Instituto Tecnológico y de Estudios Superiores de Monterrey

Actividad 2.1. Cifrado César, sustitución monoalfabética y Vigenère

Juan Pablo Echeagaray González

A00830646

Análisis de Criptograrfía y Seguridad

MA2002B.300

Dr.-Ing. Jonathan Montalvo-Urquizo

26 de mayo del 2022

1. Sobre el cifrado César

El cifrado de César fue el más sencillo de implementar de todos los métodos vistos, su tiempo de encriptado y desencriptado (aún sin conocer la llave) es mínusculo a comparación de los demás. La única barrera encontrada es la incertidumbre del lenguaje original del *plaintext*. Para la implementación sugerida solamente se toman en cuenta caracteres de lenguas romances, lo cual no nos asegura poder realizar un proceso de criptoanálisis en una situación general.

El proceso de encriptado y desencriptado con una llave conocida le tomó a la máquina de prueba un promedio de $21\mu s$; cuando no disponía de una llave se realizó un ataque por fuerza bruta, el tiempo de desencriptado incrementó a $278\mu s$. El incremento de tiempo no es tan grande porque solamente se deben de probar un total de 26 diferentes llaves para desencriptar el mensaje; al final de este proceso se necesita de un ser humano que rectifique los resultados.

2. Sobre el cifrado monoalfabético

El cifrado monoalfabético es mucho más fuerte que un simple cifrado César, se pasa de un espacio de n posibles llaves a $n!$, un espacio que tiene un tamaño de al menos 25 ordenes de magnitud superior cuando solamente se tiene un alfabeto consistente de letras.

Enumerar todas las posibles llaves e intentar desencriptar el mensaje podría ser una solución factible en equipos con un mayor poder de cómputo; sin embargo, una aproximación por fuerza bruta de ese estilo no es necesario para romper este cifrado. Dado que este método sigue siendo una simple sustitución, podemos hacer uso de herramientas estadísticas para encontrar el mensaje original, de forma más específica, se realiza un *análisis de frecuencias*.

Suponiendo que el criptoanalista conozca el lenguaje original del *plaintext*, este puede comenzar el proceso de desencriptado al analizar la frecuencia relativa de cada caracter dentro del *ciphertext*; después se pueden comparar estas frecuencias con algunas tablas de frecuencias de letras en el lenguaje a tratar. En la primera iteración se pueden remplazar las primeras n letras más frecuentes del *ciphertext* con las n primeras letras más frecuentes de las tablas conocidas.

Después de esta iteración se analiza el texto, con la esperanza de que esta sustitución torne más legibles algunas secciones del *ciphertext*. De aquí, se puede ir determinando de forma empírica las sustituciones a realizar, por ejemplo, si uno encuentra después de la primera iteración el texto *tha* y se sabe que el texto original está escrito en inglés, se propone la sustitución $a \to e$.

Este método tomó más tiempo de implementar y romper que el anterior, pero se tornó bastante sencillo de romper una vez que se conocían los remplazos adecuados. Este caso de estudio fue más sencillo de resolver dada la longitud del texto original de alrededor de $48,000$ caracteres. Con la implementación utilizada no existe una manera metódica de determinar el tiempo de ruptura del encriptado.

3. Sobre el cifrado Vigenère

De los desarrollados en esta entrega, el cifrado de Vigenère es el más fuerte de todos, por mucho tiempo fue considerado irrompible, pero con las tecnologías actuales e inteligentes análisis es posible llegar a romperlo. La fortaleza sobre los anteriores es que entra en una

categoría de métodos de cifrado conocidos como *polialfabéticos*, para este caso se utiliza un cifrado de César para cada uno de los caracteres en el *plaintext* que logra obscurecer por completo toda la información estadística del mensaje.

El algoritmo de encriptado recibe 2 secuencias de caracteres, una consistente del mensaje en claro y otra de la llave usada para cifrar, m es la longitud de la llave. El caracter iésimo del texto cifrado se calcula con la siguiente expresión:

$$C_i = (p_i + k_{i \bmod m}) \bmod 26 \tag{1}$$

Para desencriptar el mensaje se tiene una expresión simple también:

$$p_i = (C_i - k_{i \bmod m}) \bmod 26 \tag{2}$$

El espacio de búsqueda para la llave se torna más grande, si tomamos n como la longitud del alfabeto del lenguaje, y m la longitud del mensaje, el espacio de búsqueda es entonces n^m. No existe una manera clara de intentar resolver este problema por fuerza bruta, ¿Con qué llave empezar? ¿Cuál será su longitud? Son algunas de las preguntas a las que nos enfrentaríamos si enfrentáramos el problema ciegamente.

A. Código implementado y resultados

Actividad 2.1. Cifrado César, sustitución monoalfabética y Vigenère

- Juan Pablo Echeagaray González
- A00830646
- Análisis de Criptografía y Seguridad
- Profesores:
  - Dr. Alberto F. Martínez
  - Dr.-Ing. Jonathan Montalvo-Urquizo

- 24 de mayo del 2022

Dependencias

In [39]:
```
from string import ascii_letters
from random import sample, seed
```

Cifrado César

Encriptado César

In [40]:
```
def encrypt_caesar(message: str, offset: int) -> str:
    """Caesar encryption with a given offset

    Args:
        message (str): Plain text to be encrypted
        offset (int): Integer offset to be used for encryption

    Returns:
        str: Cipher text
    """

    result = ''
    for char in message:
        # Check if its uppercase
        if char.isupper():
            result += chr((ord(char) + offset - 65) % 26 + 65)
        else:
            result += chr((ord(char) + offset - 97) % 26 + 97)

    return result
```

Rompiendo cifrado César

In [41]:
```
def break_caesar(message: str, known_key: int = None) -> dict:
    """Brute force approach to break the Caesar cipher

    Args:
        message (str): Cipher text to be decrypted
        known_key (int, optional): Offset to be used for decryption. Defaults to None.

    Returns:
        dict: Dictionary that contains all the attempts taken to break the cipher
    """

    alphabet = ascii_letters[len(ascii_letters) // 2:]
    res = []

    if known_key is None:
        search_space = range(len(alphabet))
    else:
        search_space = [known_key]

    for key in search_space:
        translated = ''
        for char in message:
            if char in alphabet:
                num = alphabet.find(char)
                num -= key
                if num < 0:
                    num += len(alphabet)
                translated += alphabet[num]
            else:
                translated += char

        res.append([key, translated])
        # Map list of lists to dict
        out = dict(res)
```

```
    return out
```

Probando cifrado y desencriptado César

In [42]:
```
def caesar_encryption(plain_text: str, mode: str) -> str:
    """Driver code for Caesar cipher exercise

    Args:
        plain_text (str): Plain text to be encrypted
        mode (str): Mode of operation. Can be 'encrypt' or 'decrypt'

    Returns:
        str: Output of the process
    """

    shift = 3
    ciphert_text = encrypt_caesar(plain_text, shift)
    if mode == 'known_key':
        result_2 = break_caesar(ciphert_text, shift)
        return f'''
        Plain text: {plain_text}
        Shift: {shift}
        Cipher text: {ciphert_text}
        Known-key
        Known-key-result: {result_2}'''

    elif mode == 'brute-force':
        result_1 = break_caesar(ciphert_text)
        return f'''
        Plain text: {plain_text}
        Shift: {shift}
        Cipher text: {ciphert_text}
        Brute Force
        Result: {result_1}'''

    else:
        return 'Invalid mode'
```

In [43]:
```
plain_text = 'PERO MIRA COMO BEBEN LOS PECES EN EL RIO'
```

In [44]:
```
%%timeit -n 5000
caesar_encryption(plain_text, 'known_key')
```

```
21.9 µs ± 4.08 µs per loop (mean ± std. dev. of 7 runs, 5,000 loops each)
```

In [45]:
```
%%timeit -n 5000
caesar_encryption(plain_text, 'brute-force')
```

```
316 µs ± 30 µs per loop (mean ± std. dev. of 7 runs, 5,000 loops each)
```

Cifrado monoalfabético

Generación de alfabeto aleatorio

In [46]:
```
def random_alphabet_table(this_seed: int = 1) -> str:
    """Random key generation

    Returns:
        str: Random key. Default seed is set to 1
    """
    seed(this_seed)
    character_pool = ascii_letters[len(ascii_letters) // 2:]
    orig = list(character_pool)
    shuffled = sample(orig, len(orig))
    key = dict(zip(orig, shuffled))

    return key
```

Encriptado

In [47]:
```
def encrypt_message(message: str, key: dict) -> str:
    """Monoalphabetic encryption with a given key

    Args:
        message (str): Plain text to be encrypted
        key (dict): Dictionary containing the encryption key

    Returns:
        str: Cipher text
    """
    encrypted = []
```

```
    message = message.upper()
    for char in message:
        if char in key:
            encrypted += key[char]
        else:
            encrypted += char

    return ''.join(encrypted)
```

Inverso Alfabeto

In [48]:
```
def inv_alphabet(key: dict) -> dict:

    return {v: k for k, v in key.items()}
```

A desencriptar

In [49]:
```
def decrypt_message(message: str, key: dict):

    return encrypt_message(message, inv_alphabet(key))
```

Prueba de monoencriptado

In [50]:
```
def mono_encryption(mode: str) -> str:

    file_path = '../../homeworks/ciphers/text2.txt'

    with open(file_path, 'r') as f:
        message = f.readlines()

    message = ''.join(message)

    # Encryption
    cipher = random_alphabet_table()

    if mode == 'encrypt':
        output = encrypt_message(message, cipher)
    elif mode == 'decrypt':
        encrypted = encrypt_message(message, cipher)
        output = decrypt_message(encrypted, cipher)
    else:
        print('Invalid mode')

    return output
```

El texto original contiene más de 10,000 caracteres, por lo que su visualización dentro del entorno de desarrollo se torna tosca. El siguiente comando realiza el proceso de desencriptado pero solo imprime los primeros 1000 caracteres.

In [51]:
```
mono_encryption('decrypt')[:1000]
```

Out[51]:
```
"ACCORDING TO ALL KNOWN LAWS OF AVIATION, THERE IS NO WAY A BEE SHOULD BE ABLE TO FLY.\nITS WINGS ARE TOO SMALL TO GET ITS FAT LITTLE BO
DY OFF THE GROUND.\nTHE BEE, OF COURSE, FLIES ANYWAY BECAUSE BEES DON'T CARE WHAT HUMANS THINK IS IMPOSSIBLE.\nYELLOW, BLACK. YELLOW, BL
ACK. YELLOW, BLACK. YELLOW, BLACK.\nOOH, BLACK AND YELLOW!\nLET'S SHAKE IT UP A LITTLE.\nBARRY! BREAKFAST IS READY!\nCOMING!\nHANG ON A
SECOND.\nHELLO?\nBARRY?\nADAM?\nCAN YOU BELIEVE THIS IS HAPPENING?\nI CAN'T.\nI'LL PICK YOU UP.\nLOOKING SHARP.\nUSE THE STAIRS, YOUR FA
THER PAID GOOD MONEY FOR THOSE.\nSORRY. I'M EXCITED.\nHERE'S THE GRADUATE.\nWE'RE VERY PROUD OF YOU, SON.\nA PERFECT REPORT CARD, ALL
B'S.\nVERY PROUD.\nMA! I GOT A THING GOING HERE.\nYOU GOT LINT ON YOUR FUZZ.\nOW! THAT'S ME!\nWAVE TO US! WE'LL BE IN ROW 118,000.\nBYE!
\nBARRY, I TOLD YOU, STOP FLYING IN THE HOUSE!\nHEY, ADAM.\nHEY, BARRY.\nIS THAT FUZZ GEL?\nA LITTLE. SPECIAL DAY, GRADUATION.\nNEVER TH
OUGHT I'D MAKE IT.\nTHREE DAYS GRADE SCHOOL, THREE DAYS HIGH SCHOOL.\nTHOSE WERE AWKW"
```

Tiempo promedio de desencriptado:

In [52]:
```
%%timeit -n 500
decrypted = mono_encryption('decrypt')
```

```
11.5 ms ± 1.3 ms per loop (mean ± std. dev. of 7 runs, 500 loops each)
```

Análisis de frecuencias

In [53]:
```
def mono_frequency_analysis(cipher_text: str, language: str) -> str:

    frequencies = {'eng': 'ETAOINSHRDLCUMWFGYPBVKJXQZ',
                   'spa': 'EAOSNRILDTUCMPBHQYVGFJZXKW',
                   'fra': 'EASTIRNULODMCPVHGFBQJXZYKW'}

    symbols = [' ', ',', '.', '!', '?', ':', ';', '-', '"', "'", '\n', '\t',
               '1', '2', '3', '4', '5', '6', '7', '8', '9', '0']

    if language not in frequencies:
        print('Invalid language')
        return
```

```
    # May or may not be used
    lang_frequencies = frequencies[language]

    # Calculate the frequency of each letter in the ciphertext
    freq_table = {}
    for char in cipher_text:
        if char in freq_table:
            freq_table[char] += 1
        else:
            freq_table[char] = 1

    freq_table = dict(sorted(freq_table.items(), key=lambda x: x[1], reverse=True))
    # Drop symbols from the frequency table
    for symbol in symbols:
        if symbol in freq_table:
            freq_table.pop(symbol)


    # Replace each letter in the cipher text with the most frequent letter in the language
    # decrypted = ''
    # for char in cipher_text:
    #     if char in freq_table:
    #         decrypted += lang_frequencies[list(freq_table).index(char)]
    #     else:
    #         decrypted += char
    # Too optimistic approach, needs some human work

    """
    Defined after checking the attempts
    F --> T
    U --> H
    D --> E
    T --> R
    M --> I
    Z --> S
    Q --> N
    M --> I
    Z --> S
    R --> O
    V --> K
    P --> F
    K --> W
    E --> A
    H --> L
    O --> G
    W --> V
    S --> B
    I --> D
    B --> U
    A --> M
    J --> P
    L --> X
    """

    # Defined after iterably checking the attempts
    custom_translations = {'F': 'T', 'U': 'H', 'D': 'E', 'T': 'R', 'M': 'I',
                           'Z': 'S', 'Q': 'N', 'M': 'I', 'Z': 'S', 'R': 'O',
                           'V': 'K', 'P': 'F', 'K': 'W', 'E': 'A', 'H': 'L',
                           'O': 'G', 'W': 'V', 'S': 'B', 'I': 'D', 'B': 'U',
                           'A': 'M', 'J': 'P', 'L': 'X'}

    decrypted = ''
    for char in cipher_text:
        if char in custom_translations:
            decrypted += custom_translations[char]
        else:
            decrypted += char

    print(f'''Summary
    Frequency table: {freq_table}
    Length of the custom table: {len(custom_translations)}
    Cipher text (1st 100 chars): {cipher_text[:1000]}''')
    return decrypted


mono_frequency_analysis(mono_encryption('encrypt'), 'eng')[:1000]
```

```
Summary
    Frequency table: {'D': 4395, 'F': 3448, 'R': 3295, 'E': 2784, 'M': 2565, 'Q': 2412, 'Z': 2184, 'U': 2037, 'T': 1958, 'H': 1628, 'B': 1341, 'Y': 1253, 'I': 1088, 'K': 1017, 'A': 913, 'O': 885, 'C': 748, 'S': 744, 'P': 619, 'V': 547, 'J': 517, 'W': 377, 'G': 115, 'X': 68, 'L': 37, 'N': 26}
    Length of the custom table: 21
    Cipher text (1st 100 chars): ECCRTIMQO FR EHH VQRKQ HEKZ RP EWMEFMRQ, FUDTD MZ QR KEY E SDD ZURBHI SD ESHD FR PHY.
MFZ KMQOZ ETD FRR ZAEHH FR ODF MFZ PEF HMFFHD SRIY RPP FUD OTRBQI.
FUD SDD, RP CRBTZD, PHMDZ EQYKEY SDCEBZD SDDZ IRQ'F CETD KUEF UBAEQZ FUMQV MZ MAJRZZMSHD.
YDHHRK, SHECV. YDHHRK, SHECV. YDHHRK, SHECV. YDHHRK, SHECV.
RRU, SHECV EQI YDHHRK!
HDF'Z ZUEVD MF BJ E HMFFHD.
SETTY! STDEVPEZF MZ TDEIY!
CRAMQO!
UEQO RQ E ZDCRQI.
UDHHR?
SETTY?
EIEA?
CEQ YRB SDHMDWD FUMZ MZ UEJJDQMQO?
M CEQ'F.
M'HH JMCV YRB BJ.
HRRVMQO ZUETJ.
BZD FUD ZFEMTZ, YRBT PEFUDT JEMI ORRI ARQDY PRT FURZD.
ZRTTY. M'A DLCMFDI.
UDTD'Z FUD OTEIBEFD.
KD'TD WDTY JTRBI RP YRB, ZRQ.
E JDTPDCF TDJRTF CETI, EHH S'Z.
WDTY JTRBI.
AE! M ORF E FUMQO ORMQO UDTD.
YRB ORF HMQF RQ YRBT PBXX.
RK! FUEF'Z AD!
KEWD FR BZ! KD'HH SD MQ TRK 118,000.
SYD!
SETTY, M FRHI YRB, ZFRJ PHYMQO MQ FUD URBZD!
UDY, EIEA.
UDY, SETTY.
MZ FUEF PBXX ODH?
E HMFFHD. ZJDCMEH IEY, OTEIBEFMRQ.
QDWDT FURBOUF M'I AEVD MF.
FUTDD IEYZ OTEID ZCURRH, FUTDD IEYZ UMOU ZCURRH.
FURZD KDTD EKVK
```

Out[53]:
```
"ACCORDING TO ALL KNOWN LAWS OF AVIATION, THERE IS NO WAY A BEE SHOULD BE ABLE TO FLY.\nITS WINGS ARE TOO SMALL TO GET ITS FAT LITTLE BODY OFF THE GROUND.\nTHE BEE, OF COURSE, FLIES ANYWAY BECAUSE BEES DON'T CARE WHAT HUMANS THINK IS IMPOSSIBLE.\nYELLOW, BLACK. YELLOW, BLACK. YELLOW, BLACK. YELLOW, BLACK.\nOOH, BLACK AND YELLOW!\nLET'S SHAKE IT UP A LITTLE.\nBARRY! BREAKFAST IS READY!\nCOMING!\nHANG ON A SECOND.\nHELLO?\nBARRY?\nADAM?\nCAN YOU BELIEVE THIS IS HAPPENING?\nI CAN'T.\nI'LL PICK YOU UP.\nLOOKING SHARP.\nUSE THE STAIRS, YOUR FATHER PAID GOOD MONEY FOR THOSE.\nSORRY. I'M EXCITED.\nHERE'S THE GRADUATE.\nWE'RE VERY PROUD OF YOU, SON.\nA PERFECT REPORT CARD, ALL B'S.\nVERY PROUD.\nMA! I GOT A THING GOING HERE.\nYOU GOT LINT ON YOUR FUXX.\nOW! THAT'S ME!\nWAVE TO US! WE'LL BE IN ROW 118,000.\nBYE!\nBARRY, I TOLD YOU, STOP FLYING IN THE HOUSE!\nHEY, ADAM.\nHEY, BARRY.\nIS THAT FUXX GEL?\nA LITTLE. SPECIAL DAY, GRADUATION.\nNEVER THOUGHT I'D MAKE IT.\nTHREE DAYS GRADE SCHOOL, THREE DAYS HIGH SCHOOL.\nTHOSE WERE AWKW"
```

Cifrado Vigenère

In [54]:
```python
def transform_plain(plain_text: str, key_word: str) -> str:
    """Transform plain text with a keyword using a substitution

    Args:
        plain_text (str): Plain text to be transformed
        key_word (str): Word to map plain text with

    Returns:
        str: Mapped text
    """

    key = list(key_word)

    if len(plain_text) == len(key):
        mapped_text = ''.join(key)
        return mapped_text

    else:

        diff = len(plain_text) - len(key)

        if diff > 0:
            for i in range(len(plain_text) - len(key)):
                key.append(key[i % len(key)])
        else:
            print('Keyword is longer than plain text')
            return

    mapped_text = ''.join(key)

    return mapped_text
```

In [55]:
```python
def encrypt_vigenere(plain_text: str, key: str) -> str:
    """Vigenère encryption with a given keyword

    Args:
```

```
        plain_text (str): Plain text to be encrypted
        key (str): Transformed text with key

    Returns:
        str: Encrypted text
    """

    # Only working with uppercase letters
    alphabet = ascii_letters[len(ascii_letters) // 2:]
    cipher_text = []
    key_index = 0

    for char in plain_text:
        if char in alphabet:
            n = alphabet.find(char) + alphabet.find(key[key_index % len(key)])
            mod = n % len(alphabet)
            cipher_text.append(alphabet[mod])
            key_index += 1

        else:
            cipher_text.append(char)
            continue

    return ''.join(cipher_text)
```

In [56]:
```
def decrypt_vigenere(cipher_text: str, key_word: str) -> str:
    """Vigenère decryption with a given keyword

    Args:
        cipher_text (str): Ciphertext to be decrypted
        key_word (str): Keyword used for decryption

    Returns:
        str: Decrypted text
    """

    alphabet = ascii_letters[len(ascii_letters) // 2:]
    plain_text = []
    key_index = 0

    for char in cipher_text:
        if char in alphabet:
            n = alphabet.find(char) - alphabet.find(key_word[key_index % len(key_word)])
            mod = n % len(alphabet)
            plain_text.append(alphabet[mod])
            key_index += 1

        else:
            plain_text.append(char)
            continue

    return ''.join(plain_text)
```

In [57]:
```
def vigenere_test(case: int) -> str:
    """Vigenère encryption driver code

    Args:
        case (int): Integer representation of the case to be tested

    Returns:
        str: Report of the test
    """


    text = 'PERO MIRA COMO BEBEN LOS PECES EN EL RIO'

    if case == 1:
        key = 'M'
        key_word = transform_plain(text, key)
        encrypted = encrypt_vigenere(text, key_word)
        decrypted = decrypt_vigenere(encrypted, key_word)

    elif case == 0:
        key = 'MONTERREY'
        key_word = transform_plain(text, key)
        encrypted = encrypt_vigenere(text, key_word)
        decrypted = decrypt_vigenere(encrypted, key_word)

    else:
        print('Invalid case')
        return

    return f'''Vigenere Cipher
    Text: {text}
    Key: {key_word}
    Encrypted: {encrypted}
    Decrypted: {decrypted}'''
```

```
print(vigenere_test(1))
print(vigenere_test(0))
```

```
Vigenere Cipher
    Text: PERO MIRA COMO BEBEN LOS PECES EN EL RIO
    Key: MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
    Encrypted: BQDA YUDM OAYA NQNQZ XAE BQOQE QZ QX DUA
    Decrypted: PERO MIRA COMO BEBEN LOS PECES EN EL RIO
Vigenere Cipher
    Text: PERO MIRA COMO BEBEN LOS PECES EN EL RIO
    Key: MONTERREYMONTERREYMONTERREYMONTERREYMONT
    Encrypted: BSEH QZIE AAAB UISVR JAG CXGVJ IL QZ EBS
    Decrypted: PERO MIRA COMO BEBEN LOS PECES EN EL RIO
```