|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Information Security (116U01L602)** | **Semester:** | **VI** |
| **Date of Performance:** | **16 / 01/ 2025** | **DIV/ Batch No:** | **B2** |
| **Student Name:** | **Akshat** | **Roll No:** | **16010122221** |

|  |
| --- |
| **Title:Encryption-Decryption programs using classical cryptography**  **(Playfair cipher, Transposition cipher)** |

|  |
| --- |
| **Objectives:** |
| To write a program to convert plain text into cipher text using Caesar cipher and Transposition cipher |

|  |
| --- |
| **Expected Outcome of Experiment:** |
| **CO1:Explain various security goals, threats, vulnerabilities and controls** |

|  |
| --- |
| **Pre Lab/ Prior Concepts:** |
| Basic understanding of cryptography  Knowledge of ASCII character encoding  Understanding of modular arithmetic  Basic programming concepts (arrays, strings, matrices) |

|  |
| --- |
| **New Concepts to be learned:** |
| Substitution Ciphers:   * Monoalphabetic ciphers * Polyalphabetic ciphers * Additive/Caesar cipher * Shift ciphers   Transposition Ciphers:   * Keyless transposition methods * Keyed transposition methods * Rail fence cipher * Column transposition   Breaking Ciphers:   * Brute force attacks * Cryptanalysis techniques * Breaking shift ciphers * Breaking monoalphabetic ciphers |

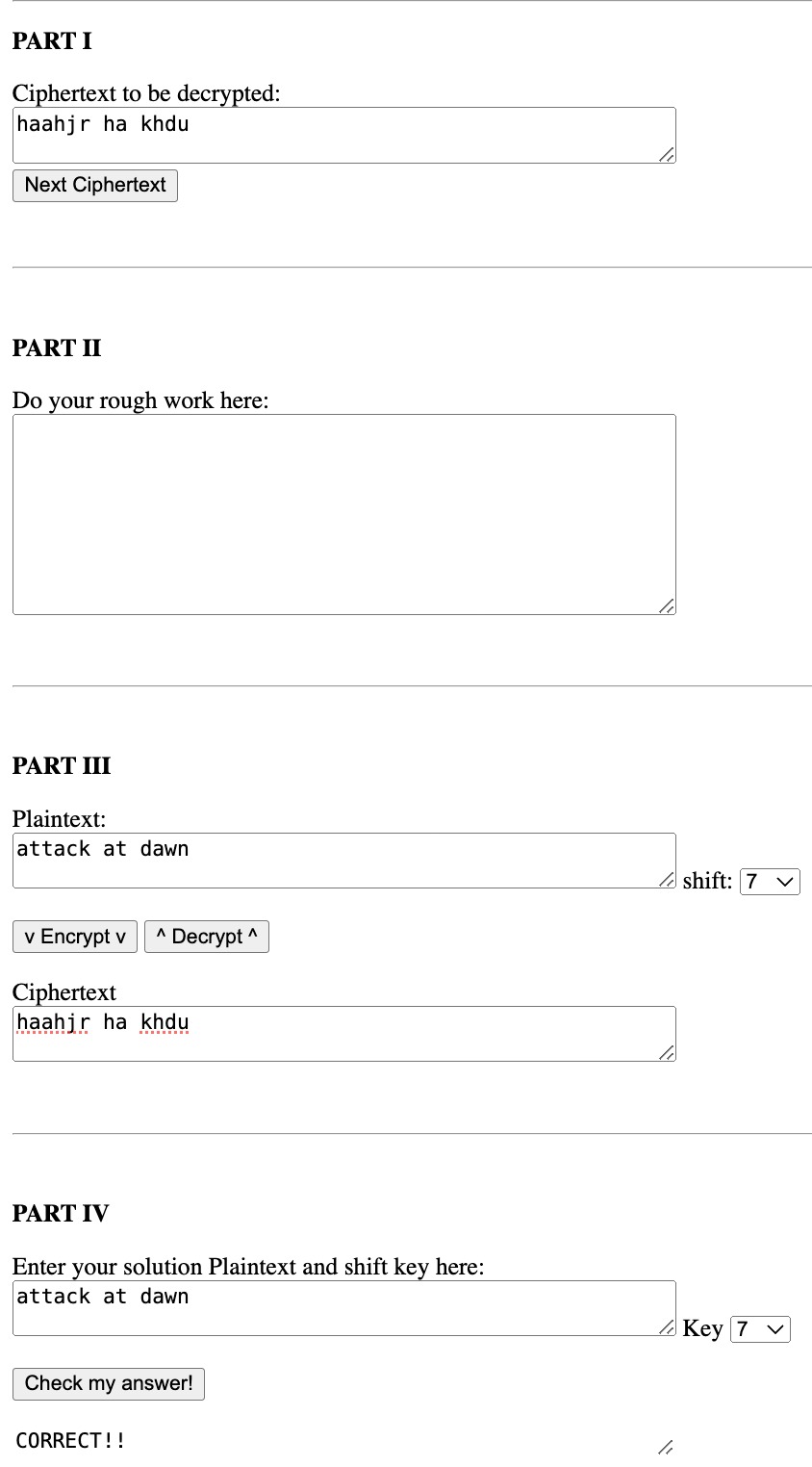
|  |
| --- |
| **Abstract:** |
| **Laboratory Experiment: Classical Cryptography**  This laboratory experiment delves into the realm of classical cryptography by focusing on the implementation of two significant ciphers: Playfair and Transposition.  **Objectives:**   * Develop and implement programs to encrypt and decrypt messages using both Playfair and Transposition ciphers. * Gain a practical understanding of key generation and message processing within the context of these historical ciphers. * Explore the fundamental principles of pattern-based encryption techniques. * Appreciate how these classical methods have contributed to the evolution of modern cryptographic systems.   **Methodology:**   * **Playfair Cipher:**   + Students will create a 5x5 key matrix based on a given keyword.   + Implement algorithms to encrypt and decrypt messages by processing pairs of letters according to the rules of the Playfair cipher. * **Transposition Cipher:**   + Implement various transposition techniques, such as columnar transposition, row transposition, and others.   + Explore different keying mechanisms for controlling the rearrangement of characters.   **Expected Outcomes:**   * Functional programs for both Playfair and Transposition ciphers, demonstrating successful encryption and decryption. * A deeper understanding of the strengths and weaknesses of classical cryptographic techniques. * An appreciation for the historical significance of these ciphers in the development of modern cryptography.   This laboratory experiment provides a valuable hands-on experience in understanding the core concepts of cryptography and their historical evolution.  **Key Improvements:**   * **Clearer Structure:** The description is now divided into sections (Objectives, Methodology, Expected Outcomes) for better readability. * **Conciseness:** The language is more concise and focuses on the key aspects of the experiment. * **Emphasis on Learning:** The emphasis is shifted towards the learning outcomes and the practical skills students will acquire. * **Enhanced Clarity:** The description is more concise and easier to understand.   . |

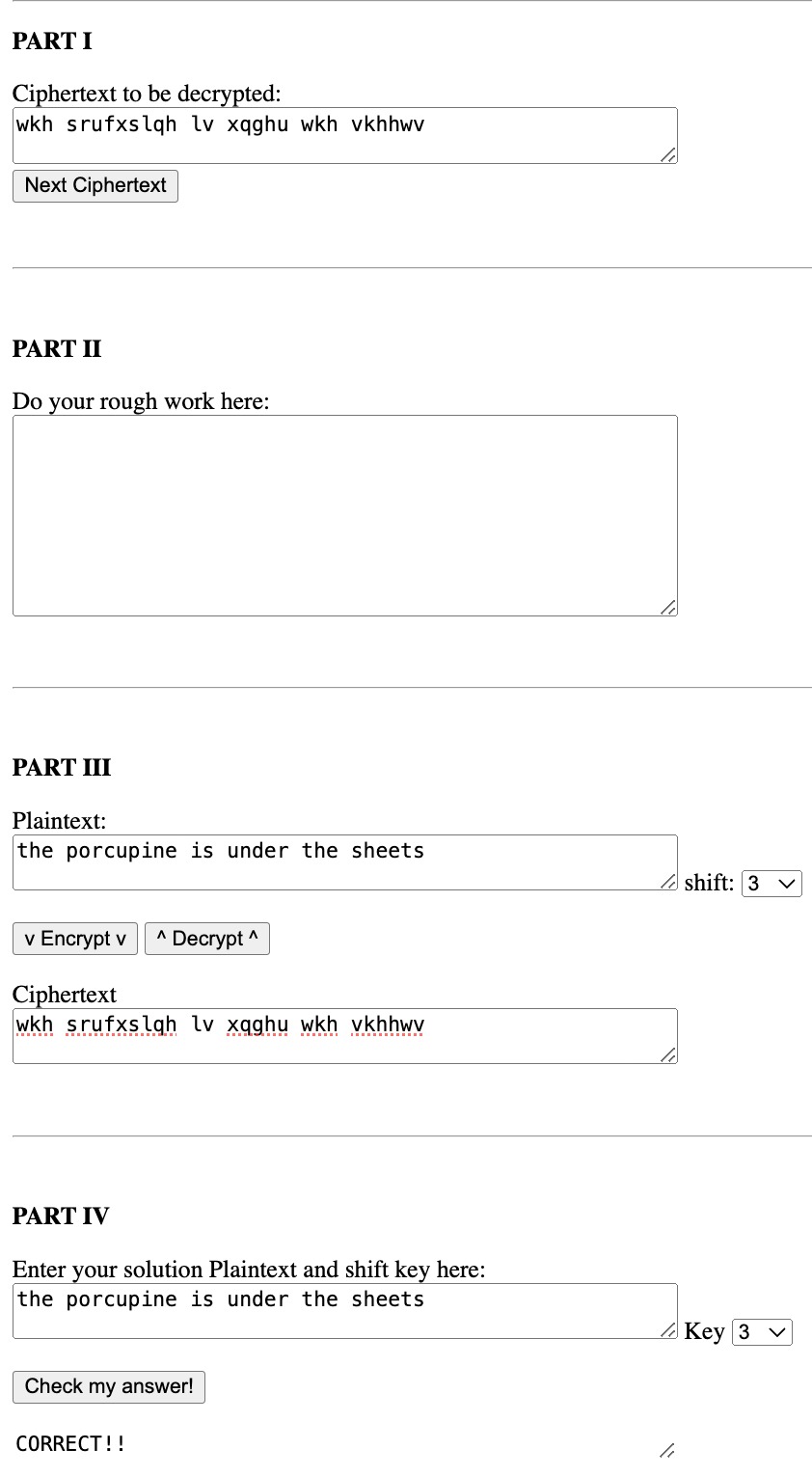
|  |
| --- |
| **Related Theory:** |
| The Playfair cipher, invented by Charles Wheatstone in 1854, operates using a 5x5 grid of letters constructed from a keyword, with each position containing a unique letter of the alphabet (I and J typically share a position). The encryption process involves breaking the plaintext into pairs of letters (digraphs) and applying specific rules based on their positions in the grid. If two letters fall in the same row, they are replaced by the letters to their right. If they appear in the same column, they are replaced by the letters below. For letters forming a rectangle, each is replaced by the letter at the opposite corner of the rectangle.  The Transposition cipher, unlike substitution ciphers, doesn't replace characters but instead rearranges them according to a specific pattern or key. This method maintains the original characters of the message but alters their positions to create the ciphertext. The most common form is the columnar transposition, where the plaintext is written in rows of fixed length, and the columns are then rearranged according to a key sequence. A simpler variant is the rail fence cipher, where text is written in a zigzag pattern across different "rails" and read off row by row.  OBJECTIVE: The primary objectives of this laboratory exercise are:   1. To implement and understand the mechanics of the Playfair cipher, including key matrix generation and digraph processing. 2. To develop programs for both columnar and rail fence transposition ciphers. 3. To analyze the strengths and weaknesses of these classical encryption methods. 4. To gain practical experience in cryptographic programming and algorithm implementation.   APPLICATIONS: While these classical ciphers are not suitable for modern security applications, their study provides valuable insights into:   1. The evolution of cryptographic systems 2. Basic principles of confusion and diffusion in encryption 3. The importance of key management and security 4. Fundamental concepts that underpin modern cryptographic algorithms |

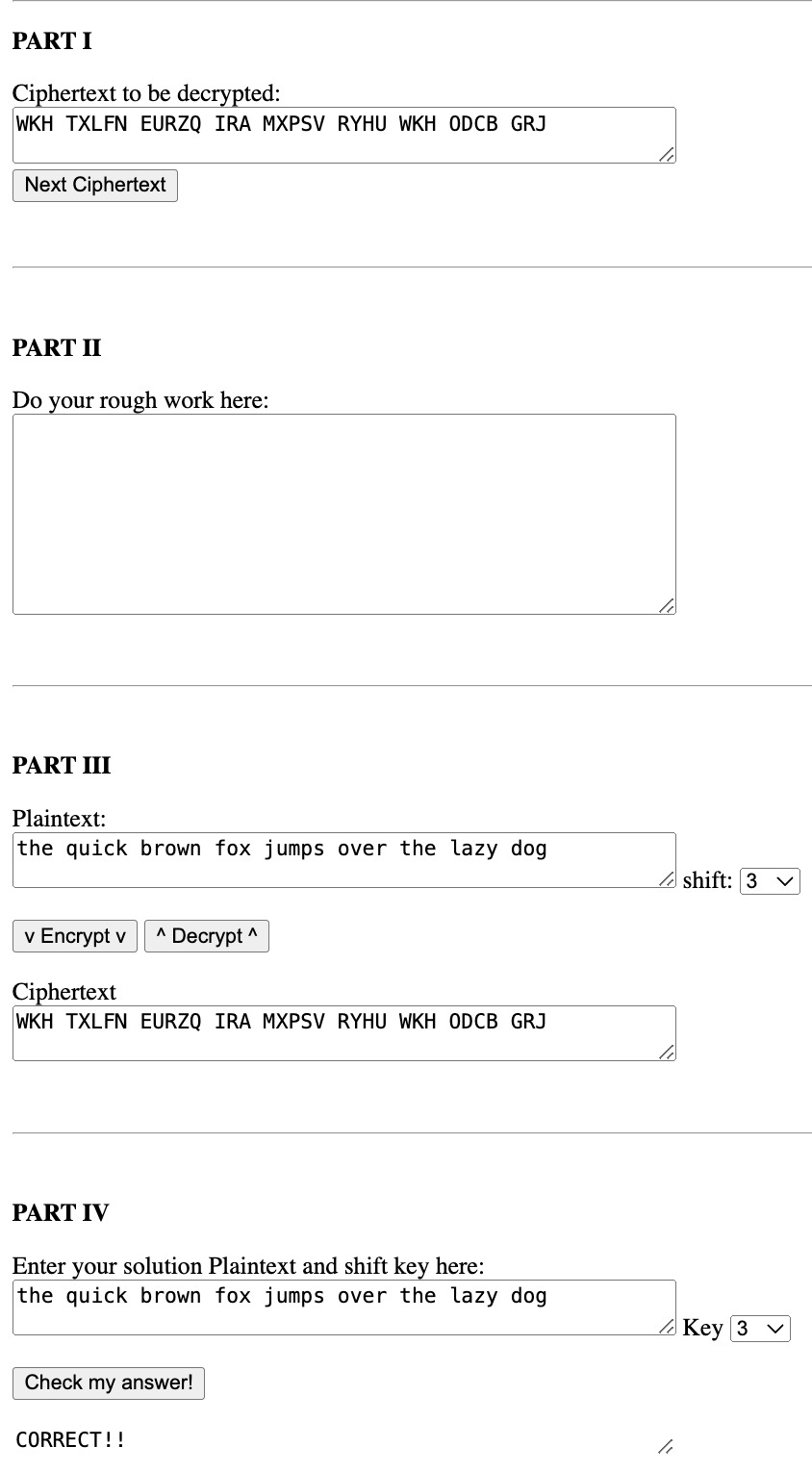
|  |
| --- |
| **Implementation Details:**  **Rail Fence** |
| #include <bits/stdc++.h>  using namespace std;  vector<int> **calculateRailLengths**(int messageLength, int rails) {      vector<int> **railLengths**(rails, 0);      int currentRail = 0;      bool goingDown = true;      for (int i = 0; i < messageLength; i++) {          railLengths**[**currentRail**]**++;          if (currentRail == 0) {              goingDown = true;          } else if (currentRail == rails - 1) {              goingDown = false;          }          currentRail = goingDown ? currentRail + 1 : currentRail - 1;      }      return railLengths;  }  string **encryptTransposition**(const string& message, int rails) {      int messageLength = message.**length**();      if (rails < 2 || rails >= messageLength) {          return "Invalid number of rails";      }      vector<string> **railContent**(rails, "");      int currentRail = 0;      bool goingDown = true;      for (char c : message) {          railContent**[**currentRail**]** **+=** c;          if (currentRail == 0) {              goingDown = true;          } else if (currentRail == rails - 1) {              goingDown = false;          }          currentRail = goingDown ? currentRail + 1 : currentRail - 1;      }      string encryptedMessage;      for (const string& rail : railContent) {          encryptedMessage **+=** rail;      }      return encryptedMessage;  }  string **decryptTransposition**(const string& encrypted, int rails) {      int messageLength = encrypted.**length**();      if (rails < 2 || rails >= messageLength) {          return "Invalid number of rails";      }      vector<int> railLengths = **calculateRailLengths**(messageLength, rails);      vector<string> **railContent**(rails);      int currentIndex = 0;      for (int i = 0; i < rails; i++) {          railContent**[**i**]** **=** encrypted.**substr**(currentIndex, railLengths**[**i**]**);          currentIndex += railLengths**[**i**]**;      }      string decrypted;      vector<int> **railIndices**(rails, 0);      currentIndex = 0;      bool goingDown = true;      for (int i = 0; i < messageLength; i++) {          decrypted **+=** railContent**[**currentIndex**][**railIndices**[**currentIndex**]**++**]**;          if (currentIndex == 0) {              goingDown = true;          } else if (currentIndex == rails - 1) {              goingDown = false;          }          currentIndex = goingDown ? currentIndex + 1 : currentIndex - 1;      }      return decrypted;  }  void **displayRailPattern**(const string& message, int rails) {      vector<vector<char>> **pattern**(rails, **vector**<char>(message.**length**(), '.'));      int currentRail = 0;      bool goingDown = true;      for (int i = 0; i < message.**length**(); i++) {          pattern**[**currentRail**][**i**]** = message**[**i**]**;          if (currentRail == 0) {              goingDown = true;          } else if (currentRail == rails - 1) {              goingDown = false;          }          currentRail = goingDown ? currentRail + 1 : currentRail - 1;      }      cout **<<** "\nRail Pattern:\n";      for (const auto& rail : pattern) {          for (char c : rail) {              cout **<<** c **<<** ' ';          }          cout **<<** '\n';      }      cout **<<** '\n';  }  int **main**() {      string message;      int rails;      cout **<<** "Enter the message (no spaces): ";  **getline**(cin, message);      cout **<<** "Enter number of rails: ";      cin **>>** rails;      cout **<<** "\nOriginal message pattern:";  **displayRailPattern**(message, rails);      string encrypted = **encryptTransposition**(message, rails);      cout **<<** "Encrypted message: " **<<** encrypted **<<** **endl**;      string decrypted = **decryptTransposition**(encrypted, rails);      cout **<<** "Decrypted message: " **<<** decrypted **<<** **endl**;      return 0;  }  **Playfair**  #include <bits/stdc++.h>  using namespace std;  // Playfair Cipher  vector<vector<char>> matrix = {      {'A', 'B', 'C', 'D', 'E'},      {'F', 'G', 'H', 'I', 'K'},      {'L', 'M', 'N', 'O', 'P'},      {'Q', 'R', 'S', 'T', 'U'},      {'V', 'W', 'X', 'Y', 'Z'}  };  string **prepareText**(const string &text) {      string cleaned = "";      for (char c : text) {          if (**isalpha**(c)) {              cleaned += **toupper**(c);          }      }      for (char &c : cleaned) {          if (c == 'J') {              c = 'I';          }      }      if (cleaned.**size**() % 2 != 0) {          cleaned += 'X';      }      return cleaned;  }  void **findPosition**(char c, int &row, int &col) {      for (int i = 0; i < 5; i++) {          for (int j = 0; j < 5; j++) {              if (matrix[i][j] == c) {                  row = i;                  col = j;                  return;              }          }      }  }  string **encryptPlayfair**(const string &plainText) {      string encryptedText = "";      string preparedText = **prepareText**(plainText);      for (size\_t i = 0; i < preparedText.**size**(); i += 2) {          char first = preparedText[i];          char second = preparedText[i + 1];          int row1, col1, row2, col2;  **findPosition**(first, row1, col1);  **findPosition**(second, row2, col2);          if (row1 == row2) {              encryptedText += matrix[row1][(col1 + 1) % 5];              encryptedText += matrix[row2][(col2 + 1) % 5];          } else if (col1 == col2) {              encryptedText += matrix[(row1 + 1) % 5][col1];              encryptedText += matrix[(row2 + 1) % 5][col2];          } else {              encryptedText += matrix[row1][col2];              encryptedText += matrix[row2][col1];          }      }      return encryptedText;  }  string **decryptPlayfair**(const string &cipherText) {      string decryptedText = "";      string preparedText = **prepareText**(cipherText);      for (size\_t i = 0; i < preparedText.**size**(); i += 2) {          char first = preparedText[i];          char second = preparedText[i + 1];          int row1, col1, row2, col2;  **findPosition**(first, row1, col1);  **findPosition**(second, row2, col2);          if (row1 == row2) {              decryptedText += matrix[row1][(col1 + 4) % 5];              decryptedText += matrix[row2][(col2 + 4) % 5];          } else if (col1 == col2) {              decryptedText += matrix[(row1 + 4) % 5][col1];              decryptedText += matrix[(row2 + 4) % 5][col2];          } else {              decryptedText += matrix[row1][col2];              decryptedText += matrix[row2][col1];          }      }      return decryptedText;  }  // Keyless Transposition Cipher  string **encryptTransposition**(string message) {      vector<int> space\_positions;      for (int i = 0; i < message.**length**(); i++) {          if (message[i] == ' ') {              space\_positions.**push\_back**(i);          }      }      message.**erase**(**remove**(message.**begin**(), message.**end**(), ' '), message.**end**());      int n = **ceil**(**sqrt**(message.**size**()));      vector<vector<char>> **grid**(n, **vector**<char>(n, '-'));      int k = 0;      for (int i = 0; i < n && k < message.**size**(); i++) {          for (int j = 0; j < n && k < message.**size**(); j++) {              grid[i][j] = message[k++];          }      }      cout << "Grid representation:" << endl;      for (int i = 0; i < n; i++) {          for (int j = 0; j < n; j++) {              cout << grid[i][j] << " ";          }          cout << endl;      }      string encrypted = "";      for (int j = 0; j < n; j++) {          for (int i = 0; i < n; i++) {              if (grid[i][j] != '-') {                  encrypted += grid[i][j];              }          }      }      for (int pos : space\_positions) {          if (pos <= encrypted.**length**()) {              encrypted.**insert**(pos, " ");          }      }      return encrypted;  }  string **decryptTransposition**(string encrypted) {      vector<int> space\_positions;      for (int i = 0; i < encrypted.**length**(); i++) {          if (encrypted[i] == ' ') {              space\_positions.**push\_back**(i);          }      }      encrypted.**erase**(**remove**(encrypted.**begin**(), encrypted.**end**(), ' '), encrypted.**end**());      int len = encrypted.**size**();      int n = **ceil**(**sqrt**(len));      int complete\_cols = len - (n \* (n - 1));      vector<int> **col\_lengths**(n, n - 1);      for (int i = 0; i < complete\_cols; i++) {          col\_lengths[i] = n;      }      vector<vector<char>> **grid**(n, **vector**<char>(n, '-'));      int k = 0;      for (int j = 0; j < n; j++) {          for (int i = 0; i < col\_lengths[j]; i++) {              grid[i][j] = encrypted[k++];          }      }      string decrypted = "";      for (int i = 0; i < n; i++) {          for (int j = 0; j < n; j++) {              if (grid[i][j] != '-') {                  decrypted += grid[i][j];              }          }      }      for (int pos : space\_positions) {          if (pos <= decrypted.**length**()) {              decrypted.**insert**(pos, " ");          }      }      return decrypted;  }  int **main**() {      string text;      cout << "Enter the text: ";  **getline**(cin, text);      // Playfair Cipher      string encryptedPlayfair = **encryptPlayfair**(text);      cout << "Playfair Encrypted text: " << encryptedPlayfair << endl;      string decryptedPlayfair = **decryptPlayfair**(encryptedPlayfair);      cout << "Playfair Decrypted text: " << decryptedPlayfair << endl;      // Keyless Transposition Cipher      string encryptedTransposition = **encryptTransposition**(text);      cout << "\nTransposition Encrypted message: " << encryptedTransposition << endl;      string decryptedTransposition = **decryptTransposition**(encryptedTransposition);      cout << "Transposition Decrypted message: " << decryptedTransposition << endl;      return 0;  }  **Transportation Keyless**  #include <iostream>  #include <string>  #include <vector>  #include <algorithm>  #include <cmath>  using namespace std;  string **encryptTransposition**(string message) {      vector<int> space\_positions;      for (int i = 0; i < message.**length**(); i++) {          if (message[i] == ' ') {              space\_positions.**push\_back**(i);          }      }      message.**erase**(**remove**(message.**begin**(), message.**end**(), ' '), message.**end**());      int n = **ceil**(**sqrt**(message.**size**()));      vector<vector<char>> **grid**(n, **vector**<char>(n, '-'));      int k = 0;      for (int i = 0; i < n && k < message.**size**(); i++) {          for (int j = 0; j < n && k < message.**size**(); j++) {              grid[i][j] = message[k++];          }      }      cout << "Grid representation:" << endl;      for (int i = 0; i < n; i++) {          for (int j = 0; j < n; j++) {              cout << grid[i][j] << " ";          }          cout << endl;      }      string encrypted = "";      for (int j = 0; j < n; j++) {          for (int i = 0; i < n; i++) {              if (grid[i][j] != '-') {                  encrypted += grid[i][j];              }          }      }      for (int pos : space\_positions) {          if (pos <= encrypted.**length**()) {              encrypted.**insert**(pos, " ");          }      }      return encrypted;  }  string **decryptTransposition**(string encrypted) {      vector<int> space\_positions;      for (int i = 0; i < encrypted.**length**(); i++) {          if (encrypted[i] == ' ') {              space\_positions.**push\_back**(i);          }      }      encrypted.**erase**(**remove**(encrypted.**begin**(), encrypted.**end**(), ' '), encrypted.**end**());      int len = encrypted.**size**();      int n = **ceil**(**sqrt**(len));      int complete\_cols = len - (n \* (n - 1));      vector<int> **col\_lengths**(n, n - 1);      for (int i = 0; i < complete\_cols; i++) {          col\_lengths[i] = n;      }      vector<vector<char>> **grid**(n, **vector**<char>(n, '-'));      int k = 0;      for (int j = 0; j < n; j++) {          for (int i = 0; i < col\_lengths[j]; i++) {              grid[i][j] = encrypted[k++];          }      }      string decrypted = "";      for (int i = 0; i < n; i++) {          for (int j = 0; j < n; j++) {              if (grid[i][j] != '-') {                  decrypted += grid[i][j];              }          }      }      for (int pos : space\_positions) {          if (pos <= decrypted.**length**()) {              decrypted.**insert**(pos, " ");          }      }      return decrypted;  }  int **main**() {      string message;      cout << "Enter message to encrypt: ";  **getline**(cin, message);      string encrypted = **encryptTransposition**(message);      cout << "\nEncrypted message: " << encrypted << endl;      string decrypted = **decryptTransposition**(encrypted);      cout << "Decrypted message: " << decrypted << endl;      return 0;  }  **Transposition keyed**  #include <iostream>  #include <string>  #include <vector>  #include <algorithm>  #include <unordered\_map>  using namespace std;  vector<int> **getKeyPos**(const string &key) {      vector<pair<char, int>> charPos;      for (size\_t i = 0; i < key.**size**(); ++i) {          charPos.**emplace\_back**(key**[**i**]**, i + 1);      }  **sort**(charPos.**begin**(), charPos.**end**());      unordered\_map<int, int> posMap;      for (size\_t newPos = 0; newPos < charPos.**size**(); ++newPos) {          posMap**[**charPos**[**newPos**]**.second**]** = newPos + 1;      }      vector<int> **keyPos**(key.**size**());      for (size\_t i = 0; i < key.**size**(); ++i) {          keyPos**[**i**]** = posMap**[**i + 1**]**;      }      return keyPos;  }  string **encrypt**(const string &message, const string &key, vector<int> &spacePos) {      string cleanMessage;      for (size\_t i = 0; i < message.**size**(); ++i) {          if (message**[**i**]** == ' ') {              spacePos.**push\_back**(i);          } else {              cleanMessage **+=** **toupper**(message**[**i**]**);          }      }      vector<int> keyPos = **getKeyPos**(key);      size\_t keyLen = key.**size**();      while (cleanMessage.**size**() % keyLen != 0) {          cleanMessage **+=** 'X';      }      vector<string> blocks;      for (size\_t i = 0; i < cleanMessage.**size**(); i += keyLen) {          blocks.**push\_back**(cleanMessage.**substr**(i, keyLen));      }      string encryptedText;      for (const string &block : blocks) {          unordered\_map<int, char> charMap;          for (size\_t i = 0; i < block.**size**(); ++i) {              charMap**[**keyPos**[**i**]]** = block**[**i**]**;          }          for (size\_t i = 1; i <= keyLen; ++i) {              encryptedText **+=** charMap**[**i**]**;          }      }      return encryptedText;  }  string **decrypt**(const string &cipher, const string &key, const vector<int> &spacePos) {      vector<int> keyPos = **getKeyPos**(key);      size\_t keyLen = key.**size**();      vector<string> blocks;      for (size\_t i = 0; i < cipher.**size**(); i += keyLen) {          blocks.**push\_back**(cipher.**substr**(i, keyLen));      }      string decryptedText;      for (const string &block : blocks) {          unordered\_map<int, char> posiMap;          for (size\_t i = 0; i < block.**size**(); ++i) {              posiMap**[**i + 1**]** = block**[**i**]**;          }          string **decryptb**(keyLen, ' ');          for (size\_t i = 0; i < keyLen; ++i) {              decryptb**[**keyPos**[**i**]** - 1**]** = posiMap**[**i + 1**]**;          }          decryptedText **+=** decryptb;      }      while (!decryptedText.**empty**() && decryptedText.**back**() == 'X') {          decryptedText.**pop\_back**();      }      for (int pos : spacePos) {          if (pos < static\_cast<int>(decryptedText.**size**())) {              decryptedText.**insert**(decryptedText.**begin**() **+** pos, ' ');          }      }      return decryptedText;  }  int **main**() {      string message, key;      cout **<<** "Enter the message to encrypt: ";  **getline**(cin, message);      cout **<<** "Enter the encryption key: ";      cin **>>** key;      vector<int> spacePos;      string encryptedText = **encrypt**(message, key, spacePos);      cout **<<** "\nEncrypted message: " **<<** encryptedText **<<** **endl**;      string decryptedText = **decrypt**(encryptedText, key, spacePos);      cout **<<** "Decrypted message: " **<<** decryptedText **<<** **endl**;      return 0;  } |

|  |
| --- |
| **Results/Output:** |
| Rail fence    Playfair    Transposition Keyless    Transposition keyed |

**VLAB**







|  |
| --- |
| **Conclusion:** |
| This experiment showcased the implementation of classical cryptographic algorithms like Playfair and Transposition ciphers, demonstrating their matrix- and position-based encryption techniques. It reinforced programming skills like string processing and matrix handling while providing a glimpse into historical cryptography. Though outdated for modern security, these methods lay a foundation for understanding the evolution of encryption and more advanced algorithms |

|  |
| --- |
| **Post-Lab Questions:** |
| **1.1 Write the points of difference between mono-alphabetic cipher and poly-alphabetic cipher.**   1. Character Mapping     * Mono-alphabetic: Each plaintext character maps to the same ciphertext character throughout the message.    * Poly-alphabetic: The mapping of a plaintext character changes depending on its position in the text. 2. Key Usage     * Mono-alphabetic: Relies on a single substitution pattern or key.    * Poly-alphabetic: Utilizes multiple substitution patterns or keys that vary with position. 3. Security Level     * Mono-alphabetic: Easier to break using frequency analysis.    * Poly-alphabetic: More secure as it disrupts frequency patterns with multiple mappings.   **1.2 Explain the working of a rail-fence cipher with the help of an example.**  To encrypt "HELLOWORLD" using a 3-rail fence:  Write it in a zigzag pattern:    For decryption, recreate the zigzag pattern, fill in the characters row by row, and then read off diagonally.We get HELLOWORLD  **1.3 Discuss any three applications of cryptography**  Secure Communication   * Email encryption * Messaging apps like WhatsApp * Military and diplomatic communications   Financial Security   * Protecting online banking transactions * Digital payment systems * Cryptocurrency operations   Data Protection   * Securing passwords * Encrypting files and databases * Digital signatures for authentication |