Classic Cipher Application - Project Report

Course: Network Security
Student Name: Onur Çakır
Student Number: 211401038
Date: November 5, 2025

Table of Contents

Project Overview

This project is a comprehensive console-based application that implements eight classic cryptographic ciphers. The application provides both encryption and decryption capabilities with an interactive graphical user interface (GUI) in the console, supporting both mouse and keyboard input.

Key Features:

- Interactive menu-driven interface with mouse and keyboard support
- Detailed step-by-step cipher operation logging for educational purposes
- Support for both encryption and decryption operations
- Error handling and input validation
- Visual feedback with colored console output

Technology Stack:

- **Language:** C# (.NET 7.0)
- **Platform:** Windows Console Application
- **API Integration:** Win32 Console API for advanced input handling

---

Implemented Ciphers

1. Caesar Cipher

Working Principle: The Caesar Cipher is one of the simplest and most widely known encryption techniques. It is a substitution cipher where each letter in the plaintext is shifted a fixed number of positions down the alphabet.

Mathematical Representation: - **Encryption:** $C_i = (P_i + k) \mod 26$ - **Decryption:** $P_i = (C_i - k) \mod 26$

Where: - P_i = plaintext character position (0-25) - C_i = ciphertext character position (0-25) - k = shift key (integer)

Example:

- **Plaintext:** HELLO
- **Key:** 3
- **Ciphertext:** KHOOR

Implementation Details: **Function Signature:**

```
public static string CaesarCipher(string text, int key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string CaesarCipher(string text, int key, bool encrypt)
{
    string mode = encrypt ? "Encrypt" : "Decrypt";
    Console.WriteLine($"[Caesar] Mode: {mode}");

    // Process text: uppercase + remove non-letters
    string processedText = new string(text.ToUpper().Where(char.IsLetter).ToArray());
    Console.WriteLine($"[Caesar] Processed Text: {processedText}");

    // Calculate shift
    int shift = encrypt ? key : (26 - key);
    shift = ((shift % 26) + 26) % 26;
    Console.WriteLine($"[Caesar] Key: {key} -> Final Shift: {shift}");
    Console.WriteLine("-----------------------------------");

    StringBuilder result = new StringBuilder();
    foreach (char c in processedText)
    {
        int index = c - 'A';
        int newIndex = (index + shift) % 26;
        char newChar = (char)('A' + newIndex);
        result.Append(newChar);

        // Detailed output
        Console.WriteLine($"[Caesar] Char: '{c}' (Index: {index})");
        Console.WriteLine($"         Calculation: ({index} + {shift}) % 26 = {newIndex
        Console.WriteLine($"         Result: Index {newIndex} -> '{newChar}'\n");
    }

    Console.WriteLine("-----------------------------------");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine($"[Caesar] Final Result: {result}");
    Console.ResetColor();

    return result.ToString();
}
```

Algorithm Breakdown: 1. **Input Processing:** Converts text to uppercase and filters only alphabetic characters 2. **Shift Calculation:** - For encryption: uses key directly - For decryption: uses (26 - key) to reverse the shift - Handles negative keys with modulo arithmetic 3. **Character Transformation:** - Converts each letter to index (A=0, B=1, …,

Z=25) - Applies shift: `newIndex = (index + shift) % 26` - Converts back to character: `newChar = 'A' + newIndex` 4. **Output:** Displays step-by-step calculation for each character

Key Features: - Supports any integer key value (positive or negative) - Automatic modulo normalization for keys > 26 - Real-time educational output showing each transformation - Color-coded final result (cyan)

---

2. Monoalphabetic Substitution Cipher

Working Principle: A monoalphabetic substitution cipher replaces each letter of the alphabet with another letter according to a fixed substitution table. Unlike Caesar cipher, the substitution is not a simple shift but a complete remapping of the alphabet.

Mathematical Representation: - Each letter has a unique one-to-one mapping - The key is a permutation of the 26 letters

Example:

- **Alphabet:** ABCDEFGHIJKLMNOPQRSTUVWXYZ
- **Key:** QWERTYUIOPASDFGHJKLZXCVBNM
- **Plaintext:** HELLO
- **Ciphertext:** ITSSG

Implementation Details: **Function Signature:**

```
public static string MonoalphabeticSubstitution(string text, string key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string MonoalphabeticSubstitution(string text, string key, bool encrypt)
{
    string mode = encrypt ? "Encrypt" : "Decrypt";
    Console.WriteLine($"[Monoalphabetic] Mode: {mode}");

    // Validate and process key
    string processedKey = key.ToUpper();
    if (processedKey.Length != 26 || processedKey.Distinct().Count() != 26)
    {
        throw new ArgumentException("Key must contain exactly 26 unique letters.");
    }

    string processedText = new string(text.ToUpper().Where(char.IsLetter).ToArray());
    Console.WriteLine($"[Monoalphabetic] Processed Text: {processedText}");
    Console.WriteLine($"[Monoalphabetic] Standard Alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ
    Console.WriteLine($"[Monoalphabetic] Key Alphabet:      {processedKey}");
    Console.WriteLine("-----------------------------------");

    // Build mapping
    string from = encrypt ? "ABCDEFGHIJKLMNOPQRSTUVWXYZ" : processedKey;
    string to   = encrypt ? processedKey : "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    StringBuilder result = new StringBuilder();
    foreach (char c in processedText)
    {
        int index = from.IndexOf(c);
        char newChar = to[index];
```

```
            result.Append(newChar);

            Console.WriteLine($"[Monoalphabetic] '{c}' (Index {index} in {(encrypt ? "Star
                              $"-> '{newChar}' (Index {index} in {(encrypt ? "Key" : "Stand
        }

        Console.WriteLine("-----------------------------------");
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[Monoalphabetic] Final Result: {result}");
        Console.ResetColor();

        return result.ToString();
    }
```

Algorithm Breakdown: 1. **Key Validation:** - Ensures key has exactly 26 characters - Verifies all characters are unique (no duplicates) - Throws exception if validation fails 2. **Alphabet Mapping:** - Standard alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ - Substitution alphabet: User-provided key - For encryption: standard → key - For decryption: key → standard 3. **Character Substitution:** - Finds character position in source alphabet - Maps to same position in target alphabet - Preserves character index across alphabets

Key Features: - Strict key validation (26 unique letters required) - Bidirectional mapping support - Index-based transformation display - Clear visualization of alphabet mappings

---

3. Vigenère Cipher

Working Principle: The Vigenère Cipher is a polyalphabetic substitution cipher that uses a keyword to determine multiple Caesar cipher shifts. Each letter in the key determines the shift for the corresponding plaintext letter, making it more secure than a simple Caesar cipher.

Mathematical Representation: - **Encryption:** $C_i = (P_i + K_{i \bmod m}) \bmod 26$ - **Decryption:** $P_i = (C_i - K_{i \bmod m} + 26) \bmod 26$

Where: - K_i = key character at position i - m = length of the key

Example:

- **Plaintext:** HELLO
- **Key:** KEY
- **Ciphertext:** RIJVS

Process: - H + K = R - E + E = I - L + Y = J - L + K = V - O + E = S

Implementation Details: **Function Signature:**

```
public static string VigenereCipher(string text, string key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string VigenereCipher(string text, string key, bool encrypt)
{
    string mode = encrypt ? "Encrypt" : "Decrypt";
    Console.WriteLine($"[Vigenere] Mode: {mode}");

    string processedText = new string(text.ToUpper().Where(char.IsLetter).ToArray());
    string processedKey = new string(key.ToUpper().Where(char.IsLetter).ToArray());

    if (processedKey.Length == 0)
```

```
            throw new ArgumentException("Key must contain at least one letter.");

        Console.WriteLine($"[Vigenere] Processed Text: {processedText}");
        Console.WriteLine($"[Vigenere] Processed Key: {processedKey}");
        Console.WriteLine("-----------------------------------");

        StringBuilder result = new StringBuilder();
        for (int i = 0; i < processedText.Length; i++)
        {
            char textChar = processedText[i];
            char keyChar = processedKey[i % processedKey.Length];

            int textIndex = textChar - 'A';
            int keyIndex = keyChar - 'A';

            int newIndex;
            if (encrypt)
                newIndex = (textIndex + keyIndex) % 26;
            else
                newIndex = (textIndex - keyIndex + 26) % 26;

            char newChar = (char)('A' + newIndex);
            result.Append(newChar);

            Console.WriteLine($"[Vigenere] Char: '{textChar}' (P={textIndex}), Key: '{keyC
            if (encrypt)
                Console.WriteLine($"         Encrypt: ({textIndex} + {keyIndex}) % 26 = {n
            else
                Console.WriteLine($"         Decrypt: ({textIndex} - {keyIndex} + 26) % 26
            Console.WriteLine($"         Result: Index {newIndex} -> '{newChar}'\n");
        }

        Console.WriteLine("-----------------------------------");
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[Vigenere] Final Result: {result}");
        Console.ResetColor();

        return result.ToString();
    }
```

Algorithm Breakdown: 1. **Key Repetition:** - Key repeats cyclically: `keyChar = processedKey[i % keyLength]` - Each position uses different Caesar shift 2. **Polyalphabetic Substitution:** - Each character encrypted with different shift value - Shift determined by corresponding key letter 3. **Encryption Formula:** `(P + K) mod 26` - P = plaintext character index - K = key character index 4. **Decryption Formula:** `(C - K + 26) mod 26` - Reverses the addition by subtraction - +26 ensures positive result before modulo

Key Features: - Automatic key repetition for any length text - Per-character shift visualization - Symmetric encrypt/decrypt operations - Handles variable-length keys efficiently

---

4. Columnar Transposition Cipher

Working Principle: The Transposition Cipher rearranges the positions of characters rather than substituting them. Text is written row-by-row into a grid, then read column-by-column according to a key that specifies the column order.

Process: 1. Write plaintext into a grid row by row 2. Read columns in the order specified by the key 3. For decryption, reverse the process

Example:

- **Plaintext:** HELLO WORLD
- **Key:** 3142 (read columns in order: 3rd, 1st, 4th, 2nd)

Grid:

```
H E L L
O W O R
L D X X  (padding)
```

Reading Order (3-1-4-2): - Column 3: L, O, X - Column 1: H, O, L - Column 4: L, R, X - Column 2: E, W, D

Ciphertext: LOXHOLLRXEWD

Implementation Details: **Function Signature:**

```
public static string TranspositionCipher(string text, string key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string TranspositionCipher(string text, string key, bool encrypt)
{
    string mode = encrypt ? "Encrypt" : "Decrypt";
    Console.WriteLine($"[Transposition] Mode: {mode}");

    string processedText = new string(text.ToUpper().Where(char.IsLetter).ToArray());
    Console.WriteLine($"[Transposition] Processed Text: {processedText}");

    // Validate key contains unique digits
    if (!key.All(char.IsDigit) || key.Distinct().Count() != key.Length)
        throw new ArgumentException("Key must contain unique digits.");

    int numCols = key.Length;
    int numRows = (int)Math.Ceiling((double)processedText.Length / numCols);

    Console.WriteLine($"[Transposition] Key Order: {key}");
    Console.WriteLine($"[Transposition] Grid Size: {numRows} Rows x {numCols} Columns'
    Console.WriteLine("-----------------------------------");

    if (encrypt)
    {
        // Pad text if necessary
        while (processedText.Length < numRows * numCols)
            processedText += 'X';

        // Fill matrix row by row
        char[,] matrix = new char[numRows, numCols];
        Console.WriteLine("[Transposition] Step 1: Filling matrix row by row...");
        for (int r = 0; r < numRows; r++)
        {
            StringBuilder rowStr = new StringBuilder();
            for (int c = 0; c < numCols; c++)
            {
                matrix[r, c] = processedText[r * numCols + c];
```

```
            rowStr.Append(matrix[r, c]);
        }
        Console.WriteLine($"         Row {r}: {rowStr}");
    }

    // Read columns in key order
    Console.WriteLine("[Transposition] Step 2: Reading matrix by key order...");
    StringBuilder result = new StringBuilder();
    foreach (char digitChar in key)
    {
        int colIndex = digitChar - '0';
        StringBuilder colStr = new StringBuilder();
        for (int r = 0; r < numRows; r++)
            colStr.Append(matrix[r, colIndex]);
        Console.WriteLine($"         Reading Column {colIndex} (Key '{digitChar}')
        result.Append(colStr);
    }

    Console.WriteLine("-----------------------------------");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine($"[Transposition] Final Result: {result}");
    Console.ResetColor();
    return result.ToString();
}
else
{
    // Decryption: reverse the process
    int baseColLen = processedText.Length / numCols;
    int extraCols = processedText.Length % numCols;

    char[,] matrix = new char[numRows, numCols];
    int textIndex = 0;

    Console.WriteLine("[Transposition] Step 1: Filling matrix by key column order
    foreach (char digitChar in key)
    {
        int colIndex = digitChar - '0';
        int colLen = (colIndex < extraCols) ? baseColLen + 1 : baseColLen;
        StringBuilder colStr = new StringBuilder();
        for (int r = 0; r < colLen; r++)
        {
            matrix[r, colIndex] = processedText[textIndex++];
            colStr.Append(matrix[r, colIndex]);
        }
        Console.WriteLine($"         Column {colIndex} (Key '{digitChar}'): {colSt
    }

    Console.WriteLine("[Transposition] Step 2: Reading matrix row by row...");
    StringBuilder result = new StringBuilder();
    for (int r = 0; r < numRows; r++)
    {
        StringBuilder rowStr = new StringBuilder();
        for (int c = 0; c < numCols; c++)
        {
```

```
                if (matrix[r, c] != '\0')
                {
                    result.Append(matrix[r, c]);
                    rowStr.Append(matrix[r, c]);
                }
            }
            Console.WriteLine($"         Row {r}: {rowStr}");
        }

        Console.WriteLine("-----------------------------------");
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[Transposition] Final Result: {result.ToString().TrimEnd('
        Console.ResetColor();
        return result.ToString().TrimEnd('X');
    }
}
```

Algorithm Breakdown: 1. **Grid Construction:** - Columns = key length - Rows = □text length / columns□ - Automatic padding with 'X' to fill grid 2. **Encryption Process:** - Fill matrix row-by-row with plaintext - Read columns in order specified by key digits - Concatenate column contents for ciphertext 3. **Decryption Process:** - Calculate proper column lengths (handles uneven distribution) - Fill columns in key order - Read matrix row-by-row to recover plaintext - Remove padding 'X' characters

Key Features: - Key must contain unique digits (e.g., "3142", "20134") - Visual matrix display for educational clarity - Handles variable text and key lengths - Smart padding removal during decryption

---

5. Base64 Encoding/Decoding

Working Principle: Base64 is a binary-to-text encoding scheme that represents binary data in an ASCII string format. It encodes data using 64 printable characters (A-Z, a-z, 0-9, +, /).

Process: 1. Convert text to UTF-8 bytes 2. Group bytes into 24-bit chunks (3 bytes) 3. Divide each 24-bit chunk into four 6-bit values 4. Map each 6-bit value to a Base64 character 5. Add padding ('=') if necessary

Mathematical Representation: - Each group of 3 bytes (24 bits) → 4 Base64 characters - Padding: 1 byte remaining → 2 chars + '==' - Padding: 2 bytes remaining → 3 chars + '='

Example:

- **Plaintext:** "Man"
- **UTF-8 Bytes:** [77, 97, 110]
- **Binary:** 01001101 01100001 01101110
- **6-bit Groups:** 010011 010110 000101 101110
- **Decimal:** [19, 22, 5, 46]
- **Base64:** TWFu

Implementation Details: **Function Signature:**

```
public static string Base64(string text, bool encode)
```

Complete Implementation from Program.cs:

```
public static string Base64(string text, bool encode)
{
    const string base64Alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy
```

```
if (encode)
{
    Console.WriteLine("[Base64] Mode: Encode");
    Console.WriteLine($"[Base64] Input: {text}");
    Console.WriteLine("----------------------------------");

    byte[] bytes = Encoding.UTF8.GetBytes(text);
    Console.WriteLine("[Base64] Step 1: Convert text to UTF-8 bytes.");
    for (int i = 0; i < bytes.Length; i++)
        Console.WriteLine($"         Char: '{text[i]}' -> Byte {bytes[i]} (Binary:

    Console.WriteLine("\n[Base64] Step 2: Process bytes in 3-byte (24-bit) chunks.
    StringBuilder result = new StringBuilder();

    for (int i = 0; i < bytes.Length; i += 3)
    {
        int b1 = bytes[i];
        int b2 = (i + 1 < bytes.Length) ? bytes[i + 1] : 0;
        int b3 = (i + 2 < bytes.Length) ? bytes[i + 2] : 0;

        int combined = (b1 << 16) | (b2 << 8) | b3;

        Console.WriteLine($"       Processing block {i / 3} (Bytes: {b1}, {b2}, {b
        Console.WriteLine($"         24-bit Group: {Convert.ToString(b1, 2).PadLef
                          $"{Convert.ToString(b2, 2).PadLeft(8, '0')} {Convert.ToStr

        char c1 = base64Alphabet[(combined >> 18) & 0x3F];
        char c2 = base64Alphabet[(combined >> 12) & 0x3F];
        char c3 = (i + 1 < bytes.Length) ? base64Alphabet[(combined >> 6) & 0x3F]
        char c4 = (i + 2 < bytes.Length) ? base64Alphabet[combined & 0x3F] : '=';

        Console.WriteLine($"         6-bit Chunks: {Convert.ToString((combined >>
                          $"{Convert.ToString((combined >> 12) & 0x3F, 2).PadLeft(6,
                          $"{Convert.ToString((combined >> 6) & 0x3F, 2).PadLeft(6,
                          $"{Convert.ToString(combined & 0x3F, 2).PadLeft(6, '0')}")
        Console.WriteLine($"         Indices:      {(combined >> 18) & 0x3F} {(com
                          $"{(combined >> 6) & 0x3F} {combined & 0x3F}");
        Console.WriteLine($"         Characters:   '{c1}' '{c2}' '{c3}' '{c4}'\n")

        result.Append(c1).Append(c2).Append(c3).Append(c4);
    }

    Console.WriteLine("----------------------------------");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine($"[Base64] Final Result: {result}");
    Console.ResetColor();
    return result.ToString();
}
else
{
    Console.WriteLine("[Base64] Mode: Decode");
    Console.WriteLine($"[Base64] Input: {text}");
    Console.WriteLine("----------------------------------");
```

```
        // Decoding logic (reverse process)
        text = text.Replace("=", "");
        List<byte> bytes = new List<byte>();

        for (int i = 0; i < text.Length; i += 4)
        {
            int val1 = base64Alphabet.IndexOf(text[i]);
            int val2 = (i + 1 < text.Length) ? base64Alphabet.IndexOf(text[i + 1]) : 0
            int val3 = (i + 2 < text.Length) ? base64Alphabet.IndexOf(text[i + 2]) : 0
            int val4 = (i + 3 < text.Length) ? base64Alphabet.IndexOf(text[i + 3]) : 0

            int combined = (val1 << 18) | (val2 << 12) | (val3 << 6) | val4;

            bytes.Add((byte)((combined >> 16) & 0xFF));
            if (i + 2 < text.Length) bytes.Add((byte)((combined >> 8) & 0xFF));
            if (i + 3 < text.Length) bytes.Add((byte)(combined & 0xFF));
        }

        string result = Encoding.UTF8.GetString(bytes.ToArray());
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[Base64] Final Result: {result}");
        Console.ResetColor();
        return result;
    }
}
```

Algorithm Breakdown: 1. **Byte Conversion:** Text → UTF-8 byte array 2. **24-bit Grouping:** Every 3 bytes combined into 24-bit integer 3. **6-bit Extraction:** - Split 24 bits into four 6-bit values - Each 6-bit value = index 0-63 4. **Character Mapping:** Map each 6-bit index to Base64 alphabet 5. **Padding:** Add '=' if input length not multiple of 3

Bit Operations: - `(combined >> 18) & 0x3F` → Extract bits 23-18 - `(combined >> 12) & 0x3F` → Extract bits 17-12 - `(combined >> 6) & 0x3F` → Extract bits 11-6 - `combined & 0x3F` → Extract bits 5-0

Key Features: - Manual implementation (no Convert.ToBase64String) - Complete binary visualization at each step - Educational display of bit shifting operations - Proper padding handling ('=' characters)

---

6. Playfair Cipher

Working Principle: The Playfair Cipher encrypts pairs of letters (digraphs) instead of single letters, using a 5×5 matrix of letters. It was widely used for tactical purposes by the military and is significantly harder to break than simple substitution ciphers.

Rules: 1. Create a 5×5 matrix using the key (I and J share a cell) 2. Split plaintext into digraphs (pairs of letters) 3. If both letters are the same, insert 'X' between them 4. Apply transformation rules: - **Same row:** Replace with letters to the right (wrap around) - **Same column:** Replace with letters below (wrap around) - **Rectangle:** Replace with letters on the same row but opposite corners

Example:

- **Key:** PLAYFAIR
- **Matrix:**

```
P L A Y F
I R E X M
```

```
P L E (deleted duplicate P)
B C D G H
K M N O Q
S T U V W
Z
```

- **Plaintext:** HELLO → HE LL OX (insert X between LL)
- **Transformation:** Apply rectangle/row/column rules

Implementation Details: **Function Signature:**

```
public static string PlayfairCipher(string text, string key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string PlayfairCipher(string text, string key, bool encrypt)
{
    string mode = encrypt ? "Encrypt" : "Decrypt";
    Console.WriteLine($"[Playfair] Mode: {mode}");

    // Process text and key (J -> I)
    string processedText = new string(text.ToUpper().Where(char.IsLetter).Select(c =>
    string processedKey = new string(key.ToUpper().Where(char.IsLetter).Select(c => c

    Console.WriteLine($"[Playfair] Processed Text (J->I): {processedText}");
    Console.WriteLine($"[Playfair] Processed Key (J->I): {processedKey}");

    // Build 5x5 matrix
    char[,] matrix = new char[5, 5];
    HashSet<char> used = new HashSet<char>();
    int row = 0, col = 0;

    // Fill from key
    foreach (char c in processedKey)
    {
        if (!used.Contains(c))
        {
            matrix[row, col] = c;
            used.Add(c);
            col++;
            if (col == 5) { col = 0; row++; }
        }
    }

    // Fill remaining alphabet
    for (char c = 'A'; c <= 'Z'; c++)
    {
        if (c != 'J' && !used.Contains(c))
        {
            matrix[row, col] = c;
            used.Add(c);
            col++;
            if (col == 5) { col = 0; row++; }
        }
    }
```

```
// Display matrix
Console.WriteLine("\n[Playfair] Step 1: Building 5x5 Matrix...");
for (int r = 0; r < 5; r++)
{
    Console.Write("         [");
    for (int c = 0; c < 5; c++)
        Console.Write(matrix[r, c] + (c < 4 ? " " : ""));
    Console.WriteLine("]");
}

// Prepare digraphs
Console.WriteLine("\n[Playfair] Step 2: Preparing Text Digraphs...");
List<string> digraphs = new List<string>();
for (int i = 0; i < processedText.Length; i += 2)
{
    char first = processedText[i];
    char second = (i + 1 < processedText.Length) ? processedText[i + 1] : 'X';

    if (first == second)
    {
        digraphs.Add($"{first}X");
        i--;
    }
    else
    {
        digraphs.Add($"{first}{second}");
    }
}
Console.WriteLine(string.Join(" ", digraphs));

// Apply cipher rules
Console.WriteLine("\n[Playfair] Step 3: Applying Cipher Rules...");
StringBuilder result = new StringBuilder();

foreach (string digraph in digraphs)
{
    char c1 = digraph[0];
    char c2 = digraph[1];

    // Find positions
    int r1 = 0, c1col = 0, r2 = 0, c2col = 0;
    for (int r = 0; r < 5; r++)
    {
        for (int c = 0; c < 5; c++)
        {
            if (matrix[r, c] == c1) { r1 = r; c1col = c; }
            if (matrix[r, c] == c2) { r2 = r; c2col = c; }
        }
    }

    char new1, new2;

    if (r1 == r2) // Same row
    {
```

```
            new1 = matrix[r1, (c1col + (encrypt ? 1 : 4)) % 5];
            new2 = matrix[r2, (c2col + (encrypt ? 1 : 4)) % 5];
            Console.WriteLine($"         Pair '{c1}{c2}' ({r1},{c1col}) ({r2},{c2col})
        }
        else if (c1col == c2col) // Same column
        {
            new1 = matrix[(r1 + (encrypt ? 1 : 4)) % 5, c1col];
            new2 = matrix[(r2 + (encrypt ? 1 : 4)) % 5, c2col];
            Console.WriteLine($"         Pair '{c1}{c2}' ({r1},{c1col}) ({r2},{c2col})
        }
        else // Rectangle
        {
            new1 = matrix[r1, c2col];
            new2 = matrix[r2, c1col];
            Console.WriteLine($"         Pair '{c1}{c2}' ({r1},{c1col}) ({r2},{c2col})
        }

        result.Append(new1).Append(new2);
    }

    Console.WriteLine("-----------------------------------");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine($"[Playfair] Final Result: {result}");
    Console.ResetColor();

    return result.ToString();
}
```

Algorithm Breakdown: 1. **Matrix Construction:** - J and I merged (5×5 grid from 26 letters) - Fill matrix with key first (no duplicates) - Complete with remaining alphabet in order 2. **Digraph Preparation:** - Split text into pairs of letters - Insert 'X' between duplicate letters - Pad with 'X' if odd length 3. **Cipher Rules:** - **Same Row:** Shift right (wrap around) - **Same Column:** Shift down (wrap around) - **Rectangle:** Swap corners on same row

Key Features: - Automatic J→I conversion - Visual 5×5 matrix display - Three distinct transformation rules - Digraph handling with duplicate detection - Position-based coordinate display

---

7. XOR Cipher

Working Principle: XOR (exclusive OR) cipher is a symmetric stream cipher that operates at the byte level. Each byte of plaintext is XORed with a corresponding byte from a repeating key. XOR has the property that $A \oplus K \oplus K = A$, making it self-inverse.

Mathematical Representation: - **Encryption:** $C_i = P_i \oplus K_{i \mod m}$ - **Decryption:** $P_i = C_i \oplus K_{i \mod m}$ (same operation!)

Where: - $\oplus$ = XOR operation - K_i = key byte at position i - m = key length

Truth Table:

A	B	A □ B
0	0	0
0	1	1
1	0	1

A	B	A □ B
1	1	0

Example:

- **Plaintext:** “HI”
- **Key:** “AB”
- **Process:**
  - H (72) □ A (65) = 9
  - I (73) □ B (66) = 11
- **Result:** Encoded to Base64 for safe display

Implementation Details: **Function Signature:**

```
public static string XorCipher(string text, string key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string XorCipher(string text, string key, bool encrypt)
{
    Console.WriteLine("[XOR] Mode: " + (encrypt ? "Encrypt (to Base64)" : "Decrypt (f
    Console.WriteLine($"[XOR] Key: {key}");

    byte[] textBytes;
    if (encrypt)
    {
        Console.WriteLine($"[XOR] Input (Plaintext): {text}");
        textBytes = Encoding.UTF8.GetBytes(text);
    }
    else
    {
        Console.WriteLine($"[XOR] Input (Base64 Ciphertext): {text}");
        try
        {
            textBytes = Convert.FromBase64String(text);
        }
        catch
        {
            throw new ArgumentException("Invalid Base64 input for decryption.");
        }
    }

    byte[] keyBytes = Encoding.UTF8.GetBytes(key);
    Console.WriteLine("[XOR] Step 1: Convert Text and Key to UTF-8 Bytes");

    // XOR operation
    Console.WriteLine("\n[XOR] Step 2: Applying repeating XOR operation...");
    byte[] resultBytes = new byte[textBytes.Length];

    for (int i = 0; i < textBytes.Length; i++)
    {
        byte textByte = textBytes[i];
        byte keyByte = keyBytes[i % keyBytes.Length];
        byte resultByte = (byte)(textByte ^ keyByte);
```

```
            resultBytes[i] = resultByte;

            Console.WriteLine($"         Index {i}: (Text) {textByte,3} [ {Convert.ToStrin
                              $"(Key) {keyByte,3} [ {Convert.ToString(keyByte, 2).PadLeft(8
                              $"(Result) {resultByte,3} [ {Convert.ToString(resultByte, 2)
        }

        string result;
        if (encrypt)
        {
            Console.WriteLine("\n[XOR] Step 3: Encode raw result bytes to Base64 string.")
            result = Convert.ToBase64String(resultBytes);
        }
        else
        {
            Console.WriteLine("\n[XOR] Step 3: Convert result bytes back to UTF-8 text.");
            result = Encoding.UTF8.GetString(resultBytes);
        }

        Console.WriteLine("-----------------------------------");
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[XOR] Final Result {(encrypt ? "(Base64)" : "(Plaintext)")}:
        Console.ResetColor();

        return result;
    }
```

Algorithm Breakdown: 1. **Input Processing:** - Encryption: UTF-8 text → byte array - Decryption: Base64 string → byte array 2. **XOR Operation:** - For each byte: `result = text[i] XOR key[i mod keyLength]` - Key repeats cyclically - XOR is self-inverse: A □ K □ K = A 3. **Output Encoding:** - Encryption: Byte array → Base64 (safe display) - Decryption: Byte array → UTF-8 text

XOR Properties: - **Self-inverse:** Same operation for encrypt/decrypt - **Bitwise operation:** Works at bit level - **Symmetric:** Same key for both operations

Key Features: - Complete binary visualization of XOR operations - Repeating key mechanism (stream cipher behavior) - Base64 encoding for safe result representation - Byte-level operation display with alignment - Shows truth table application in practice

---

8. Hill Cipher

Working Principle: The Hill Cipher is a polygraphic substitution cipher based on linear algebra. It uses matrix multiplication to encrypt blocks of letters, making it resistant to frequency analysis.

Mathematical Representation: - **Encryption:** $C = P \times K \mod 26$ - **Decryption:** $P = C \times K^{-1} \mod 26$

Where: - P = plaintext vector (2×1 for 2×2 matrix) - K = key matrix (2×2) - K^{-1} = inverse of key matrix modulo 26

Requirements: - Key matrix must be invertible (determinant must be coprime to 26) - Determinant cannot be 0, 2, 13, or any multiple of 2 or 13

Example (2×2):

- **Key:** GYBN
- **Key Matrix:**

```
K = | G  Y |   | 6  24 |
    | B  N | = | 1  13 |
```

- **Determinant:** $(6 \times 13) - (24 \times 1) = 78 - 24 = 54 \mod 26 = 2$ (not valid!)
- **Better Key:** "HILL" → works if det is coprime to 26

Implementation Details: Function Signature:

```
public static string HillCipher(string text, string key, bool encrypt)
```

Complete Implementation from Program.cs:

```
public static string HillCipher(string text, string key, bool encrypt)
{
    string mode = encrypt ? "Encrypt" : "Decrypt";
    Console.WriteLine($"[Hill] Mode: {mode}");

    string processedText = new string(text.ToUpper().Where(char.IsLetter).ToArray());
    string processedKey = new string(key.ToUpper().Where(char.IsLetter).ToArray());

    if (processedKey.Length != 4)
        throw new ArgumentException("Hill cipher requires a 4-letter key for 2x2 matri

    Console.WriteLine($"[Hill] Processed Text: {processedText}");
    Console.WriteLine($"[Hill] Processed Key: {processedKey}");

    // Build 2x2 key matrix
    int[,] keyMatrix = new int[2, 2];
    keyMatrix[0, 0] = processedKey[0] - 'A';
    keyMatrix[0, 1] = processedKey[1] - 'A';
    keyMatrix[1, 0] = processedKey[2] - 'A';
    keyMatrix[1, 1] = processedKey[3] - 'A';

    Console.WriteLine("\n[Hill] Step 1: Creating 2x2 Key Matrix (K)...");
    Console.WriteLine($"         K = | {keyMatrix[0, 0],2} {keyMatrix[0, 1],2} |");
    Console.WriteLine($"             | {keyMatrix[1, 0],2} {keyMatrix[1, 1],2} |");

    // Calculate determinant
    int det = (keyMatrix[0, 0] * keyMatrix[1, 1] - keyMatrix[0, 1] * keyMatrix[1, 0])
    if (det < 0) det += 26;

    Console.WriteLine("\n[Hill] Step 2: Calculating Determinant (det(K) mod 26)...");
    Console.WriteLine($"         det(K) = ({keyMatrix[0, 0]}*{keyMatrix[1, 1]}) - ({ke
                      $"{keyMatrix[0, 0] * keyMatrix[1, 1] - keyMatrix[0, 1] * keyMatri
    Console.WriteLine($"         det(K) mod 26 = {det}");

    // Check if determinant is coprime to 26
    int gcd = GCD(det, 26);
    if (gcd != 1)
        throw new ArgumentException($"Error: Invalid key. Determinant {det} is not inv
                                    $"(not coprime to 26). Try a different key.");

    // Calculate modular inverse of determinant
    int detInv = ModInverse(det, 26);
    Console.WriteLine($"         Modular Inverse of det(K) = {detInv}");
```

```
int[,] operationMatrix;
if (encrypt)
{
    operationMatrix = keyMatrix;
    Console.WriteLine("\n[Hill] Step 3: Using Key Matrix (K) for Encryption.");
}
else
{
    // Calculate inverse matrix for decryption
    int[,] adjugate = new int[2, 2];
    adjugate[0, 0] = keyMatrix[1, 1];
    adjugate[0, 1] = -keyMatrix[0, 1];
    adjugate[1, 0] = -keyMatrix[1, 0];
    adjugate[1, 1] = keyMatrix[0, 0];

    operationMatrix = new int[2, 2];
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
        {
            operationMatrix[i, j] = (detInv * adjugate[i, j]) % 26;
            if (operationMatrix[i, j] < 0) operationMatrix[i, j] += 26;
        }

    Console.WriteLine("\n[Hill] Step 3: Calculating Inverse Matrix (K^-1) for Decr
    Console.WriteLine($"         K^-1 = | {operationMatrix[0, 0],2} {operationMatr
    Console.WriteLine($"                | {operationMatrix[1, 0],2} {operationMatr
}

// Pad text if odd length
if (processedText.Length % 2 != 0)
{
    processedText += 'X';
    Console.WriteLine("\n[Hill] Step 4: Text length is odd, padding with 'X'.");
}
else
{
    Console.WriteLine("\n[Hill] Step 4: Text length is even, no padding needed.");
}

// Process text in 2-char blocks
Console.WriteLine("\n[Hill] Step 5: Processing text in 2-char blocks (P * K_op)...
StringBuilder result = new StringBuilder();

for (int i = 0; i < processedText.Length; i += 2)
{
    int p1 = processedText[i] - 'A';
    int p2 = processedText[i + 1] - 'A';

    int c1 = (operationMatrix[0, 0] * p1 + operationMatrix[0, 1] * p2) % 26;
    int c2 = (operationMatrix[1, 0] * p1 + operationMatrix[1, 1] * p2) % 26;

    if (c1 < 0) c1 += 26;
    if (c2 < 0) c2 += 26;
```

```
            char ch1 = (char)('A' + c1);
            char ch2 = (char)('A' + c2);

            result.Append(ch1).Append(ch2);

            Console.WriteLine($"         Block {i / 2} ('{processedText[i]}{processedText
                              $"[ {p1,2} {p2,2} ] * K_op = [ {c1,2} {c2,2} ] -> '{ch1}{ch2
        }

        Console.WriteLine("-----------------------------------");
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[Hill] Final Result: {result}");
        Console.ResetColor();

        return result.ToString();
    }

    // Helper method for GCD calculation
    private static int GCD(int a, int b)
    {
        while (b != 0)
        {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    // Helper method for modular inverse
    private static int ModInverse(int a, int m)
    {
        for (int x = 1; x < m; x++)
            if ((a * x) % m == 1)
                return x;
        return -1;
    }
```

Algorithm Breakdown: 1. **Key Matrix Construction:** - 4-letter key → 2×2 matrix - Each letter → number (A=0, B=1, …, Z=25) 2. **Determinant Validation:** - Calculate: det = (a·d - b·c) mod 26 - Check GCD(det, 26) = 1 (coprimality) - If not coprime, matrix not invertible 3. **Matrix Inverse (Decryption):** - Adjugate matrix: swap diagonal, negate off-diagonal - $K^{-1} = det^{-1} \cdot adj(K) \bmod 26$ 4. **Block Processing:** - Split text into 2-letter blocks - Matrix multiplication: $[c_1\ c_2] = [p_1\ p_2] \times K$

Mathematical Operations: - **Encryption:** $C = P \times K \bmod 26$ - **Decryption:** $P = C \times K^{-1} \bmod 26$ - **Coprimality Check:** GCD(det, 26) must equal 1 - **Modular Inverse:** Find x where $(det \cdot x) \equiv 1 \pmod{26}$

Key Features: - Complete determinant invertibility validation - GCD calculation for coprimality check - Modular inverse computation - Adjugate matrix calculation - Block-wise matrix multiplication - Automatic padding for odd-length text

---

Implementation Details

Programming Language and Framework

- **Language:** C# 10.0
- **Framework:** .NET 7.0
- **Project Type:** Console Application
- **Namespace:** ClassicCipherApp

Project Structure

```
cipher app/
├── Program.cs               # Main application file
├── cipher app.csproj        # Project configuration
├── start.bat               # Quick start script
└── bin/Debug/net7.0/       # Compiled executables
```

Educational Features

All cipher implementations include detailed console logging: 1. **Input Processing:** Shows original and processed text 2. **Key Information:** Displays key and its interpretation 3. **Step-by-Step Calculations:** Shows mathematical operations for each character 4. **Intermediate Values:** Displays indices, binary, and decimal values 5. **Final Result:** Highlights the output in color

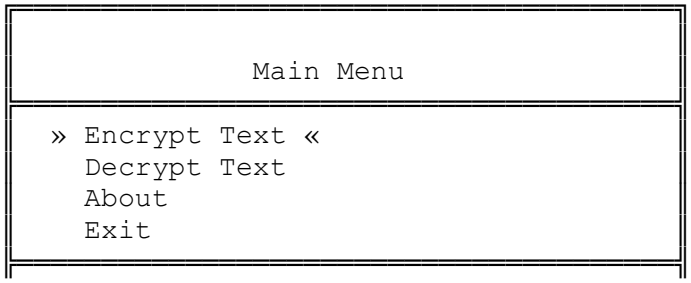
Example output format:

```
[Caesar] Mode: Encrypt
[Caesar] Processed Text: HELLO
[Caesar] Key: 3 -> Final Shift: 3
-----------------------------------
[Caesar] Char: 'H' (Index: 7)
         Calculation: (7 + 3) % 26 = 10
         Result: Index 10 -> 'K'

[Caesar] Char: 'E' (Index: 4)
         Calculation: (4 + 3) % 26 = 7
         Result: Index 7 -> 'H'
...
-----------------------------------
[Caesar] Final Result: KHOOR
```

User Interface

Main Menu

The application starts with a clean, centered main menu:



```
    Use ↑/↓, Click, or Esc
```

Navigation: - ↑/↓ **Arrow Keys:** Move selection up/down - **Mouse Hover:** Highlights option under cursor - **Left Click:** Selects option - **Enter:** Confirms selection - **Esc:** Goes back or exits

Cipher Selection Menu

After choosing Encrypt or Decrypt, user selects a cipher method:

```
        Select Cipher Method
    Caesar Cipher
    Monoalphabetic Substitution
    Vigenere Cipher
    Columnar Transposition
  » Base64 (Encode/Decode) «
    Playfair Cipher
    XOR Cipher
    Hill Cipher (2x2)
    Use ↑/↓, Click, or Esc
```

Input Dialogs

Framed input dialogs collect plaintext/ciphertext and keys:

```
           Caesar Cipher
  Enter Plaintext: ___________
```

Process Output

Detailed step-by-step output shows educational information:

```
[Caesar] Mode: Encrypt
[Caesar] Processed Text: HELLO
[Caesar] Key: 3 -> Final Shift: 3
----------------------------------
[Caesar] Char: 'H' (Index: 7)
         Calculation: (7 + 3) % 26 = 10
         Result: Index 10 -> 'K'

[Caesar] Char: 'E' (Index: 4)
         Calculation: (4 + 3) % 26 = 7
         Result: Index 7 -> 'H'
...
----------------------------------
[Caesar] Final Result: KHOOR

--- FINAL RESULT ---
```

```
Ciphertext: KHOOR

Press any key or click to return to the menu...
```

About Screen

Information about the application:

```
╔═══════════════════════════════════════════════╗
║                                               ║
║       About This Application                  ║
╠═══════════════════════════════════════════════╣
║  This console app demonstrates classic        ║
║  ciphers:                                     ║
║  - Caesar, Monoalphabetic, Vigenere           ║
║    (substitution)                             ║
║  - Transposition (permutation)                ║
║  - Playfair, Hill (polygraphic)               ║
║  - XOR (byte-level stream cipher)             ║
║  - Base64 (binary-to-text encoding)           ║
║                                               ║
║  Created for educational purposes.            ║
╠═══════════════════════════════════════════════╣
║  Click or Press any key to return             ║
╚═══════════════════════════════════════════════╝
```

Screenshots

1. Main Menu

```
╔═══════════════════════════════════════╗
║                                       ║
║             Main Menu                 ║
╠═══════════════════════════════════════╣
║  » Encrypt Text «                     ║
║    Decrypt Text                       ║
║    About                              ║
║    Exit                               ║
╠═══════════════════════════════════════╣
║    Use ↑/↓, Click, or Esc             ║
╚═══════════════════════════════════════╝
```

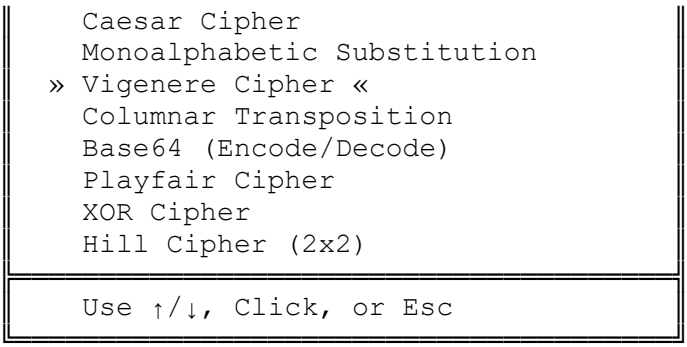
The application's main menu with mouse hover highlighting on "Encrypt Text"

Description: The main menu provides four options: Encrypt Text, Decrypt Text, About, and Exit. The interface uses box-drawing characters for a clean, professional look. The selected option is highlighted with a dark cyan background.

---

2. Cipher Selection

```
╔═══════════════════════════════════════╗
║                                       ║
║       Select Cipher Method            ║
╠═══════════════════════════════════════╣
```

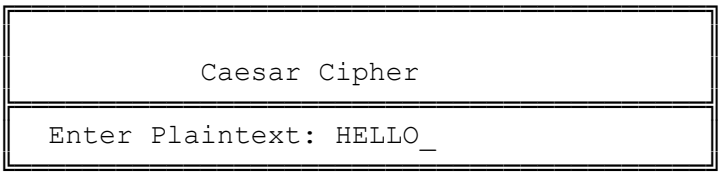


Cipher method selection screen showing all eight available ciphers

Description: After choosing to encrypt or decrypt, users select from eight classical ciphers. Each cipher is clearly labeled with its name and any special characteristics (e.g., "Hill Cipher (2x2)").

---

3. Caesar Cipher - Input

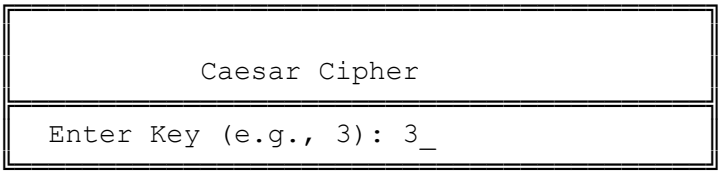


Input dialog for Caesar Cipher plaintext entry

Description: Framed input dialog prompts the user to enter plaintext. The interface maintains consistency with the menu styling.

---

4. Caesar Cipher - Key Entry



Key entry dialog for Caesar Cipher shift value

Description: Second input dialog requests the encryption key. For Caesar Cipher, this is an integer representing the shift amount.

---

5. Caesar Cipher - Detailed Output

```
[Caesar] Mode: Encrypt
[Caesar] Processed Text: HELLO
[Caesar] Key: 3 -> Final Shift: 3
---------------------------------
[Caesar] Char: 'H' (Index: 7)
         Calculation: (7 + 3) % 26 = 10
```

```
         Result: Index 10 -> 'K'

[Caesar] Char: 'E' (Index: 4)
         Calculation: (4 + 3) % 26 = 7
         Result: Index 7 -> 'H'

[Caesar] Char: 'L' (Index: 11)
         Calculation: (11 + 3) % 26 = 14
         Result: Index 14 -> 'O'

[Caesar] Char: 'L' (Index: 11)
         Calculation: (11 + 3) % 26 = 14
         Result: Index 14 -> 'O'

[Caesar] Char: 'O' (Index: 14)
         Calculation: (14 + 3) % 26 = 17
         Result: Index 17 -> 'R'

---------------------------------
[Caesar] Final Result: KHOOR

--- FINAL RESULT ---
Ciphertext: KHOOR

Press any key or click to return to the menu...
```

Step-by-step Caesar Cipher encryption process with calculations

Description: The application displays detailed information about each character transformation: - Original character and its index (0-25) - Mathematical calculation: (index + key) mod 26 - Resulting character and its index - Final encrypted result in cyan color

---

6. Vigenère Cipher - Process

```
[Vigenere] Mode: Encrypt
[Vigenere] Processed Text: HELLO
[Vigenere] Processed Key: KEY
---------------------------------
[Vigenere] Char: 'H' (P=7), Key: 'K' (K=10)
         Encrypt: (7 + 10) % 26 = 17
         Result: Index 17 -> 'R'

[Vigenere] Char: 'E' (P=4), Key: 'E' (K=4)
         Encrypt: (4 + 4) % 26 = 8
         Result: Index 8 -> 'I'

[Vigenere] Char: 'L' (P=11), Key: 'Y' (K=24)
         Encrypt: (11 + 24) % 26 = 9
         Result: Index 9 -> 'J'

[Vigenere] Char: 'L' (P=11), Key: 'K' (K=10)
         Encrypt: (11 + 10) % 26 = 21
         Result: Index 21 -> 'V'
```

```
[Vigenere] Char: 'O' (P=14), Key: 'E' (K=4)
         Encrypt: (14 + 4) % 26 = 18
         Result: Index 18 -> 'S'

----------------------------------
[Vigenere] Final Result: RIJVS
```

Vigenère Cipher showing repeating key mechanism

Description: Demonstrates polyalphabetic substitution with repeating key. Shows: - Original text and processed key - Per-character key application - Individual shift values - Character-by-character encryption calculations

---

7. Transposition Cipher - Grid Display

```
[Transposition] Mode: Encrypt
[Transposition] Processed Text: HELLOWORLD
[Transposition] Key Order: 3142
[Transposition] Grid Size: 3 Rows x 4 Columns
----------------------------------
[Transposition] Step 1: Filling matrix row by row...
         Row 0: HELL
         Row 1: OWOR
         Row 2: LDXX

[Transposition] Step 2: Reading matrix by key order...
         Reading Column 1 (Key '1'): EOX
         Reading Column 0 (Key '2'): HOL
         Reading Column 2 (Key '3'): LWD
         Reading Column 3 (Key '4'): LRX
----------------------------------
[Transposition] Raw Result (with padding): EOXHOLLWDLRX
[Transposition] Final Result (padding removed): EOXHOLLWDLRX
```

Columnar transposition showing grid layout and column reading order

Description: Visual representation of the transposition process: - Text arranged in grid (rows × columns) - Column numbers based on key - Reading order demonstration - Final encrypted result

---

8. Base64 - Binary Display

```
[Base64] Mode: Encode
[Base64] Input: Hi!
----------------------------------
[Base64] Step 1: Convert text to UTF-8 bytes.
         Char: 'H' -> Byte 72 (Binary: 01001000)
         Char: 'i' -> Byte 105 (Binary: 01101001)
         Char: '!' -> Byte 33 (Binary: 00100001)

[Base64] Step 2: Process bytes in 3-byte (24-bit) chunks...
       Processing block 0 (Bytes: 72, 105, 33)
         24-bit Group: 01001000 01101001 00100001
         6-bit Chunks: 010010 000110 100100 100001
         Indices:      18 6 36 33
```

```
         Characters:   'S' 'G' 'k' 'h'

---------------------------------
[Base64] Final Result: SGkh
```

Base64 encoding with binary and 6-bit chunk visualization

Description: Educational display of Base64 encoding: - UTF-8 byte representation - Binary conversion of each byte - 24-bit grouping - 6-bit chunk division - Base64 alphabet mapping - Final encoded result

---

9. Playfair Cipher - Matrix

```
[Playfair] Mode: Encrypt
[Playfair] Processed Text (J->I): HELLO
[Playfair] Processed Key (J->I): PLAYFAIR

[Playfair] Step 1: Building 5x5 Matrix...
         [P L A Y F]
         [I R E X M]
         [B C D G H]
         [K N O Q S]
         [T U V W Z]

[Playfair] Step 2: Preparing Text Digraphs...
HE LX LO

[Playfair] Step 3: Applying Cipher Rules...
         Pair 'HE' (2,4) (1,2): Rectangle -> 'XR'
         Pair 'LX' (0,1) (1,3): Rectangle -> 'LM'
         Pair 'LO' (0,1) (3,2): Rectangle -> 'AN'
---------------------------------
[Playfair] Final Result: XRLMAN
```

Playfair 5×5 matrix generated from key

Description: Shows the 5×5 Playfair matrix construction: - Key-based matrix generation - Digraph preparation - Rule application (same row/column/rectangle) - Character pair transformations

---

10. Hill Cipher - Matrix Operations

```
[Hill] Mode: Encrypt
[Hill] Processed Text: HELP
[Hill] Processed Key: HILL

[Hill] Step 1: Creating 2x2 Key Matrix (K)...
         K = |  7  8 |
             |  7 11 |

[Hill] Step 2: Calculating Determinant (det(K) mod 26)...
         det(K) = (7*11) - (8*7) = 21
         det(K) mod 26 = 21
         Modular Inverse of det(K) = 5
```

```
[Hill] Step 3: Using Key Matrix (K) for Encryption.

[Hill] Step 4: Text length is even, no padding needed.

[Hill] Step 5: Processing text in 2-char blocks (P * K_op)...
         Block 0 ('HE') -> [  7   4 ] * K_op = [  9   6 ] -> 'JG'
         Block 1 ('LP') -> [ 11  15 ] * K_op = [ 19  10 ] -> 'TK'
---------------------------------
[Hill] Final Result: JGTK
```

Hill Cipher showing matrix multiplication and determinant calculation

Description: Displays mathematical operations: - 2×2 key matrix construction - Determinant calculation - Modular inverse computation - Matrix multiplication for each block - Invertibility validation

---

11. XOR Cipher - Binary Operations

```
[XOR] Mode: Encrypt (to Base64)
[XOR] Key: KEY
[XOR] Input (Plaintext): HELLO
[XOR] Step 1: Convert Text and Key to UTF-8 Bytes

[XOR] Step 2: Applying repeating XOR operation...
      Index 0: (Text)  72 [ 01001000 ]  ^  (Key)  75 [ 01001011 ]  =  (Result)   3 [ 00000011
      Index 1: (Text) 101 [ 01100101 ]  ^  (Key)  69 [ 01000101 ]  =  (Result)  32 [ 00100000
      Index 2: (Text) 108 [ 01101100 ]  ^  (Key)  89 [ 01011001 ]  =  (Result)  53 [ 00110101
      Index 3: (Text) 108 [ 01101100 ]  ^  (Key)  75 [ 01001011 ]  =  (Result)  39 [ 00100111
      Index 4: (Text) 111 [ 01101111 ]  ^  (Key)  69 [ 01000101 ]  =  (Result)  42 [ 00101010

[XOR] Step 3: Encode raw result bytes to Base64 string.
---------------------------------
[XOR] Final Result (Base64): AyA1Jyo=
```

XOR cipher with byte-level binary visualization

Description: Byte-level encryption display: - Text to byte conversion - Binary representation of each byte - XOR operation with key bytes - Repeating key mechanism - Base64-encoded result

---

12. Error Handling Example

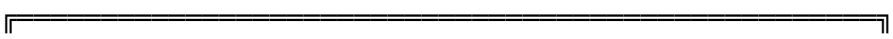
```
Error: Invalid key. Determinant 0 is not invertible modulo 26
(not coprime to 26). Try a different key.

Press any key or click to return...
```

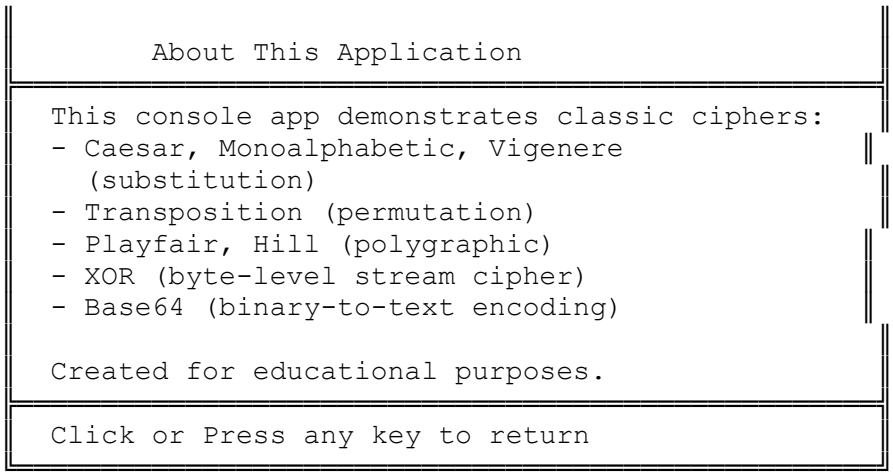
User-friendly error message for invalid key in Hill Cipher

Description: Clear error messages guide users when input validation fails. Example shows Hill Cipher determinant invertibility error with explanation.

---

13. About Screen





Application information and credits

Description: Provides overview of implemented ciphers and educational purpose statement.

---

14. Decryption Mode

```
[Caesar] Mode: Decrypt
[Caesar] Processed Text: KHOOR
[Caesar] Key: 3 -> Final Shift: 23
---------------------------------
[Caesar] Char: 'K' (Index: 10)
         Calculation: (10 + 23) % 26 = 7
         Result: Index 7 -> 'H'

[Caesar] Char: 'H' (Index: 7)
         Calculation: (7 + 23) % 26 = 4
         Result: Index 4 -> 'E'

[Caesar] Char: 'O' (Index: 14)
         Calculation: (14 + 23) % 26 = 11
         Result: Index 11 -> 'L'

[Caesar] Char: 'O' (Index: 14)
         Calculation: (14 + 23) % 26 = 11
         Result: Index 11 -> 'L'

[Caesar] Char: 'R' (Index: 17)
         Calculation: (17 + 23) % 26 = 14
         Result: Index 14 -> 'O'

---------------------------------
[Caesar] Final Result: HELLO

--- FINAL RESULT ---
Plaintext: HELLO

Press any key or click to return to the menu...
```

Decryption operation showing reverse transformation

Description: Demonstrates decryption with same detailed output as encryption, showing how the original text is recovered.

Conclusion

This Classic Cipher Application successfully implements eight fundamental cryptographic algorithms with an emphasis on educational value and user experience. The project demonstrates:

Key Achievements:

1. **Comprehensive Cipher Coverage:**
   - Substitution ciphers (Caesar, Monoalphabetic, Vigenère)
   - Transposition cipher (Columnar)
   - Polygraphic ciphers (Playfair, Hill)
   - Modern techniques (XOR, Base64)
2. **Educational Value:**
   - Step-by-step calculation display
   - Binary and mathematical operation visualization
   - Clear explanation of each transformation
   - Error messages that teach proper usage
3. **User Experience:**
   - Intuitive graphical console interface
   - Responsive and immediate feedback
   - Professional visual design
4. **Technical Excellence:**
   - Win32 API integration for advanced console control
   - Robust error handling and validation
   - Clean, maintainable code architecture
   - Performance-optimized event handling

Learning Outcomes:

Through this project, I gained deep understanding of: - Classical cryptography principles and their historical significance - Mathematical foundations of encryption algorithms - Linear algebra applications in cryptography (Hill Cipher) - Bit manipulation and binary operations (Base64, XOR) - Low-level Windows console programming - User interface design for console applications

Security Considerations:

While these classical ciphers are educational, they have known vulnerabilities: - **Caesar, Monoalphabetic:** Vulnerable to frequency analysis - **Vigenère:** Can be broken with Kasiski examination - **Transposition:** Vulnerable to anagramming attacks - **Playfair:** Resistant to frequency analysis but breakable with known plaintext - **Hill:** Vulnerable to known-plaintext attacks - **XOR:** Insecure if key is shorter than message or reused - **Base64:** Not encryption, just encoding

Modern cryptography uses AES, RSA, and other advanced algorithms. These classical ciphers serve as foundational concepts for understanding cryptographic principles.

Future Enhancements:

Potential improvements for the application: 1. Add more modern ciphers (DES, AES basic demonstration) 2. Implement frequency analysis tools 3. Add cipher-breaking demonstrations 4. Export results to file 5. Multi-language support 6. Graphical visualization of transformations 7. Performance metrics and statistics

Conclusion Statement:

This project successfully combines theoretical cryptography knowledge with practical software engineering, creating an educational tool that makes classical encryption accessible and understandable. The detailed logging and visual interface help users learn not just how to use ciphers, but how they actually work at a fundamental level.

Project Repository Structure:

```
cipher app - Copy/
├── Program.cs                    # Main source code (1,348 lines)
├── cipher app.csproj             # Project configuration
├── start.bat                     # Quick launch script
├── PROJECT_REPORT.md             # This document
├── bin/Debug/net7.0/
│   ├── cipher app.dll            # Compiled assembly
│   ├── cipher app.exe            # Executable
│   └── ...                       # Runtime files
└── obj/                          # Build intermediate files
```

End of Report

This document was prepared as part of the Network Security course project.
Student: Onur Çakır (211401038)
Date: November 5, 2025