Lista de Exercícios 01

Leandro Beloti Kornelius

21 de abril de 2025

Resumo

Esta lista de exercícios 01 é da Disciplina de Segurança Computacional da UnB e busca aprofundar através da prática conceitos aprendidos na disciplina.

1 Introdução

Neste relatório abordaremos duas cifras: a por deslocamento e por transposição. Ademais, será visto também dois possíveis ataques para tais cifras: por força bruta e por frequência. Assim, será possível contrastar tanto as cifras quanto os possíveis ataques.

Ambos algoritmos a serem detalhados neste relatório são criptografias simétricas. Isso significa que o remetente e destinatário possuem a mesma chave para criptografar e descriptografar a mensagem. Logo, devem ter os seguintes componentes em suas implementações:

- Geração da chave privada
- Função de Encriptação da mensagem
- Função de Decriptação da mensagem

Com isso, os algoritmos de quebra devem buscar através de diferentes técnicas descobrir a chave usada para criptografia, um pedaço ou a mensagem por completo.

Todas as implementações utilizadas neste relatório estão presentes no repositório:
Repositório com códigos da Lista de Exercícios 01

2 Exercício 1 - Quebrando Shift Cipher

2.1 Implementação da Cifra por Deslocamento

A Cifra por Deslocamento consiste em deslocar as letras da mensagem por um número. Com isso, em uma Cifra de César por exemplo todas letras da mensagem são deslocadas em três que é a chave da cifra.

Com base nisso, iremos iniciar a implementação definindo um dicionário que atribui em ordem números à letras correspondentes.

Para não haver diferenciação entre maiúsculo e minúsculo todos caracteres das mensagens das funções no relatório foram reduzidos para sua versão minúscula.

Em seguida, iteramos pelos caracteres da mensagem. Caso o caracter esteja nas letras da língua portuguesa será necessário realizar o deslocamento, se não iremos apenas inserir o caracter sem deslocá-lo à mensagem cifrada.

Para os caractéres que devem ser deslocados é necessário identificar o inteiro correspondente, somar ao valor da chave e utilizar o operador de resto com a quantidade de letras no dicionário para retornar somas que ultrapassam esta quantidade para um caracter inicial. Logo, estaremos fazendo aritmética modular para fazer este deslocamento.

Tal implementação pode ser consultada abaixo:

```python
def enc(m, k):
 m = m.lower()
 print(f'Message being encrypted: {m}')
 c = ''
 for char in m:
  if char in letters.values():
   for num, letter in letters.items():
    if char == letter:
     c += letters[(num + k) % len(letters)]
  else:
   c += char
 print(f'Message encrypted: {c}\n')
 return c
```

Para verificar o funcionamento da função, foram feitos três testes. Dois com a Cifra de César em que K=3 e uma com K=8 com o primeiro parágrafo do primeiro livro do Harry Potter. Estas mesmas mensagens serão usadas para o restante dos testes. No código, é possível averiguar o correto funcionamento desta função.

Para descriptografar é necessário fazer o processo inverso, ou seja retornar o deslocamento. Como na função de encriptação foi feito a soma para realizar o deslocamento, será feito a subtração. O restante tem funcionamento análogo, como pode ser visualizado abaixo:

```
def dec(c, k):
 c = c.lower()
 print(f'Cipher being decrypted: {c}')
 m = ''
 for char in c:
  if char in letters.values():
   for num, letter in letters.items():
    if char == letter:
     m += letters[(num - k) % len(letters)]
  else:
   m += char
 print(f'Cipher decrypted: {m}\n')
 return m
```

Observa-se que ao utilizar tal função com os testes definidos anteriores obtemos os resultados esperados.

Em ambos algoritmos a complexidade do algoritmo é definida pelo tamanho da mensagem e a quantidade de letras do alfabeto. Por parte, obtemos:

- Primeiro loop (linha 5) com complexidade n em que n é o tamanho da mensagem
- Condicional (linha 6) com complexidade k em que k é o número de letras
- Último loop (linha 7) com complexidade k em que k é o número de letras

No algoritmo em questão, o k é constante. Ou seja, a complexidade desta função é **O(n)** em que n é o tamanho da mensagem a ser criptografada.

2.2 Ataque por Força Bruta

O ataque por Força Bruta consiste em testar todas as chaves possíveis até obter tradução inteligível para o texto. Supondo que a pessoa realizando o algoritmos de quebra saiba que a mensagem foi originada no Brasil a quantidade de tentativas para as chaves será igual a quantidade de letras no alfabeto brasileiro.

Iniciaremos iterarando sob todas possibilidades de valores para K, de 0 a 25 para o alfabeto brasileiro. Para cada uma das chaves testadas iremos fazer o processo de descriptografia e adicionar à um vetor de possíveis mensagens. Na função, apenas retornamos todas possíveis mensagens, assim o "hacker"poderá averiguar todas mensagens para ver aquela que faz sentido.

O algoritmo implementado pode ser visualizado abaixo:

```
def brute_force_attack(c):
 c = c.lower()
 print("Testing all possible keys:")
 possible_messages = []
 for k in range(len(letters)):
  attempted_message = dec(c, k)
  possible_messages.append(attempted_message)
 return possible_messages
```

Ao testar esta quebra com os testes estabelecidos anteriormente, obtemos os resultados esperados.

Para este algoritmo, a complexidade é definida pela quantidade de letras do alfabeto e função "dec"utilizada. Por parte, obtemos:

- Primeiro loop (linha 5) com complexidade k em que k é o número de letras
- Função "dec"a qual tem sua complexidade definida por n em que n é o tamanho da cifra a ser descriptografada

No algoritmo em questão, o k é constante. Ou seja, a complexidade desta função é definida pela função "dec"utilizada a qual tem complexidade **O(n)** em que n é o tamanho da mensagem a ser quebrada.

2.3 Ataque por Distribuição de Frequência

O ataque por Distribuição de Frequência consiste em comparar a frequência de letras do texto cifrado com a esperada na língua da mensagem. Com isso, é possível descobrir o deslocamento para que o texto cifrado se aproxime mais com a frequência da língua. Usaremos de referência para a implementação a tabela de frequência da língua portuguesa e definimos um dicionário com a frequência de cada letra.

Para a implementação da quebra, faz-se necessário duas funções auxiliares. A primeira realizar a distribuição de frequência para um texto qualquer. Esta função será usada para podermos ter a frequência de caracteres para o texto cifrado e possui complexidade **O(n)** em que n é o tamanho da cifra. Além dessa, faz-se necessário haver uma função que meça o quão parecido são as duas distribuições de frequência a qual também tem complexidade linear **O(n)** em que n é a quantidade de caracteres da primeira frequência.

```
def text_frequency(t):
 t = t.lower()
 cipher_char_frequency = {}

 for char in letters.values():
  cipher_char_frequency[char] = 0

 # Count occurences of chars
 for char in t:
  if char in letters.values():
   cipher_char_frequency[char] += 1

 # Converts to percentage
 total_chars = sum(cipher_char_frequency.values())
 for char in cipher_char_frequency:
  cipher_char_frequency[char] = (cipher_char_frequency[char] / total_chars) * 100

 return cipher_char_frequency
```

```
def frequency_score(frequency_one, frequency_two):
 return sum((frequency_one.get(char, 0) - frequency_two.get(char, 0)) ** 2 for char in frequency_one)
```

A quebra por Distribuição de Frequência poderia fazer uma equivalência entre algumas das letras mais frequentes e com isso realizar a quebra. Entretanto, esta tentativa pode não ser a melhor, pois podemos ter uma presença irregular de um certo caracter na mensagem de origem. Portanto, optei por iterar sob todas as chaves e realizar o "score"para todos os caracteres a fim de escolher a mensagem que mais se aproxima de distribuição de frequência da língua portuguesa, conforme a implementação abaixo.

```
def frequency_distribution_attack(c):
 smaller_frequency_score = float('inf')
 c = c.lower()
 for k in range(len(letters)):
  attempted_message = dec(c, k)
  frequency_attempted_message = text_frequency(attempted_message)
  frequency_score_attempted_message = frequency_score(frequency_attempted_message, port_char_frequency)
  if frequency_score_attempted_message < smaller_frequency_score:
   best_key = k
   smaller_frequency_score = frequency_score_attempted_message
 return dec(c, best_key)
```

Ao testar o algoritmo para a mensagem do Harry Potter, obtemos os resultados esperados.

Para este algoritmo, a complexidade é definida pela quantidade de letras do alfabeto e as funções auxiliares utilizadas. Por parte, obtemos:

- Primeiro loop (linha 4) com complexidade k em que k é o número de letras
- Função "dec"a qual tem sua complexidade definida por n em que n é o tamanho da cifra a ser descriptografada
- Função "textfrequency"a qual tem sua complexidade definida por n em que n é o tamanho da mensagem a ser feita a frequência
- Função "frequencyscore"a qual tem sua complexidade definida por n em que n é o tamanho do texto da primeira frequência

No algoritmo em questão, o k é constante. Ou seja, a complexidade desta função é definida pelas funções auxiliares as quais tem complexidade **O(n)** em que n é o tamanho da mensagem a ser quebrada.

2.4 Conclusão

Com base no analisado, nota-se os seguintes tempos de execução em segundos obtidos através de uma função implementada:

```
{'Enc': 0.0005018472671508789,
 'Dec': 0.0005079984664916992,
 'Brute Force': 0.008291339874267578,
 'Distribution Frequency': 0.012526249885559082}
```

Em suma, nota-se que a Cifra por Deslocamento é facilmente quebrada por ambas técnicas através de soluções polinomiais e, por isso, não é uma técnica confiável de criptografia. A quebra por força bruta exige que alguém esteja supervisionando o ataque a fim de confirmar qual mensagem é inteligível. Ademais, é mais rápida do que a quebra por Distribuição de Frequência. Em contrapartida, a quebra por Distribuição de Frequência não exige essa supervisão, pois faz uso da língua da mensagem para prover a melhor aproximação. Entretanto, caso a mensagem seja irregular ou pequena ela pode se tornar pouco efetiva, pois a distribuição da mensagem cifrada pode ser bem distante da língua.

3 Exercício 2 - Quebrando Cifra por Transposição

3.1 Implementação da Cifra por Transposição

A Cifra por Transposição consiste em rearranjar as letras da mensagem por colunas definidas pelo K (chave da criptografia).

Com base nisso, iremos iniciar a implementação retirando espaços em branco e deixando todas letras minúsculas.

Em seguida, como será necessário fazer o rearranjo com base nas colunas, será necessário verificar se a quantidade de caracteres é divisível pela chave K, pois caso não seja, precisaremos completar através de um caracter. Na implementação foi escolhido o "x".

Agora, iremos fazer a troca da mensagem que antes estava em linhas para colunas. Tal processo foi feito com uma lógica de resto da divisão, pois os caracteres que estão na mesma coluna na transposição terão o mesmo resto da divisão por K.

Por fim, por a implementação ter usado o método de permutação é necessário realizar a permutação conforme o que foi recebido como parâmetro rearranjando as colunas.

Tal implementação pode ser consultada abaixo:

```
def encTransp(m, k, perm):
 print(f'Message being encrypted is: {m}')
 m = m.replace(' ', '').lower()
 aux = [''] * k
 if len(m) % k != 0:
  m += 'x' * (k - (len(m) % k))
 for i in range(len(m)):
  aux[i % k] += m[i]
 c = ''
 for number in perm:
  c += aux[number]
 return c
```

Para verificar o funcionamento da função, foram feitos alguns testes e um foi deixado no código para fácil verificação do funcionamento desta função.

A complexidade deste algoritmo é definida pelo tamanho da mensagem e a quantidade números da permutação. Por parte, obtemos:

- Primeiro loop (linha 7) com complexidade n em que n é o tamanho da mensagem
- Segundo (linha 10) com complexidade p em que p é o número de números da permutação

No algoritmo em questão, a complexidade desta função é **O(n * p)** em que n é o tamanho da mensagem a ser criptografada e p é a quantidade de números da permutação.

3.2 Quebra da Cifra por Transposição por Força Bruta

Como mencionado anteriormente, a quebra por Força Bruta consiste em testar todas possibilidades. Para isso, foi necessário definir duas funções auxiliares:

- A primeira estabelece todos divisores até um número n
- A segunda gera todas permutações de 0 a um número k

```python
def find_divisors(n):
 import math
 divisors = []
 for d in range(1, math.isqrt(n) + 1):
  if n % d == 0:
   divisors.append(d)
   divisors.append(n // d)
 return sorted(divisors)
```

Executed at 2025.04.21 16:51:10 in 199ms

```python
def find_permutations_up_to_k(k):
    from itertools import permutations
    items = list(range(0, k))
    for perm in permutations(items):
        yield perm
```

Com isso, o código da quebra inicia encontrando todas chaves possíveis para gerar a transposição do texto cifrado. Essas chaves são todos os divisores do tamanhos do texto cifrado afinal foi usado um padding na função de encriptação para completar a mensagem caso a quantidade de caracteres não fosse divisível.

Em seguida, passamos por todas estas chaves possíveis e fazemos o mesmo processo da função de encriptação em que usamos uma lógica de resto da divisão para separar caracteres da mesma coluna. Portanto, as colunas voltar a ser as linhas, pois ao transpor a tabela, retornamos a sua forma original novamente.

Por fim, é feito todas permutações até a chave que está sendo testada para compor todas possibilidades de quebra. Tal processo pode ser visualizado na imagem abaixo:

```python
def transposition_cipher_brute_force_attack(c):
    c = c.lower()
    possible_messages = []
    possible_keys = find_divisors(len(c))
    for k in possible_keys:
        aux = [''] * k
        for i in range(len(c)):
            aux[i % k] += c[i]
        possible_perms_for_k = find_permutations_up_to_k(k)
        for perm in possible_perms_for_k:
            m = ''
            for number in perm:
                m += aux[number]
            possible_messages.append(m)
    return possible_messages
```

Para verificar o funcionamento da função, foram feitos alguns testes e um foi deixado no código para fácil verificação do funcionamento desta função.

Para este algoritmo, a complexidade é definida pelo tamanho do texto criptografado e as funções auxiliares utilizadas. Por parte, obtemos:

- Primeira função auxiliar dos divisores (linha 4) a qual tem complexidade $O(\sqrt{n})$ em que n é o tamanho do texto cifrado
- Segundo loop (linha 7) a qual tem complexidade definida por n em que n é o tamanho da mensagem cifrada
- Segunda função auxiliar e terceiro loop (linhas 9 e 10) que tem complexidade fatorial **O(k!)** devido as permutações a serem geradas a partir da chave k
- Último loop o qual tem complexidade definido pelo tamano da permutação p

No algoritmo em questão, a complexidade desta função é definida pela função auxiliar de permutação a qual tem complexidade **O(k!)** em que k são todas possibilidades de chave para o texto cifrado.

3.3 Conclusão

Com base no analisado, nota-se os seguintes tempos de execução em segundos obtidos através de uma função implementada:

```
{'Enc': 0.00010006427764892578, 'Brute Force': 0.0010593891143798827}
```

Em suma, nota-se que mesmo sem a implementação da Quebra por Distribuição de Frequência para a Cifra por Transposição, nota-se que a por Força Bruta não é uma boa escolha, pois com textos grandes o seu tempo de execução aumenta consideravalmente e se torna inviável.