Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Дисциплина «Защита информации и надежность информационных систем»

**Отчёт по лабораторной работе №9**

Сжатие/распаковка данных на основе статических методов

Студент: Жук С.С.

ФИТ 3 курс 2 группа

Преподаватель: Савельева М.Г.

**Содержание**

[1 Теоретические сведения 3](#_Toc196602042)

[2 Метод Шеннона-Фано 6](#_Toc196602043)

[3 Метод Хаффмана 11](#_Toc196602044)

[Вывод 15](#_Toc196602045)

# **1 Теоретические сведения**

Мы неоднократно подчеркивали, что каждый из естественных языков обладает избыточностью. Среди европейских языков белорусский и русский обладают одним из самых высоких уровней избыточности.

До появления уже упоминавшихся работ К. Шеннона кодирование символов алфавита при передаче сообщения по каналам связи осуществлялось одинаковым количеством битов, получаемым по формуле Хартли. Позднее начали появляться способы, кодирующие символы разным числом битов в зависимости от вероятности появления их в тексте. Таким образом, за счет использования для каждого значения байта кодов ASCII (символа алфавита) кода различной длины в соответствии с частостью (вероятностью появления этого символа в сообщении) можно значительно уменьшить общий размер данных. Эта базовая идея лежит в основе алгоритмов статистических (вероятностных) методов сжатия: Шеннона − Фано и Хаффмана.

Статистические алгоритмы позволяют создавать более короткие коды для часто встречающихся и более длинные – для редко встречающихся символов алфавита или конкретного сообщения. В первом случае метод считается статическим статистическим, во втором – динамическим статистическим: вероятностные свойства символов подсчитываются для конкретного сообщения или потока данных.

Частота или вероятность появления того или иного символа алфавита в произвольном сообщении, лежащая в основе алгоритмов, дали название этим алгоритмам и соответствующим методам.

Иногда эти методы называют также префиксными.

К примеру, если имеется некоторый код, который записывается как *Х*1 = *А*1*А*2, и другой код – *Х*2 = *А*1, то говорят, что *Х*2 является префиксом *Х*1. Или если *Х*1 = 1010, а *Х*2 = 10101100, то *Х*2 также является префиксом *Х*1.

Таким образом, использование описываемых методов предусматривает создание кодовой таблицы (подобно кодам ASCII или base64). Формально процедура сжатия (прямое преобразование) состоит в подстановке соответствующего бинарного кода вместо символа исходного алфавита и наоборот – при обратном преобразовании.

Методы относятся к классу «сжатие без потерь». Различие между двумя рассматриваемыми методами состоит лишь в особенностях формирования таблицы бинарных кодов. При формировании этой таблицы для обоих методов можно воспользоваться статистическими свойствами алфавитов, полученными при выполнении лабораторной работы № 2.

Код Шеннона – Фано не является оптимальным (обеспечивает минимальную избыточность) в общем смысле, хотя и дает оптимальные результаты при некоторых распределениях вероятностей. Для одного и того же распределения вероятностей можно построить, вообще говоря, несколько кодов Шеннона – Фано, и все они могут дать различные результаты.

Итак, необходимо выполнить следующие действия:

1. подсчитать вероятностные параметры символов алфавита *А* = {*ai*} (реализуется статическая версия алгоритма);
2. отсортировать – обычно в порядке убывания (невозрастания, т. е. могут иметь место повторяющиеся значения) вероятностей *р*(*аi*); *р*(*аi*) – вероятность появления в сжимаемом сообщении на произвольной позиции символа *аi* алфавита, т. е. создать таблицу символов алфавита, на основе которого генерируется сжимаемое сообщение;
3. каждому символу отсортированного множества поставить в соответствие бинарный код, для чего это множество (таблица) символов делится на две группы таким образом, чтобы каждая из групп имела приблизительно одинаковую суммарную частоту (вероятность). Очевидно, на первом шаге такая суммарная вероятность в каждой из групп должна быть максимально близка к 0,5. Первому из полученных подмножеств устанавливается первый символ бинарного кода: 0, второй − 1 (или наоборот). Для вычисления следующих битов кодов данная процедура повторяется рекурсивно для каждого из полученных на текущем шаге подмножеств, в котором содержится больше одного символа. Получим таблицу, в которой длина кодовых комбинаций меняется от минимального (*l*min) до максимального (*l*max) значений.

Символам с большими вероятностями соответствуют коды меньшей длины и наоборот. Основное требование: все кодовые комбинации разные. Требование префикса: ни одна из кодовых комбинаций меньшей длины не является началом кодовой комбинации большей длины. Именно последняя таблица используется в неизменном виде (речь о кодах) в процессах прямого и обратного преобразований.

Алгоритм прямого преобразования: необходимо выполнить одну операцию: заменить символы входного сообщения соответствующими бинарными кодами.

Алгоритм обратного преобразования: на входе – сообщение в виде бинарной последовательности.

Шаг 1. Анализируются *l*min начальных бинарных символов: осуществляется поиск в таблице соответствующего совпадения. Если такое будет найдено, то на выходе будет символ исходного алфавита с совпадающим кодом. После этого процедура повторяется, т. е. анализируются очередные *l*min символов. Если не найдено в таблице совпадения, переходим к шагу 2.

Шаг 2. Длина анализируемой последовательности увеличивается на 1 бит: *l*min + 1. Осуществляется поиск совпадающей бинарной комбинации такой же длины в таблице. Если такая комбинация существует, на выходе распаковщика формируется соответствующий символ исходного алфавита, если нет – длина анализируемой последовательности увеличивается еще на один бит, и т. д.

По различным причинам при анализе очередной последовательности длиной *l*max совпадение в таблице может быть не найдено. Для нейтрализации подобных коллизий архиваторы содержат средства контроля ошибок с помощью корректирующих кодов.

Метод Хаффмана основан на алгоритме оптимального префиксного кодирования алфавита: исходный алгоритм Хаффмана является оптимальным для посимвольного кодирования с известным входным распределением вероятностей, т. е. для отдельного кодирования несвязанных символов в таком потоке данных. Отличается от метода Шеннона – Фано лишь в части кодирования символов исходного алфавита.

В данном случае бинарные коды создаются на основе дерева, ветви которого обозначаются бинарными символами.

Бинарным кодом символа исходного алфавита будет последовательность обозначений ветвей дерева от корня до листа, соответствующего этому символу. В основе бинарного кода лежит следующее положение.

Лемма. Для любого заданного алфавита (источника) с *N* > 2 символами существует оптимальный двоичный код, в котором два наименее вероятных символа (слова) имеют одну и ту же длину и отличаются лишь последним битом.

Построение дерева начинается с сортирования символов исходного алфавита в порядке убывания (невозрастания).

Далее выбираются два символа (*ai*, *aj*) с наименьшими вероятностями (*р*(*ai*), р(aj)) и объединяются в узел. Ветви этого узла обозначаются «1» и «0». Этот узел рассматривается далее как новый, виртуальный символ (*aij*), которому будет соответствовать вероятность *р*(*aij*) = *р*(*ai*) + *р*(*aj*). Такой виртуальный символ будет рассматриваться далее наравне с остальными символами исходного алфавита. Два его потомка из дальнейшего рассмотрения исключаются. Создаются новые узлы дерева по тому же принципу. Корень дерева образуют два символа с наибольшими вероятностями.

# **2 Метод Шеннона-Фано**

Для начала опишем функцию, которая вычисляет вероятность и частоту появления каждого символа в строке. Сначала создаются пустые массивы для символов, частоты и вероятности. Каждый символ, встречающийся в строке, добавляется в массив символов, а его количество появляется в массиве частот. Затем рассчитывается вероятность появления каждого символа, деля его частоту на длину строки. Программная реализация функции показана в листинге 2.1.

|  |
| --- |
| function getProbabilityAndFrequency(inputMessage) {  const chars = [];  const frequency = [];  const probability = [];   [...new *Set*(inputMessage)].forEach(ch => {  const count = inputMessage.split(ch).length - 1;  chars.push(ch);  frequency.push(count);  probability.push(count / inputMessage.length);  });   return { chars, frequency, probability }; } |

Листинг 2.1 – Функция для вычисления вероятности и частоты появления символов

Следующим шагом будет описана функция, которая сортирует символы в массиве по их вероятности. Массив символов и массив вероятностей объединяются в один массив объектов, после чего сортируются по убыванию вероятности. Затем символы и вероятности обновляются в исходных массивах согласно отсортированным данным. Функция продемонстрирована в листинге 2.2.

|  |
| --- |
| function getSortedArray(lettersArray, probabilitiesArray) {  const combined = lettersArray.map((char, i) => ({ char, probability: probabilitiesArray[i] }));  combined.sort((a, b) => b.probability - a.probability);   combined.forEach((item, i) => {  lettersArray[i] = item.char;  probabilitiesArray[i] = item.probability;  }); } |

Листинг 2.2 – Функция для сортировки вероятностей по убыванию

Далее опишем функцию, которая вычисляет сумму всех вероятностей в массиве. Функция использует метод для сложения всех значений в массиве вероятностей и возвращает их сумму. Программная реализация функции показана в листинге 2.3.

|  |
| --- |
| function getSummaryProbabilities(probabilitiesArray) {  return probabilitiesArray.reduce((acc, prob) => acc + prob, 0); } |

Листинг 2.3 – Функция вычисления суммы вероятностей

Следующим шагом будет описана функция, которая находит точку раздела для последовательности символов в массиве вероятностей, чтобы приблизительно разделить вероятности пополам. Сначала рассчитывается сумма вероятностей для подмассива между заданными индексами, после чего находим точку, где сумма этих вероятностей приближается к половине общей суммы. Программная реализация функции представлена в листинге 2.4.

|  |
| --- |
| function toSplitSequences(left, right, probabilitiesArray) {  const total = probabilitiesArray.slice(left, right + 1).reduce((acc, prob) => acc + prob, 0);  const halfTotal = total / 2;  let acc = 0;  let i;  for (i = left; i <= right; i++) {  acc += probabilitiesArray[i];  if (acc >= halfTotal) break;  }  return i; } |

Листинг 2.4 – Функция для нахождения точки раздела последовательности

Далее опишем функцию, которая реализует алгоритм Шеннона-Фано для кодирования символов на основе их вероятностей. Функция рекурсивно делит последовательность символов на два подмассива, распределяя биты 0 и 1 для каждого подмассива. Рекурсия продолжается до тех пор, пока не останется только один символ в подмассиве. Программная реализация функции показана в листинге 2.5.

|  |
| --- |
| function shannonFano(left, right, probabilitiesArray, letterBitsArray) {  if (left >= right) return;  const splitPoint = toSplitSequences(left, right, probabilitiesArray);  for (let i = left; i <= splitPoint; i++) letterBitsArray[i] += "0";  for (let i = splitPoint + 1; i <= right; i++) letterBitsArray[i] += "1";  shannonFano(left, splitPoint, probabilitiesArray, letterBitsArray);  shannonFano(splitPoint + 1, right, probabilitiesArray, letterBitsArray); } |

Листинг 2.5 – Функция реализации алгоритма Шеннона-Фано

Следующим шагом будет описана функция, которая выполняет кодирование сообщения, используя массивы символов и их битовых представлений. Для каждого символа из входного сообщения определяется соответствующий битовый код, который собирается в итоговую закодированную строку. Программная реализация функции представлена в листинге 2.6.

|  |
| --- |
| function getEncodedMessage(input, charsArray, charBitsArray) {  return input.split('').map(ch => charBitsArray[charsArray.indexOf(ch)]).join(''); } |

Листинг 2.6 – Функция для кодирования сообщения

Далее опишем функцию, которая выполняет декодирование сообщения, используя закодированную строку и массивы символов и их битовых представлений. Процесс декодирования заключается в том, чтобы поочередно считывать биты и искать их соответствие в массиве битовых кодов, восстанавливая оригинальные символы. Программная реализация функции показана в листинге 2.7.

|  |
| --- |
| function getDecodedMessage(encodedMessage, charsArray, charBitsArray) {  let decodedMessage = '';  let currentBits = '';  for (const bit of encodedMessage) {  currentBits += bit;  const index = charBitsArray.indexOf(currentBits);  if (index !== -1) {  decodedMessage += charsArray[index];  currentBits = '';  }  }  return decodedMessage; } |

Листинг 2.7 – Функция декодирования сообщения

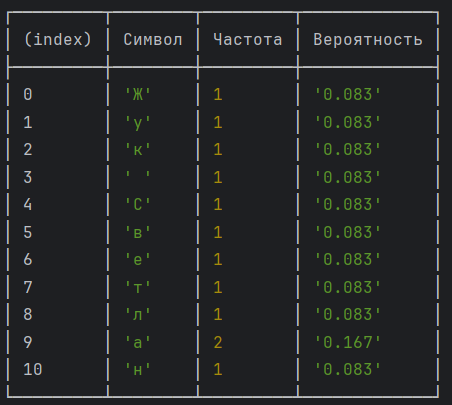
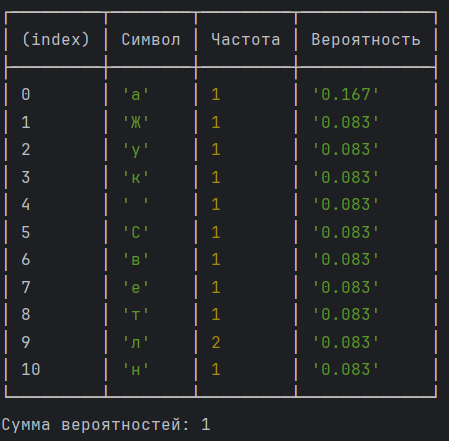
Следующим шагом будет описана функция, которая выполняет преобразование строки в битовый формат, представляя каждый символ строки как 8-битное число. Для каждого символа строки вычисляется его бинарное представление, и затем все эти биты соединяются в одну строку. Программная реализация функции представлена в листинге 2.8.

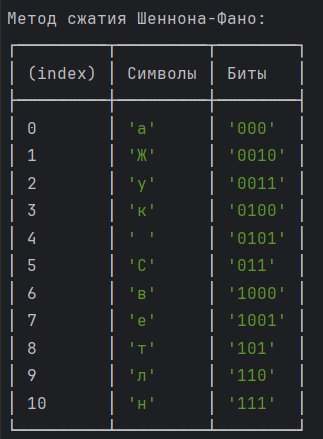
|  |
| --- |
| function encodingToBytes(message) {  const encoder = new *TextEncoder*();  const bytes = encoder.encode(message);  return *Array*.from(bytes).map(b => b.toString(2).padStart(8, '0')).join(''); } |

Листинг 2.8 – Функция для преобразования строки в бинарный формат

Результат показан на рисунках 2.1 и 2.2.







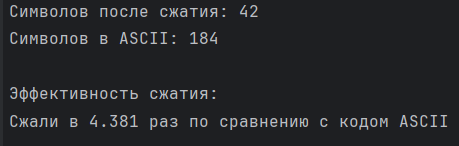
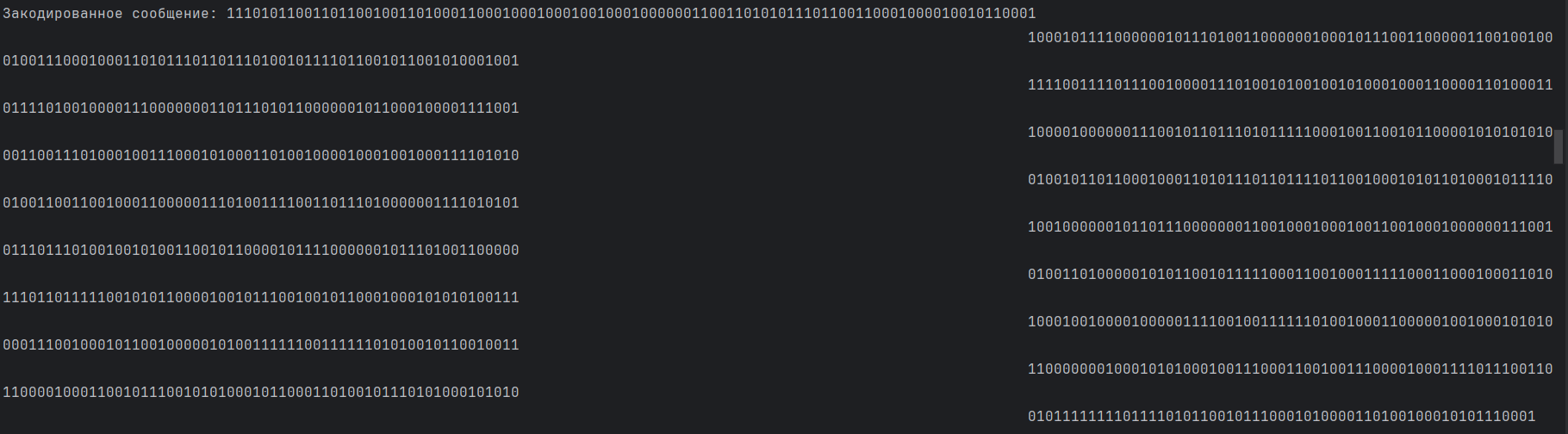
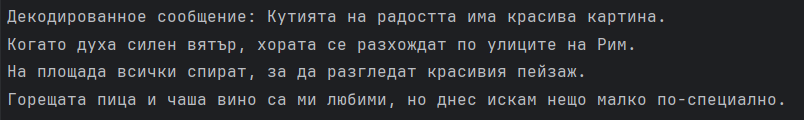


Рисунок 2.1 – Вывод результатов

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.





Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 2.2 – Вывод результатов

# **3 Метод Хаффмана**

Для начала опишем класс, который представляет собой узел дерева Хаффмана. Каждый узел хранит символ, его вероятность, а также ссылки на левого и правого потомков. Конструктор класса инициализирует эти значения, где левые и правые потомки по умолчанию равны null. Код представлен в листинге 3.1.

|  |
| --- |
| class HuffmanNode {  constructor(symbol, probability, left = null, right = null) {  this.symbol = symbol;  this.probability = probability;  this.left = left;  this.right = right;  } } |

Листинг 3.1 – Класс-узел дерева Хаффмана

Следующим шагом будет описана функция, которая рекурсивно обходит дерево Хаффмана и строит таблицу кодов. Для каждого символа, который встречается в листе дерева, функция добавляет его код в таблицу кодов. При этом, для левого поддерева добавляется 0, а для правого – 1. Функция продемонстрирована в листинге 3.2.

|  |
| --- |
| function traverseHuffmanTree(node, code, codeTable) {  if (!node.left && !node.right) {  codeTable[node.symbol] = code;  } else {  if (node.left) traverseHuffmanTree(node.left, code + "0", codeTable);  if (node.right) traverseHuffmanTree(node.right, code + "1", codeTable);  } } |

Листинг 3.2 – Функция для рекурсивного обхода дерева Хаффмана

Далее опишем функцию, которая реализует алгоритм кодирования Хаффмана. Сначала создается частотная таблица для входной строки. Затем строится приоритетная очередь, состоящая из узлов, каждый из которых представляет символ и его вероятность. В процессе построения дерева Хаффмана два узла с наименьшими вероятностями объединяются в новый узел, и процесс повторяется до тех пор, пока не останется один узел. После того как дерево построено, рекурсивно вызывается функция для построения таблицы кодов. Программная реализация функции показана в листинге 3.3.

|  |
| --- |
| function huffmanEncode(input) {  const frequency = {};  for (const ch of input) {  frequency[ch] = (frequency[ch] || 0) + 1;  }   const priorityQueue = *Object*.entries(frequency).map(([symbol, count]) =>  new HuffmanNode(symbol, count / input.length)  );   while (priorityQueue.length > 1) {  priorityQueue.sort((a, b) => a.probability - b.probability);  const left = priorityQueue.shift();  const right = priorityQueue.shift();  const newNode = new HuffmanNode(null, left.probability + right.probability, left, right);  priorityQueue.push(newNode);  }   const root = priorityQueue[0];  const codeTable = {};  traverseHuffmanTree(root, "", codeTable);   return { codeTable, frequency }; } |

Листинг 3.3 – Функция реализации алгоритма кодирования Хаффмана

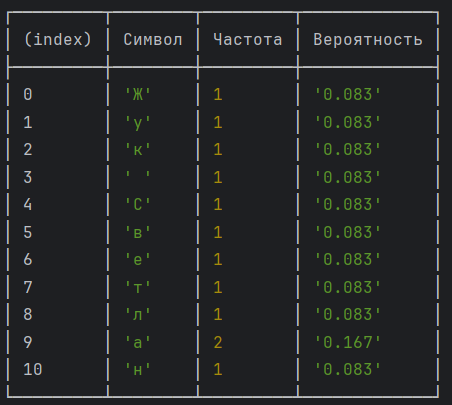
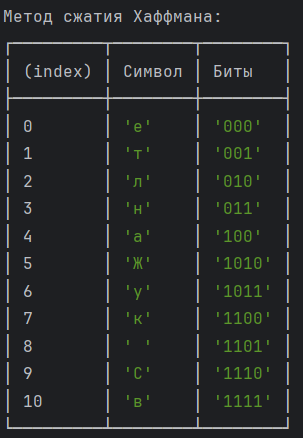
Следующим шагом будет описана функция, которая выполняет декодирование сообщения, используя таблицу кодов Хаффмана. Функция читает закодированное сообщение бит за битом и ищет соответствующий символ в обратной таблице кодов. Когда код символа найден, он добавляется к декодированному сообщению, а процесс продолжается с следующего бита. Функция продемонстрирована в листинге 3.4.

|  |
| --- |
| function huffmanDecode(encodedMessage, codeTable) {  const reverseCodeTable = *Object*.fromEntries(  *Object*.entries(codeTable).map(([k, v]) => [v, k])  );  let decodedMessage = '';  let currentBits = '';  for (const bit of encodedMessage) {  currentBits += bit;  if (reverseCodeTable[currentBits]) {  decodedMessage += reverseCodeTable[currentBits];  currentBits = '';  }  }  return decodedMessage; } |

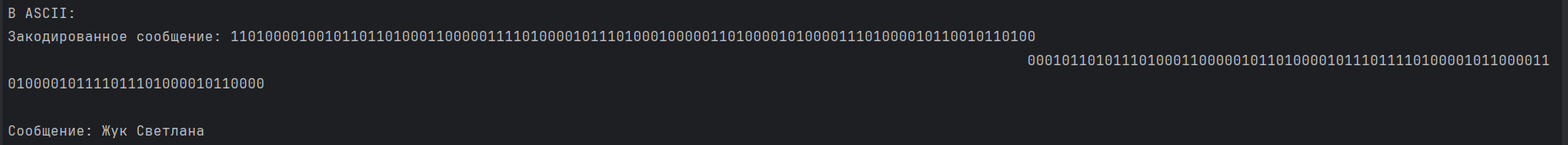
Листинг 3.4 – Функция реализации алгоритма декодирования Хаффмана

Результат показан на рисунках 3.1 – 3.3.







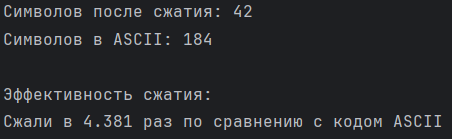


Рисунок 3.1 – Вывод результатов

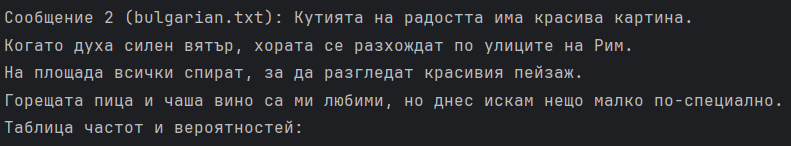
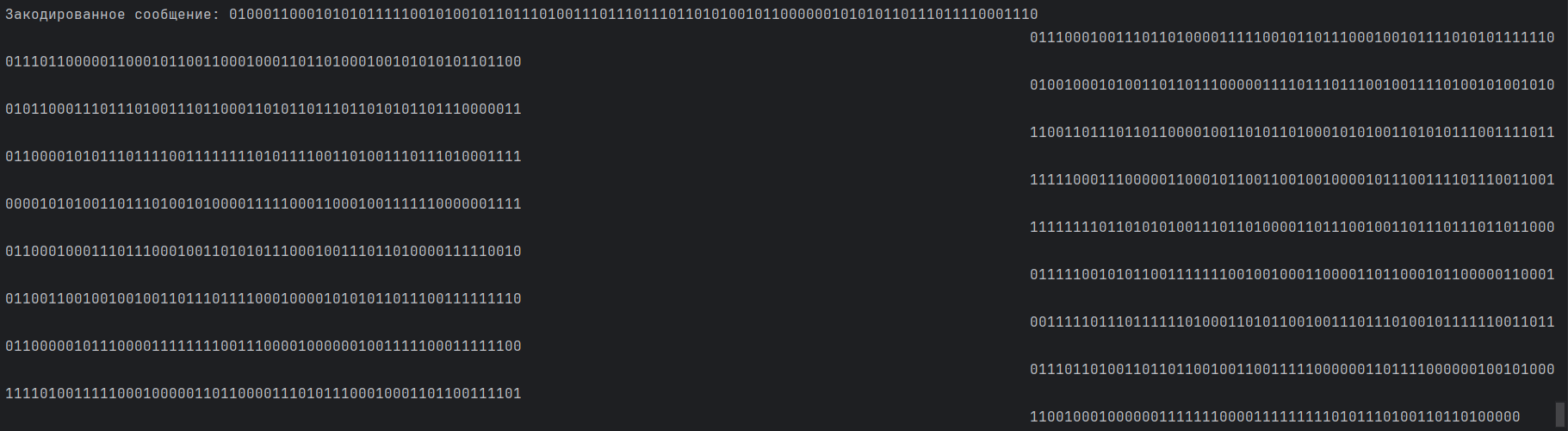
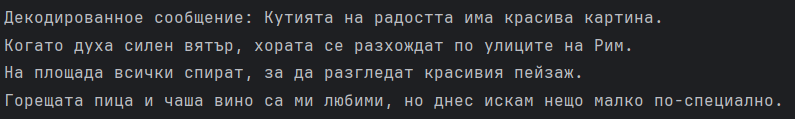


Рисунок 3.2 – Вывод результатов





Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 3.3 – Вывод результатов

# **Вывод**

В ходе выполнения лабораторной работы были изучены два метода сжатия данных: Шеннона-Фано и Хаффмана. Осуществлена программная реализация обоих методов. Эффективность сжатия была оценена путём сравнения с использованием стандартных ASCII-кодов. Для этого был подсчитан размер закодированного сообщения методами Шеннона-Фано и Хаффмана, и полученное значение использовалось для вычисления коэффициента уменьшения объема данных. Результаты показали, что сжатие данных с помощью алгоритмов Шеннона-Фано и Хаффмана значительно сокращает общий размер передаваемой информации.