCharacter(ciphertextGroup[i])):
13                 numberOfNormalCharacters[ciphertextGroup[i]] += 1
14                 totalOfNormalCharacters += 1
15
16         for key in numberOfNormalCharacters:
17             FrequencyOfNormalCharacters[key] = 100 * (numberOfNormalCharacters[key] / totalOfNormalCharacters)
18
19         return FrequencyOfNormalCharacters
20
```

- Repositórios no github
  - <https://github.com/Leir-Cruz/VigenereCipher>

