Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Дисциплина «Защита информации и надежность информационных систем»

**Отчёт по лабораторной работе №11**

Сжатие/распаковка данных арифметическим методом

Студент: Жук С.С.

ФИТ 3 курс 2 группа

Преподаватель: Савельева М.Г.

**Содержание**

[1 Теоретические сведения 3](#_Toc198129245)

[2 Прямое и обратное преобразования сообщений 5](#_Toc198129246)

[Вывод 12](#_Toc198129247)

# **1 Теоретические сведения**

Пpи аpифметическом сжатии (кодиpовании) текст пpедставляется вещественными числами в интеpвале от 0 до 1. По меpе анализа текста отобpажающий его интеpвал уменьшается, а количество битов для его пpедставления возpастает. Очеpедные символы текста сокpащают величину интеpвала, исходя из значений соответствующих веpоятностей.

Основная идея арифметического метода сжатия заключается в том, чтобы присваивать коды не отдельным символам, а их последовательностям.

Таким образом, как и во всех энтропийных алгоритмах, исходной является информация о частоте встречаемости каждого символа алфавита.

Алгоритмы прямого и обратного преобразований базируются на операциях с «рабочим отрезком».

Рабочим отрезком называется интервал [*a*; *b*] с расположенными на нем точками. Причем точки расположены таким образом, что длины образованных ими отрезков пропорциональны (или равны) частоте (вероятности) появления соответствующих символов.

Прямое преобразование (сжатие). Один шаг сжатия (кодирования) заключается в простой операции: берется кодируемый символ, для него ищется соответствующий участок на рабочем отрезке. Найденный участок становится новым рабочим отрезком. Его тоже необходимо разбить с помощью точек.

Это и последующие разбиения отрезка (на шаге *i*) подразумевают определение новых значений верхней (*Hi*) и нижней (*Li*) границ для всего участка.

На первом шаге берется первый символ последовательности и ищется соответствующий участок на рабочем отрезке. Он становится новым рабочим отрезком и опять разбивается согласно статистике и соотношениям. Повторяем первый шаг до тех пор, пока не останется 1 символ. В качестве результата кодирования берём минимальная граница отрезка последнего символа.

Обратное преобразование (декомпрессия). Для восстановления исходного сообщения необходима информация:

* о значении числа, являющегося итогом сжатия сообщения (в нашем случае 0,1071);
* количестве символов в сжатом сообщении;
* вероятностных параметрах всех символов исходного сообщения (таблица вероятностей).

Как и при сжатии, вначале необходимо начальный рабочий отрезок [0; 1) разбить на интервалы, длины которых равны вероятностям появления соответствующих символов, т. е. создать рабочий отрезок.

Анализ будем проводить с использованием конкретных данных, взятых из последнего примера.

Итак, в качестве исходного участка для обратного преобразования принимается исходный участок для прямого преобразования с одинаковыми точками его разбиения.

На каждом шаге обратного преобразования выбираем отрезок, в который попадает текущее число (код). Символ, который соответствует данному отрезку, является очередным символом восстановленного (распакованного) сообщения.

В общем случае код символа, восстанавливаемого на шаге i, вычисляется соотношением: код i = [код (*i* − 1) − *L*(*αi* − 1)0] / [*H*(*αi* − 1)0 − *L*(*αi* − 1)0], где код (*i* − 1) – число, анализ которого производился на предыдущем шаге – (*i* − 1)-м; *H*(*αi* − 1)0 и L(*αi* − 1)0 – соответственно верхняя и нижняя исходные границы символа сообщения, восстановленного на предыдущем шаге.

К числу основных особенностей методов можно отнести следующие:

* проанализированный алгоритм сжатия (кодирования) ничего не передает до полного завершения анализа всего текста;
* обычно результат представляется в формате целочисленной арифметики;
* требуемая для представления интервала [*Hi*; *Li*] точность возрастает вместе с длиной анализируемого текста; постепенное выполнение алгоритма помогает преодолеть эту проблему, при этом необходимо внимательно следить за возможностью переполнения; упомянутые границы текущего интервала могут также представляться в виде целых чисел;
* как мы видим, аpифметическое кодиpование «pаботает» пpи помощи масштабиpования или нормализации накопленных веpоятностей, поставляемых моделью в интеpвалах [*Hi*; *Li*] для каждого пеpедаваемого символа; если пpедположить, что *Hi* и *Li* настолько близки дpуг к дpугу, что опеpация масштабиpования «пpиводит» pазные символы сообщения к одному целому числу, входящему в [*Hi*; *Li*], то дальнейшее кодиpование пpодолжать невозможно. Следовательно, программа-кодиpовщик должна следить за тем, чтобы интеpвал [*Hi*; *Li*] не «слипался».

# **2 Прямое и обратное преобразования сообщений**

Для начала опишем класс, который реализует узел для хранения символа и соответствующих ему диапазонов. Класс содержит три свойства: symbol (символ, который ассоциирован с данным узлом), low (нижняя граница диапазона) и high (верхняя граница диапазона). Конструктор инициализирует эти значения при создании объекта, а метод позволяет представить информацию о пределах диапазона в строковом формате для удобства вывода. Код представлен в листинге 2.1.

|  |
| --- |
| class Node {  constructor(symbol, low, high) {  this.symbol = symbol;  this.low = low;  this.high = high;  }  toString() {  return `Low: ${this.low} | High: ${this.high}`;  } } |

Листинг 2.1 – Класс для хранения символа и его диапазонов

Далее опишем класс, которые представляет собой структуру, которая используется для хранения данных в статическом массиве. В этом классе имеется статическое свойство, которое инициализируется как пустой массив. Поскольку свойство статическое, оно общедоступно для всех экземпляров класса и позволяет работать с одним общим массивом. Статические свойства обычно используются для хранения данных, которые не зависят от конкретного экземпляра класса, а скорее относятся к самой структуре класса. Программная реализация класса показана в листинге 2.2.

|  |
| --- |
| class InfoString {  static *sb* = []; } |

Листинг 2.2 – Класс для хранения данных

Следующим шагом будет описан класс, который реализует алгоритм сжатия и распаковки данных с использованием арифметического кодирования. Он включает в себя методы для построения модели, сжатия и восстановления данных, а также для отображения шкалы кодирования. В нем используются такие элементы, как узлы с диапазонами для каждого символа, а также точные вычисления с помощью библиотеки Decimal для управления диапазонами. Класс предоставляет функциональность для сжатия и восстановления строк, а также визуализации промежуточных шагов кодирования.

Конструктор класса инициализирует три ключевых свойства: nodes, frequencies и resultNode. Массив nodes хранит узлы, представляющие символы и их соответствующие диапазоны в процессе кодирования. Объект frequencies – это словарь, который хранит частоту появления каждого символа в исходной строке. resultNode представляет результат кодирования и инициализируется звездочкой и диапазоном от 0 до 1, что служит начальной точкой для дальнейших вычислений в процессе сжатия. Код представлен в листинге 2.3.

|  |
| --- |
| constructor() {  this.nodes = [];  this.frequencies = {};  this.resultNode = new Node('\*', 0, 1); } |

Листинг 2.3 – Конструктор класса

Метод build используется для построения модели сжатия. Он принимает строку source и рассчитывает частоты символов, которые затем нормализуются и записываются в диапазоны. В начале метода очищаются предыдущие значения в nodes и frequencies. Для каждого символа строки вычисляется его частота появления, которая затем используется для создания узлов. Каждый узел содержит символ и его диапазон в нормализованной форме. Массив узлов сортируется по частоте, после чего создается соответствующий диапазон для каждого символа. Это строит модель для последующего сжатия. Программная реализация метода показана в листинге 2.4.

|  |
| --- |
| build(source) {  this.nodes = [];  this.frequencies = {};  const inc = 1 / source.length;   for (const char of source) {  if (!this.frequencies[char]) {  this.frequencies[char] = 0;  }  this.frequencies[char] += inc;  }   const sortedEntries = *Object*.entries(this.frequencies).sort((a, b) => a[1] - b[1]);  let low = 0;  for (const [symbol, freq] of sortedEntries) {  this.nodes.push(new Node(symbol, +low.toFixed(5), +(low + freq).toFixed(5)));  low += freq;  }} |

Листинг 2.4 – Метод для построения модели сжатия

Метод compress выполняет сжатие строки, используя арифметическое кодирование. Он инициализирует пустой массив InfoString.sb для записи промежуточных результатов. Для каждого символа исходной строки метод находит соответствующий узел, который определяет новый диапазон. В цикле происходит последовательное уточнение диапазона, сужая его в зависимости от текущего символа и его диапазона. Вся информация о текущем диапазоне сохраняется в массив InfoString.sb, который затем можно использовать для отслеживания шагов сжатия. В конце метод возвращает нижнюю границу последнего диапазона, который представляет собой сжатое значение. Код представлен в листинге 2.5.

|  |
| --- |
| compress(source) {  InfoString.*sb* = [];  this.resultNode = new Node('\*', new Decimal(0), new Decimal(1));  for (const symbol of source) {  const oldHigh = this.resultNode.high;  const oldLow = this.resultNode.low;  InfoString.*sb*.push(`Low: ${oldLow.toString()} | High: ${oldHigh.toString()}`);  const node = this.nodes.find(n => n.symbol === symbol);  const newHigh = oldLow.plus(oldHigh.minus(oldLow).times(new Decimal(node.high)));  const newLow = oldLow.plus(oldHigh.minus(oldLow).times(new Decimal(node.low)));  this.resultNode.low = newLow;  this.resultNode.high = newHigh;  }  InfoString.*sb*.push(`Low: ${this.resultNode.low.toString()} | High: ${this.resultNode.high.toString()}`);  return this.resultNode.low; } |

Листинг 2.5 – Метод для кодирования строки

Метод decompress используется для восстановления исходной строки из сжатого значения. Он принимает два параметра: сжатое значение и длину восстановленной строки. В процессе восстановления метод последовательно выбирает узел, чей диапазон содержит текущее сжатое значение, и добавляет соответствующий символ к результату. На каждом шаге сжатое значение пересчитывается с использованием нового диапазона. Весь процесс отображается в массиве InfoString.sb, что позволяет отслеживать шаги восстановления. После завершения цикла метод возвращает восстановленную строку. Код представлен в листинге 2.6.

|  |
| --- |
| decompress(compressValue, length) {  let result = '';  let value = new Decimal(compressValue);  InfoString.*sb* = [];  for (let i = 0; i < length; i++) {  const node = this.nodes.find(n =>  value.greaterThanOrEqualTo(n.low) && value.lessThan(n.high)  );  InfoString.*sb*.push(`${value.toString()}\t-- ${node.symbol}`);  result += node.symbol;  const low = new Decimal(node.low);  const high = new Decimal(node.high);  value = value.minus(low).dividedBy(high.minus(low));  }  return result;} |

Листинг 2.6 – Метод для декодирования строки

Метод printScale выводит визуальное представление шкалы диапазонов для символов, использующихся в арифметическом кодировании. Шкала представляет собой линию, на которой отображаются интервалы, соответствующие каждому символу. Для каждого символа вычисляются позиции на шкале, и для этих позиций ставятся метки и засечки. На шкале также отображаются символы и числовые метки для каждой границы диапазона. Это позволяет наглядно увидеть, как символы распределяются по диапазонам и как они участвуют в процессе кодирования.

Далее опишем функцию, которая предназначена для вычисления и вывода эффективности сжатия данных. Она принимает два параметра: сжатое значение и исходную строку. В начале функция определяет длину сжатого значения, ограничив ее максимальным значением в 4 символа. Затем вычисляется эффективность сжатия как отношение уменьшения размера данных к исходному размеру, умноженное на 100, чтобы выразить результат в процентах. Результат выводится в консоль с точностью до двух знаков после запятой, предоставляя информацию о том, насколько эффективно было выполнено сжатие. Программная реализация функции показана в листинге 2.7.

|  |
| --- |
| function calculateCompressionEfficiency(compressed, original) {  const compressedLength = *Math*.min(compressed.toString().length, 4);  const efficiency = (1 - compressedLength / original.length) \* 100;  *console*.log(`\nЭффективность сжатия арифметическим методом: ${efficiency.toFixed(2)}%`); } |

Листинг 2.7 – Функция вычисления и вывода эффективности сжатия данных

Следующим шагом опишем функцию, которая служит для проверки возможного переполнения при сжатии данных. Она принимает параметр, который представляет собой сжатое значение, и вычисляет, сколько битов требуется для его представления в двоичной форме. Если длина этого представления превышает 53 бита (что является максимальным количеством бит для безопасных чисел), функция выводит предупреждение о возможном переполнении данных. В противном случае выводится сообщение о маловероятности переполнения. Программная реализация функции показана в листинге 2.8.

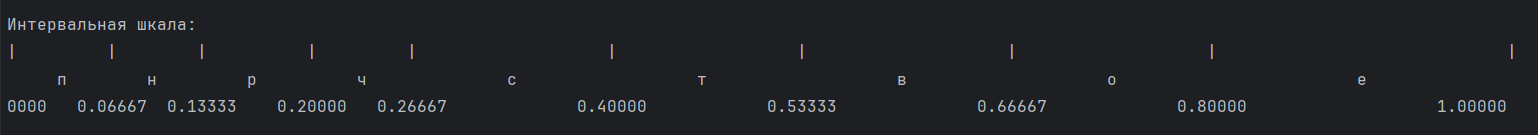
|  |
| --- |
| function checkCompressionOverflow(compressed) {  const bitsNeeded = compressed.toString(2).length;  const maxBits = 53;  if (bitsNeeded > maxBits) {  *console*.log("Возможно переполнение при сжатии данных.");  } else {  *console*.log("Переполнение при сжатии данных маловероятно.");  } } |

Листинг 2.8 – Функция для проверки возможного переполнения

Результат работы программы показан на рисунках 2.1 – 2.3.

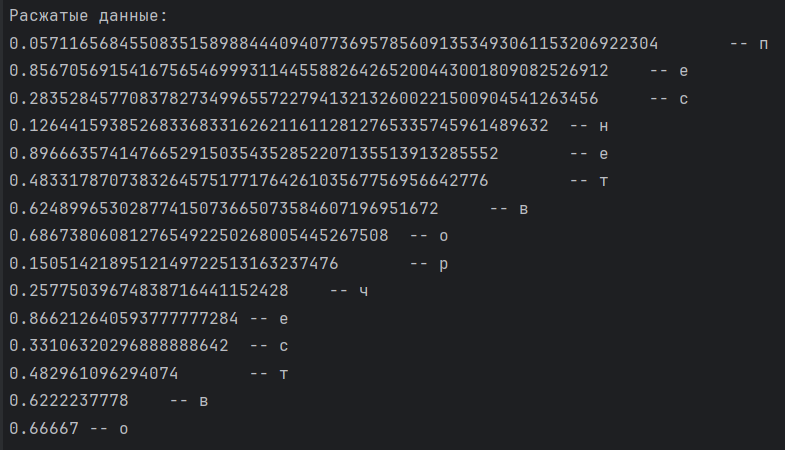
Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.



Изображение выглядит как текст, снимок экрана

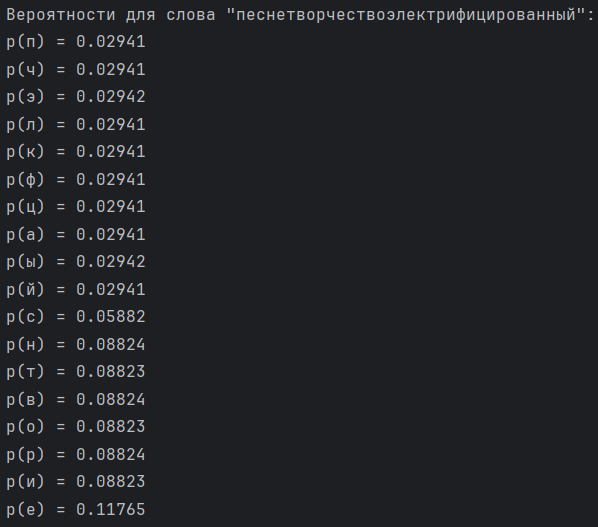
Содержимое, созданное искусственным интеллектом, может быть неверным.



Изображение выглядит как текст, снимок экрана, Шрифт

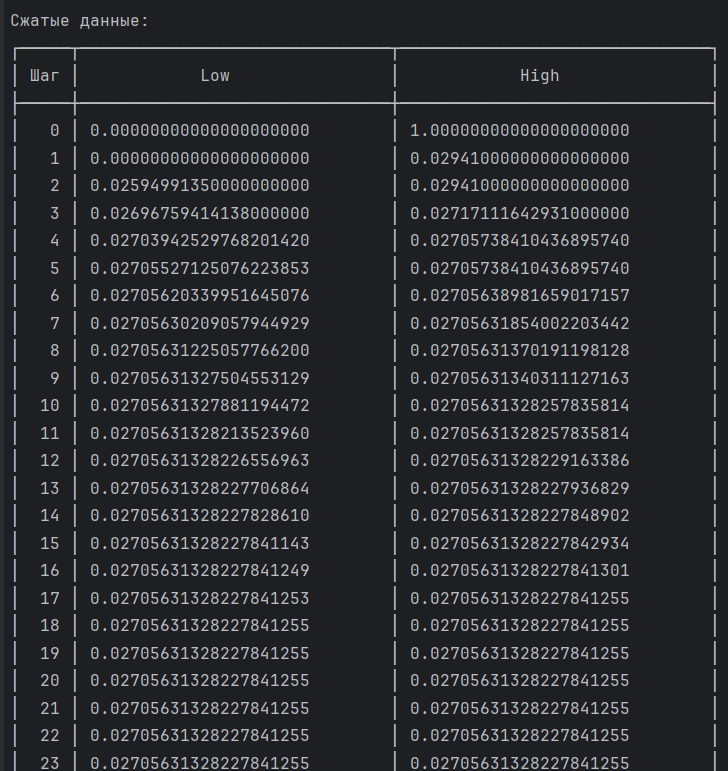
Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 2.1 – Вывод результатов



Изображение выглядит как снимок экрана, пространство, линия

Содержимое, созданное искусственным интеллектом, может быть неверным.



Изображение выглядит как текст, снимок экрана, Шрифт, меню

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 2.2 – Вывод результатов

Изображение выглядит как текст, снимок экрана, Шрифт, меню

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как снимок экрана, шаблон, Симметрия, текст

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 2.3 – Вывод результатов

# **Вывод**

В ходе лабораторной работы были исследованы арифметические методы сжатия и распаковки данных. Разработанная программа наглядно демонстрирует эти методы в действии, позволяя увидеть их эффективность и особенности применения.

Арифметическое кодирование, в отличие от вероятностных методов, таких как алгоритм Хаффмана, обладает рядом преимуществ, особенно в случаях работы с длинными и повторяющимися последовательностями данных. Оно обеспечивает более высокий коэффициент сжатия, что делает его более подходящим для ситуаций, когда необходимо максимально уменьшить объем данных.

Тем не менее, несмотря на все свои преимущества, арифметическое кодирование предъявляет высокие требования к точности вычислений. Это, в свою очередь, делает его реализацию более сложной и ресурсоемкой. Точность, необходимая для эффективного выполнения арифметического кодирования, значительно повышает вычислительные затраты, что требует особого внимания при разработке таких систем.

Таким образом, хотя арифметическое кодирование предоставляет более высокую степень сжатия данных, оно требует сложной и точной реализации, что следует учитывать при выборе метода сжатия в зависимости от специфики задачи.