

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ»

**Избыточное кодирование данных в информационных систем в
информационных системах. Код Хемминга**

Студент: Водчиц Анастасия
ФИТ 3 курс 1 группа
Преподаватель: Нистюк О.А.

Минск 2025

Цель: приобретение практических навыков кодирования/декодирования двоичных данных при использовании кода Хемминга.

Задачи:

– Закрепить теоретические знания по использованию методов помехоустойчивого кодирования для повышения надежности передачи и хранения в памяти компьютера двоичных данных.

– Разработать приложение для кодирования/декодирования двоичной информации кодом Хемминга с минимальным кодовым расстоянием 3 или 4.

– Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

Теоретические сведения

Надежность системы – характеристика способности программного, аппаратного, аппаратно-программного средства выполнить при определенных условиях требуемые функции в течение конкретного периода времени.

Достоверность работы системы (устройства) – свойство, характеризующее истинность конечного (выходного) результата работы (выполнения программы), определяемое способностью средств контроля фиксировать правильность или ошибочность работы.

Ошибка устройства – неправильное значение сигнала (бита – в цифровом устройстве) на внешних выходах устройства или отдельного его узла, вызванное технической неисправностью, или воздействующими на него помехами (преднамеренными либо непреднамеренными), или иным способом.

Ошибка программы – проявляется в не соответствующем реальному (требуемому) промежуточному или конечному значению(результате) вследствие неправильно запрограммированного алгоритма или неправильно составленной программы.

Надежность является комплексным свойством, включающим в себя единичные свойства: безотказность, ремонтпригодность, сохраняемость, долговечность.

Безотказность – это свойство технического объекта непрерывно сохранять работоспособное состояние в течение некоторого времени (или наработки).

Ремонтпригодность – это свойство технического объекта, заключающееся в приспособленности к поддержанию и восстановлению работоспособного состояния путем технического обслуживания, ремонта (или с помощью дополнительных, избыточных технических средств, функционирующих параллельно с объектом).

Изначальной причиной нарушения нормальной работы цифрового устройства являются технические дефекты (неисправности), возникающие внутри узлов или блоков устройства либо в каналах связи между ними.

Дефекты или неисправности могут приводить либо к кратковременному нарушению достоверности работы устройства (сбой), либо к полной и окончательной потере достоверности (отказ).

В каждом из этих случаев следствием неисправности являются ошибки в информации (информационные ошибки). Чаще всего причиной ошибок бывают внешние помехи. Количество таких ошибок (количество ошибочных двоичных символов) принято называть кратностью ошибки. Обнаружение и/или исправление подобных ошибок как раз и призваны обеспечить кодер и декодер.

При использовании избыточных кодов исходные данные делятся на блоки из k битов (называются информационными битами). В процессе кодирования каждый k -битный блок данных преобразуется, как было отмечено выше, в блок из n битов (кодовое слово). Число k часто называется размерностью кода. Таким образом, к каждому блоку данных в процессе кодирования присоединяются $r = n - k$ битов, которые называют избыточными битами (redundant bits), битами четности (parity bits) или контрольными битами (check bits); новой информации они не несут.

Для обозначения описанного кода обычно пользуются записью (n, k) и говорят, что данный код использует n символов для передачи (хранения) k символов сообщения. Отношение числа битов данных к общему числу битов k/n именуется степенью кодирования (code rate) – доля кода, которая приходится на полезную информацию. Еще одним важным параметром кода является расстояние Хемминга (d), которое показывает, что два кодовых слова различаются по крайней мере в d позициях.

В общем случае код позволяет обнаруживать t_0 ошибок:

$$t_0 = \begin{cases} \frac{d}{2}, & d - \text{четное}; \\ \frac{d-1}{2}, & d - \text{нечетное}. \end{cases}$$

Количество исправляемых кодом ошибок t_n определяется следующим образом:

$$t_n = \begin{cases} \frac{d-1}{2}, & d - \text{нечетное}; \\ \frac{d-2}{2}, & d - \text{четное}. \end{cases}$$

К. Шеннон сформулировал теорему для случая передачи дискретной информации по каналу связи с помехами, утверждающую, что вероятность ошибочного декодирования принимаемых сигналов может быть обеспечена сколь угодно малой путем выбора соответствующего способа кодирования сигналов. В теореме Шеннона не говорится о том, как нужно строить

необходимые помехоустойчивые коды. Однако в ней указывается на принципиальную возможность кодирования, при котором может быть обеспечена сколь угодно высокая надежность передачи.

Код Хемминга относится к классу линейных блочных кодов. Линейные блочные коды – это класс кодов с контролем четности, которые можно описать парой чисел (n, k) .

Для формирования r проверочных символов (кодирования), т. е. вычисления проверочного слова X_r , используется порождающая матрица G : совокупность базисных векторов будем далее записывать в виде матрицы G размерностью $k \times n$ с единичной подматрицей (I) в первых k строках и столбцах: $G = [P|I]$.

Более точно матрица G называется порождающей матрицей линейного корректирующего кода в приведенно-ступенчатой форме. Кодовые слова являются линейными комбинациями строк матрицы G (кроме слова, состоящего из нулевых символов).

Кодирование заключается в умножении вектора сообщения X_k длиной k на порождающую матрицу по правилам матричного умножения (все операции выполняются по модулю 2). Очевидно, что при этом первые k символов кодового слова равны соответствующим символам сообщения, а последние r символов образуются как линейные комбинации первых.

Для всякой порождающей матрицы G существует матрица H размерности $r \times n$, задающая базис нулевого пространства кода и удовлетворяющая равенству $G * H^T = 0$. Справедливо также $X_n * H^T = H * X_n^T = 0$. Матрица H , называемая проверочной, равна $H = [-P^T|I]$.

В коде Хемминга с минимальным кодовым расстоянием $d_{\min} = 3$ проверочная матрица H имеет классический вид и состоит из двух подматриц: P' размером $k \times r$ и I размером $r \times r$ соответственно.

Общее число всех возможных комбинаций 2^r должно удовлетворять неравенству $2^r \geq n + 1$.

Результат умножения сообщения на выходе канала передачи (Y_n) или (что равнозначно) сообщения, считываемого из памяти, на проверочную матрицу (H) называется синдромом (вектором ошибки) S : $S = H * (Y_n)^T = Y_n * H^T$.

Синдром – это результат проверки четности, выполняемой над сообщением Y_n для определения его принадлежности заданному набору кодовых слов. При положительном результате проверки синдром S равен 0, т. е. $Y_n = X_n$. Если Y_n содержит ошибки, которые можно исправить, то синдром имеет определенное ненулевое значение, что позволяет обнаружить и исправить конкретную ошибочную комбинацию.

Важно запомнить, что ненулевой синдром всегда равен сумме по модулю 2 тех векторстолбцов матрицы H , номера которых соответствуют номерам ошибочных битов в слове Y_n .

Практические задания

Задание 1. На основе информационного сообщения, представленного символами русского/английского алфавитов, служебными символами и цифрами, содержащегося в некотором текстовом файле, сформировать информационное сообщение в двоичном виде; длина сообщения в бинарном виде должна быть не менее 16 символов. Для выполнения этого задания можно использовать коды ASCII символов алфавита либо результаты лабораторной работы № 3.

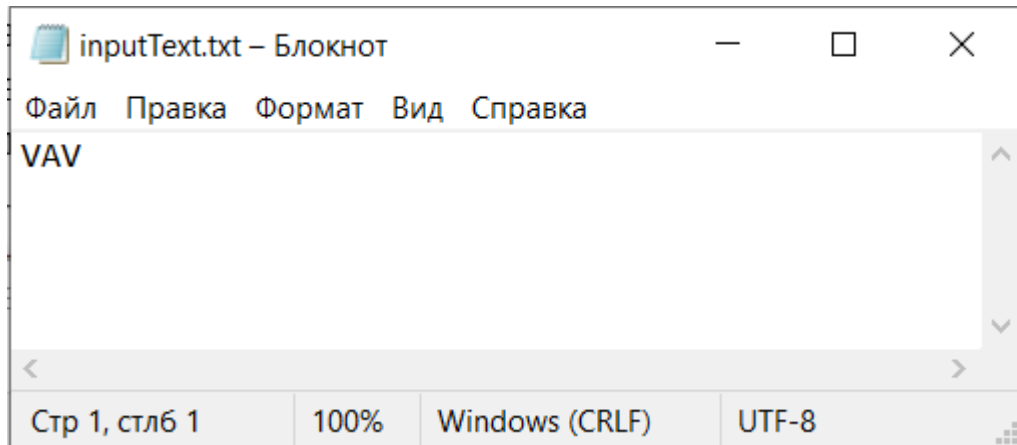


Рисунок 1.1 – Исходный файл на латинице (латышский)

```
class FileReader
{
    public static string ReadTextFromFile(string filePath)
    {
        try
        {
            FileInfo fileInfo = new FileInfo(filePath);
            return File.ReadAllText(filePath);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка при чтении файла: {ex.Message}");
            return string.Empty;
        }
    }

    public static int[] TextToBinaryArray(string text)
    {
        List<int> binaryArray = new List<int>();

        foreach (char c in text)
        {
            int asciiValue = (int)c;
            string binaryValue = Convert.ToString(asciiValue, 2).PadLeft(8,
'0');

            // Добавляем каждый бит как отдельный элемент массива
            foreach (char bit in binaryValue)
            {
                binaryArray.Add(int.Parse(bit.ToString())); // Преобразуем
'0'/'1' в 0/1
            }
        }
    }
}
```

```

    }

    return binaryArray.ToArray();
}
}

```

Листинг 1.1 – Класс для работы с файлом

```

string inputText = FileReader.ReadTextFromFile("inputText.txt");
int[] Xk = FileReader.TextToBinaryArray(inputText);
foreach (var x in Xk) Console.Write(x);
Console.WriteLine();
int k = Xk.Length;

Console.WriteLine("=====");

```

Листинг 1.2 – Обработка текстового файла

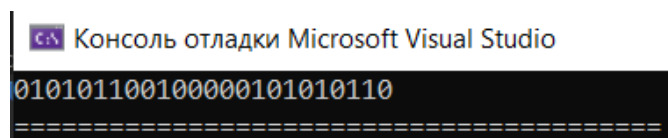


Рисунок 1.2 – Результат работы

Задание 2. Для полученного информационного слова построить проверочную матрицу Хемминга.

```

public static int[,]? GenerateMatrix(int k)
{
    Console.WriteLine("Длина информационного слова: " + k);
    int r = CalculateRedundantBits(k);
    Console.WriteLine("Длина избыточного слова: " + r);
    int n = k + r;
    Console.WriteLine("Длина кодового слова: " + n);

    int[,] iMatrix = CreateIMatrix(r);
    int[,]? pMatrix = CreatePMatrix(k, r);

    if (pMatrix == null) return null;
    // Соединяем identityMatrix и pMatrix
    int[,] Matrix = ConcatenateMatrices(pMatrix, iMatrix);
    return Matrix;
}

private static int[,]? CreatePMatrix(int k, int r)
{
    // Шаг 1: Посчитать общее количество возможных столбцов
    int totalColumns = 0;
    for (int p = 2; p <= r; p++)
    {
        totalColumns += Combinations(r, p);
    }

    Console.WriteLine($"Общее количество возможных столбцов: {totalColumns}");

    // Шаг 2: Проверить, что количество столбцов больше k
    if (totalColumns < k) // Corrected condition: <= changed to <
    {
        Console.WriteLine("Ошибка: количество возможных столбцов меньше k");
    }
}

```

```

        return null;
    }

    // Шаг 3: Построить матрицу
    // Генерация всех возможных столбцов с весом от двух
    var allColumns = GenerateAllColumns(r);
    // Выбор k случайных уникальных столбцов
    var selectedColumns = SelectRandomColumns(allColumns, k);
    // Построение матрицы
    int[,] matrix = new int[r, k];
    Console.WriteLine($"Матрица P:");
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < k; j++)
        {
            matrix[i, j] = selectedColumns[j][i]; // Corrected indexing:
selectedColumns[j][i]
            Console.Write(matrix[i, j] + " ");
        }
        Console.WriteLine();
    }
    return matrix;
}

// Функция для вычисления числа сочетаний C(n, k)
static int Combinations(int n, int k)
{
    if (k > n) return 0;
    if (k == 0 || k == n) return 1;

    int result = 1;
    for (int i = 1; i <= k; i++)
    {
        result = result * (n - k + i) / i;
    }
    return result;
}

// Функция для выбора k случайных уникальных столбцов
static List<int[]> SelectRandomColumns(List<int[]> allColumns, int k)
{
    var random = new Random();
    return allColumns.OrderBy(x => random.Next()).Take(k).ToList();
}

// Функция для генерации всех возможных столбцов с минимум w единицами
static List<int[]> GenerateAllColumns(int r)
{
    var columns = new List<int[]>();
    for (int p = 2; p <= r; p++)
    {
        // Генерация всех комбинаций из r по p
        var combinations = GenerateCombinations(r, p);
        foreach (var combination in combinations)
        {
            // Создание столбца на основе комбинации
            int[] column = new int[r];
            foreach (int index in combination)
            {
                column[index] = 1;
            }
            columns.Add(column);
        }
    }
}

```

```

        return columns;
    }

    // Функция для генерации всех комбинаций из n по k
    static IEnumerable<int[]> GenerateCombinations(int n, int k)
    {
        int[] result = new int[k];
        Stack<int> stack = new Stack<int>();
        stack.Push(0);

        while (stack.Count > 0)
        {
            int index = stack.Count - 1;
            int value = stack.Pop();

            while (value < n)
            {
                result[index++] = value++;
                stack.Push(value);

                if (index == k)
                {
                    yield return result.ToArray();
                    break;
                }
            }
        }
    }

    public static int[,] ConcatenateMatrices(int[,] matrixA, int[,] matrixB)
    {
        int rowsA = matrixA.GetLength(0);
        int colsA = matrixA.GetLength(1);
        int rowsB = matrixB.GetLength(0);
        int colsB = matrixB.GetLength(1);

        if (rowsA != rowsB)
        {
            throw new ArgumentException("Матрицы должны иметь одинаковое количество строк для горизонтального объединения.");
        }

        int[,] concatenatedMatrix = new int[rowsA, colsA + colsB];

        // Копируем элементы из matrixA
        for (int i = 0; i < rowsA; i++)
        {
            for (int j = 0; j < colsA; j++)
            {
                concatenatedMatrix[i, j] = matrixA[i, j];
            }
        }

        // Копируем элементы из matrixB
        for (int i = 0; i < rowsA; i++)
        {
            for (int j = 0; j < colsB; j++)
            {
                concatenatedMatrix[i, colsA + j] = matrixB[i, j];
            }
        }

        return concatenatedMatrix;
    }
}

```



```

public static int[,] CreateIMatrix(int r)
{
    if (r <= 0)
    {
        throw new ArgumentException("Размер матрицы должен быть положительным числом");
    }

    int[,] identityMatrix = new int[r, r];
    Console.WriteLine($"Матрица I:");
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < r; j++)
        {
            if (i == j) identityMatrix[i, j] = 1;
            else identityMatrix[i, j] = 0;
            Console.Write(identityMatrix[i, j] + " ");
        }
        Console.WriteLine();
    }
    return identityMatrix;
}

private static int CalculateRedundantBits(int k)
{
    return (int)Math.Ceiling(Math.Log2(k + Math.Ceiling(Math.Log2(k)) + 1));
    // More correct calculation of redundant bits
}

public static void PrintMatrix(int[,] matrix)
{
    int rows = matrix.GetLength(0);
    int cols = matrix.GetLength(1);

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++) Console.Write(matrix[i, j] + " ");
        Console.WriteLine();
    }
}

```

Листинг 2.1 – Функции для вычисления проверочной матрицы

```

int[,]? H = Hemming.GenerateMatrix(k);

Console.WriteLine("=====");

Console.WriteLine("Проверочная матрица H:");
Hemming.PrintMatrix(H);

Console.WriteLine("=====");

```

Листинг 2.2 – Вычисление проверочной матрицы

Консоль отладки Microsoft Visual Studio

```
010101100100000101010110
=====
Длина информационного слова: 24
Длина избыточного слова: 5
Длина кодового слова: 29
Матрица I:
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
Общее количество возможных столбцов: 26
Матрица P:
1 0 1 0 1 1 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 1 1 1
1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1
0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 1 1 0 0 1 0 1 1 1
0 1 1 0 1 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 1 1 0 1
0 1 0 1 1 1 1 1 0 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0
=====
Проверочная матрица H:
1 0 1 0 1 1 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0 0 0
0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 0
0 1 1 0 1 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 0 1 0
0 1 0 1 1 1 1 1 0 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0 0 0 0 0 1
=====
```

Рисунок 2.1 – Результат работы

Задание 3. Используя построенную матрицу, вычислить избыточные символы (слово Xr).

```
public static int[] SolveHamming(int[,] h, int[] knownBits, out int[]
unknownBits)
{
    int k = knownBits.Length;
    int r = CalculateRedundantBits(k);
    unknownBits = new int[r];

    // Проверка размерностей
    if (h.GetLength(0) != r || h.GetLength(1) != r + k || knownBits.Length !=
k)
    {
        throw new ArgumentException("Неверные размерности матриц или вектора
известных битов.");
    }

    // Создаем полный вектор сообщения x, добавляя нули для неизвестных
    int[] x = new int[r + k];
    for (int i = 0; i < k; i++)
    {
        x[i] = knownBits[i];
    }
    // Остальные элементы (неизвестные) уже инициализированы нулями при
создании массива
}
```

```

// Вычисляем неизвестные элементы (теперь работаем с полным вектором x)
for (int i = 0; i < r; i++)
{
    int x_ri = 0;
    for (int j = 0; j < k; j++)
    {
        x_ri ^= h[i, j] * x[j];
    }
    unknownBits[i] = x_ri; // Сохраняем неизвестный бит
    x[k + i] = x_ri;
}

return x;
}

```

Листинг 3.1 – Функция для вычисления избыточного слова матрицы

```

int[] Xr;
int[] Xn = Hemming.SolveHamming(H, Xk, out Xr);

Console.WriteLine("Избыточное слово:");
foreach (var x in Xr) Console.Write(x);
Console.WriteLine();

Console.WriteLine("Кодовое слово:");
foreach (var x in Xn) Console.Write(x);
Console.WriteLine();

```

Листинг 3.2 – Вычисление избыточного слова матрицы

```

=====
Избыточное слово:
01111
Кодовое слово:
01010110010000010101011001111
=====

```

Рисунок 3.1 – Результат работы

Задание 4. Принять исходное слово со следующим числом ошибок: 0, 1, 2. Позиция ошибки определяется (генерируется) случайным образом.

```

public static void ChangeValue(int[] array)
{
    int d = random.Next(0, array.Length + 1);
    if (array[d] == 0) array[d] = 1;
    else array[d] = 0;
}

```

Листинг 4.1 – Функция для генерации случайной ошибки

```

int[] Yn1 = new int [Xn.Length];
int[] Yn2 = new int [Xn.Length];
int[] Yn3 = new int [Xn.Length];
Xn.CopyTo(Yn1, 0);

```

```
Xn.CopyTo(Yn2, 0);
Xn.CopyTo(Yn3, 0);
Hemming.ChangeValue(Yn2);
Hemming.ChangeValue(Yn3);
Hemming.ChangeValue(Yn3);
```

Листинг 4.2 – Генерация слов для декодирования

Задание 5. Для полученного слова $Y_n = Y_k, Y_r$, используя уже известную проверочную матрицу Хемминга, вновь вычислить избыточные символы (обозначим их Y_r').

Задание 6. Вычислить и проанализировать синдром. В случае, если анализ синдрома показал, что информационное сообщение было передано с ошибкой (или 2 ошибками), сгенерировать унарный вектор ошибки $E_n = e_1, e_2, \dots, e_n$ и исправить одиночную ошибку; проанализировать ситуацию при возникновении ошибки в 2 битах.

```
public static (int[] firstPart, int[] secondPart) SplitArray(int[] array, int
k)
{
    // Проверка входных данных
    if (array == null)
    {
        throw new ArgumentNullException(nameof(array), "Входной массив не
может быть null.");
    }
    if (k < 0 || k > array.Length)
    {
        throw new ArgumentOutOfRangeException(nameof(k), "Значение k должно
быть неотрицательным и не больше длины массива.");
    }

    // Создаем первый подмассив
    int[] firstPart = new int[k];
    Array.Copy(array, 0, firstPart, 0, k);

    // Создаем второй подмассив
    int[] secondPart = new int[array.Length - k];
    Array.Copy(array, k, secondPart, 0, array.Length - k);

    return (firstPart, secondPart);
}
public static int[] XorVectors(int[] vector1, int[] vector2)
{
    if (vector1 == null || vector2 == null)
    {
        throw new ArgumentNullException("Один или оба входных вектора равны
null.");
    }
    if (vector1.Length != vector2.Length)
    {
        throw new ArgumentException("Векторы должны иметь одинаковую
длину.");
    }

    int[] result = new int[vector1.Length];

    for (int i = 0; i < vector1.Length; i++)
    {
```

```

        result[i] = vector1[i] ^ vector2[i];
    }

    return result;
}

public static void CheckYn(int[] Yn, int k, int[,] H)
{
    Console.WriteLine("Поступило слово:");
    foreach (var x in Yn) Console.Write(x);
    Console.WriteLine();
    (int[] Yk, int[] Yr) = SplitArray(Yn, k);
    Console.WriteLine("Yk:");
    foreach (var x in Yk) Console.Write(x);
    Console.WriteLine();
    Console.WriteLine("Yr:");
    foreach (var x in Yr) Console.Write(x);
    Console.WriteLine();
    Console.WriteLine("Вычисляем Yr'");
    int[] Yrq;
    int[] Ynq = SolveHamming(H, Yk, out Yrq);
    foreach (var x in Yrq) Console.Write(x);
    Console.WriteLine();
    int[] S = XorVectors(Yr, Yrq);
    Console.WriteLine("Синдром:");
    foreach (var x in S) Console.Write(x);
    Console.WriteLine();

    // Проверка на нулевой вектор
    bool isZeroVector = true;
    foreach (int element in S)
    {
        if (element != 0)
        {
            isZeroVector = false;
            break;
        }
    }

    if (isZeroVector)
    {
        Console.WriteLine("Синдром нулевой, значит ошибок нет");
        return;
    }

    if (S.Length != H.GetLength(0))
    {
        throw new ArgumentException("Длина вектора должна быть равна количеству строк матрицы.");
    }

    // Поиск совпадающего столбца
    for (int j = 0; j < H.GetLength(1); j++) // Перебираем столбцы
    {
        bool match = true;
        for (int i = 0; i < H.GetLength(0); i++) // Перебираем строки
        {
            if (H[i, j] != S[i])
            {
                match = false;
                break;
            }
        }
    }
}

```

```

        if (match)
        {
            Console.WriteLine($"Найден совпадающий столбец: {j}");
            int[] E = new int[Yn.Length];
            for (var e = 0; e < E.Length; e++) E[e] = 0;
            E[j] = 1;
            Console.WriteLine($"Вектор ошибки:");
            for (var e = 0; e < E.Length; e++) Console.Write(E[e]);
            Console.WriteLine();
            Console.WriteLine($"Исправленный Yn:");
            int[] YnFIX = XorVectors(Yn, E);
            for (var i = 0; i < YnFIX.Length; i++) Console.Write(YnFIX[i]);
            Console.WriteLine();
            return;
        }
    }
    Console.WriteLine("Совпадающих столбцов не найдено")
}

```

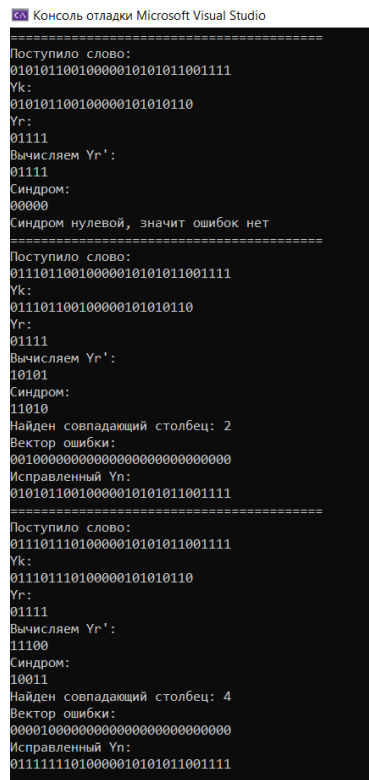
Листинг 6.1 – Функции для декодирования

```

Console.WriteLine("=====");
Hemming.CheckYn(Yn1, k, H);
Console.WriteLine("=====");
Hemming.CheckYn(Yn2, k, H);
Console.WriteLine("=====");
Hemming.CheckYn(Yn3, k, H);
Console.WriteLine("=====");

```

Листинг 6.2 – Декодирование



```

Консоль отладки Microsoft Visual Studio
=====
Поступило слово:
010101100100000101011001111
Yk:
0101011001000001010110
Yr:
01111
Вычисляем Yr':
01111
Синдром:
00000
Синдром нулевой, значит ошибок нет
=====
Поступило слово:
011101100100000101011001111
Yk:
0111011001000001010110
Yr:
01111
Вычисляем Yr':
10101
Синдром:
11010
Найден совпадающий столбец: 2
Вектор ошибки:
00100000000000000000000000000000
Исправленный Yn:
010101100100000101011001111
=====
Поступило слово:
011101110100000101011001111
Yk:
0111011101000001010110
Yr:
01111
Вычисляем Yr':
11100
Синдром:
10011
Найден совпадающий столбец: 4
Вектор ошибки:
00001000000000000000000000000000
Исправленный Yn:
011111110100000101011001111
=====

```

Рисунок 6.1 – Результат работы

Вывод: В ходе лабораторной работы были закреплены теоретические знания по основам теории информации, а также разработано программное средство для расчета энтропии по Шеннону и определения количества информации для латышского, таджикского и бинарного алфавитов.

Выполнены вычисления, позволяющие проанализировать объем информации в зависимости от используемого алфавита. Дополнительно исследовано влияние вероятности ошибок на передаваемую информацию. Установлено, что формула эффективной энтропии применима исключительно к бинарному алфавиту, так как только в двоичной системе представления информации возможно однозначное изменение символа на противоположный при наличии ошибки, что делает данный метод неприменимым для многосимвольных алфавитов.