Lista de Exercícios 01

Quebrando Shift Cipher

A cifra de César, ou Shift Cipher, é um dos métodos mais simples de criptografia. Ela funciona deslocando cada letra do texto original um certo número de posições no alfabeto. Por exemplo, com um deslocamento de 3, a letra A se torna D, B se torna E, e assim por diante.

Neste trabalho, usamos a lógica modular para quebrar a cifra de César (Shift Cipher). O código leva em conta a diferença entre letras maiúsculas e minúsculas usando seus valores ASCII. A pontuação é mantida, mas caracteres especiais como é, ã e ç não são considerados. O código é explicado nos comentários abaixo.

A complexidade do algoritmo é determinada pelo laço 4-26 que itera nos caracteres do texto fornecido, o que significa que a complexidade do algoritmo é *O(n)*, isto é, linear e depende do tamanho do texto de entrada.

```
def shift_cipher(plain_text: str, k=3):
    letters = []  # Lista para armazenar os caracteres cifrados

    for c in plain_text:
        letter_ascii = ord(c)  # Obtém o código ASCII do caractere

        # Verifica se o caractere é uma letra maiúscula (A-Z)
        if letter_ascii in range(65, 91):
            offset = 65  # Offset para letras maiúsculas (A = 65)
        # Verifica se o caractere é uma letra minúscula (a-z)
        elif letter_ascii in range(97, 123):
            offset = 97  # Offset para letras minúsculas (a = 97)
        else:
            offset = None  # Não é uma letra do alfabeto

        # Se for uma letra do alfabeto, aplica o deslocamento
        if offset is not None:
            letter_code = (
                letter_ascii - offset
            )  # Converte o caractere para um valor de 0 a 25
            shifted_letter = chr(
                (letter_code + k) % 26 + offset
            )  # Aplica o deslocamento e converte de volta para
caractere
            letters.append(shifted_letter)  # Adiciona o caractere
cifrado à lista
        else:  # Se não for letra, mantém o caractere original
(pontuação, espaço etc.)
            letters.append(c)
```

```
    cipher_text = "".join(letters)  # Junta todos os caracteres
cifrados em uma string
    return cipher_text  # Retorna o texto cifrado


# Exemplo de uso:
plain_text_example = "Duas coisas são infinitas: o universo e a
estupidez humana. Mas, em relação ao universo, ainda não tenho certeza
absoluta."
shift_cipher(plain_text_example, 3)
```

```
'Gxdv frlvdv vãr lqilqlwdv: r xqlyhuvr h d hvwxslghc kxpdqd. Pdv, hp
uhodçãr dr xqlyhuvr, dlqgd qãr whqkr fhuwhcd devroxwd.'
```

Para decifrar o texto, basta deslocarmos os caracteres de volta para o seu formato original. Para isso basta reutilizarmos o algoritmo de cifra original, mas dessa vez com o valor negativo da chave. Como a cifra usa uma lógica modular, também poderíamos deslocar as letras *26-key* para frente. Como ele utilizada a função de cifração, ambos possuem a mesma complexidade.

```
def shift_decipher(cipher_text: str, k=3):
    return shift_cipher(cipher_text,-k)

cipher_text_example = "Gxdv frlvdv vãr lqilqlwdv: r xqlyhuvr h d
hvwxslghc kxpdqd. Pdv, hp uhodçãr dr xqlyhuvr, dlqgd qãr whqkr fhuwhcd
devroxwd."
shift_decipher(cipher_text_example, 3)
```

```
'Duas coisas são infinitas: o universo e a estupidez humana. Mas, em
relação ao universo, ainda não tenho certeza absoluta.'
```

Quebra por força bruta

Antes de abordar o algoritmo de quebra, primeiro vou explicar a função para saber se o algoritmo foi quebrado. Na database "combinacoes.db" utilizada anteriormente, existe a tabela "palavras_comuns" que contém uma lista de algumas palavras frequentes que podem aparecer nos textos em português brasileiro.

```
import sqlite3

def get_frequent_words():
    db_path = "combinacoes.db"
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()

    cursor.execute("SELECT palavra FROM palavras_comuns")

    return [p[0] for p in cursor.fetchall()]

#get_frequent_words()
```

Criamos uma função rudimentar para checar se o resultado é em texto claro inteligível verificando se as palavras comuns aparecem pelo menos 3 vezes no texto fornecido.

```
def not_gibberish(text: str):
    common_words = get_frequent_words()
    words = text.lower().split()
    words_found = sum([common_words.count(word) for word in words])
    return words_found > 3


not_gibberish(
    "Sim, eu quero queijo. Como você sabia? Talvez você consiga ler
mentes?!!"
)
```

```
True
```

A quebra por força bruta é feita utilizando cada uma das 26 chaves possíveis para o shift cipher, o que faz com que o algoritmo possua uma complexidade computacional constante minúscula. Para cada chave, tentamos decifrar o texto e verificar se o resultado é inteligível.

```
def break_shift_cipher_v1(cipher_text: str):
    for k in range(1,26):
        plain_text = shift_decipher(cipher_text,k)
        if not_gibberish(plain_text):
            return k
    return None


break_shift_cipher_v1(cipher_text_example)
```

```
3
```

Quebra por distribuição frequência

Primeiro, precisamos obter a distribuição das frequências fornecidas. Os dados foram armazenados em uma database e são acessados utilizando a função *get_letter_frequency()*

```
def get_letter_frequency():
    db_path = "combinacoes.db"
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()

    cursor.execute(
        "SELECT combinacao, porcentagem FROM combinacoes ORDER BY
porcentagem DESC"
    )

    # Criar dicionários separados por tamanho da chave
    freq_1 = {}  # Para combinações de 1 letra
```

```
    freq_2 = {}  # Para combinações de 2 letras
    freq_3 = {}  # Para combinações de 3 letras

    # Separar os dados nos dicionários apropriados
    for combinacao, porcentagem in cursor.fetchall():
        if len(combinacao) == 1:
            freq_1[combinacao] = porcentagem
        elif len(combinacao) == 2:
            freq_2[combinacao] = porcentagem
        elif len(combinacao) == 3:
            freq_3[combinacao] = porcentagem

    conn.close()

    return freq_1, freq_2, freq_3  # Retorna os três dicionários
separados
```

A função find_best_matches compara as letras ou combinações de letras da cifra com uma base de dados, utilizando as frequências de ocorrência. Dessa forma, tenta encontrar a melhor correspondência entre o texto cifrado e as letras mais comuns da língua, ajudando a decifrar a mensagem.

```
def find_best_matches(cipher_letter_frequency):
    freq_1, freq_2, freq_3 = get_letter_frequency()
    cfreq_1, cfreq_2, cfreq_3 = {}, {}, {}

    # Ordena as frequências do texto cifrado do maior para o menor
    cipher_letter_frequency_ordered = dict(
        sorted(cipher_letter_frequency.items(), key=lambda item:
item[1], reverse=True)
    )

    for combinacao, porcentagem in
cipher_letter_frequency_ordered.items():
        if len(combinacao) == 1:
            cfreq_1[combinacao] = porcentagem
        elif len(combinacao) == 2:
            cfreq_2[combinacao] = porcentagem
        elif len(combinacao) == 3:
            cfreq_3[combinacao] = porcentagem

    matches = {}
    matches.update(dict(zip(cfreq_1, freq_1)))
    matches.update(dict(zip(cfreq_2, freq_2)))
    matches.update(dict(zip(cfreq_3, freq_3)))

    return matches
```

O código tenta descobrir qual foi o número de posições usado para cifrar uma mensagem com a cifra de César. Para isso, ele analisa a frequência de letras, pares e trios de letras no texto cifrado

e compara com uma base de dados. Quando encontra um resultado que faz sentido, ele retorna o texto decifrado. O processo é mais lento porque testa muitas combinações de letras. A complexidade desse algoritmo é determinada pelo método *count* da linha 22, que tem complexidade linear O(n).

```
import itertools


def break_shift_cipher_v2(cipher_text: str):
    # Cria uma lista com todas as letras minúsculas do alfabeto
    alphabet = [chr(i) for i in range(ord("a"), ord("z") + 1)]

    # Gera todas as combinações possíveis de 2 letras (digraphs), como
'aa', 'ab', ..., 'zz'
    digraphs = ["".join(pair) for pair in itertools.product(alphabet,
repeat=2)]

    # Gera todas as combinações possíveis de 3 letras (trigraphs),
como 'aaa', 'aab', ..., 'zzz'
    trigraphs = ["".join(triplet) for triplet in
itertools.product(alphabet, repeat=3)]

    # Conta quantas letras (apenas letras) existem no texto cifrado
    total_letters = sum(1 for c in cipher_text if c.isalpha())

    # Dicionário para armazenar a frequência de cada letra ou
combinação de letras
    cipher_letter_frequency = {}
    for letter_combination in alphabet + digraphs + trigraphs:
        # Calcula a frequência relativa de cada combinação no texto
        cipher_letter_frequency[letter_combination] = round(
            cipher_text.count(letter_combination) / total_letters, 2
        )

    # Usa as frequências calculadas para tentar encontrar as melhores
correspondências
    # com uma base de dados de frequências da língua (função externa)
    matches = find_best_matches(cipher_letter_frequency)

    # Para cada par encontrado (caractere da cifra e seu possível
correspondente real)
    for k, v in matches.items():
        # Calcula o deslocamento (diferença entre os caracteres)
        key = abs(ord(k) - ord(v))

        # Tenta decifrar o texto com a chave calculada
        clear_text = shift_decipher(cipher_text, key)

        # Verifica se o texto decifrado faz sentido (função externa)
        if not_gibberish(clear_text):
```

```
            return clear_text  # Se for legível, retorna o texto

    return None


# Exemplo de uso
break_shift_cipher_v2(cipher_text_example)
```

```
'Duas coisas são infinitas: o universo e a estupidez humana. Mas, em relação ao universo, ainda não tenho certeza absoluta.'
```

Quebrando Cifra por Transposição

Essa função implementa a cifra de transposição em colunas (columnar transposition cipher), um método de criptografia que reorganiza as letras do texto original com base em uma chave. Primeiro, o texto é distribuído em 5 colunas, uma letra por vez em ordem. Depois, as colunas são embaralhadas conforme a ordem alfabética da chave fornecida. Por fim, o texto cifrado é montado lendo as letras coluna por coluna na nova ordem. Esse método não altera as letras, apenas muda suas posições. Esse algoritmo possui complexidade O(n), onde n é o tamanho do texto de entrada.

Como eu tenho *n* colunas que são organizadas conforme a ordem das letras da palavra, temos apenas *n!* possibilidades de chave que podem ser utilizadas. Por exemplo, usamos "cargo" para codificar a mensagem anterior, o que no algoritmo atribui os valores "21534" para embaralhar as colunas, o qual também pode ser obtido pela chave "baecd".

Logo, só é necessário achar uma permutação de "abcde" que gere a sequência de ordenação correta para encontrarmos uma chave de decriptação válida. Supondo que a chave tenha um tamanho variável e possa utilizar somente letras minúsculas, teremos *26! + 25! + ... + 2! + 1!* possibilidades de chaves possíveis.

```
def columnar_transposition_cipher(plain_text: str, key: str):
    # Cria 5 colunas vazias para organizar o texto
    columns = [[] for _ in range(5)]

    # Ordena a chave alfabeticamente e cria uma ordem numérica para as colunas
    key_order = key
    for c, i in zip(list(sorted(key)), range(5)):
        # Substitui cada letra da chave por seu número na ordem alfabética (1 a 5)
        key_order = key_order.replace(c, str(i + 1))

    # Distribui o texto nas colunas: cada letra vai para uma coluna alternadamente
    for i in range(len(plain_text)):
        columns[i % 5].append(plain_text[i])

    # Reorganiza as colunas de acordo com a ordem definida pela chave
```

```
    shuffled_columns = []
    for n in list(map(int, key_order)):
        shuffled_columns.append(columns[n - 1])

    # Lê as letras das colunas embaralhadas para formar o texto
cifrado
    shuffled_letters = []
    for i in range(len(plain_text)):
        shuffled_letters.append(shuffled_columns[i % 5][i // 5])

    # Junta todas as letras para formar o texto cifrado final
    cipher_text = "".join(shuffled_letters)
    return cipher_text


# Exemplo de uso com uma chave de 5 letras
columnar_transposition_cipher(plain_text_example, "cargo")
```

```
'uD asocais sosãi infinsta :uo inrveos  e atespueid zmhuna a.aM s,mee
raloçãa uo inrveosa, ni daãnto ne hoecertazb aostlu.a'
```

A função *columnar_transposition_decipher* usa a mesma chave que foi usada para cifrar. Primeiro, ela separa o texto em colunas embaralhadas, depois reorganiza essas colunas de volta à ordem original. Por fim, ela monta o texto original juntando as letras na ordem certa.

```
def columnar_transposition_decipher(cipher_text: str, key: str):
    # Cria 5 colunas vazias para reorganizar o texto
    columns = [[] for _ in range(5)]

    # Cria a ordem numérica da chave com base na ordem alfabética das
letras
    key_order = key
    for c, i in zip(list(sorted(key)), range(len(key))):
        # Substitui cada letra da chave por um número de 1 a 5,
conforme a ordem alfabética
        key_order = key_order.replace(c, str(i + 1))

    # Distribui as letras do texto cifrado nas colunas (como se fossem
embaralhadas)
    for i in range(len(cipher_text)):
        columns[i % 5].append(cipher_text[i])

    # Reorganiza as colunas de volta à ordem original, com base na
posição na chave
    unshuffled_columns = []
    for i in range(1, 6):  # de 1 a 5
        # Encontra o índice original da coluna com base no número da
chave
        unshuffled_columns.append(columns[key_order.index(str(i))])
```

```
    # Reconstrói o texto original lendo as letras na ordem correta
    unshuffled_letters = []
    for i in range(len(cipher_text)):
        unshuffled_letters.append(unshuffled_columns[i % 5][i // 5])

    # Junta todas as letras para formar o texto decifrado
    cipher_text = "".join(unshuffled_letters)
    return cipher_text


# Exemplo de uso da função
cipher_text_example2 = "uD asocais sosãi infinsta :uo inrveos  e
atespueid zmhuna a.aM s,mee raloçãa uo inrveosa, ni daãnto ne
hoecertazb aostlu.aeqw"
columnar_transposition_decipher(cipher_text_example2, "cargo")
```

```
'Duas coisas são infinitas: o universo e a estupidez humana. Mas, em
relação ao universo, ainda não tenho certeza absoluta.qwe'
```

Quebra por força bruta

O algoritmo de quebra por força bruta é relativamente simples, ele itera nas combinações não repetidas de "abcde" (supondo que eu sei que a chave tem tamanho fixo 5), e tenta decifrar o texto e verificar se ele é legível. Obviamente uma das fraquezas da cifra é que, como dito anteriormente, podemos ter inúmeras chaves que geram a mesma ordem de embaralhamento.

Isso significa que apesar dessa cifra por transposição ser extremamente simples de implementar, ela é rudimentar. Se a chave utilizada for muito simples ou tiver poucas letras e variações de caracteres é possível quebrá-la rapidamente pelo método de força bruta.

```
import string

def break_columnar_transposition_cipher_v1(cipher_text: str):
    key_combinations = ["".join(p) for p in
itertools.permutations("abcde")]
    for key in key_combinations:
        plain_text = columnar_transposition_decipher(cipher_text, key)
        if not_gibberish(plain_text):
            return key
    return None


key = break_columnar_transposition_cipher_v1(cipher_text_example2)
columnar_transposition_decipher(cipher_text_example2, key)
```

```
'Duas coisas são infinitas: o universo e a estupidez humana. Mas, em
relação ao universo, ainda não tenho certeza absoluta.qwe'
```

Quebra por distribuição frequência

Como a chave é uma palavra comum, sem repetição de letras, podemos usar uma lista com a frequência das palavras mais usadas que atendem a esses critérios. Para isso, seria ideal criar um contador de palavras, alimentado com exemplos de textos, para construir uma base de dados com a distribuição desejada. Porém, como o foco deste exercício é a lógica do algoritmo, utilizei um conjunto de dados fictício (dummy data) para a quebra.

Além disso, precisamos conhecer o tamanho da chave para gerar as permutações necessárias. Supondo que a chave seja composta apenas por letras (maiúsculas e minúsculas) e números, temos um total de $62! + 61! + ... + 1!$ chaves possíveis, o que resulta em um número ENORME. Esse conceito pode ser comparado com senhas de usuários que são consideradas inseguras, como aquelas que utilizam palavras comuns ou sequências de números simples como '123' ou 'abc', o que reduz significativamente o espaço de busca.

Embora seja possível realizar uma análise de frequência das letras presente na cifra, não podemos utilizar essa informação para desembaralhar as letras diretamente. Por isso, podemos analisar a probabilidade/frequência com que determinadas chaves são utilizadas.

No caso em que o tamanho da chave é variável utilizar a segunda tática de quebra é muito mais viável por conta da quantidade gigantesca de tentativas que seria necessári para quebra com força bruta. Supondo que a chave inclua diversos caracteres, seria necessário outras técnicas de quebra de senhas comuns.

```
word_frequency = {
    "bravo": 0.045,
    "cargo": 0.082,
    "claro": 0.047,
    "custo": 0.058,
    "grato": 0.052,
    "limpo": 0.061,
    "lenta": 0.076,
    "poder": 0.065,
    "tempo": 0.054,
    "troca": 0.049,
}

def break_columnar_transposition_cipher_v2(cipher_text: str):
    word_frequency_ordered = dict(
        sorted(word_frequency.items(), key=lambda item: item[1],
reverse=True)
    )

    for word in word_frequency_ordered:
        plain_text = columnar_transposition_decipher(cipher_text,
word)
        if not_gibberish(plain_text):
            return word
    return None
```

```
break_columnar_transposition_cipher_v2(cipher_text_example2)
'cargo'
```