CHRIST
(DEEMED TO BE UNIVERSITY)
DELHI NCR · INDIA

School of Sciences

BDA202-3N: JAVA

B.Sc.(Data Science & Artificial Intelligence)

Sem III

Academic Year : 2024-25

BDA202-3N

Enhanced Caesar Cipher with GUI in Java

A Comprehensive Report on Encryption, Decryption, and Frequency Analysis using Graphical User Interface

Submitted to
Dr. Preety

Submitted by
John Anugrah Peter
23215206

Introduction

The **Caesar Cipher**, named after Julius Caesar, is one of the simplest and earliest forms of encryption techniques. It is a substitution cipher in which each letter in the plaintext is shifted by a certain number of positions down or up the alphabet. While it was used historically by Roman military leaders for secure communication, the Caesar Cipher is no longer considered secure due to its simplicity and vulnerability to brute-force attacks. Despite its limitations, it provides a foundation for understanding more complex encryption methods.

In this report, we delve into an Enhanced Caesar Cipher Program implemented in Java, incorporating both text-based and graphical user interfaces (GUI). The project aims to provide users with a more interactive experience when encrypting and decrypting messages. A Caesar Cipher is a classic encryption technique that shifts letters in a message by a specified number. The enhanced version supports uppercase, lowercase, numbers, custom shift values, and features a robust GUI, making the encryption process user-friendly and visually appealing.

The primary goals of the project include:

- Implementing a Caesar Cipher with support for custom shifts, handling letters and numbers.
- Introducing frequency analysis for cracking encrypted text.
- Developing a user-friendly GUI to allow users to encrypt, decrypt, and analyze text effortlessly.
- Providing a visual interface for brute-force and frequency analysis methods to improve user experience.

Classes Used

The Enhanced Caesar Cipher Program is implemented in Java and consists of several key classes that handle different aspects of the encryption and decryption process. Below is a description of the main classes and their responsibilities within the project.

1. `EnhancedCaesarCipher` **Class**:
    - **Purpose**: This is the main class that controls the flow of the program. It is responsible for performing encryption, decryption, and cracking operations.
    - **Key Responsibilities**:
        - Encryption and decryption based on the shift value.
        - Handling dynamic shift calculations based on user-provided passwords.
        - Performing frequency analysis for breaking the cipher.
        - Managing the user interface (CLI or GUI) for input and output.

    1.1 Helper Methods:

    #### `1.1.1 calculateShiftFromPassword(String password)`:

    - This method calculates a shift value based on the ASCII values of the characters in the password entered by the user. This dynamic shift ensures that the encryption key changes based on the password provided.

    #### `1.1.2 findShiftByFrequency(String cipherText)`:

    - A crucial method for performing frequency analysis. It examines the frequency distribution of letters in the encrypted text to estimate the shift value used during encryption. This method is important for cracking ciphers without knowing the key in advance.

2. `XORCipher` **Class (Optional)**:

    - **Purpose**: This class implements an XOR-based encryption method. It is used to further obscure the shift key for enhanced security, ensuring that even if someone knows the shift value, they cannot easily decrypt the message without the XOR key.
    - **Key Responsibilities**:
        - Encrypting and decrypting the shift value using XOR encryption.
3. **Utility Methods**:
    - These methods handle specific tasks such as shifting letters in the alphabet, converting text to lowercase, handling custom shift ranges, and displaying results in a user-friendly format.

The program follows an object-oriented design where each class serves a specific purpose, making the code modular, maintainable, and easy to extend with additional features. The `EnhancedCaesarCipher` class acts as the backbone, calling the necessary methods for encryption, decryption, and analysis, while the helper methods support the program's functionality.

4. T**he project utilizes several Java classes, particularly `JFrame`, `JButton`, `JTextField`, `JTextArea`, and `JLabel` from the Swing library for GUI implementation. The main classes include:**

1. **EnhancedCaesarCipherGUI**: Manages the GUI components, handling user input and integrating encryption/decryption methods.
2. **EnhancedCaesarCipher**: Contains the core functionality of the cipher, such as encryption, decryption, and frequency analysis.
3. **Utilities**: Provides helper methods for handling custom shift values and calculations.

The GUI elements ensure users have a seamless and interactive experience with clear input fields, output displays, and responsive buttons.

Methods

The Enhanced Caesar Cipher program employs several methods to handle the various aspects of encryption, decryption, and cryptanalysis. Below is a detailed explanation of the key methods used in the program:

1. `encrypt(String plainText, int shift)`:
    - **Purpose**: Encrypts a given plaintext string by shifting each letter by the specified shift value.
    - **Parameters**:
        - `plainText`: The text to be encrypted (a string of letters).
        - `shift`: The number of positions each letter in the plaintext will be shifted.
    - **Description**:
        - This method iterates over each character in the plaintext, calculates the new character based on the shift value, and constructs the encrypted message.

```
public static String encrypt(String plainText, int shift) {
    StringBuilder cipherText = new StringBuilder();
    for (char ch : plainText.toCharArray()) {
        if (Character.isLetter(ch)) {
            char shiftedChar = (char) (ch + shift);
            cipherText.append(shiftedChar);
        } else {
            cipherText.append(ch); // Non-letter characters are
added as is
        }
    }
    return cipherText.toString();
}
```

2. `decrypt(String cipherText, int shift)`:

- **Purpose**: Decrypts a given cipher text by reversing the shift applied during encryption.
- **Parameters**:
  - `cipherText`: The encrypted message to be decrypted.
  - `shift`: The number of positions each letter will be shifted back.
- **Description**:
  - This method iterates over each character in the cipher text and shifts each character backwards by the shift value to recover the original message.

```
public static String decrypt(String cipherText, int shift) {
    StringBuilder originalText = new StringBuilder();
    for (char ch : cipherText.toCharArray()) {
        if (Character.isLetter(ch)) {
            char shiftedChar = (char) (ch - shift);
            originalText.append(shiftedChar);
        } else {
            originalText.append(ch); // Non-letter characters
are added as is
        }
    }
    return originalText.toString();
}
```

3. `findShiftByFrequency(String cipherText)`:

- **Purpose**: Performs frequency analysis on the cipher text to estimate the shift value used for encryption.
- **Parameters**:
  - `cipherText`: The encrypted message to be analyzed.
- **Description**:
  - This method analyzes the frequency distribution of characters in the cipher text. Based on the most frequent letter, the program estimates the likely shift by comparing it to the expected frequency of letters in the English language (or other languages, if applicable).

```
public static int findShiftByFrequency(String cipherText) {
    Map<Character, Integer> letterFrequency = new HashMap<>();
    for (char c : cipherText.toCharArray()) {
        if (Character.isLetter(c)) {
            letterFrequency.put(c,
letterFrequency.getOrDefault(c, 0) + 1);
        }
    }
    // Assuming most frequent letter is 'e' (in English)
    char mostFrequent =
Collections.max(letterFrequency.entrySet(),
Map.Entry.comparingByValue()).getKey();
    int shift = mostFrequent - 'e'; // Calculate shift based on
'e'
    return shift < 0 ? shift + 26 : shift; // Handle negative
shifts
}
```

4. `crackCipherBruteForce(String cipherText)`:

- **Purpose**: Attempts to decrypt the cipher text using all possible shifts (1 to 25) and displays the results.
- **Parameters**:
  - `cipherText`: The encrypted message to be cracked.
- **Description**:
  - This method performs a brute-force attack by trying every possible shift and displaying the decrypted output for each. It helps identify the correct shift if the user doesn't know it.

```
public static void crackCipherBruteForce(String cipherText) {
    for (int i = 1; i <= 25; i++) {
        String decryptedMessage = decrypt(cipherText, i);
        System.out.println("Shift " + i + ": " +
decryptedMessage);
    }
}
```

5. `handleCustomShift(int shift)`:

- **Purpose**: Handles shift values that are either too large or negative by converting them into valid values within the 0-25 range.
- **Parameters**:
  - `shift`: The shift value, which can be negative or greater than 25.
- **Description**:
  - This method ensures that the shift is within the valid range (0 to 25). For negative shifts, it wraps around by adding 26 until the shift is non-negative.

```
public static int handleCustomShift(int shift) { return (shift % 26 + 26) % 26; // Handles both positive and negative shifts }
```

Implementation Overview

The Enhanced Caesar Cipher Program was developed to improve upon the traditional Caesar Cipher by implementing several advanced features like dynamic shift values based on passwords, frequency analysis for cipher cracking, and more flexible input handling (including numbers and special characters). The implementation follows an object-oriented approach, where different methods work together to achieve the encryption and decryption tasks.

Program Flow

1. **User Input**:
    - The user is prompted to input a message and specify whether they want to encrypt or decrypt it.
    - If encryption is selected, the program asks for a shift value, either fixed or password-based.
    - If decryption is selected, the program either uses a known shift or attempts to crack the cipher through frequency analysis or brute-force methods.
2. **Encryption**:
    - The encryption process shifts each letter in the plaintext by the specified shift amount (either static or dynamic).
    - Special characters, spaces, and numbers remain unchanged in the encrypted output.
3. **Decryption**:
    - Decryption involves shifting each character back by the shift value used during encryption.
    - If the shift is unknown, the program uses frequency analysis or brute-force techniques to estimate and apply the correct shift.
4. **Cracking the Cipher**:
    - The brute-force method attempts all possible shifts (1-25), decrypting the text for each.
    - Frequency analysis compares the most frequent character in the encrypted message to the expected frequency of letters in the English language, estimating the shift value.

5. **GUI Implementation**

The GUI is implemented using Java Swing, providing a user-friendly interface that includes:

- **Input Fields**: For the user to enter plain text or cipher text.
- **Buttons**: For choosing encryption, decryption, brute-force cracking, or frequency analysis.
- **Output Area**: A text area that displays the results and any errors or messages.
- **Visual Feedback**: The GUI updates in real-time, providing a clear, responsive interface that enhances the overall user experience.

CODE

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.*;



public class EnhancedCaesarCipherGUI extends JFrame implements ActionListener {

    // English letter frequency (approximate)

    private static final double[] ENGLISH_FREQ = {

        8.167, 1.492, 2.782, 4.253, 12.702, 2.228, 2.015, 6.094,

        6.966, 0.153, 0.772, 4.025, 2.406, 6.749, 7.507, 1.929,

        0.095, 5.987, 6.327, 9.056, 2.758, 0.978, 2.360, 0.150,
```

```
        1.974, 0.074
    };

    // GUI components
    private JTextField messageField, shiftField;
    private JTextArea resultArea;
    private JButton encryptButton, decryptButton, bruteForceButton,
freqAnalysisButton;

    public EnhancedCaesarCipherGUI() {
        // Setting up the GUI
        setTitle("Enhanced Caesar Cipher Program");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(7, 1));

        panel.add(new JLabel("Enter your message:"));
        messageField = new JTextField();
        panel.add(messageField);

        panel.add(new JLabel("Enter shift value (leave empty for cracking):"));
```

```
        shiftField = new JTextField();
        panel.add(shiftField);

        encryptButton = new JButton("Encrypt");
        decryptButton = new JButton("Decrypt");
        bruteForceButton = new JButton("Crack Cipher (Brute-force)");
        freqAnalysisButton = new JButton("Crack Cipher (Frequency Analysis)");

        panel.add(encryptButton);
        panel.add(decryptButton);
        panel.add(bruteForceButton);
        panel.add(freqAnalysisButton);

        resultArea = new JTextArea();
        resultArea.setEditable(false);
        panel.add(new JScrollPane(resultArea));

        add(panel);

        // Adding action listeners
        encryptButton.addActionListener(this);
        decryptButton.addActionListener(this);
        bruteForceButton.addActionListener(this);
```

```
        freqAnalysisButton.addActionListener(this);
    }

    // Encrypt function
    public static String encrypt(String plainText, int shift) {
        StringBuilder cipherText = new StringBuilder();
        for (char letter : plainText.toCharArray()) {
            if (Character.isLetter(letter)) {
                char base = Character.isLowerCase(letter) ? 'a' : 'A';
                char encryptedLetter = (char) ((letter - base + shift + 26) % 26
+ base);
                cipherText.append(encryptedLetter);
            } else if (Character.isDigit(letter)) {
                char encryptedDigit = (char) ((letter - '0' + shift + 10) % 10 +
'0');
                cipherText.append(encryptedDigit);
            } else {
                cipherText.append(letter);
            }
        }
        return cipherText.toString();
    }
```

```java
// Decrypt function
public static String decrypt(String cipherText, int shift) {
    return encrypt(cipherText, -shift);
}

// Frequency Analysis for cracking the cipher
public static int findShiftByFrequency(String cipherText) {
    int probableShift = 0;
    double minChiSquared = Double.MAX_VALUE;

    for (int shift = 0; shift < 26; shift++) {
        double chiSquared = calculateChiSquared(cipherText, shift);
        if (chiSquared < minChiSquared) {
            minChiSquared = chiSquared;
            probableShift = shift;
        }
    }
    return probableShift;
}

// Chi-squared calculation for frequency analysis
private static double calculateChiSquared(String text, int shift) {
    int[] letterCount = new int[26];
```

```
        int totalLetters = 0;


        for (char c : text.toCharArray()) {
            if (Character.isLetter(c)) {
                c = Character.toLowerCase(c);
                letterCount[(c - 'a' - shift + 26) % 26]++;
                totalLetters++;
            }
        }


        double chiSquared = 0.0;
        for (int i = 0; i < 26; i++) {
            double expected = totalLetters * ENGLISH_FREQ[i] / 100;
            double observed = letterCount[i];
            chiSquared += Math.pow(observed - expected, 2) / expected;
        }


        return chiSquared;
    }


    // Brute-force cracking with output
    public static String crackCipherBruteForce(String cipherText) {
```

```
        StringBuilder result = new StringBuilder("Brute-force cracking
results:\n");

        for (int shift = 1; shift < 26; shift++) {

            String possiblePlainText = decrypt(cipherText, shift);

            result.append("Shift ").append(shift).append(":
").append(possiblePlainText).append("\n");

        }

        return result.toString();

    }



    @Override

    public void actionPerformed(ActionEvent e) {

        String message = messageField.getText();

        int shift = 0;

        if (!shiftField.getText().isEmpty()) {

            shift = Integer.parseInt(shiftField.getText()) % 36;

        }

        String result = "";



        try {

            if (e.getSource() == encryptButton) {

                result = "Encrypted message: " + encrypt(message, shift);

            } else if (e.getSource() == decryptButton) {
```

```
                result = "Decrypted message: " + decrypt(message, shift);
            } else if (e.getSource() == bruteForceButton) {
                result = crackCipherBruteForce(message);
            } else if (e.getSource() == freqAnalysisButton) {
                int predictedShift = findShiftByFrequency(message);
                String crackedMessage = decrypt(message, predictedShift);
                result = String.format("Predicted Shift: %d\nCracked message:
%s", predictedShift, crackedMessage);
            }
        } catch (Exception ex) {
            result = "Error: " + ex.getMessage();
        }

        resultArea.setText(result);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            EnhancedCaesarCipherGUI gui = new EnhancedCaesarCipherGUI();
            gui.setVisible(true);
        });
    }
```

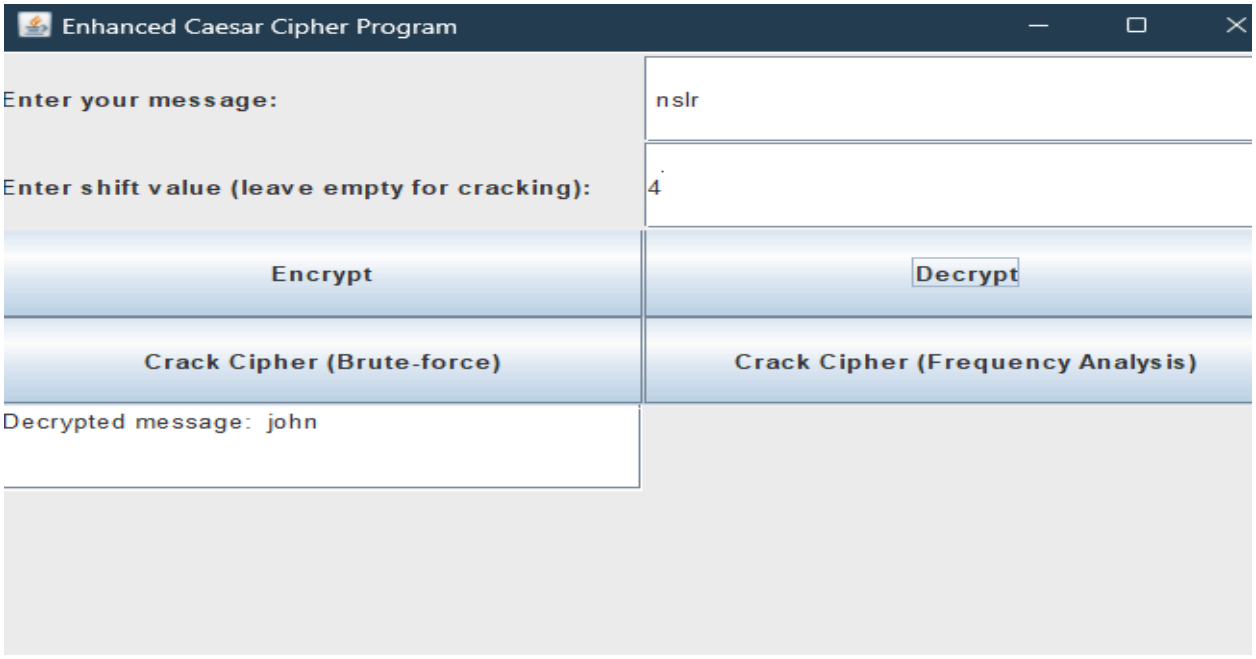
Complexity Analysis

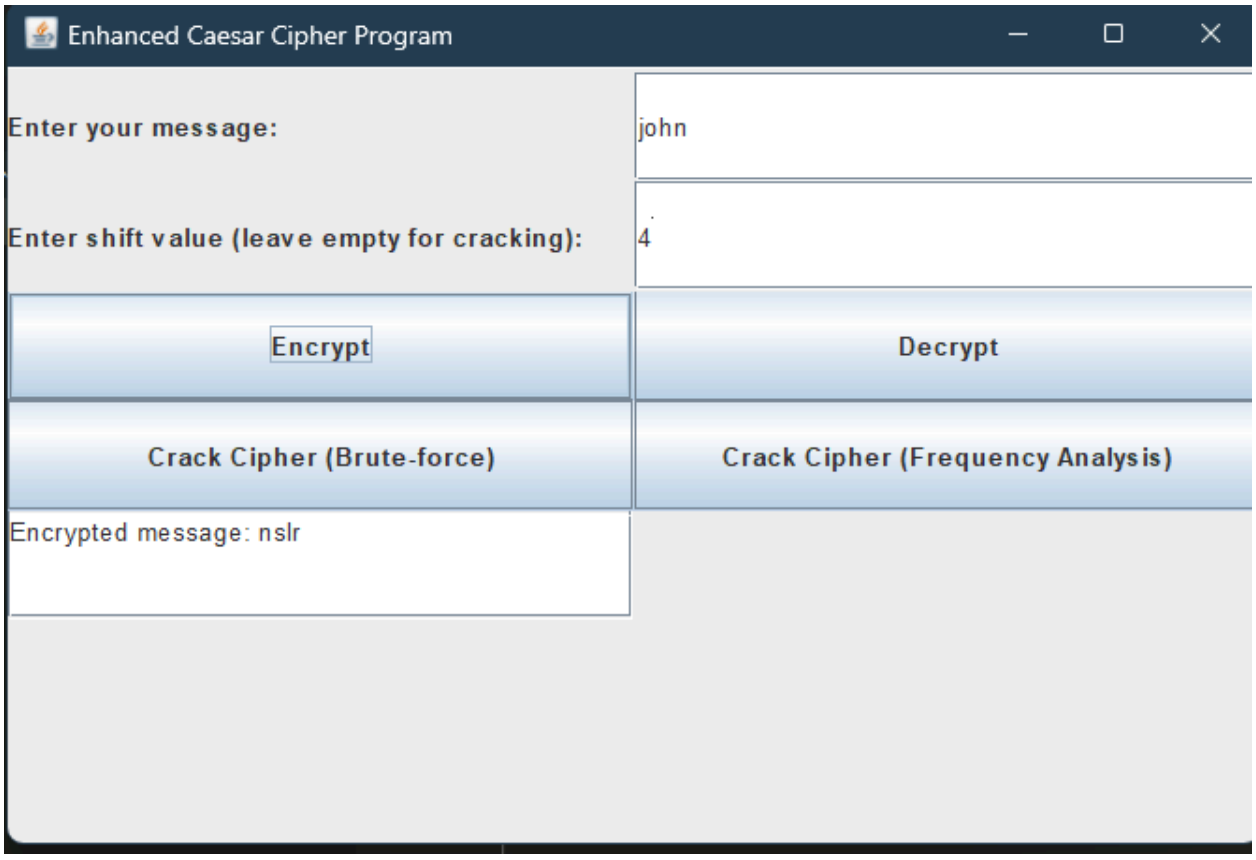
Time Complexity

The time complexity of the methods depends on the operations performed within each function. Below is an analysis of the time complexity for the key methods:

1. **Encryption and Decryption (`encrypt`, `decrypt`)**:
    - The program iterates over each character in the message, applying a shift. Therefore, the time complexity for both encryption and decryption is **O(n)**, where **n** is the length of the message.
    - This is efficient, as each character is processed once.
2. **Frequency Analysis (`findShiftByFrequency`)**:
    - The frequency analysis involves scanning through the cipher text and counting occurrences of each character, which is done in **O(n)** time. However, it also involves finding the maximum frequency in the letter map, which is an additional constant time operation.
    - Therefore, the overall time complexity is **O(n)**.
3. **Brute-Force Cracking (`crackCipherBruteForce`)**:
    - The brute-force method involves decrypting the cipher text with every possible shift (1-25). Since each decryption operation takes **O(n)** time (where **n** is the length of the cipher text), the time complexity for brute-force cracking is **O(25 * n)**, which simplifies to **O(n)**.
    - While this is linear, it may still take a considerable amount of time for large messages.
4. **Shift Calculation (`calculateShiftFromPassword`)**:
    - This method sums the ASCII values of each character in the password, which takes **O(m)** time, where **m** is the length of the password.
    - Since the shift value is computed modulo 26, the overall complexity remains **O(m)**.

OUTPUT


Enhanced Caesar Cipher Program
Enter your message:
nslr
Enter shift value (leave empty for cracking):
4
Encrypt
Decrypt
Crack Cipher (Brute-force)
Crack Cipher (Frequency Analysis)
Decrypted message: john


Enhanced Caesar Cipher Program
Enter your message:
john
Enter shift value (leave empty for cracking):
4
Encrypt
Decrypt
Crack Cipher (Brute-force)
Crack Cipher (Frequency Analysis)
Encrypted message: nslr

Conclusion

The Enhanced Caesar Cipher Program offers a robust solution for learning about encryption techniques. By incorporating dynamic shifts, frequency analysis, and brute-force cracking, the program not only teaches the fundamentals of cryptography but also enhances the security of the basic Caesar Cipher. It is a valuable tool for both educational and practical applications, demonstrating the importance of secure communication and encryption in the digital age.

This project provided a deep dive into encryption, helping me understand both the theoretical and practical aspects of cryptography. It also highlighted the challenges involved in creating a secure encryption system, especially in a world where computational power is rapidly increasing.