Lista de Exercícios 01

Lucca Magalhães Boselli Couto - 222011552

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
Segurança Computacional, 2025/1

1. Introdução

Criptografia é a prática de proteger informações por meio da transformação de dados legíveis (texto claro / plain text) em uma forma codificada (texto cifrado / texto escuro / cipher text), de modo que apenas pessoas autorizadas possam acessá-los. O principal objetivo da criptografia é garantir a confidencialidade, integridade e autenticidade das informações.

Ela é amplamente utilizada em diversas áreas, como comunicações seguras, transações bancárias, armazenamento de senhas e proteção de dados pessoais na internet.

Existem dois tipos principais de criptografia:

- **Criptografia simétrica**: utiliza a mesma chave para cifrar e decifrar a informação.
- **Criptografia assimétrica**: utiliza um par de chaves, sendo uma pública e outra privada

Nessa lista de exercícios abordaremos as cifras de deslocamento e transposição, tal como o funcionamento, estratégias de implementação e especificidades de cada uma.

2. Exercício 1 - Quebrando Shift Cipher

A Shift Cipher (ou cifra de César) é uma cifra de substituição simples e simétrica usada na criptografia clássica. Ela funciona deslocando cada letra do texto original um número fixo de posições no alfabeto. Por exemplo, com um deslocamento de 3 (cifra de César), a letra A vira D, B vira E, e assim por diante. Quando se chega ao fim do alfabeto, voltamos ao início da sequência de letras, ou seja, Z com deslocamento 3 vira C. Nesse exercício, iremos elaborar códigos para criptografia, descriptografia, ataque por força bruta e por distribuição de frequência.

A cifra de césar é bem simples de ser implementada e utiliza poucos recursos computacionais. Em contrapartida, não é uma técnica de criptografia muito segura, de modo que ataques por força bruta e análise de frequência podem quebrá-la facilmente.

Comentários sobre a implementação, análises de viabilidade, complexidade de algoritmo e tempo de execução serão abordados nas subseções abaixo.

O código-fonte completo pode ser acessado em:
https://jupyter.org/try-jupyter/lab/index.html?path=Lista01-Ex1.ipynb

Caso não seja possível acessar diretamente pelo link acima, o código-fonte pode ser encontrado no arquivo **"Lista01-Ex1.ipynb"** localizado no seguinte repositório do GitHub:
https://github.com/Luccambc/Seguranca-Computacional/tree/main/Lista%2001

2.1. Encriptação

Inicialmente, foi definido um dicionário `caracteres = {...}`, contendo todas as 26 letras do alfabeto e suas respectivas posições, e um dicionário invertido `caracteres_invertidos = {...}` para facilitar a manipulação dos dados ao longo da implementação.

A função `encriptacao(plain_text, key)`, criada para encriptar um texto claro qualquer, recebe uma string qualquer e uma chave qualquer no intervalo de 0 a 16. Primeiramente, colocamos todos os caracteres do texto em letra minúscula para não haver inconsistências durante a execução. Em seguida, iremos percorrer todos os caracteres do texto claro e, para cada letra, verificaremos sua posição no dicionário invertido e calcularemos sua nova posição de acordo com a chave fornecida no parâmetro "key" da função. Com a nova posição em mãos, iremos acessar o dicionário de caracteres e verificar qual letra corresponde àquela posição. Por fim, adicionamos a letra à nova string e repetimos esse processo até que o texto claro seja completamente encriptado.

Abaixo é possível visualizar a implementação da encriptação com a validação para a cifra de César (key = 3).

```
[5]: # Dicionário
caracteres = {
    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'
}

# Dicionário invertido
caracteres_invertido = {v: k for k, v in caracteres.items()}

def encriptacao(plain_text, key):
    plain_text = plain_text.lower()
    cipher_text = ''

    for letra in plain_text:
        if letra in caracteres_invertido:
            pos_letra = caracteres_invertido[letra]
            nova_pos = (pos_letra + key) % 26
            nova_letra = caracteres[nova_pos]
            cipher_text += nova_letra
        else:
            cipher_text += letra

    return cipher_text

# Teste
print(encriptacao("Lucca", 3))  # Saída esperada: "oxffd"

oxffd
```

Figure 1. Função de Encriptação

2.2. Decriptação

A função `decriptacao(cipher_text, key)` segue a mesma lógica de implementação da função `encriptacao(plain_text, key)`. Entretanto, como o processo de decriptação consiste em descobrir o texto claro, ao invés de, por exemplo, deslocarmos os caracteres 3 posições para frente, iremos deslocá-los 3 posições para trás. Logo, nesse exemplo a letra ”Y” seria decriptada como ”V”.

Abaixo podemos ver a implementação da função de decriptação com a validação para a cifra de César:

```
[6]: def decriptacao(cipher_text, key):
         cipher_text = cipher_text.lower()
         plain_text = ''

         for letra in cipher_text:
             if letra in caracteres_invertido:
                 pos_letra = caracteres_invertido[letra]
                 nova_pos = (pos_letra - key) % 26
                 nova_letra = caracteres[nova_pos]
                 plain_text += nova_letra
             else:
                 plain_text += letra

         return plain_text

     # Teste
     print(decriptacao("oxffd", 3))  # Saída esperada: "lucca"

     lucca
```

Figure 2. Função de Decriptação

2.3. Quebra por Força Bruta

O algoritmo de quebra por força bruta consiste em testar todas as chaves possíveis até encontrar a correta. Tendo em vista que existem apenas 26 chaves possíveis no shift cipher, é viável testar todos os deslocamentos possíveis em um curto período de tempo.

Geralmente, essa técnica é utilizada em casos em que os textos são curtos ou quando queremos descobrir rapidamente qual a chave. É possível utilizá-la também para decifrar textos maiores, porém o tempo de quebra poderá ser maior.

Devido à sua pequena quantidade de chaves, temos que, de acordo com os conceitos de complexidade de algoritmo em notação Big-O, sua complexidade é O(1).

Para saber se encontramos a chave correta, podemos fazer uma análise humana dos textos claros obtidos, análise de frequência de letras ou verificação com dicionário. A

implementação feita abaixo utiliza a análise visual dos textos para verificar qual deles é o mais provável de ser o plain text. Tal método não é tão eficaz como os outros, porém é eficiente para textos curtos.

```
[17]: texto = "seguranca"
texto_enc = encriptacao(texto, 23)
texto_dec = decriptacao(texto_enc, 23)

def forca_bruta(cipher_text):
    plain_text_for_analysis = list()
    for i in range(27):
        plain_text = decriptacao(cipher_text, i)
        plain_text_for_analysis.append((i, plain_text))

        if plain_text == texto_dec:
            posicao = i

    print("Abaixo temos possíveis correspondências para o texto cifrado analisado.")

    for text in plain_text_for_analysis:
        print(f"{text}")

    print(f"Após uma análise dos textos, percebemos que a i={posicao} é a chave mais provável!")

forca_bruta(texto_enc)
```

```
Abaixo temos possíveis correspondências para o texto cifrado analisado.
(0, 'pbdroxkzx')
(1, 'oacqnwjyw')
(2, 'nzbpmvixv')
(3, 'myaoluhwu')
(4, 'lxznktgvt')
(5, 'kwymjsfus')
(6, 'jvxliretr')
(7, 'iuwkhqdsq')
(8, 'htvjgpcrp')
(9, 'gsuifobqo')
(10, 'frthenapn')
(11, 'eqsgdmzom')
(12, 'dprfclynl')
(13, 'coqebkxmk')
(14, 'bnpdajwlj')
(15, 'amoczivki')
(16, 'zlnbyhujh')
(17, 'ykmaxgtig')
(18, 'xjlzwfshf')
(19, 'wikyverge')
(20, 'vhjxudqfd')
(21, 'ugiwtcpec')
(22, 'tfhvsbodb')
(23, 'seguranca')
(24, 'rdftqzmbz')
(25, 'qcespylay')
(26, 'pbdroxkzx')
Após uma análise dos textos, percebemos que a i=23 é a chave mais provável!
```

Figure 3. Função de Ataque por Força Bruta

2.4. Quebra por Análise de Frequência

A análise de frequência é uma técnica probabilística que visa descobrir as letras e trechos de palavras mais frequentes em um idioma. No caso do português, vê-se que as letras ’a’, ’e’ e ’o’ são as letras com maior frequência. A partir disso, essa técnica consiste em analisar combinações de letras e possíveis padrões para decifrar um texto.

Inicialmente, um texto é encriptado com uma chave específica e desconhecida

para o atacante. Para descobrir a chave, é necessário descobrir a letra mais frequente no texto encriptado e associá-la com a letra mais frequente do alfabeto utilizado. No caso do português, temos 26 letras e, consequentemente, de 0 a 25 deslocamentos possíveis. Para a letra mais frequente do cipher text, calculamos seu deslocamento com as letras mais frequentes do alfabeto.

No código disponibilizado posteriormente, utilizam-se dicionários para armazenar as frequências de letras e calcular os deslocamentos para achar a chave.

Esse tipo de ataque é viável para decifrar uma cifra de César, pois essa cifra mantém as frequências relativas de letras inalteradas, de modo que o texto cifrado ainda reflete o padrão de distribuição de letras. Entretanto, essa estratégia depende do tamanho do texto, de modo que não é recomendado aplicá-la em textos muito curtos, pois a frequência de letras pode não ser representativa. Além disso, é fundamental conhecer o idioma que está sendo utilizado para fazer o ataque por análise de frequência.

Embora tenhamos apenas 26 letras no alfabeto, é necessário decodificar cada letra do texto cifrado para descobrir o plain text. Nesse sentido, quanto maior a palavra, maior será o tempo e dificuldade para fazer a quebra. Portanto, sua complexidade em notação Big-O é O(n), sendo ”n” o tamanho do texto.

Por fim, é importante ressaltar que esse método é altamente eficaz contra cifras simples, porém, quando precisamos quebrar cifras mais complexas e modernas, é preferível utilizar outros métodos.

Além da identificação da letra mais frequente, uma melhoria na análise consiste em utilizar uma função de score para comparar a distribuição de letras do texto decifrado com a distribuição esperada da língua portuguesa. A função calcula um valor numérico que representa o quão distante o padrão do texto decifrado está do padrão típico do idioma, utilizando para isso a soma dos erros quadráticos entre as frequências observadas e as esperadas. Assim, para cada chave possível, gera-se um texto decifrado e calcula-se um score; quanto menor o score, maior a probabilidade de o texto estar correto. Essa abordagem permite automatizar a escolha da chave mais plausível com base em critérios estatísticos, aumentando significativamente a eficácia do ataque, mesmo quando há múltiplas candidatas aparentes.

```
[6]: texto2 = "essetextocontemmuitasletrasmeuscontemporaneos"
     texto_enc2 = encriptacao(texto2, 7)

     # Dicionário com as frequências de cada letra em Português
     frequencia_pt = {
         'a': 13.9, 'b': 1.0, 'c': 4.4, 'd': 5.4, 'e': 12.2,
         'f': 1.0, 'g': 1.2, 'h': 0.8, 'i': 6.9, 'j': 0.4,
         'k': 0.1, 'l': 2.8, 'm': 4.2, 'n': 5.3, 'o': 10.8,
         'p': 2.9, 'q': 0.9, 'r': 6.9, 's': 7.9, 't': 4.9,
         'u': 4.0, 'v': 1.3, 'w': 0.0, 'x': 0.3, 'y': 0.0, 'z': 0.4
     }

     # Ordenando em ordem decrescente de frequencia
     frequencia_pt_ord = dict(sorted(frequencia_pt.items(), key=lambda item: (-item[1], item[0])))

     dic_frequencias = {}

     def dist_frequencia(cipher_text):
         dic_frequencias.clear()

         for letra in cipher_text:
             dic_frequencias[letra] = dic_frequencias.get(letra, 0) + 1

         dic_frequencias_ord = dict(
             sorted(dic_frequencias.items(), key=lambda item: (-item[1], item[0]))
         )

         letra_mais_freq_cifrada = next(iter(dic_frequencias_ord))
         pos_cifrada = caracteres_invertido[letra_mais_freq_cifrada]

         resultados = []

         for letra_pt in frequencia_pt_ord:
             pos_pt = caracteres_invertido[letra_pt]
             deslocamento = (pos_cifrada - pos_pt) % 26
             texto_decifrado = decriptacao(cipher_text, deslocamento)
             score = calcular_score(texto_decifrado)
             resultados.append((deslocamento, letra_pt, texto_decifrado, score))

         resultados.sort(key=lambda x: x[3])  # Ordena pelo menor score (melhor correspondência)
         return resultados
```

Figure 4. Função de Ataque por Análise de Frequência

```python
def calcular_score(texto_decifrado):
    # Conta a frequência das letras no texto decifrado
    total = len(texto_decifrado)
    freq_decifrado = {}
    for letra in texto_decifrado:
        if letra in frequencia_pt:
            freq_decifrado[letra] = freq_decifrado.get(letra, 0) + 1

    # Converte para porcentagem
    for letra in freq_decifrado:
        freq_decifrado[letra] = (freq_decifrado[letra] / total) * 100

    # Calcula o erro quadrático entre as frequências
    score = 0.0
    for letra in frequencia_pt:
        freq_real = frequencia_pt[letra]
        freq_observada = freq_decifrado.get(letra, 0)
        score += (freq_real - freq_observada) ** 2

    return score


possiveis = dist_frequencia(texto_enc2)

print("\nMelhores decifrações (ordenadas por score):\n")
for chave, letra, tentativa, score in possiveis[:5]:  # Mostra as 5 melhores
    print(f"KEY: {chave:2} - Letra: {letra} | SCORE: {score:.2f} | {tentativa}")
```

Figure 5. Função para calcular Score

```
Melhores decifrações (ordenadas por score):

KEY:  7 - Letra: e | SCORE: 276.69 | essetextocontemmuitasletrasmeuscontemporaneos
KEY: 11 - Letra: a | SCORE: 678.92 | aooapatpkykjpaiiqepwohapnwoiaqoykjpailknwjako
KEY: 19 - Letra: s | SCORE: 862.92 | sggshslhcqcbhsaaiwhogzshfogasigqcbhsadcfobscg
KEY: 21 - Letra: q | SCORE: 881.58 | qeeqfqjfaoazfqyygufmexqfdmeyqgeoazfqybadmzqae
KEY:  8 - Letra: d | SCORE: 887.81 | drrdsdwsnbnmsdllthszrkdsqzrldtrbnmsdlonqzmdnr
```

Figure 6. Output da Função de Ataque por Análise de Frequência

Pela análise do resultado obtido acima, podemos ver que a letra ”e” é a mais frequente do texto claro. Calculando o deslocamento de acordo com a chave obtida, podemos perceber que, no texto escuro, a letra mais frequente estava codificada como ”l”.

3. Exercício 2 - Quebrando Cifra por Transposição

A cifra de transposição é uma técnica de criptografia clássica que atua reorganizando a ordem das letras de uma mensagem, sem alterar os caracteres em si. Diferente das cifras de substituição — que trocam letras por outras — a cifra de transposição mantém as letras originais, apenas modificando sua posição no texto. O objetivo é embaralhar a mensagem de forma que apenas alguém com o conhecimento da chave possa restaurar a ordem correta e entender o conteúdo.

A segurança da cifra de transposição depende do segredo da chave e do tamanho da mensagem. Ela sozinha não é muito segura para proteger mensagens modernas, porém ainda é usada como base em sistemas mais complexos, e tem valor didático no ensino de criptografia.

Comentários sobre a implementação, análises de viabilidade, complexidade de algoritmo e tempo de execução serão abordados nas subseções abaixo.

O código-fonte completo pode ser acessado em:
https://jupyter.org/try-jupyter/lab/index.html?path=Lista01-Ex2.ipynb

Caso não seja possível acessar diretamente pelo link acima, o código-fonte pode ser encontrado no arquivo **"Lista01-Ex2.ipynb"** localizado no seguinte repositório do GitHub:
https://github.com/Luccambc/Seguranca-Computacional/tree/main/Lista%2001

3.1. Encriptação

O processo de encriptação de uma mensagem consiste em reorganizar as letras existentes no texto. Inicialmente, cria-se uma matriz quadrada de tamanho n referente ao tamanho da chave utilizada. Cada espaço dessa matriz receberá um caractere do texto claro. Porém, antes de encriptar a mensagem, é preciso verificar se o tamanho da mensagem é grande o suficiente para preencher toda a matriz e, caso não seja, colocamos caracteres extras que não alteram o sentido da mensagem original (processo de padding) para garantir que a encriptação seja realizada corretamente.

Após realizar esses passos, a mensagem estará disposta em linhas e colunas, sendo que cada coluna dessa matriz representa uma letra da chave. Tendo isso em mente, iremos analisar a chave e, em ordem alfabética, pegaremos a coluna referente a cada uma das letras da chave e criaremos o texto cifrado.

O algoritmo de encriptação feito possui uma complexidade O(n), pois estaremos trabalhando com matrizes.

Abaixo podemos verificar o código que realiza esse processo.

```
•[4]: import math

def enc_transp(plain_text, key):
    texto = plain_text.strip().replace(" ", "").upper()
    #print(texto)

    # Tratando caracteres faltantes
    len_key = len(key)
    len_text = len(texto)
    diff = abs(len_key - abs(len_text - len_key))

    if diff % len_key != 0:
        for i in range(diff):
            texto = texto + 'X'

    # Criando a matriz
    n_columns = len_key
    n_rows = math.ceil(len_text / len_key)

    matrix = [[] for _ in range(n_rows)]
    index = 0

    for i in range(n_rows):
        for j in range(n_columns):
            matrix[i].append(texto[index])
            index += 1

    # Cria uma lista com (caractere, índice original)
    key_indexed = list(enumerate(key))

    # Ordena essa lista por caractere (usa o segundo elemento da tupla)
    ord_list = sorted(key_indexed, key=lambda x: x[1])

    # Agora temos os índices das colunas na ordem correta:
    ord_columns = [i[0] for i in ord_list]

    cipher_text = ""
    for col in ord_columns:
        for row in matrix:
            cipher_text += row[col]

    return cipher_text

enc_transp("PLEASETRANSFERONEMILLIONDOLLARSTOMYSWISSBANKACCOUNTSIXTWOTWO", "MEGABUCK")
```

```
[4]: 'AFLLSKSOSELAWAIXTOOSSCTXLNMOMANTESILYNTWRNNTSOWXPAEDOBUOERIRICXX'
```

Figure 7. Função de Encriptação da Cifra de Transposição

3.2. Ataque por Força Bruta

O ataque por força bruta em uma cifra de transposição é uma abordagem em que o atacante tenta todas as permutações possíveis da chave para decifrar o texto cifrado, até encontrar a combinação correta que revela o texto claro.

A grande vantagem dessa técnica é que ela não requer conhecimento prévio da

chave, embora o número de tentativas possa crescer exponencialmente à medida que o tamanho da chave aumenta. Sob essa perspectiva, temos que uma chave de n elementos possui n! permutações possíveis, de modo que pode ser extremamente demorado decriptar mensagens longas ou com chaves muito grandes.

Um possível problema envolvendo o algoritmo de ataque por força bruta feito em Python é o estouro de memória oriundo da grande quantidade de permutações existentes.

Abaixo temos os códigos de duas funções auxiliares. A função `find_div(n)` retorna uma lista com os divisores de um inteiro n qualquer e a função `find_permutations(n)` utiliza a biblioteca itertools para gerar uma lista com as permutações de uma lista com n elementos.

[2]:
```python
# Função para achar divisores (possíveis tamanhos de chave)
def find_div(n):
    divs = []
    for i in range(1, n + 1):
        if n % i == 0:
            divs.append(i)

    return divs

# Exemplo de uso:
print(find_div(17))
```

```
[1, 17]
```

[4]:
```python
import itertools

def find_permutations(n):
    nums = list(range(0, n))
    perms = list(itertools.permutations(nums))

    return perms

resultado = find_permutations(3)
print(resultado)
```

```
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
```

Figure 8. Funções Auxiliares

Por fim, temos o código que realiza o ataque por força bruta. Junto ao código, podemos observar o output de um texto de 6 caracteres. Não foi possível mostrar todo o output devido à quantidade de linhas existentes na saída do código.

```
•[9]: def apply_permutation(block, permutation):
          return ''.join(block[i] for i in permutation)

      def trasp_brute_force(cipher_text):
          len_cipher = len(cipher_text)
          possible_keys = find_div(len_cipher)

          for key in possible_keys:
              print(f"\n -= Testando chave de tamanho {key} -=")
              permutations = find_permutations(key)

              # Quebra o texto cifrado em blocos do tamanho da chave
              blocks = [cipher_text[i:i+key] for i in range(0, len_cipher, key)]

              for perm in permutations:
                  decrypted = ''

                  # Para cada bloco, aplica a permutação reversa
                  for block in blocks:
                      if len(block) < key:
                          continue
                      decrypted += apply_permutation(block, perm)

                  print(f"Chave {perm}: {decrypted}")

      ct = "VASCOX"
      trasp_brute_force(ct)
```

```
 -= Testando chave de tamanho 1 -=
Chave (0,): VASCOX

 -= Testando chave de tamanho 2 -=
Chave (0, 1): VASCOX
Chave (1, 0): AVCSXO

 -= Testando chave de tamanho 3 -=
Chave (0, 1, 2): VASCOX
Chave (0, 2, 1): VSACXO
Chave (1, 0, 2): AVSOCX
Chave (1, 2, 0): ASVOXC
Chave (2, 0, 1): SVAXCO
Chave (2, 1, 0): SAVXOC

 -= Testando chave de tamanho 6 -=
Chave (0, 1, 2, 3, 4, 5): VASCOX
Chave (0, 1, 2, 3, 5, 4): VASCXO
Chave (0, 1, 2, 4, 3, 5): VASOCX
Chave (0, 1, 2, 4, 5, 3): VASOXC
Chave (0, 1, 2, 5, 3, 4): VASXCO
```

Figure 9. Funções do Ataque por Força Bruta

3.3. Ataque por Distribuição de Frequência

A análise de frequência permite identificar padrões típicos da língua, mesmo com a ordem das letras embaralhada. Ao testar diferentes tamanhos de chave e permutações possíveis, é possível reorganizar os blocos de texto e verificar quais combinações resultam em mensagens legíveis. Assim, a análise de frequência, combinada com tentativas sistemáticas, torna-se uma ferramenta eficaz na criptoanálise de cifras por transposição. Esse processo se torna ainda mais viável quando se conhece o idioma da mensagem original, já que a presença de palavras ou combinações recorrentes pode indicar aproximações corretas da chave. Mesmo sendo uma técnica simples, a transposição continua relevante no estudo da segurança da informação por sua contribuição histórica e pelo valor educacional no entendimento e estudo dos princípios da criptografia.