МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационной безопасности

ОТЧЕТ

по лабораторной работе №3

на тему: «**Основы классической криптографии**»

по дисциплине «Информационная безопасность»

Выполнили: Кожухова О.А. Шифр: 170582

Шорин В.Д. Шифр: 171406

Институт приборостроения, автоматизации и информационных технологий

Направление: 09.03.04 «Программная инженерия»

Группа: 71-ПГ

Проверил: Еременко В.Т.

Отметка о зачете:_____

Дата «____» _____ 2021г.

Орел, 2021 г.

Задание

1. Написать программу, осуществляющую кодирование текста путем шифра замены. Для реализации алгоритма используйте собственный генератор псевдослучайных чисел.

2. Написать программу, шифрующую сообщение методом перестановки. Для реализации алгоритма используйте собственный генератор псевдослучайных чисел.

3. Создать алгоритм «взбивания» сообщения, используя систему шифрования DES. Для реализации алгоритма используйте собственный генератор псевдослучайных чисел.

4. Создайте компьютерную модель криптографической машины Энигма. Для реализации алгоритма используйте собственный генератор псевдослучайных чисел.

Ход работы

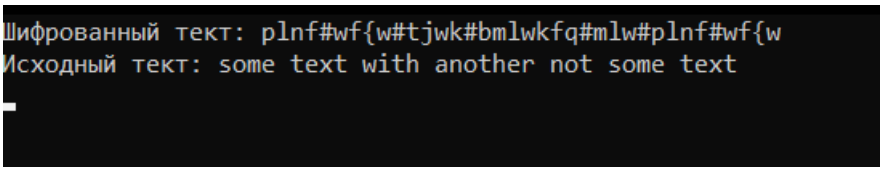

Рисунок 1 – Шифр замены

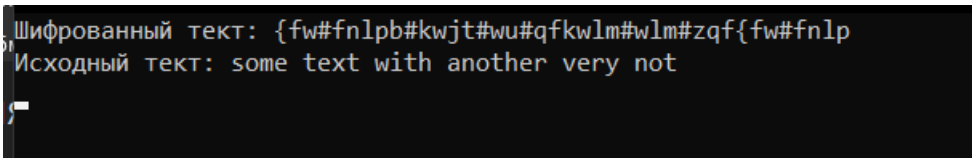

Рисунок 2 – Метод перестановки

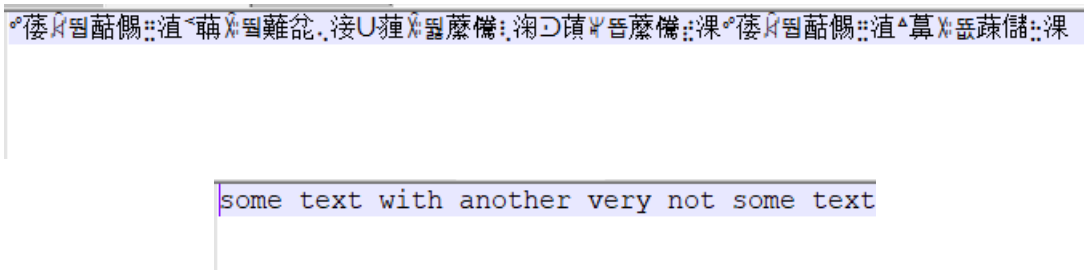

Рисунок 3 – Алгоритм DES (сверху зашифрованное сообщение, снизу расшифрованное)

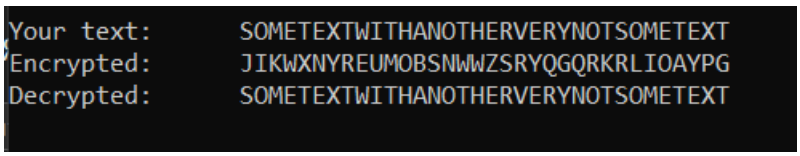

Рисунок 4 - Энигма

Код

«Program.cs»

```
using System;
using System.Collections.Generic;

namespace IS_L_3
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.Clear();
                Console.WriteLine("1 - Шифр замены");
                Console.WriteLine("2 - Метод перестановки");
                Console.WriteLine("3 - DES");
                Console.WriteLine("4 - Энигма");
                Console.WriteLine("0 - Выход");

                int res = Convert.ToInt32(Console.ReadLine());

                switch (res)
                {
                    case 1:
                        {
                            Console.Clear();

                            task1 t1 = new task1();
                            int n = 5;
                            string result = t1.Replacement("some text with another not
some text", n);
                            Console.WriteLine($"Шифрованный текст: {result}");
                            Console.WriteLine($"Исходный текст: {t1.Replacement(result,
n)}");


                            Console.ReadLine();
                            break;
                        }
                    case 2:
                        {
                            Console.Clear();

                            task1 t1 = new task1();
                            task2 t2 = new task2();
                            int n = 5;
                            string key = "87654321";
                            string text = "some text with another very not some text";

                            string result = t2.EncryptPermutation(t1.Replacement(text,
n), key);
                            Console.WriteLine($"Шифрованный текст: {result}");
                            Console.WriteLine($"Исходный текст:
{t1.Replacement(t2.EncryptPermutation(result, key), n)}");
```

```
                            Console.ReadLine();
                            break;
                        }
                    case 3:
                        {
                            Console.Clear();

                            task3 t3 = new task3();
                            string key = "somekey";
                            string text = "some text with another very not some text";

                            string keyDecoded = t3.EncryptDES(text, key);
                            Console.WriteLine($"Исходный текст:
{t3.DecryptDES(keyDecoded)}");

                            Console.ReadLine();
                            break;
                        }
                    case 4:
                        {
                            Console.Clear();

                            task4 t4 = new task4();
                            string text = "some text with another very not some text";

                            t4.Enigma(text);

                            Console.ReadLine();
                            break;
                        }
                    case 0:
                        return;
                    default:
                        Console.WriteLine("Нет такой команды");
                        break;
                }
            }
        }

        public static List<int> MZ(int n)
        {
            List<int> nums = new List<int>();
            List<int> fib = new List<int>();
            fib.Add(1);
            fib.Add(1);

            for (int i = 2; i < 1000; i++)
            {
                fib.Add(fib[i - 2] + fib[i - 1]);
            }

            for (int i = 0; i < fib.Count; i++)
            {
                string str = fib[i].ToString();
                str = str[str.Length - 1].ToString();
                fib[i] = Convert.ToInt32(str);
            }

            for (int i = 0; i < fib.Count; i += n)
            {
                nums.Add(fib[i]);
            }

            return nums;
        }
```

```
    }
}
```

«task1.cs»

```
using System;
using System.Collections.Generic;

namespace IS_L_3
{
    class task1
    {
        public string Replacement(string text, int n)
        {
            string result = "";
            List<int> nums = Program.MZ(n);

            for (int i = 0; i < text.Length; i++)
            {
                var tmp = Convert.ToChar(text[i] ^ nums[5]);
                result += tmp;
            }

            return result;
        }
    }
}
```

«task2.cs»

```
namespace IS_L_3
{
    class task2
    {
        public string EncryptPermutation(string text, string key)
        {
            string result = "";
            while (key.Length < text.Length)
            {
                int l = (text.Length > key.Length) ? key.Length : text.Length;
                string tmp = text.Substring(0, l);
                text = text.Remove(0, l);

                for (int i = 0; i < key.Length; i++)
                {
                    int index = key[i] - '0' - 1;
                    result += tmp[index];
                }
            }

            return result;
        }

    }
}
```

«task3.cs»

```
using System;
using System.IO;

namespace IS_L_3
```

```
{

    class task3
    {
        private const int sizeOfBlock = 128;
        private const int sizeOfChar = 16;

        private const int shiftKey = 2;

        private const int quantityOfRounds = 16;

        string[] Blocks;

        public string EncryptDES(string text, string key)
        {
            string result = "";

            string newText = StringToRightLength(text);
            CutStringIntoBlocks(newText);

            key = CorrectKeyWord(key, newText.Length / (2 * Blocks.Length));
            string tmpKey = key;
            Console.WriteLine($"tmpKey:{tmpKey}");
            key = StringToBinaryFormat(key);

            for (int j = 0; j < quantityOfRounds; j++)
            {
                for (int i = 0; i < Blocks.Length; i++)
                    Blocks[i] = EncodeDES_One_Round(Blocks[i], key);

                key = KeyToNextRound(key);
            }

            key = KeyToPrevRound(key);

            string keyDecoded = StringFromBinaryToNormalFormat(key);
            Console.WriteLine($"keyEncoded: {keyDecoded}");

            for (int i = 0; i < Blocks.Length; i++)
                result += Blocks[i];

            StreamWriter sw = new StreamWriter("out1.txt");
            sw.WriteLine(StringFromBinaryToNormalFormat(result));
            sw.Close();


            return keyDecoded;
        }

        public string DecryptDES(string keyDecoded)
        {
            string result = "";
            string key = StringToBinaryFormat(keyDecoded);

            string text = "";
            StreamReader sr = new StreamReader("out1.txt");

            while (!sr.EndOfStream)
            {
                text += sr.ReadLine();
            }

            sr.Close();

            text = StringToBinaryFormat(text);
```

```
        CutBinaryStringIntoBlocks(text);

        for (int j = 0; j < quantityOfRounds; j++)
        {
            for (int i = 0; i < Blocks.Length; i++)
                Blocks[i] = DecodeDES_One_Round(Blocks[i], key);

            key = KeyToPrevRound(key);
        }

        key = KeyToNextRound(key);

        string oldKey = StringFromBinaryToNormalFormat(key);
        Console.WriteLine($"oldKey: {oldKey}");

        for (int i = 0; i < Blocks.Length; i++)
            result += Blocks[i];

        StreamWriter sw = new StreamWriter("out2.txt");
        sw.WriteLine(StringFromBinaryToNormalFormat(result));
        sw.Close();

        return result;
    }

    private string StringToRightLength(string input)
    {
        while (((input.Length * sizeOfChar) % sizeOfBlock) != 0)
            input += "#";

        return input;
    }
    private void CutStringIntoBlocks(string input)
    {
        Blocks = new string[(input.Length * sizeOfChar) / sizeOfBlock];

        int lengthOfBlock = input.Length / Blocks.Length;

        for (int i = 0; i < Blocks.Length; i++)
        {
            Blocks[i] = input.Substring(i * lengthOfBlock, lengthOfBlock);
            Blocks[i] = StringToBinaryFormat(Blocks[i]);
        }
    }

    private void CutBinaryStringIntoBlocks(string input)
    {
        Blocks = new string[input.Length / sizeOfBlock];

        int lengthOfBlock = input.Length / Blocks.Length;

        for (int i = 0; i < Blocks.Length; i++)
            Blocks[i] = input.Substring(i * lengthOfBlock, lengthOfBlock);
    }

    private string StringToBinaryFormat(string input)
    {
        string output = "";

        for (int i = 0; i < input.Length; i++)
        {
            string char_binary = Convert.ToString(input[i], 2);

            while (char_binary.Length < sizeOfChar)
                char_binary = "0" + char_binary;
```

```
            output += char_binary;
        }

        return output;
    }

    private string CorrectKeyWord(string input, int lengthKey)
    {
        if (input.Length > lengthKey)
            input = input.Substring(0, lengthKey);
        else
            while (input.Length < lengthKey)
                input = "0" + input;

        return input;
    }
    private string EncodeDES_One_Round(string input, string key)
    {
        string L = input.Substring(0, input.Length / 2);
        string R = input.Substring(input.Length / 2, input.Length / 2);

        return (R + XOR(L, f(R, key)));
    }

    private string DecodeDES_One_Round(string input, string key)
    {
        string L = input.Substring(0, input.Length / 2);
        string R = input.Substring(input.Length / 2, input.Length / 2);

        return (XOR(f(L, key), R) + L);
    }

    private string XOR(string s1, string s2)
    {
        string result = "";

        for (int i = 0; i < s1.Length; i++)
        {
            bool a = Convert.ToBoolean(Convert.ToInt32(s1[i].ToString()));
            bool b = Convert.ToBoolean(Convert.ToInt32(s2[i].ToString()));

            if (a ^ b)
                result += "1";
            else
                result += "0";
        }
        return result;
    }

    private string f(string s1, string s2)
    {
        return XOR(s1, s2);
    }

    private string KeyToNextRound(string key)
    {
        for (int i = 0; i < shiftKey; i++)
        {
            key = key[key.Length - 1] + key;
            key = key.Remove(key.Length - 1);
        }

        return key;
    }
```

```
        private string KeyToPrevRound(string key)
        {
            for (int i = 0; i < shiftKey; i++)
            {
                key = key + key[0];
                key = key.Remove(0, 1);
            }

            return key;
        }
        private string StringFromBinaryToNormalFormat(string input)
        {
            string output = "";

            while (input.Length > 0)
            {
                string char_binary = input.Substring(0, sizeOfChar);
                input = input.Remove(0, sizeOfChar);

                int a = 0;
                int degree = char_binary.Length - 1;

                foreach (char c in char_binary)
                    a += Convert.ToInt32(c.ToString()) * (int)Math.Pow(2, degree--);

                output += ((char)a).ToString();
            }

            return output;
        }

    }
}
```

«task4.cs»

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Text.RegularExpressions;

namespace IS_L_3
{
   class task4
   {
      public void Enigma(string text)
      {
         EnigmaMachine machine = new EnigmaMachine();
         EnigmaSettings eSettings = new EnigmaSettings();

         querySettings(eSettings);
```

```
    string message = text;
    while (!Regex.IsMatch(message, @"^[a-zA-Z ]+$"))
    {
        Console.Write("Only letters A-Z is allowed, try again: ");
        message = Console.ReadLine();
    }
    message = message.Replace(" ", "").ToUpper();

    // Enter settings on machine
    machine.setSettings(eSettings.rings, eSettings.grund, eSettings.order, eSettings.reflector);

    // The plugboard settings
    foreach (string plug in eSettings.plugs)
    {
        char[] p = plug.ToCharArray();
        machine.addPlug(p[0], p[1]);
    }

    // Message encrypt
    Console.WriteLine();
    Console.WriteLine("Your text:\t" + message);
    string enc = machine.runEnigma(message);
    Console.WriteLine("Encrypted:\t" + enc);

    // Reset the settings before decrypting!
    machine.setSettings(eSettings.rings, eSettings.grund, eSettings.order, eSettings.reflector);

    // Message decrypt
    string dec = machine.runEnigma(enc);
    Console.WriteLine("Decrypted:\t" + dec);
    Console.WriteLine();

    Console.ReadLine();
}
```

```
        private static void querySettings(EnigmaSettings e)
        {
            e.setDefault();

            //string r;
            //Console.Write("Do you want to: [1] Specify settings [2] Use default settings? (Default:
[2]): ");
            //r = Console.ReadLine();
            //while (r != "1" && r != "2" && r != "")
            //{
            //    Console.Write("Invalid input, enter 1, 2 or 3 ");
            //    r = Console.ReadLine();
            //}
            //if (r == "1")
            //{
            //    Console.Write("Enter the ring settings (Ex. AAA, MCK, Default: AAA): ");
            //    r = Console.ReadLine();
            //    if (r == "")
            //        e.rings = new char[] { 'A', 'A', 'A' };
            //    else
            //        e.rings = r.ToCharArray();

            //    Console.Write("Enter the inital rotor start settings (Ex. AAA, MCK, Default: AAA):
");
            //    r = Console.ReadLine();
            //    if (r == "")
            //        e.grund = new char[] { 'A', 'A', 'A' };
            //    else
            //        e.grund = r.ToCharArray();

            //    Console.Write("Enter the order of the rotors (Ex. I-II-III, III-I-II, Default: I-II-III): ");
            //    r = Console.ReadLine();
            //    if (r == "")
            //        e.order = "I-II-III";
```

```
        //    else
        //        e.order = r;

        //    Console.Write("Enter the reflector to use (A, B, or C, Default: B): ");
        //    r = Console.ReadLine();
        //    if (r == "")
        //        e.reflector = 'B';
        //    else
        //        e.reflector = r.ToCharArray()[0];

        //    Console.Write("Enter the plugboard configuration (Ex. KH AB CE IJ, Default:
None): ");
        //    r = Console.ReadLine();
        //    if (r == "")
        //        e.plugs.Clear();
        //    else
        //    {
        //        string[] plugs = r.Split(' ');
        //        foreach (string s in plugs)
        //        {
        //            e.plugs.Add(s);
        //        }
        //    }

        //}
        //else if (r == "2" || r == "")
        //{
        //    e.setDefault();
        //}

        Console.WriteLine();
    }

    private class EnigmaSettings
    {
```

```
        public char[] rings { get; set; }
        public char[] grund { get; set; }
        public string order { get; set; }
        public char reflector { get; set; }
        public List<string> plugs = new List<string>();

        public EnigmaSettings()
        {
            setDefault();
        }

        public void setDefault()
        {
            rings = new char[] { 'A', 'A', 'A' };
            grund = new char[] { 'A', 'A', 'A' };
            order = "I-II-III";
            reflector = 'B';
            plugs.Clear();
        }
    }

}

class EnigmaMachine
{
    /* Enigma Machine
       Modelled after Enigma I, from ~1930
    */
    private Dictionary<Char, Char> plugBoard;

    // The machine has three rotors and a reflector
    private Rotor[] rotors;
    private Rotor reflector;

    private const string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
    // Rotor and reflectors. These configurations are constant and the same on every Enigma machine
    private const string rotorIconf = "EKMFLGDQVZNTOWYHXUSPAIBRCJ";
    private const string rotorIIconf = "AJDKSIRUXBLHWTMCQGZNPYFVOE";
    private const string rotorIIIconf = "BDFHJLCPRTXVZNYEIWGAKMUSQO";

    private const string reflectorAconf = "EJMZALYXVBWFCRQUONTSPIKHGD";
    private const string reflectorBconf = "YRUHQSLDPXNGOKMIEBFZCWVJAT";
    private const string reflectorCconf = "FVPJIAOYEDRZXWGCTKUQSBNMHL";

    // Rotor class representing one rotor
    private class Rotor
    {
        // The current char of the alphabet, and position of it. This char is visible outside the machine
        private int outerPosition;
        public char outerChar { get; set; }

        // The fixed alphabet of the rotor
        private string wiring;

        // turnOver is the notch on which letter the rotors turnover point is
        private char turnOver;

        public string name { get; }

        // Ring is the wiring setting relative to the turnover notch and position
        // Basically part of the initialization vector
        public char ring { get; set; }

        public int[] map { get; }
        public int[] revMap { get; }

        public Rotor(string w, char to, string n)
```

```
{
    turnOver = to;
    outerPosition = 0;

    ring = 'A'; // A default ring setting
    name = n;

    map = new int[26];
    revMap = new int[26];

    setWiring(w);
}

public void setWiring(string newW)
{
    wiring = newW;
    outerChar = wiring.ToCharArray()[outerPosition];

    // Fill the mapping arrays
    for (int i = 0; i < 26; i++)
    {
        int match = ((int)wiring.ToCharArray()[i]) - 65;
        map[i] = (26 + match - i) % 26;
        revMap[match] = (26 + i - match) % 26;
    }
}

public void setOuterPosition(int i)
{
    outerPosition = i;
    outerChar = alphabet.ToCharArray()[outerPosition];
}

public int getOuterPosition()
{
```

```
        return outerPosition;
    }

    public void setOuterChar(char c)
    {
        outerChar = c;
        outerPosition = alphabet.IndexOf(outerChar);
    }

    public void step()
    {
        outerPosition = (outerPosition + 1) % 26;
        outerChar = alphabet.ToCharArray()[outerPosition];
    }

    public bool isInTurnOver()
    {
        return outerChar == turnOver;
    }
}

private void rotateRotors(Rotor[] r)
{
    if (r.Length == 3)
    {
        if (r[1].isInTurnOver())
        {
            // If rotor II is on turnOver, all rotors step
            r[0].step();
            r[1].step();
        }
        else if (r[2].isInTurnOver())
        {
            // If rotor III is on turnOver, the two rotors to the right step
            r[1].step();
```

```
        }

        // Rotor III always steps
        r[2].step();
    }
}

// Apply the rotor scramble to character using all three rotors
// Argumentent reverse decides which direction we are scrambling
private char rotorMap(char c, bool reverse)
{
    int cPos = (int)c - 65;
    if (!reverse)
    {
        for (int i = rotors.Length - 1; i >= 0; i--)
        {
            cPos = rotorValue(rotors[i], cPos, reverse);
        }
    }
    else
    {
        for (int i = 0; i < rotors.Length; i++)
        {
            cPos = rotorValue(rotors[i], cPos, reverse);
        }
    }

    return alphabet.ToCharArray()[cPos];
}

private int rotorValue(Rotor r, int cPos, bool reverse)
{
    int rPos = (int)r.ring - 65;
    int d;
    if (!reverse)
```

```
        d = r.map[(26 + cPos + r.getOuterPosition() - rPos) % 26];
    else
        d = r.revMap[(26 + cPos + r.getOuterPosition() - rPos) % 26];

    return (cPos + d) % 26;
}

// Apply the reflector, the part that comes after the rotors
private char reflectorMap(char c)
{
    int cPos = (int)c - 65;
    cPos = (cPos + reflector.map[cPos]) % 26;
    return alphabet.ToCharArray()[cPos];
}

// Constructor
public EnigmaMachine()
{
    plugBoard = new Dictionary<char, char>();

    // Notch and alphabet are fixed on the rotor
    // First argument is alphabet, second is the turnover notch
    Rotor rI = new Rotor(rotorIconf, 'Q', "I");
    Rotor rII = new Rotor(rotorIIconf, 'E', "II");
    Rotor rIII = new Rotor(rotorIIIconf, 'V', "III");
    rotors = new Rotor[] { rI, rII, rIII }; // Default ordering of rotors
    reflector = new Rotor(reflectorAconf, ' ', "");
}

public void setReflector(char conf)
{
    if (conf != 'A' && conf != 'B' && conf != 'C')
    {
        throw new ArgumentException("Invalid argument");
    }
```

```
    string wiring = "";
    switch (conf)
    {
        case 'A':
            wiring = reflectorAconf;
            break;
        case 'B':
            wiring = reflectorBconf;
            break;
        case 'C':
            wiring = reflectorCconf;
            break;
    }
    reflector.setWiring(wiring);
}

// Enter the ring settings and initial rotor positions
public void setSettings(char[] rings, char[] grund)
{
    if (rings.Length != rotors.Length || grund.Length != rotors.Length)
    {
        throw new ArgumentException("Invalid argument lengths");
    }

    for (int i = 0; i < rotors.Length; i++)
    {
        rotors[i].ring = Char.ToUpper(rings[i]);
        rotors[i].setOuterChar(Char.ToUpper(grund[i]));
    }
}

public void setSettings(char[] rings, char[] grund, string rotorOrder)
{
    Rotor rI = null;
```

```
    Rotor rII = null;
    Rotor rIII = null;

    // Get the current ordering
    for (int i = 0; i < rotors.Length; i++)
    {
        if (rotors[i].name == "I")
            rI = rotors[i];
        if (rotors[i].name == "II")
            rII = rotors[i];
        if (rotors[i].name == "III")
            rIII = rotors[i];
    }

    string[] order = rotorOrder.Split('-');

    // Set the new ordering
    for (int i = 0; i < order.Length; i++)
    {
        if (order[i] == "I")
            rotors[i] = rI;
        if (order[i] == "II")
            rotors[i] = rII;
        if (order[i] == "III")
            rotors[i] = rIII;
    }

    setSettings(rings, grund);
}

public void setSettings(char[] rings, char[] grund, string rotorOrder, char reflectorConf)
{
    setReflector(reflectorConf);
    setSettings(rings, grund, rotorOrder);
}
```

```
// Encrypts or decrypts a message
public string runEnigma(string msg)
{
    StringBuilder encryptedMessage = new StringBuilder();

    msg = msg.ToUpper();

    foreach (char c in msg)
    {
        encryptedMessage.Append(encryptChar(c));
    }

    return encryptedMessage.ToString();
}

// Encrypts (or decrypts) a single character
private char encryptChar(char c)
{

    // Rotate the rotors before scrambling
    rotateRotors(rotors);

    // Into plugboard from keyboard <--
    if (plugBoard.ContainsKey(c))
    {
        c = plugBoard[c];
    }

    // Scramble with rotors
    // First we go all the way through the rotors <--
    c = rotorMap(c, false);

    // Reflect at the end so we don't just unscramble it again when we go back
    // If the line below is commented out, the cipher will be equal to the message
```

```
            c = reflectorMap(c);

            // Go back through all the rotors the other way -->
            c = rotorMap(c, true);

            // Plugboard again, from other direction -->
            if (plugBoard.ContainsKey(c))
            {
                c = plugBoard[c];
            }

            // Character is now encrypted
            return c;
        }

        // Add a character pair into the plugboard
        public void addPlug(char c, char cc)
        {
            if (Char.IsLetter(c) && Char.IsLetter(cc))
            {
                c = Char.ToUpper(c);
                cc = Char.ToUpper(cc);
                if (c != cc && !plugBoard.ContainsKey(c))
                {
                    plugBoard.Add(c, cc);
                    plugBoard.Add(cc, c);
                }
            }
            else
            {
                throw new ArgumentException("Invalid character");
            }
        }
    }
}
```