南京信息工程大学 实验（实习）报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 日期 |  | 得分 |  | 指导教师 |  |
| 班级 |  | 学号 |  | 姓名 |  |

单表密码实现与分析

1．实验目的：

1. 掌握单表密码加密算法；
2. 掌握单表密码常见的攻击方法。

2．实验内容：

1. 实现单表密码的加解密算法对英文文本进行加密；
2. 统计明文和密文文本字母出现概率，对单表密码进行攻击分析，用实际的数据来说明问题。

3．Experiment Content

[Code]

import string

import random

def generate\_cipher\_table():

"""

Generates a random substitution table for single table cipher.

Returns:

dict: A mapping of plaintext letters to ciphertext letters.

"""

alphabet = string.ascii\_lowercase # Generate the alphabet

shuffled = list(alphabet)

random.shuffle(shuffled) # Randomly shuffle the alphabet

return dict(zip(alphabet, shuffled))

def encrypt\_single\_table\_cipher\_with\_table(*plaintext*, *cipher\_table*):

"""

Encrypts the plaintext using a substitution cipher based on a cipher table.

Parameters:

plaintext (str): The text to be encrypted.

cipher\_table (dict): The substitution cipher table.

Returns:

str: The encrypted ciphertext.

"""

ciphertext = ""

for char in plaintext:

if char.isalpha():

is\_upper = char.isupper()

mapped\_char = cipher\_table[char.lower()] # Map lowercase character

ciphertext += mapped\_char.upper() if is\_upper else mapped\_char

else:

ciphertext += char # Non-alphabetic characters remain unchanged

return ciphertext

def decrypt\_single\_table\_cipher\_with\_table(*ciphertext*, *cipher\_table*):

"""

Decrypts the ciphertext using a substitution cipher based on a cipher table.

Parameters:

ciphertext (str): The text to be decrypted.

cipher\_table (dict): The substitution cipher table.

Returns:

str: The decrypted plaintext.

"""

reverse\_table = {v: k for k, v in cipher\_table.items()} # Reverse the cipher table

plaintext = ""

for char in ciphertext:

if char.isalpha():

is\_upper = char.isupper()

mapped\_char = reverse\_table[char.lower()] # Map using reverse table

plaintext += mapped\_char.upper() if is\_upper else mapped\_char

else:

plaintext += char # Non-alphabetic characters remain unchanged

return plaintext

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# User input

plaintext = input("Enter the plaintext: ")

# Generate cipher table

cipher\_table = generate\_cipher\_table()

print("Cipher Table:", cipher\_table)

# Encrypt the plaintext

encrypted\_text = encrypt\_single\_table\_cipher\_with\_table(plaintext, cipher\_table)

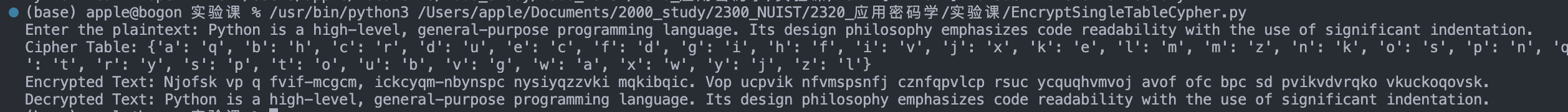
print("Encrypted Text:", encrypted\_text)

# Decrypt the ciphertext

decrypted\_text = decrypt\_single\_table\_cipher\_with\_table(encrypted\_text, cipher\_table)

print("Decrypted Text:", decrypted\_text)

[Output]

 The long string of garbled text above is the encrypted ciphertext, and below is the key book used for this encryption. When using this program for communication, both parties should each hold a copy of the key book to facilitate encryption and decryption. The key book should not be disclosed.

4．Experiment Analyzation and Summarization

[1]Plaintext Selection

In the code, a plaintext string is input by the user, representing the message to be encrypted. For better encryption analysis, the plaintext should ideally be long and meaningful, such as the first chapter of a story or an excerpt from a document.

[2]Cypher Table Generation

The single-table substitution cipher requires a cipher table, referred to as the “cipher book.” The cipher table is essentially a one-to-one mapping between the English alphabet’s 26 letters. Each letter in the plaintext is substituted with a corresponding letter from the cipher book.

• In this implementation, the cipher book is randomly generated using Python’s random.shuffle function.

• For instance, if the cipher table maps 'a' -> 'x', 'b' -> 'y', …, 'z' -> 'q', then plaintext letters are replaced according to this mapping.

• Both the ciphertext and the cipher table are output by the program to ensure transparency during the encryption process.

[3]Caesar Cypher Comparison

The Caesar cipher is a simple substitution cipher that shifts plaintext letters by a fixed number of positions (e.g., k positions). While straightforward, the Caesar cipher is vulnerable to brute-force attacks due to its small keyspace of size 26.

Example:

Input: ATTACKATSUNSET

Key: k = 3

Output: DWWDFNDWVXQVHW

By contrast, the single-table substitution cipher implemented here provides a significantly larger keyspace of 26! (over 10^26), making brute-force decryption infeasible.

[4]Decryption Challenges

Although the single-table substitution cipher resists brute-force attacks, it does not hide the statistical distribution of letters in the ciphertext.

• For example, the letter 'e' is the most frequent in English, often appearing in over 10% of text. A ciphertext with an unusually high frequency of a particular letter (e.g., 'q') strongly suggests a mapping to 'e'.

• Using frequency analysis, an attacker can deduce probable mappings and gradually decrypt the ciphertext.

Using the features of the encrypted messages, decrypting program can be built like this:

import matplotlib.pyplot as plt

from collections import Counter

def analyze\_message(*msg*):

"""

Analyzes the frequency of each letter in the given message.

Parameters:

msg (str): The message to be analyzed.

Returns:

dict: A dictionary with letters as keys and their frequencies as values.

"""

# Filter out only alphabetic characters and convert to lowercase

filtered\_msg = [char.lower() for char in msg if char.isalpha()]

# Count occurrences of each letter

letter\_count = Counter(filtered\_msg)

# Calculate total letters

total\_letters = sum(letter\_count.values())

# Calculate frequency for each letter

letter\_frequency = {char: count / total\_letters for char, count in letter\_count.items()}

return letter\_count, letter\_frequency

def plot\_letter\_frequency(*letter\_frequency*):

"""

Plots the frequency of letters using a bar chart.

Parameters:

letter\_frequency (dict): A dictionary with letters as keys and their frequencies as values.

"""

# Sort the frequency dictionary by letters

sorted\_letters = sorted(letter\_frequency.items(), *key*=lambda *x*: x[0])

letters, frequencies = zip(\*sorted\_letters)

# Plot the bar chart

plt.figure(*figsize*=(10, 6))

plt.bar(letters, frequencies, *color*='skyblue', *edgecolor*='black')

# Add titles and labels

plt.title('Letter Frequency in Message', *fontsize*=16)

plt.xlabel('Letters', *fontsize*=14)

plt.ylabel('Frequency', *fontsize*=14)

# Display grid and chart

plt.grid(*axis*='y', *linestyle*='--', *alpha*=0.7)

plt.xticks(*fontsize*=12)

plt.yticks(*fontsize*=12)

plt.show()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Input the message

msg = input("Enter the encrypted message: ")

# Analyze the message

letter\_count, letter\_frequency = analyze\_message(msg)

# Print results

print("\nLetter Count:")

for char, count in letter\_count.items():

print(f"{char}: {count}")

print("\nLetter Frequency:")

for char, freq in sorted(letter\_frequency.items(), *key*=lambda *x*: x[1], *reverse*=True):

print(f"{char}: {freq:.4f}")

# Plot the letter frequency

plot\_letter\_frequency(letter\_frequency)

**Code Composition**

1. **Dependencies:**

• matplotlib.pyplot: Used for visualizing data with charts (bar charts).

• collections.Counter: Quickly counts the occurrences of letters.

2. **Main Functions:**

• analyze\_message(msg)

• **Input**: A string message (plaintext or ciphertext).

• **Output**: Two results:

• **Letter Count (**letter\_count**)**: A dictionary that counts the occurrences of each letter.

• **Letter Frequency (**letter\_frequency**)**: A dictionary that calculates the frequency of each letter (count divided by the total number of letters).

• **Logic**:

• Filters out non-alphabet characters and converts the remaining characters to lowercase.

• Uses Counter to count the occurrences of each letter.

• Computes the frequency of each letter based on the total number of letters.

• plot\_letter\_frequency(letter\_frequency)

• **Input**: Letter frequency dictionary (letter\_frequency).

• **Function**: Plots the letter frequency data as a bar chart, showing the relative frequency of each letter.

• **Visualization Details**:

• X-axis: Letters (sorted alphabetically).

• Y-axis: Frequency of the letters.

• Bar chart style: Sky blue fill, black borders, with grid lines.

• Title and axis labels for clarity.

3. **Main Program:**

• The user provides a message via input (either ciphertext or plain text).

• Calls analyze\_message to analyze letter counts and frequencies.

• Prints the results for letter counts and frequencies.

• Calls plot\_letter\_frequency to generate the bar chart of letter frequencies.

**Functionality of the Code**

1. **Letter Counting:**

• Counts the occurrences of each letter in the input message.

• Ignores non-alphabetic characters (such as digits and punctuation marks).

• Converts all letters to lowercase, treating them as case-insensitive.

2. **Letter Frequency Analysis:**

• Calculates the relative frequency of each letter based on its count divided by the total number of letters.

• The output is sorted in descending order by frequency, making it easy to observe the most frequent letters.

3. **Data Visualization:**

• The letter frequencies are displayed in a bar chart, offering a clear visual representation of the letter distribution.

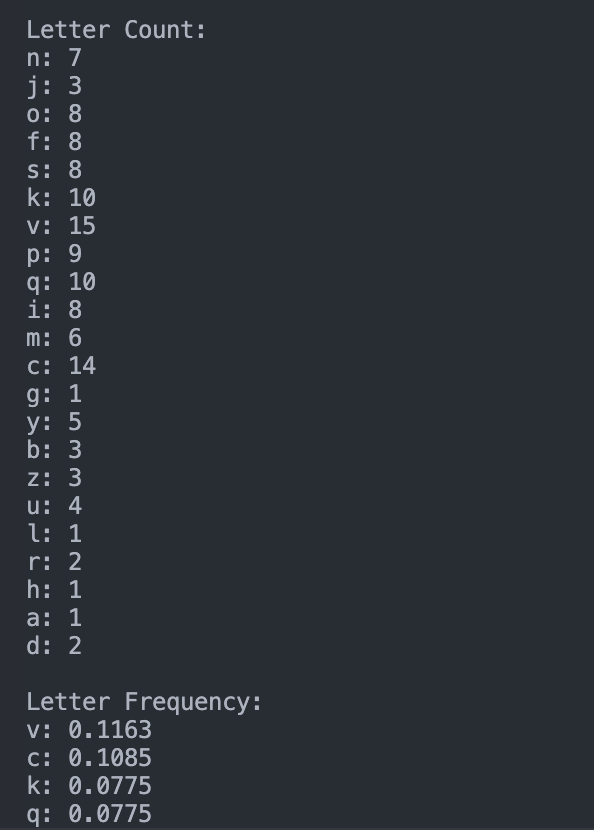
• The chart is designed to be visually appealing and easy to interpret, making it ideal for analyzing ciphertext characteristics.

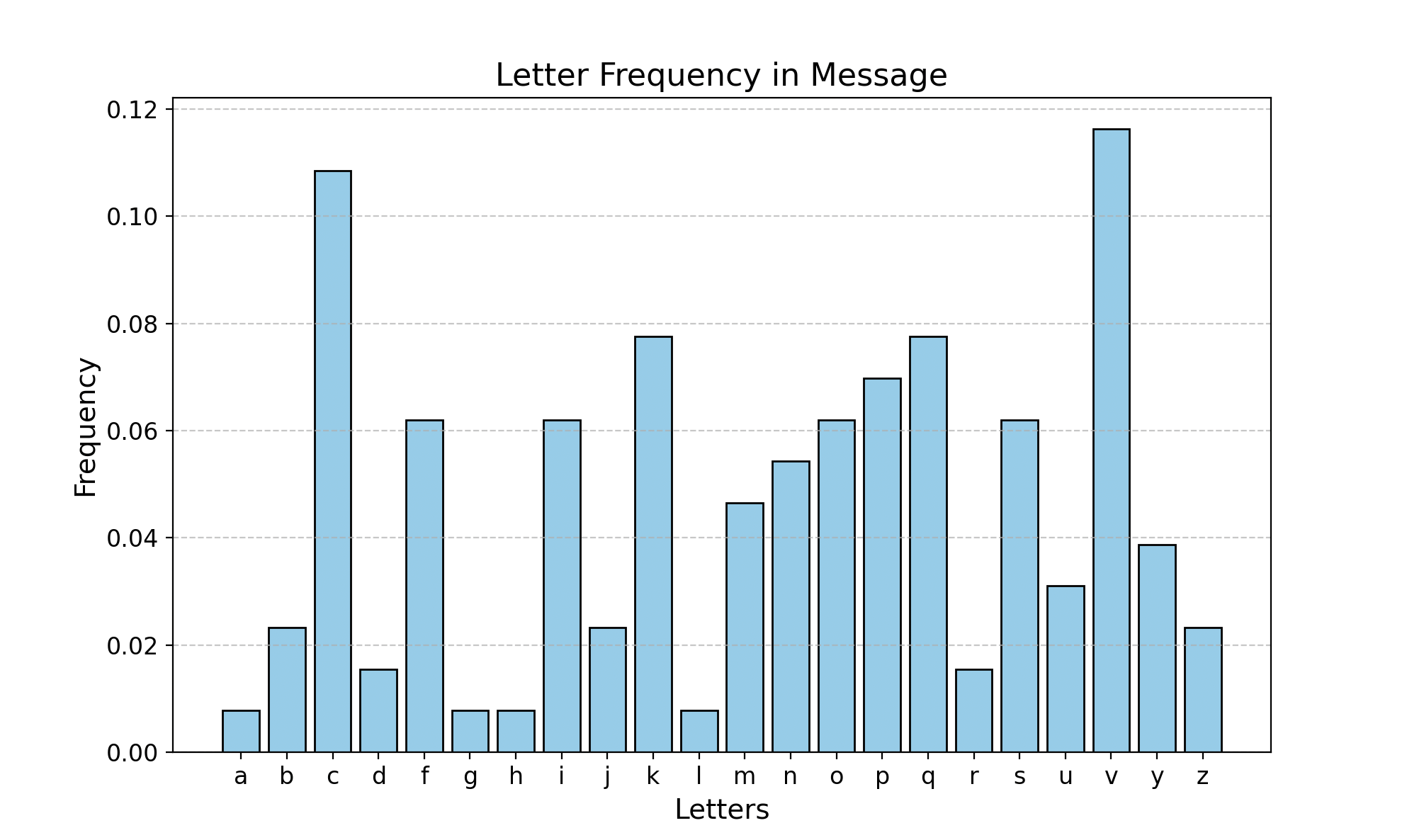
**Run Example**

Entering the encrypted message: Njofsk vp q fvif-mcgcm, ickcyqm-nbynspc nysiyqzzvki mqkibqic. Vop ucpvik nfvmspsnfj cznfqpvlcp rsuc ycquqhvmvoj avof ofc bpc sd pvikvdvrqko vkuckoqovsk.

Plaintext: Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.

Output:





The decryption approach based on the program output can be carried out through the following steps:

**1. Collect Letter Frequency Statistics**

First, the program will output the frequency of each letter in the ciphertext. These frequencies will help us determine which letters are the most common (usually, in English, the letter ‘e’ is the most frequent). By examining the frequency distribution of letters in the ciphertext, we can make educated guesses about the possible mappings of certain letters.

**2. Compare the Frequency Distribution with Known Language Letter Frequencies**

In English, there is a known frequency distribution of letters, with e, t, a, o, i being the most frequent. By comparing the ciphertext’s frequency with this known distribution, we can make initial assumptions. For example:

• If the letter q has the highest frequency in the ciphertext (e.g., 0.1183), and in English, e is the most common letter, we can guess that q might represent e.

• If the letter p also has a high frequency, it could represent t or a, depending on the commonality of those letters.

**3. Gradually Replace Letters**

Once some letter mappings are guessed, we can start replacing those letters. For example:

• If we assume q corresponds to e, we replace all occurrences of q with e.

• If we assume p corresponds to t, we replace all occurrences of p with t.

**4. Check the Validity of the Guesses**

After replacing some letters, we check whether the resulting ciphertext makes sense in English grammar. If the text starts to resemble English words and phrases, this indicates our guesses might be correct.

For example, if replacing q with e results in repeated occurrences of q in the ciphertext, it might form common words or word endings (like the, ate), which suggests the replacement is valid.

**5. Use Context to Further Replace Letters**

As the replacements progress, we can rely on context to infer the mappings of additional letters. For instance, if the ciphertext contains the word gix, we might guess that g is likely s, i is likely i, and x is likely x. This combination could lead us to further accurate substitutions based on common word structures.

**6. Adjust Guesses Iteratively**

As we continue the process, we can adjust guesses based on the structure of the decrypted message. If some letter substitutions do not immediately result in meaningful words, we can try alternative common letters.

**7. Final Decryption**

Through repeated substitutions and educated guessing, the letters of the ciphertext gradually reveal the plaintext message. This step-by-step process relies on frequency analysis and an understanding of language structure. Eventually, as more letters are mapped correctly, the entire ciphertext will be decoded into plaintext.

**Conclusion**

By analyzing letter frequencies and gradually replacing them, we can deduce the letter mappings in the ciphertext. This is a common method for breaking single-table substitution ciphers, especially when the keyspace is large (such as 26!). While brute force becomes infeasible, frequency analysis provides an effective decryption technique.