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.

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.
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 ignorada pelo decifrador.

generateKey.py > generateKeystream

```python
def generateKeystream (message, key):
    messageSize = len(message)
    keySize = len(key)
    if messageSize == keySize:
        return key
    elif messageSize > keySize:
        keyStream =  [letter for letter in key]
        for i in range(len(message) - len(key)):
            keyStream.append(keyStream[i % len(key)])
        return "".join(keyStream)
    else:
        return key[0: messageSize]
```

cypher.py > cypher

```python
from generateKey import generateKeystream
from checkEspecialCharacter import checkEspecialCharacter

def cypher(message, key):
  messageSize = len(message)
  cyphertext = ""
  try:
    keystream = generateKeystream(message, key)
  except:
    print("Erro ao gerar KeyStream, tente novamente")
  else:
    print(f"Keystream gerado: {keystream}")
    normalCharacterIndex = 0
    for i in range(messageSize):
      if checkEspecialCharacter(message[i]):
        cyphertext += message[i]
      else:
        cypherCharacter = (ord(message[i]) + ord(keystream[normalCharacterIndex]) - 2 * ord('a')) % 26
        cypherCharacter += ord('a')  # shift
        cyphertext += chr(cypherCharacter)
        normalCharacterIndex += 1
  return cyphertext, keystream
```

decypher.py > decypher

```python
from checkEspecialCharacter import checkEspecialCharacter

def decypher(cyphertext, keystream):
  cyphertextSize = len(cyphertext)
  originalMessage = ""
  normalCharacterIndex = 0
  for i in range(cyphertextSize):
    if checkEspecialCharacter(cyphertext[i]):
      originalMessage += cyphertext[i]
    else:
      decypherCharacter = (ord(cyphertext[i]) - ord(keystream[normalCharacterIndex]) + 26) % 26
      decypherCharacter += ord('a') #* we have to add 'a' to shift our character after module operation
      originalMessage += (chr(decypherCharacter))
      normalCharacterIndex += 1
  return originalMessage
```

checkEspecialCharacter.py > checkEspecialCharacter

```python
def checkEspecialCharacter(character):
  asciiCharacter = ord(character)
  if asciiCharacter < 97 or asciiCharacter > 122:
    return True
  else:
    return False
```

Descrição do código - Quebra cifra Vigenere análise de frequência
A classe Kasiski possui as funções para encontrarmos o comprimento da chave do criptograma fornecido. Primeiro, dividimos o criptograma em Ngram (default n=3) para calcularmos seu índice de reincidência e posições. Após isso, utilizamos as posições de cada Ngram que se repete mais de uma vez para calcularmos as distâncias entre elas. Após calcularmos as distâncias entre as ocorrências dos ngrams, guardamos os valores e calculamos os divisores mais comuns desses valores, retornando para o usuário uma lista ordenada com eles.
O usuário é responsável por escolher o tamanho da chave com base nos valores fornecidos.

```python
class KasiskiAttack:

  def checkValidNgram(ngram):
    return all(checkEspecialCharacter(char) for char in ngram)

  def findDistanceBetweenNgrams(arrayOfPositions):
    arrayOfPositionsSize = len(arrayOfPositions)
    arrayOfDistances = []
    for i in range(arrayOfPositionsSize - 1):
      distance = arrayOfPositions[i + 1] - arrayOfPositions[i]
      arrayOfDistances.append(distance)
    return arrayOfDistances


  def findCommonNgrams(ciphertext, ngramSize=3):
    ngramsPositions = {}
    ngramsDistances = {}
    ciphertextSize = len(ciphertext)
    normalCharacterIndex = 0
    for i in range(ciphertextSize):
      ngram=""
      for j in range(ciphertextSize - i):
        if len(ngram) >= ngramSize or (checkEspecialCharacter(ciphertext[i]) and ngram==""):
          break
        if not checkEspecialCharacter(ciphertext[j + i]):
          if ngram == "":
            normalCharacterIndex += 1
          ngram += ciphertext[j + i]
      if ngram != "":
        if ngram not in ngramsPositions.keys():
          ngramsPositions[ngram] = [normalCharacterIndex]
        else:
          ngramsPositions[ngram].append(normalCharacterIndex)
    for key in list(ngramsPositions.keys()):
      if len(ngramsPositions[key]) > 1:
        ngramsDistances[key] = KasiskiAttack.findDistanceBetweenNgrams(ngramsPositions[key])
    return ngramsDistances
```

```python
def findPotentialDividers(number):
  factors = set()
  for i in range(1, number):
      if number % i == 0:
          factors.add(i)
          factors.add(number//i)
  return sorted(factors)

def findKeyLength(ngramsDistancesDict):
  dividers = []
  for key in ngramsDistancesDict:
    for i in range(len(ngramsDistancesDict[key])):
      potentialDividers = KasiskiAttack.findPotentialDividers(ngramsDistancesDict[key][i])
      for potentialDivider in potentialDividers:
        dividers.append(potentialDivider)
  countedDividers = Counter(dividers).most_common()
  print(f"Escolha um tamanho de chave com base indice de coincidências: \n Primeiro número da tupla é o candidato a tamanho da chave \n{countedDividers}")
  keyLength = int(input("Tamanho: "))
  return keyLength
```

Após encontrarmos o tamanho da chave, dividimos o ciphertext em cosets, no qual cada coset representa uma das letras da chave. Para cada coset calculamos a frequência alfabética de cada letra e depois utilizamos o método X^2 para encontrar as possíveis letras mais próximas utilizando também a cifra de césar. O usuário é responsável por escolher a letra e temos então a chave retornada.

```python
class FrequencyAnalysis:
  def calculateFrequency(coset):
    numberOfNormalCharacters = collections.defaultdict(int)
    FrequencyOfNormalCharacters = {}
    ciphertextGroupSize = len(coset)
    totalOfNormalCharacters = 0

    for i in range(ciphertextGroupSize):
      if(not checkEspecialCharacter(coset[i])):
        numberOfNormalCharacters[coset[i]] += 1
        totalOfNormalCharacters += 1

    for key in numberOfNormalCharacters:
      FrequencyOfNormalCharacters[key] = 100 * (numberOfNormalCharacters[key] / totalOfNormalCharacters)

    return FrequencyOfNormalCharacters

  def caesarCipher(ciphertext, shift):
    if shift < 0:
      shift += 25
    ciphertextSize = len(ciphertext)
    shiftedCipher = ""
    for i in range(ciphertextSize):
      deslocatedCharacter = (ord(ciphertext[i]) + shift - ord('a')) % 26 + ord('a')
      shiftedCipher += chr(deslocatedCharacter)
    return shiftedCipher

  def calculateShift(firstLetter, secondLetter):
    shift = ord(firstLetter) - ord(secondLetter)
    return shift

  def generateCoset(ciphertext, spacing):
    coset = ""
    normalCharacterIndex = 0
    for i in range(len(ciphertext)):
      if(not checkEspecialCharacter(ciphertext[i])):
        if normalCharacterIndex % (spacing) == 0:
          coset += ciphertext[i]
        normalCharacterIndex += 1
    return coset
```

```python
def x2CosetValue(cosetFrequency, languageFrequency):
  value = 0
  for key in languageFrequency.keys():
    if key in cosetFrequency.keys():
      value += ((cosetFrequency[key] - languageFrequency[key]) ** 2) / languageFrequency[key]
    else:
      value += languageFrequency[key]
  return value

def x2CosetFindPossibleLetters(coset, languageFrequency, keyLength):
  shiftDict = {}
  possibleLetters = []
  for i in range(25):
    shiftedCoset = FrequencyAnalysis.caesarCipher(coset, i)
    cosetFrequency = FrequencyAnalysis.calculateFrequency(shiftedCoset)
    cosetX2Value = FrequencyAnalysis.x2CosetValue(cosetFrequency, languageFrequency)
    shiftDict[i] = cosetX2Value
  shiftItems = list(shiftDict.items())
  shiftItems.sort(key=lambda item: item[1])
  for j in range(keyLength):
    shift = shiftItems[j][0]
    letter = FrequencyAnalysis.caesarCipher('a', shift)
    possibleLetters.append(letter)
  return possibleLetters


def findKey(ciphertext, keyLength, languageType="english"):
  key = ""
  if languageType == "english":
    languageFrequency = english
  else:
    languageFrequency = portuguese
  for i in range(keyLength):
    coset = FrequencyAnalysis.generateCoset(ciphertext[i:len(ciphertext)], keyLength)
    possibleletters = FrequencyAnalysis.x2CosetFindPossibleLetters(coset, languageFrequency, keyLength)
    letter = input(f"Segue as possíveis letras para esse caractere, {possibleletters}, escolha uma: ")
    key += letter
  return key
```