INTERNATIONAL ISLAMIC UNIVERSITY, ISLAMABAD

SOFTWARE ENGINEERING DEPARTMENT

Assignment 03

Submitted To: Mr. Niaz Muhammad

Student Name: Syed Muhammad Oan Kazmi

Reg. Number: 4425-FOC/BSSE/F22

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <algorithm>

using namespace std;

// Function prototypes
void displayMenu();
string additiveCipher(const string &text, int key, bool encrypt);
string multiplicativeCipher(const string &text, int key, bool encrypt);
string affineCipher(const string &text, int a, int b, bool encrypt);
string vigenereCipher(const string &text, const string &key, bool encrypt);
int modInverse(int a, int m);
void bruteForceAdditive(const string &text);
string rowKeylessTranspositionCipher(const string &text, bool encrypt);
string columnKeylessTranspositionCipher(const string &text, bool encrypt);

int main()
{
    int choice;
    string text, key;
    int a, b;

    while (true)
    {
        displayMenu();
        cin >> choice;

        cout << "\n\nEnter your text: ";
        cin.ignore();
        getline(cin, text);

        int option;
        cout << "\n\nSelect : 1 for Encryption, 2 for Decryption    ";
        cin >> option;
        bool encrypt = (option == 1);

        switch (choice)
        {
        case 1:
        { // Vigenere cipher
            cout << "Enter the key: ";
            cin >> key;
            cout << "Result: " << vigenereCipher(text, key, encrypt) << endl;
            break;
        }
        case 2:
        { //brute force
            bruteForceAdditive(text);
            break;
        }
        case 3:
        { // Multiplicative cipher
            int key;
            cout << "Enter the key: ";
            cin >> key;
            cout << "Result: " << multiplicativeCipher(text, key, encrypt) << endl;
            break;
```

```
        }
        case 4:
        { // Additive cipher
            int key;
            cout << "Enter the key: ";
            cin >> key;
            cout << "Result: " << additiveCipher(text, key, encrypt) << endl;
            break;
        }
        case 5:
        { // Affine cipher
            cout << "Enter 'a' and 'b' keys: ";
            cin >> a >> b;
            cout << "Result: " << affineCipher(text, a, b, encrypt) << endl;
            break;
        }
        case 6:
        { // Column Keyless Transposition
            cout << "Result: " << columnKeylessTranspositionCipher(text, encrypt) << endl;
            break;
        }
        case 7:
        { // Row Keyless Transposition
            cout << "Result: " << rowKeylessTranspositionCipher(text, encrypt) << endl;
            break;
        }
        case 8:
        {   //exit
            cout << "Exiting...\n";
            break;
        }
        default:
            cout << "Invalid choice! Enter a Valid Number" << endl;
        }
    }

    return 0;
}

// Display menu
void displayMenu()
{
    cout << "\n    Encryption/Decryption Tool    \n";
    cout << "\nList of ciphers:\n\n";
    cout << "1. Vigenere\n";
    cout << "2. Brute Force Attack (Additive)\n";
    cout << "3. Multiplicative \n";
    cout << "4. Additive\n";
    cout << "5. Affine\n";
    cout << "6. Column Keyless Transposition\n";
    cout << "7. Row Keyless Transposition\n";
    cout << "8. Exit\n";
    cout << "\n\nEnter a number: ";
}

// Additive Cipher
string additiveCipher(const string &text, int key, bool encrypt)
{
    string result;
    int shift = encrypt ? key : -key;
```

```
    for (char c : text)
    {
        if (isalpha(c))
        {
            char base = isupper(c) ? 'A' : 'a';
            result += (c - base + shift + 26) % 26 + base;
        }
        else
        {
            result += c;
        }
    }
    return result;
}

// Modular Inverse
int modInverse(int a, int m)
{
    int m0 = m, t, q;
    int x0 = 0, x1 = 1;

    if (m == 1) return 0;

    while (a > 1)
    {
        // q is quotient
        q = a / m;
        t = m;

        // m is remainder now, process same as Euclid's algorithm
        m = a % m, a = t;
        t = x0;

        x0 = x1 - q * x0;
        x1 = t;
    }

    // Make x1 positive
    if (x1 < 0) x1 += m0;

    return x1;
}

// Multiplicative Cipher
string multiplicativeCipher(const string &text, int key, bool encrypt)
{
    // Check if the key is coprime with 26
    if (__gcd(key, 26) != 1)
    {
        cout << "Error: Key must be coprime with 26. Please choose a different key.\n";
        return "";
    }

    string result;
    int shift = encrypt ? key : modInverse(key, 26);

    if (shift == -1)
    {
        cout << "Error: No modular inverse exists for the given key.\n";
        return "";
    }
```

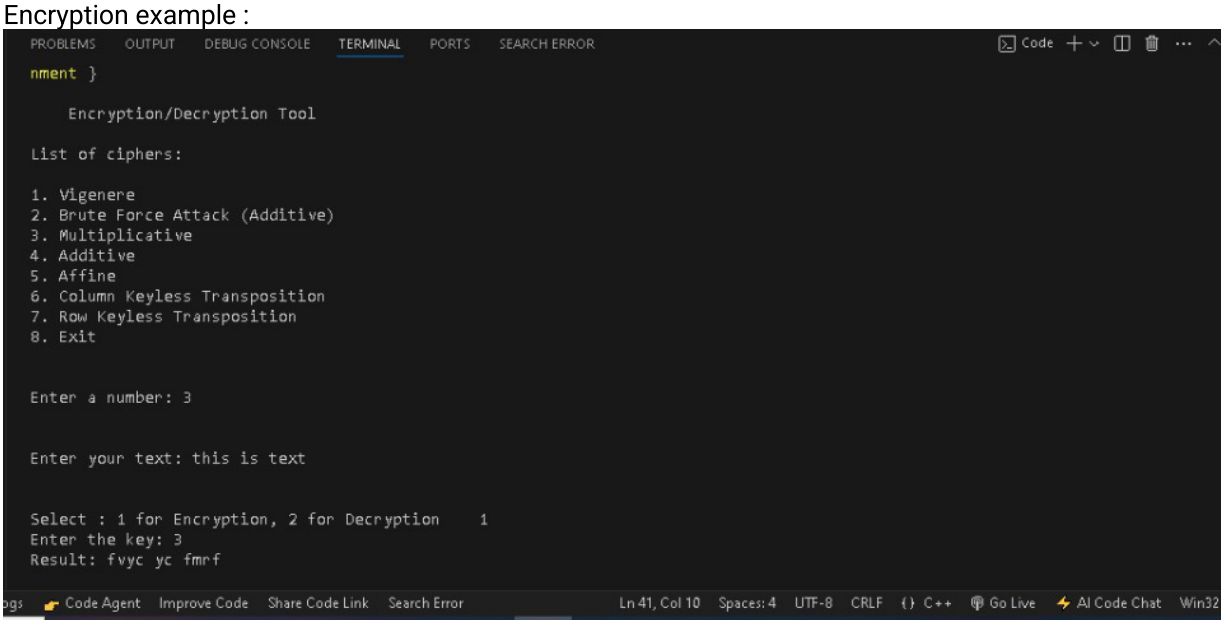
```
    for (char c : text)
    {
        if (isalpha(c))
        {
            char base = isupper(c) ? 'A' : 'a';
            int value = c - base;
            if (encrypt)
            {
                result += ((value * shift) % 26) + base;
            }
            else
            {
                // Ensure positive modulo result
                result += ((value * shift) % 26 + 26) % 26 + base;
            }
        }
        else
        {
            result += c;
        }
    }
    return result;
}

// Affine Cipher
string affineCipher(const string &text, int a, int b, bool encrypt)
{
    // Ensure that 'a' is coprime with 26
    if (__gcd(a, 26) != 1)
    {
        cout << "Error: 'a' must be coprime with 26. Please choose a different value for 'a'.\n";
        return "";
    }

    string result;
    int invA = modInverse(a, 26);

    // If decrypting and no modular inverse exists
    if (!encrypt && invA == -1)
    {
        cout << "Error: No modular inverse exists for the given value of 'a'.\n";
        return "";
    }

    for (char c : text)
    {
        if (isalpha(c))
        {
            char base = isupper(c) ? 'A' : 'a';
            int value = c - base;

            if (encrypt)
            {
                // Encryption formula: E(x) = (a*x + b) % 26
                result += ((a * value + b) % 26) + base;
            }
            else
            {
                // Decryption formula: D(x) = invA * (x - b) % 26
                result += ((invA * (value - b + 26)) % 26) + base;
```

```
            }
        }
        else
        {
            result += c; // Retain non-alphabet characters as-is
        }
    }

    return result;
}

// Vigenere Cipher
string vigenereCipher(const string &text, const string &key, bool encrypt)
{
    string result;
    int keyLength = key.length();

    for (size_t i = 0; i < text.length(); ++i)
    {
        char c = text[i];
        char k = key[i % keyLength];

        if (isalpha(c))
        {
            char base = isupper(c) ? 'A' : 'a';
            char kbase = isupper(k) ? 'A' : 'a';
            int shift = encrypt ? (k - kbase) : -(k - kbase);
            result += (c - base + shift + 26) % 26 + base;
        }
        else
        {
            result += c;
        }
    }
    return result;
}

// Brute Force for Additive Cipher
void bruteForceAdditive(const string &text)
{
    cout << "Brute Force Results:\n";
    for (int key = 1; key < 26; ++key)
    {
        cout << "Key " << key << ": " << additiveCipher(text, key, false) << endl;
    }
}

// Row Keyless Transposition Cipher
string rowKeylessTranspositionCipher(const string &text, bool encrypt)
{
    string result;
    int n = text.size();
    int numRows = 2;
    vector<string> rows(numRows);

    if (encrypt)
    {
        for (int i = 0; i < n; ++i)
        {
            rows[i % numRows] += text[i];
        }
```

```
        for (const string &row : rows)
        {
            result += row;
        }
    }
    else
    {
        int half = (n + 1) / 2;
        rows[0] = text.substr(0, half);
        rows[1] = text.substr(half);

        for (int i = 0; i < half; ++i)
        {
            if (i < rows[0].size())
                result += rows[0][i];
            if (i < rows[1].size())
                result += rows[1][i];
        }
    }
    return result;
}

// Column Keyless Transposition Cipher
string columnKeylessTranspositionCipher(const string &text, bool encrypt)
{
    string result;
    int n = text.size();
    int numCols = 4;
    int numRows = (n + numCols - 1) / numCols;
    vector<string> grid(numRows, string(numCols, ' '));

    if (encrypt)
    {
        for (int i = 0; i < n; ++i)
        {
            grid[i / numCols][i % numCols] = text[i];
        }
        for (int col = 0; col < numCols; ++col)
        {
            for (int row = 0; row < numRows; ++row)
            {
                if (grid[row][col] != ' ')
                    result += grid[row][col];
            }
        }
    }
    else
    {
        int fullCols = n % numCols;
        int fullRows = fullCols == 0 ? numRows : numRows - 1;
        int idx = 0;

        for (int col = 0; col < numCols; ++col)
        {
            for (int row = 0; row < numRows; ++row)
            {
                if (row < fullRows || col < fullCols)
                {
                    grid[row][col] = text[idx++];
                }
            }
```

```
        }

        for (const string &row : grid)
        {
            result += row;
        }
    }
    return result;
}
```

Encryption example :



Decryption example: