Politechnika Białostocka

Sprawozdanie z pracowni specjalistycznej

Bezpieczeństwo Sieci Komputerowych

Ćwiczenie numer: 3

Temat: **Implementacja podstawowych modułów kryptograficznych**

Wykonujący ćwiczenie: **Daniel Pietrzeniuk**

Studia dzienne

Kierunek: Informatyka

Semestr: VI

Grupa zajęciowa: 5

Prowadzący ćwiczenie: mgr inż. Dariusz Jankowski

Data wykonania ćwiczenia:
18 marca 2022r.

Wstęp:

Algorytmy zaimplementowałem w języku C# 9.0, do którego uruchomienia wymagany jest framework .NET 6 (z racji, że jest to język pół-interpretowany).

Testowanie algorytmów przeprowadziłem poprzez napisanie testów jednostkowych z pomocą biblioteki Xunit (wszystkie pliki testów nazwyają się w konwencji „[NazwaKlasy].test.cs“). Żeby uruchomić testy należy przejść terminalem do root-folderu kodu i użyć komendy „dotnet test“. Po użyciu komendy projekt się skompiluje, zostaną wykonane wszystkie testy i na terminal zostaną wypisane wyniki, w przypadku niepowodzenia jakiegoś testu, na terminal zostanie wypisany powód niepowodzenia algorytmu. Jeśli ktoś chciałby analizować działanie algorytmu polecam użycie do tego debugowania testów.

UWAGA: Jeden z testów (MatrixShift2_Tests.WordEncryption_ZeSprawdzarki()) posiada błędny.

Testować algorytmy można także manualnie poprzez napisany przeze mnie prosty TUI, jednak należy uważać podczas schodzenia z happy-path.

Projekt rozwijałem na repozytorium github: https://github.com/ProgramistycznySwir/BSK-3—Basic-Cryptography-. Na tym repozytorium są zaimplementowane także algorytmy Caesar Cipher i Vigenere Cipher które wyciąłem ze sprawozdania bo ich nie dotyczyło zadanie. Do tego 2 z 5 algorytmów implementują interfejs IEncryptor który pozwala na zahashowanie dowolnego ciągu obiektów.

Omówienie kodu:

Interfejsy IEncryptor i IStringEncryptor:

W celu ułatwienia w korzystaniu z algorytmów stworzyłem 2 interfejsy (gdzie IEncryptor jest uszczególnieniem IStringEncryptor). Interfejsy te definiują metody to szyfrowania i deszyfrowania ciągów znaków (lub dowolnych obiektów w przypadku IEncryptor), klucze są przechowywane wewnątrz klas implementujących algorytmy i są podawane w konstruktorze.

Zaimplementowane algorytmy:

- Szyfr RailFence - src/RailFence.cs
- Przestawienie macierzowe przykład A - src/MatrixShift.cs
- Przestawienie macierzowe przykład B - src/MatrixShift2.cs
- Szyfr cezara – src/CaesarCipher.cs
- Szyfr Vigenere’a - src/VigenereCipher.cs

Wyniki testów:

```
[ps@Lestaro Code]$ dotnet test
  Determining projects to restore...
  All projects are up-to-date for restore.
/home/ps/.nuget/packages/microsoft.net.test.sdk/16.11.0/build/netcoreapp2.1/Microsoft.NET.Test.Sdk.Program.cs(4,41): warning CS7022: The entry point of the program is global code; ignoring 'AutoGeneratedProgram.Main
(string[])' entry point. [/home/ps/Studia/Semestr_6/BSK/Sprawozdanie_03_(Basic_Cryptography)/Code/Code.csproj]
  Code -> /home/ps/Studia/Semestr_6/BSK/Sprawozdanie_03_(Basic_Cryptography)/Code/bin/Debug/net6.0/Code.dll
Test run for /home/ps/Studia/Semestr_6/BSK/Sprawozdanie_03_(Basic_Cryptography)/Code/bin/Debug/net6.0/Code.dll (.NETCoreApp,Version=v6.0)
Microsoft (R) Test Execution Command Line Tool Version 17.0.0+68bd10d3aee862a9fbb0bac8b3d474bc323024f3
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:    30, Skipped:     0, Total:    30, Duration: 54 ms - /home/ps/Studia/Semestr_6/BSK/Sprawozdanie_03_(Basic_Cryptography)/Code/bin/Debug/net6.0/Code.dll (net6.0)
[ps@Lestaro Code]$
```

Wszystkie testy jednostkowe nie wykazują żadnych błędów w implementacji

Szyfr RailFence:

Zaimplementowany w pliku: **src/RailFence.cs**

Testy:

```
public class RailFence_Tests
{
    [Theory]
    [InlineData("CRYPTOGRAPHY", "CTARPORPYYGH", 3)]
    [InlineData("ABCDEFGHIJKLMNOPRSTUWVXYZ", "AEIMRWZBDFHJLNPSUVYCGKOTX", 3)]
    Run Test | Debug Test
    public void WordEncryption(string word, string encryptedWord_expected, int railCount)
    {
        // Arrange:
        RailFence encryptor = new(railCount: railCount);
        // Act:
        string encryptedWord = encryptor.Encrypt(word);
        // Assert:
        Assert.Equal(encryptedWord_expected, encryptedWord);
    }

    [Theory]
    [InlineData("CTARPORPYYGH", "CRYPTOGRAPHY", 3)]
    [InlineData("AEIMRWZBDFHJLNPSUVYCGKOTX", "ABCDEFGHIJKLMNOPRSTUWVXYZ", 3)]
    Run Test | Debug Test
    public void WordDecryption(string word, string decryptedWord_expected, int railCount)
    {
        // Arrange:
        RailFence encryptor = new(railCount: railCount);
        // Act:
        string decryptedWord = encryptor.Decrypt(word);
        // Assert:
        Assert.Equal(decryptedWord_expected, decryptedWord);
    }
}
```

Testy dla tego algorytmy są w sumie 4, po jednym używając fraz z instrukcji, oraz po jednej używając alfabetu angielskiego.

Enkrypcja:

```
public IEnumerable<T> Encrypt<T>(IEnumerable<T> sequence)
{
    // Jeśli jest tylko jedna szyna od razu zwróć input.
    if(RailCount == 1)
        return sequence;

    // Deklaracji i inicjalizacja szyn.
    Queue<T>[] layers = new Queue<T>[RailCount].Select(e => new Queue<T>()).ToArray();

    // Deklaracja zmiennych używanych w iteracji przez sekwencję.
    var(bounceBack, currRail) = (false, 0);
    foreach(T element in sequence)
    {
        // Algorytm w currRail przechowuje do której szyny ma zapisać dany element, poniższy blok służy do "odbijania"
        //  tej zmiennej pomiędzy 0 i RailCount.
        if(currRail is 0)
            bounceBack = false;
        else if(currRail == RailCount-1)
            bounceBack = true;
        // Umieszczanie elementu na szynie.
        layers[currRail].Enqueue(element);
        currRail += bounceBack ? -1 : 1;
    }

    // Zebranie wszystkich szyn do jednej kolekcji. W innych językach to powinno być .reduce().
    return layers.SelectMany(e => e);
}
```

Algorytm najpierw przygotowuje szyny (w kodzie nazwane *layers*). Następnie, posiłkując się zmiennymi tymczasowymi *bounceBack* i *currRail*, umieszcza w iteracji wszystkie elementy na odpowiednich szynach (szyny są kolejką FIFO). Na koniec wszystkie szyny są zbierane (czytane po kolei wszystkie kolejki) i zwracane. Właściwie nie wiem czemu, ale w przypadku przechowywania klucza zdecydowałem się na podejście obiektowe i RailCount jest właściwością obiektu.

Dekrypcja:

```
public IEnumerable<T> Decrypt<T>(IEnumerable<T> sequence)
{
    // Jeśli jest tylko jedna szyna od razu zwróć input.
    if(RailCount == 1)
        return sequence;

    List<T>[] lines = Enumerable.Range(0, RailCount).Select(e => new List<T>()).ToArray();
    int[] lines_Lenght = Enumerable.Repeat(0, RailCount).ToArray();

    var(bounceBack, currRail) = (false, 0);
    foreach(T element in sequence)
    {
        lines_Lenght[currRail]++;

        if(currRail is 0)
            bounceBack = false;
        else if(currRail == RailCount-1)
            bounceBack = true;

        currRail += bounceBack ? -1 : 1;
    }

    using (var enumerator = sequence.GetEnumerator().Init())
    for (int line = 0; line < RailCount; line++)
        for (int c = 0; c < lines_Lenght[line]; c++)
            lines[line].Add(enumerator.PopMoveNext());

    int sequence_Count = sequence.Count(); // Cache
    List<T> result = new List<T>(sequence_Count);
    (bounceBack, currRail) = (false, 0);

    var lines_enumerator = lines.Select(e => e.GetEnumerator().Init()).ToArray();
    for (int i = 0; i < sequence_Count; i++)
    {
        result.Add(lines_enumerator[currRail].PopMoveNext());

        if(currRail is 0)
            bounceBack = false;
        else if(currRail == RailCount-1)
            bounceBack = true;

        currRail += bounceBack ? -1 : 1;
    }
    foreach (var enumerator in lines_enumerator)
        enumerator.Dispose();

    return result.AsEnumerable();
}
```

Z racji, że zdecydowałem się na zoptymalizowane rozwiązanie czytelność kodu cierpi. W pierwszej pętli określana jest długość szyn i zapisywana do kolejnych indeksów tablicy *lines_Lenght,* jest to robione w podobny sposób jak w szyfrowaniu. W następnych pętlach tworzony jest iterator z sekwencji wejściowej algorytmu, a następnie do *lines* zapisywane są kolejne obiekty z sekwencji, tak, że brane jest po n=*lines_Lenght*[line] znaków. Na koniec do result zbierane są elementy ze wszystkich szyn tak, gdzie zmienna currRail określa z której szyny wziąć znak.

Przestawienie macierzowe przykład A:

Zaimplementowany w pliku: **src/MatrixShift.cs**

Testy:

W sumie napisałem 8 testów dla tego algorytmu, po jednym dla enkrypcji i dekrypcji, 4 testy 2-stronne (sprawdzające, czy proces szyfrowania jest odwracalny przy tym samym kluczu), oraz 2 testy na podstawie sprawdzarki z https://serwer2155200.home.pl. Z racji, że testy wyglądają identycznie jak w przypadku RailFence, zamieszczę tutaj tylko ich deklarację, w razie czego można podejrzeć ich kod w pliku *.test.cs.

```
[Theory]
[InlineData("CRYPTOGRAPHYOSA", "YCPRGTROHAYPAOS", new int[] { 3, 1, 4, 2 })]
Run Test | Debug Test
public void WordEncryption(string word, string encryptedWord_expected, int[] key)
[Theory]
[InlineData("YCPRGTROHAYPAOS", "CRYPTOGRAPHYOSA", new int[] { 3, 1, 4, 2 })]
Run Test | Debug Test
public void WordDecryption(string word, string decryptedWord_expected, int[] key)
[Theory]
[InlineData("CRYPTOGRAPHYOSA", new int[] { 3, 1, 4, 2 })]
[InlineData("ABCDEFGHIJKLMNOPQRSTUVWXYZ", new int[] { 3, 1, 4, 2 })]
[InlineData("CRYPTOGRAPHYOSA", new int[] { 3, 4, 1, 5, 2 })]
[InlineData("ABCDEFGHIJKLMNOPQRSTUVWXYZ", new int[] { 3, 4, 1, 5, 2 })]
Run Test | Debug Test
public void TwoWayEncryption(string word, int[] key)
[InlineData("CRYPTOGRAPHYOSA", "YCPRGTROHAYPAOS", new int[] { 3, 1, 4, 2 })]
[InlineData("CRYPTOGRAPHYOSA", "YPCTRRAOPGOSHAY", new int[] { 3, 4, 1, 5, 2 })]
Run Test | Debug Test | Run | Debug | Show in Test Explorer
public void WordEncryption_ZeSprawdzarki(string word, string encryptedWord_expected, int[] key)
```

Implementacja:

```
// Since MatrixShift is only series manipulation, we could easily invert it by inverting the key, and bool at the
//  end is doing that for us. Look out for use of reverse flag to seek differences between encryption and decryption
public static IEnumerable<T> Hash<T>(IEnumerable<T> sequence, int[] key, bool reverse = false)
{
    if(key.Length is 1)
        return sequence;

    List<T> result = new List<T>(sequence.Count());
    int[] inversedKey = InverseKey(key).ToArray();

    T[] buffer = new T[key.Length];
    int bufferIndex = 0;
    foreach(T element in sequence)
    {
        buffer[(reverse ? key : inversedKey)[bufferIndex]] = element;
        bufferIndex++;
        if(bufferIndex == key.Length)
        {
            bufferIndex = 0;
            result.AddRange(buffer);
        }
    }
    // After normal pass of the algorithm we have to collect last incomplete segment:
    if(bufferIndex is not 0) // If buffer is not empty. (i states number of elements in buffer)
    {
        if(reverse is false) // Encrypt
        {
            for(int ii = 0; ii < key.Length; ii++)
                if(key[ii] < bufferIndex) // Skip if key literal is bigger than buffer elements.
                    result.Add(buffer[ii]);
        }
        // If we want to reverse encryption we have to:
        // Important bit about this whole operation is that the last sequence is often encrypted in a way:
        //  ABCX -(3,1,4,2)> CAXB - where X is just empty element, null.
        //  But in output sequence elements are collected to something like this CAB and we lose information about X
        //  When we decrypt this sequence we get CABX -(inverse(3,1,4,2))> BCXA => BCA - no bueno.
        //  In step 1 we have to revert last shift inserting X into correct place, we do this by skipping iterations
        //   where key element is greater than last sequence lenght.
        else // Decrypt
        {
            // 1. Revert shift in last segment.
            var temp = new T[key.Length];
            for(int i0 = 0, i1 = 0; i0 < key.Length; i0++)
            {
                if(key[i0] >= bufferIndex) // 1.1. Most important bit is this part.
                    continue;

                temp[i0] = buffer[key[i1]];
                i1++;
            }
            // 2. We have to redo shift, now with elements in correct places.
            for(int i0 = 0; i0 < key.Length; i0++)
                buffer[key[i0]] = temp[i0];
            // 3. Voila, now we just add elements
            for(int i2 = 0; i2 < bufferIndex; i2++)
                result.Add(buffer[i2]);
        }
    }

    return result.AsEnumerable();
}
```

W przypadku tego algorytmu proces enkrypcji i dekrypcji jest do siebie podobny i różni się jedynie kolejnością klucza, oraz tym jak należy potraktować ostatni blok. W pierwszej pętli algorytmu iteruje się po wszystkich elementach ciągu wejściowego, wrzuca się je na tablicę (*buffer*) zgodnie z kluczem, która to tablica po zapełnieniu zrzucana jest do ciągu wyjściowego (*result*). To była główna część algorytmu, następną częścią jest przetworzenie ostatniego ciągu jeśli jest on mniejszy od długości klucza. Z racji, na zrezygnowanie z użycia paddingu ten krok jest nieco bardziej skomplikowany. W przypadku szyfrowania wystarczy zwyczajnie przenieść elementy bufor do *result* jednak w przypadku deszyfrowania należy odwrócić ostatni etap przestawienia macierzowego, dodać padding i na koniec na nowo wykonać ostatni etap przestawienia.

Przestawienie macierzowe przykład B:

Zaimplementowany w pliku: **src/MatrixShift2.cs**

Testy:

W sumie napisałem 6 testów dla tego algorytmu, po jednym dla enkrypcji i dekrypcji, 4 testy 2-stronne (sprawdzające, czy proces szyfrowania jest odwracalny przy tym samym kluczu). Z racji, że testy wyglądają identycznie jak w przypadku RailFence, zamieszczę tutaj tylko ich deklarację, w razie czego można podejrzeć ich kod w pliku *.test.cs.

```
[Theory]
[InlineData("HERE IS A SECRET MESSAGE ENCIPHERED BY TRANSPOSITION",
        "HECRN CEYI ISEP SGDI RNTO AAES RMPN SSRO EEBT ETIA EEHS",
        "CONVENIENCE")]
Run Test | Debug Test
public void WordEncryption(string word, string encryptedWord_expected, string key)
[Theory]
[InlineData("HECRN CEYI ISEP SGDI RNTO AAES RMPN SSRO EEBT ETIA EEHS",
        "HEREISASECRETMESSAGEENCIPHEREDBYTRANSPOSITION",
        "CONVENIENCE")]
Run Test | Debug Test
public void WordDecryption(string word, string decryptedWord_expected, string key)
[Theory]
[InlineData("CRYPTOGRAPHYOSA", "ROSHAR")]
[InlineData("ABCDEFGHIJKLMNOPQRSTUVWXYZ", "ROSHAR")]
[InlineData("CRYPTOGRAPHYOSA", "CONVENIENCE")]
[InlineData("ABCDEFGHIJKLMNOPQRSTUVWXYZ", "CONVENIENCE")]
Run Test | Debug Test
public void TwoWayEncryption(string word, string key)
```

Szyfrowanie:

```
public string Encrypt(string word)
{
    word = string.Join("", word.Split(default(string[]), StringSplitOptions.RemoveEmptyEntries));

    int queueLenght = (int) Math.Ceiling((float)word.Length / Key.Length);
    (Queue<char> column, char keyLiteral)[] matrix = new Queue<char>[Key.Length]
            .Select(e => new Queue<char>(queueLenght))
            .Zip(Key)
            .ToArray();

    for(int i = 0; i < word.Length; i++)
        matrix[i%matrix.Length].column.Enqueue(word[i]);

    return string.Join(' ', matrix.OrderBy(e => e.keyLiteral).Select(e => e.column.CollectString()));
}
```

W przypadku implementacji tego algorytmu zdecydowałem się na programowanie funkcyjne. Na początku z słowa wejściowego usuwane są puste znaki (takie jak spacje). Następnie tworzona jest tablica kolejek *matrix* z przypisanych im kolejnym literom klucza *keyLiteral* (w iteratorze Select, który działa jak map z innych języków, jedynie deklarowane są kolejki). Po tym w pętli wrzucane są na kolejne kolejki kolejne litery z słowa wejściowego. Na koniec dochodzi do właściwego przestawienia matrycowego, gdzie *matrix* jest sortowana po *keyLiteral* i następnie każda kolumna matrycy jest łączona ze sobą.

Deszyfrowanie:

Deszyfrowanie jest troche bardziej skomplikowanym procesem. Na początek słowo wejściowe jest zapisywane do matrycy i każdej kolumnie przypisywane są litery z klucza w kolejności alfabetycznej. Następnie te kolumny są przemieszczane tak by ich przypisane litery zgadzały się z kluczem. Na koniec zczytywane są rzędy matrycy.

```
public string Decrypt(string word)
{
    // Prepare input for organizing.
    LinkedList<(char, Queue<char>)> rawMatrix = new(
            Key.OrderBy(e => e)
            .Zip(word.Split(' ').Select(e => new Queue<char>(e))));

    // Now we have to place matrix columns in order defined by key.
    Queue<Queue<char>> matrix = new();
    foreach(var keyLiteral in Key)
    {
        var node = rawMatrix.First;
        while(node is not null)
        {
            if (node.Value.Item1 == keyLiteral)
            {
                matrix.Enqueue(node!.Value.Item2);
                rawMatrix.Remove(node);
                break;
            }

            node = node.Next;
        }
    }
    // Collect result by reading matrix row by row.
    Queue<char> result = new();
    while(true)
        foreach(var column in matrix)
        {
            if(column.Count is 0)
                return result.CollectString();
            result.Enqueue(column.Dequeue());
        }
}
```

Szyfr Cezara:

Zaimplementowany w pliku: **src/CaesarCipher.cs**

Testy:

W sumie napisałem 6 testów dla tego algorytmu, po jednym dla enkrypcji i dekrypcji, 4 testy 2-stronne (sprawdzające, czy proces szyfrowania jest odwracalny przy tym samym kluczu). Z racji, że testy wyglądają identycznie jak w przypadku RailFence, zamieszczę tutaj tylko ich deklarację, w razie czego można podejrzeć ich kod w pliku *.test.cs.

```
[Theory]
[InlineData("CRYPTOGRAPHY", "FUBSWRJUDSKB", 3)]
Run Test | Debug Test | Run | Debug | Show in Test Explorer
public void WordEncryption(string word, string encryptedWord_expected, int key)
[Theory]
[InlineData("FUBSWRJUDSKB", "CRYPTOGRAPHY", 3)]
Run Test | Debug Test | Run | Debug | Show in Test Explorer
public void WordDecryption(string word, string decryptedWord_expected, int key)
```

```
[Theory]
[InlineData("CRYPTOGRAPHYOSA", 3)]
[InlineData("ABCDEFGHIJKLMNOPRSTUWVXYZ", 3)]
[InlineData("CRYPTOGRAPHYOSA", 5)]
[InlineData("ABCDEFGHIJKLMNOPRSTUWVXYZ", 5)]
Run Test | Debug Test | Run | Debug | Show in Test Explorer
public void TwoWayEncryption(string word, int key)
```

Implementacja:

```
public string Alphabet { get; init; }
public Dictionary<char, int> Alphabet_Dict { get; init; }

public CaesarCipher(int key = Default_Key, string alphabet = Default_Alphabet)
{
    (Key, Alphabet) = (key, alphabet);
    Alphabet_Dict = Alphabet
            .Zip(Enumerable.Range(0, Alphabet.Length))
            .ToDictionary(e => e.First, e => e.Second);
}
```

Na początku algorytmu by ułatwić jego późniejszą implementację, wszystkim literom alfabetu przypisuje rosnąco liczby od 0 (algorytm został zaimplementowany w taki sposób, że można mu podać dowolny alfabet).

```
public string Shift(string word, int shift)
{
    if(shift is 0)
        return word;

    if(word.ToHashSet().Any(letter => Alphabet_Dict.ContainsKey(letter) is false))
        throw new ArgumentException($"First argument [{nameof(word)}] contains letters which are not in this encryptor alphabet!");

    return word.Select(letter => ShiftLetter(letter, shift)).CollectString();
}
```

Kiedy korzystamy z algorytmu podajemy mu zwyczajnie słowo wejściowe *word* i liczbę *shift* o jaką chcemy przesunąć to słowo. Algorytm wtedy na każdej literze słowa wykonuje funkcję *ShiftLetter()*. Która wygląda następująco:

```
public char ShiftLetter(char letter, int shift)
    => Alphabet[MyMath.ClampMod(Alphabet_Dict[letter] + shift, mod: Alphabet.Length)];
```

Funkcja ta najpierw z słownika *Alphabet_Dict* pobiera indeks w *Alphabet* obecnie przetwarzanej litery, dodaje do tego indeksu wartość przesunięcia *shift* i zwraca z *Alphabet'*u przesunięty indeks zapewniając by znajdował się on w rozmiarze alfabetu.

Szyfr Vigenere'a:

Zaimplementowany w pliku: **src/VigenereCipher.cs**

Testy:

W sumie napisałem 6 testów dla tego algorytmu, po jednym dla enkrypcji i dekrypcji, 4 testy 2-stronne (sprawdzające, czy proces szyfrowania jest odwracalny przy tym samym kluczu). Z racji, że testy wyglądają identycznie jak w przypadku RailFence, zamieszczę tutaj tylko ich deklarację, w razie czego można podejrzeć ich kod w pliku *.test.cs.

```
[Theory]
[InlineData("CRYPTOGRAPHY", "DICPDPXVAZIP", "BREAKBREAKBR")]
Run Test | Debug Test
public void WordEncryption(string word, string encryptedWord_expected, string key)
[Theory]
[InlineData("DICPDPXVAZIP", "CRYPTOGRAPHY", "BREAKBREAKBR")]
Run Test | Debug Test
public void WordDecryption(string word, string decryptedWord_expected, string key)
[Theory]
[InlineData("CRYPTOGRAPHY", "BREAKBREAKBR")]
[InlineData("ABCDEFGHIJKLMNOPRSTUWVXYZ", "BREAKBREAKBR")]
[InlineData("CRYPTOGRAPHYOSA", "OTHERKEY")]
[InlineData("ABCDEFGHIJKLMNOPRSTUWVXYZ", "OTHERKEY")]
Run Test | Debug Test
public void TwoWayEncryption(string word, string key)
```

Implementacja:

```
public VigenereCipher(string key = Default_Key, string alphabet = Default_Alphabet)
{
    (Key, Alphabet) = (key, alphabet);
    Alphabet_Dict = Alphabet
            .Zip(Enumerable.Range(0, Alphabet.Length))
            .ToDictionary(e => e.First, e => e.Second);
}
```

Identycznie jak w przypadku algorytmu szyfru cezara, na początku algorytmu by ułatwić jego późniejszą implementację, wszystkim literom alfabetu przypisuje rosnąco liczby od 0 (algorytm został zaimplementowany w taki sposób, że można mu podać dowolny alfabet).

```
public string Hash(string word, string key, bool reverse = false)
{
    List<char> result = new();

    for (int i = 0; i < word.Length; i++)
        result.Add(ShiftLetter(word[i], key[i % key.Length], reverse));

    return result.CollectString();
}
```

Algorytm w swojej głównej metodzie jedyne co robi to wykonuje *ShiftLetter()* na każdej literze słowa wejściowego *word*. Flaga *reverse* służy do określenia czy algorytm ma szyfrować, czy odwrotnie – deszyfrować.

```
public char ShiftLetter(char letter, char key, bool revert)
{
    if (Alphabet_Dict.ContainsKey(letter) is false)
        return letter;

    int index = Alphabet_Dict[letter];
    index += revert is false ? Alphabet_Dict[key] : -Alphabet_Dict[key];
    index = MyMath.ClampMod(index, Alphabet.Length);

    return Alphabet[index];
}
```

Funkcja *ShiftLetter()* znajduje indeks litery do przesunięcia w alfabecie, następnie przestawia go o indeks litery klucza (zależne od tego czy algorytm szyfruje czy deszyfruje w prawo lub w lewo) tak, żeby ten indeks znajdował się w długości alfabetu i na koniec zwraca element alfabetu na który wskazuje ten indeks.

Wnioski:

Udało się zaimplementować wszystkie algorytmy szyfrujące za wyjątkiem przestawienia macierzowego C. Z mojej wiedzy wynika, że żaden z tych algorytmów nie jest uważany za bezpieczny, jednak to ćwiczenie pomogło w zrozumieniu jak nawet tak proste szyfry pomagają w zabezpieczaniu informacji.