Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Защита информации и надёжность информационных систем

Студент: Лопатнюк П.В.

ФИТ 3 курс 1 группа

Преподаватель: Нистюк О.А.

Минск 2025

# Лабораторная работа № 11

# СЖАТИЕ/РАСПАКОВКА ДАННЫХ АРИФМЕТИЧЕСКИМ МЕТОДОМ

**Цель:** приобретение практических навыков использования арифметических методов сжатия/распаковки данных.

**Задачи:**

1. Закрепить теоретические знания по алгебраическому описанию и использованию арифметических методов сжатия/распаковки (архивации/разархивации) данных.
2. Разработать приложение для реализации арифметических методов.
3. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

# Теоретические сведения

Арифметическое кодирование представляет собой метод сжатия данных без потерь, который отличается от вероятностных (префиксных) методов, таких как код Хаффмана. В то время как префиксные коды присваивают каждому символу отдельный битовый код и наиболее эффективны, когда вероятности символов являются степенями двойки (1/2, 1/4, 1/8 и т.д.), арифметическое кодирование способно более точно использовать статистику символов, даже если их вероятности не так удобны.

Основная идея арифметического кодирования заключается в том, чтобы представить всю последовательность входных символов (сообщение) одним вещественным числом, находящимся в интервале [0, 1).

Процесс кодирования:

1. Инициализация: Начинаем с «рабочего отрезка» [a, b], который изначально равен [0, 1).
2. Модель вероятностей: Для каждого символа алфавита известна его вероятность (частота) появления. На основе этих вероятностей начальный рабочий отрезок [0, 1) разбивается на подынтервалы, пропорциональные вероятностям символов. Например, если символ 'м' имеет вероятность 0.1666, то ему будет соответствовать начальный подынтервал [0, 0.1666).
3. Итеративное сужение: Для каждого кодируемого символа αj из входной последовательности выбирается соответствующий ему подынтервал внутри текущего рабочего отрезка [Li-1, Hi-1].

Границы этого нового подынтервала [Li(αj), Hi(αj)] вычисляются по формулам:  
 Li(αj) = Li-1 + (Hi-1 – Li-1) \* L(αj)0  
 Hi(αj) = Li-1 + (Hi-1 – Li-1) \* H(αj)0  
 где L(αj)0 и H(αj)0 — это нижняя и верхняя границы исходного подынтервала символа αj (в рамках отрезка [0, 1)).

Этот новый, более узкий подынтервал становится текущим рабочим отрезком для кодирования следующего символа.

Результат: После обработки всех символов сообщения, любое число из финального рабочего отрезка (часто выбирают его нижнюю границу) является закодированным представлением всего сообщения.

Процесс декодирования:

1. Необходимые данные: Закодированное число, исходная длина сообщения и та же модель вероятностей символов, что использовалась при кодировании.
2. Инициализация: Начинаем с рабочего отрезка [0, 1), разбитого на подынтервалы согласно модели вероятностей.
3. Итеративное восстановление: Определяется, в какой из подынтервалов попадает текущее закодированное число. Символ, соответствующий этому подынтервалу, является очередным восстановленным символом.

Этот подынтервал становится новым рабочим отрезком.

Закодированное число «масштабируется» относительно этого нового рабочего отрезка для следующего шага. Если код\_i-1 — это значение на предыдущем шаге, а символ αi-1 был восстановлен с исходными границами L(αi-1)0 и H(αi-1)0, то новое значение для поиска код\_i вычисляется как: код\_i = (код\_i-1 - L(αi-1)0) / (H(αi-1)0 - L(αi-1)0)

Завершение: Процесс повторяется, пока не будет восстановлено исходное количество символов.

Арифметическое кодирование обрабатывает всё сообщение целиком, прежде чем выдать результат.

Требуется высокая точность вычислений. На практике часто используют целочисленную арифметику и техники масштабирования, чтобы избежать проблем с плавающей точкой и переполнением/исчезновением точности, когда интервал становится слишком мал («слипается»).

Точность представления границ интервала [Hi, Li] должна расти с длиной анализируемого текста.

Арифметическое кодирование, как правило, обеспечивает лучшее сжатие, чем метод Хаффмана, особенно для источников с сильно смещенными или не степенями двойки вероятностями.

# Практическое задание

1. **Разработать авторское приложение в соответствии с целью лабораторной работы.**

Функция для построения таблицы символов, содержащей для каждого символа его вероятность и соответствующий ему кумулятивный диапазон [Low, High) на отрезке [0, 1).

|  |
| --- |
| public static List<SymbolInfo> BuildSymbolInfoTable(Dictionary<char, decimal> charProbabilities)  {  var symbolInfoTable = new List<SymbolInfo>();  decimal cumulativeLow = 0m;  var sortedSymbols = charProbabilities.Keys.OrderBy(c => c).ToList();  for (int i = 0; i < sortedSymbols.Count; i++)  {  char symbol = sortedSymbols[i];  decimal probability = charProbabilities[symbol];  if (i == sortedSymbols.Count - 1 && sortedSymbols.Count > 1)  {  if (cumulativeLow < 1.0m)  {  probability = 1.0m - cumulativeLow;  }  if (probability < 0) probability = 0;  }  else if (sortedSymbols.Count == 1)  {  probability = 1.0m;  }  symbolInfoTable.Add(new SymbolInfo  {  Symbol = symbol,  Probability = probability,  Low = cumulativeLow,  High = cumulativeLow + probability  });  cumulativeLow += probability;  }  if (symbolInfoTable.Any())  {  var lastSymbol = symbolInfoTable.Last();  if (lastSymbol.High != 1.0m && Math.Abs(lastSymbol.High - 1.0m) < 0.000000000000001m)  {  lastSymbol.High = 1.0m;  }  if (lastSymbol.High < 1.0m && sortedSymbols.Count > 0)  {  if (lastSymbol.Low < 1.0m)  {  lastSymbol.High = 1.0m;  lastSymbol.Probability = 1.0m - lastSymbol.Low;  }  }  if (symbolInfoTable.Count == 1)  {  lastSymbol.Low = 0m;  lastSymbol.High = 1.0m;  lastSymbol.Probability = 1.0m;  }  }  return symbolInfoTable;  } |

Листинг 1.1 – Метод для построения таблицы символов.

Символы сортируются (обычно по алфавиту для консистентности). Затем для каждого символа итеративно вычисляются его кумулятивные диапазоны. Нижняя граница диапазона текущего символа равна верхней границе предыдущего (или 0 для первого символа). Верхняя граница равна Low + Probability. Производится коррекция для последнего символа, чтобы его High был точно 1.0m, компенсируя возможные ошибки округления decimal.

Функция для кодирования входной строки текста в одно число типа decimal с использованием таблицы символов.

|  |
| --- |
| public decimal Encode(string textToEncode, List<SymbolInfo> symbolInfoTable, bool verbose = false)  {  decimal low = 0.0m;  decimal high = 1.0m;  if (verbose) Console.WriteLine("\n>>> НАЧАЛО ПРОЦЕССА КОДИРОВАНИЯ <<<");  if (verbose) Console.WriteLine($"Исходный интервал: Low = {low:F28}, High = {high:F28}");  int step = 1;  foreach (char symbol in textToEncode)  {  decimal currentRange = high - low;  SymbolInfo sInfo = symbolInfoTable.FirstOrDefault(s => s.Symbol == symbol);  if (sInfo == null) throw new ArgumentException($"Символ '{symbol}' не найден в таблице вероятностей.");  decimal oldLow = low;  decimal oldHigh = high;  high = low + currentRange \* sInfo.High;  low = low + currentRange \* sInfo.Low;  if (verbose)  {  Console.WriteLine($"\n--- Шаг кодирования {step} ---");  Console.WriteLine($" Кодируемый символ: '{sInfo.Symbol}' (Базовый диапазон: [{sInfo.Low:F6} - {sInfo.High:F6}])");  Console.WriteLine($" Предыдущий интервал: [{oldLow:F28} - {oldHigh:F28}], Диапазон: {currentRange:F28}");  Console.WriteLine($" Новый интервал: [{low:F28} - {high:F28}]");  }  step++;  }  if (verbose) Console.WriteLine("\n>>> КОНЕЦ ПРОЦЕССА КОДИРОВАНИЯ <<<\n");  return low;  } |

Листинг 1.2 – Метод для кодирования входной строки текста.

Инициализируется рабочий диапазон [low, high) как [0.0m, 1.0m). Для каждого символа в кодируемой строке. Находится информация о символе (sInfo) в symbolInfoTable. Текущий диапазон currentRange = high - low пересчитывается. Новые high и low вычисляются по формулам.

Таким образом, рабочий диапазон сужается с каждой итерацией. Финальное значение low является результатом кодирования.

Функция для декодирования числа decimal обратно в исходную строку текста, зная её первоначальную длину и используя таблицу символов.

|  |
| --- |
| public string Decode(decimal encodedValue, int originalLength, List<SymbolInfo> symbolInfoTable, bool verbose = false)  {  StringBuilder decodedText = new StringBuilder();  decimal low = 0.0m;  decimal high = 1.0m;  const decimal EPSILON = 1e-27m;  if (verbose) Console.WriteLine("\n>>> НАЧАЛО ПРОЦЕССА ДЕКОДИРОВАНИЯ <<<");  if (verbose) Console.WriteLine($"Исходное значение для декодирования: {encodedValue:F28}");  if (verbose) Console.WriteLine($"Начальный интервал: Low = {low:F28}, High = {high:F28}");  for (int i = 0; i < originalLength; i++)  {  decimal currentRange = high - low;  if (verbose) Console.WriteLine($"\n--- Шаг декодирования {i + 1} ---");  if (verbose) Console.WriteLine($" Текущий интервал: [{low:F28} - {high:F28}], Диапазон: {currentRange:F28}");  if (currentRange <= 0 || currentRange < EPSILON \* symbolInfoTable.Count \* 1e-5m)  {  Console.WriteLine($" ВНИМАНИЕ: Критически малый или нулевой диапазон ({currentRange}). Декодирование может быть неточным.");  decimal valueInCurrentRangeForEmergency = (currentRange > 0) ? (encodedValue - low) / currentRange : 0;  SymbolInfo emergencySymbol = symbolInfoTable  .OrderBy(s => Math.Abs(valueInCurrentRangeForEmergency - s.Low))  .ThenBy(s => s.Symbol)  .First();  decodedText.Append(emergencySymbol.Symbol);  if (verbose) Console.WriteLine($" Аварийно выбран символ '{emergencySymbol.Symbol}' из-за малого диапазона. Масштаб. значение: {valueInCurrentRangeForEmergency:F28}");  if (currentRange > 0)  {  decimal temp\_high\_emergency = low + currentRange \* emergencySymbol.High;  low = low + currentRange \* emergencySymbol.Low;  high = temp\_high\_emergency;  }  else  {  if (verbose) Console.WriteLine(" Диапазон равен нулю, дальнейшее декодирование невозможно стандартным методом.");  if (i < originalLength - 1 && verbose)  {  Console.WriteLine($" ПРЕДУПРЕЖДЕНИЕ: Цикл будет продолжен, но результаты могут быть неверными.");  }  }  continue;  }  decimal valueInCurrentRange = (encodedValue - low) / currentRange;  if (verbose) Console.WriteLine($" Масштабированное значение (encodedValue - Low) / Range = {valueInCurrentRange:F28}");  SymbolInfo foundSymbol = null;  foreach (var sInfo in symbolInfoTable)  {  bool isMatch = false;  if (valueInCurrentRange >= sInfo.Low - EPSILON)  {  if (Math.Abs(sInfo.High - 1.0m) < EPSILON)  {  if (valueInCurrentRange <= sInfo.High + EPSILON)  {  isMatch = true;  }  }  else  {  if (valueInCurrentRange < sInfo.High - EPSILON)  {  isMatch = true;  }  }  }  if (isMatch)  {  foundSymbol = sInfo;  break;  }  }  if (foundSymbol == null)  {  if (verbose) Console.WriteLine($" ПРЕДУПРЕЖДЕНИЕ: Символ не найден для масштабированного значения {valueInCurrentRange:F28}. EncodedValue: {encodedValue:F28}");  SymbolInfo bestCandidate = null;  decimal minDiff = decimal.MaxValue;  foreach (var sInfoCand in symbolInfoTable)  {  bool isContainedRelaxed = (valueInCurrentRange >= sInfoCand.Low - EPSILON && valueInCurrentRange <= sInfoCand.High + EPSILON);  if (isContainedRelaxed)  {  if (bestCandidate == null) bestCandidate = sInfoCand;  decimal midPoint = sInfoCand.Low + (sInfoCand.High - sInfoCand.Low) / 2;  decimal diff = Math.Abs(valueInCurrentRange - midPoint);  if (diff < minDiff)  {  minDiff = diff;  }  }  }  if (bestCandidate != null)  {  foundSymbol = bestCandidate;  if (verbose) Console.WriteLine($" Аварийно (расширенный поиск) выбран символ '{foundSymbol.Symbol}'");  }  else  {  foundSymbol = symbolInfoTable.OrderBy(s => Math.Abs(valueInCurrentRange - s.Low)).First();  if (verbose) Console.WriteLine($" Аварийно (ближайший по L0) выбран символ '{foundSymbol.Symbol}'");  }  }  if (foundSymbol == null)  {  throw new InvalidOperationException($"Критическая ошибка декодирования: не удалось найти символ на шаге {i + 1}. " +  $"EncodedValue={encodedValue:F28}, Low={low:F28}, High={high:F28}, Range={currentRange:F28}, ValueInCurrentRange={valueInCurrentRange:F28}");  }  decodedText.Append(foundSymbol.Symbol);  if (verbose) Console.WriteLine($" Найден символ: '{foundSymbol.Symbol}' (Базовый диапазон: [{foundSymbol.Low:F6} - {foundSymbol.High:F6}])");  decimal oldLow = low;  high = low + currentRange \* foundSymbol.High;  low = oldLow + currentRange \* foundSymbol.Low;  if (verbose) Console.WriteLine($" Обновленный интервал: [{low:F28} - {high:F28}]");  if (low > high || Math.Abs(low - high) < 1e-28m && i < originalLength - 1)  {  if (verbose) Console.WriteLine($" ПРЕДУПРЕЖДЕНИЕ: Невалидный или слишком узкий интервал после обновления: Low={low:F28}, High={high:F28}. Следующая итерация может быть неточной.");  }  }  if (verbose) Console.WriteLine("\n>>> КОНЕЦ ПРОЦЕССА ДЕКОДИРОВАНИЯ <<<\n");  return decodedText.ToString();  } |

Листинг 1.3 – Метод для декодирования числа decimal.

Инициализируется рабочий диапазон [low, high) как [0.0m, 1.0m). Цикл выполняется originalLength раз:

* Вычисляется текущий диапазон currentRange = high - low.
* Закодированное значение encodedValue масштабируется к текущему диапазону для поиска символа: valueInCurrentRange = (encodedValue - low) / currentRange.
* В symbolInfoTable ищется символ, в чей кумулятивный диапазон [sInfo.Low, sInfo.High) попадает valueInCurrentRange. Применяются поправки на точность (EPSILON) для граничных условий.
* Найденный символ добавляется к результату декодирования.
* Рабочий диапазон [low, high) сужается до диапазона найденного символа.

Включены механизмы обработки ситуаций, когда диапазон становится критически малым или символ не находится стандартным способом из-за проблем точности.

1. **С помощью приложения выполнить прямое и обратное преобразования сообщений в соответствии с таблицей. Каждый студент выполняет задание, состоящее из двух частей. Первая часть предусматривает кодирование/декодирование сообщения, указанного в 2-м столбце, вторая часть – составного сообщения, полученного конкатенацией последовательностей из 2-го столбца, указанных в 3-м столбце.**

Первая часть: кодирование/декодирование сообщения «самообороноспособность».

Вторая часть: кодирование/декодирование составного сообщения, полученного конкатенацией последовательностей «самообороноспособность» и «малосимпатичный».

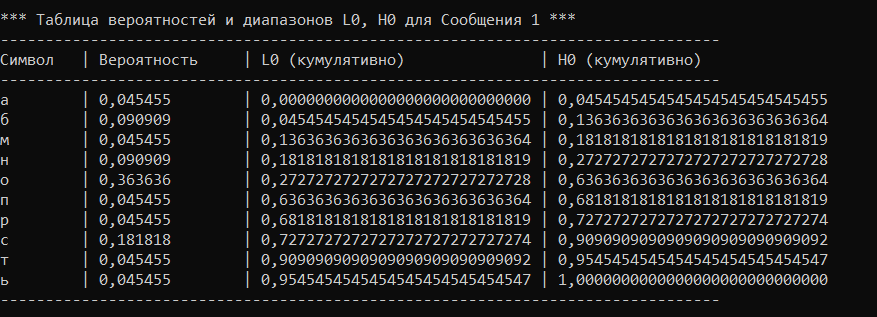
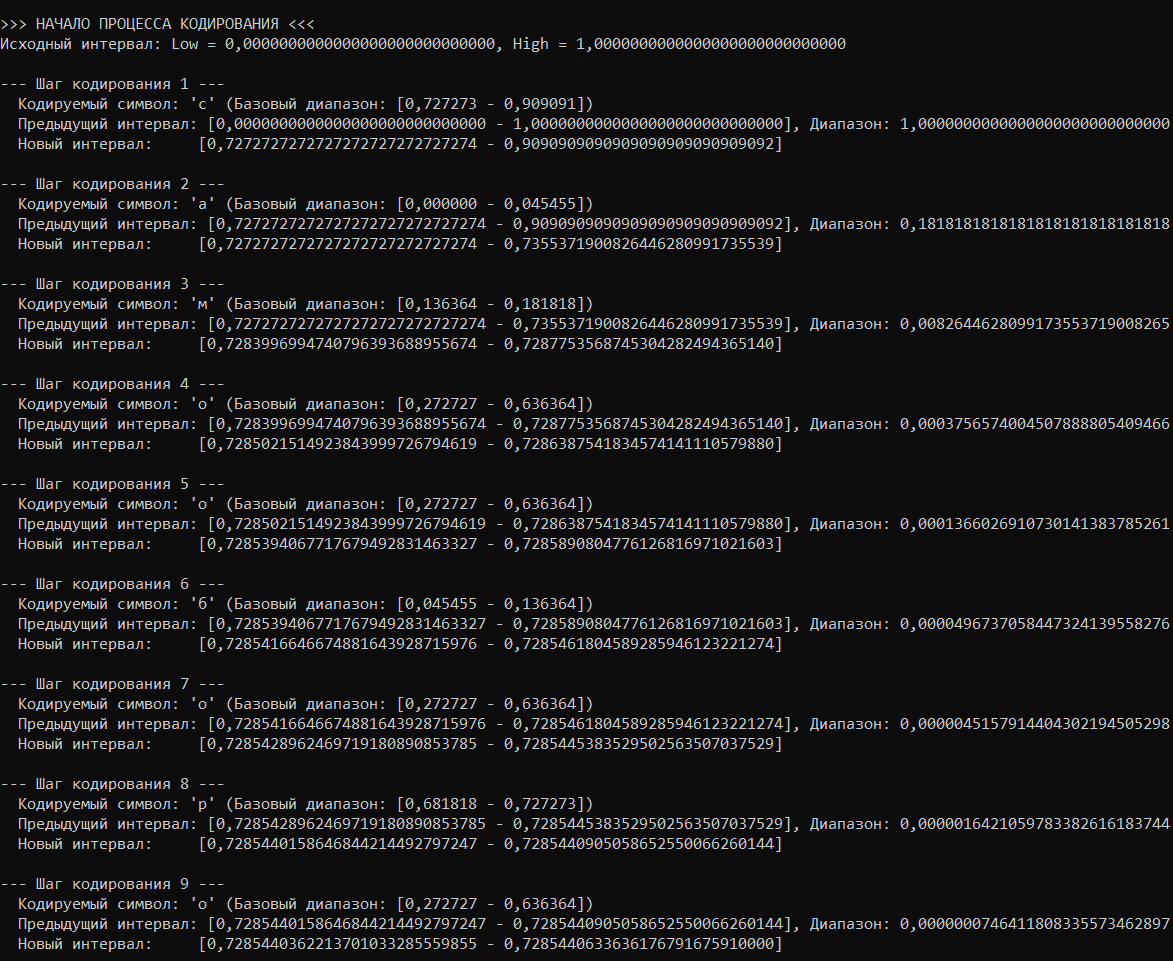
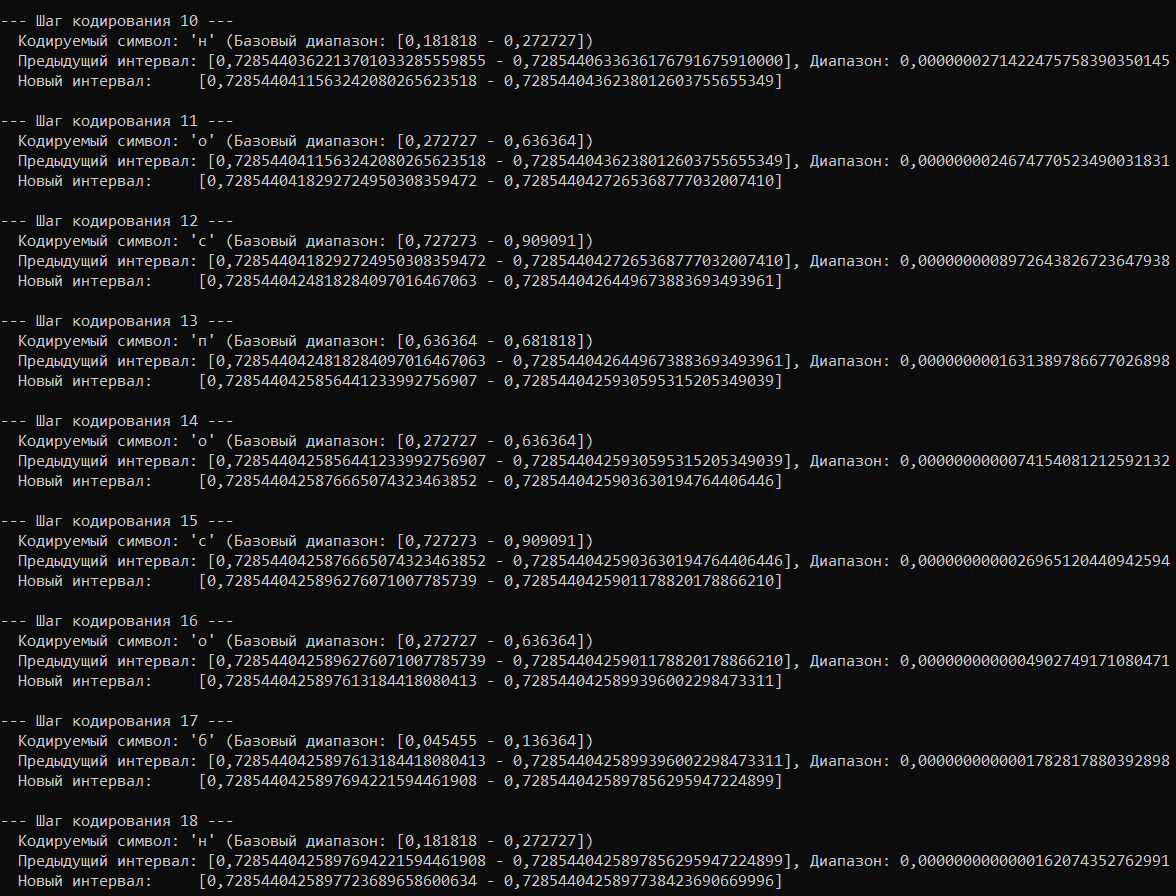


Рисунок 2.1 – Первая часть: список интервалов





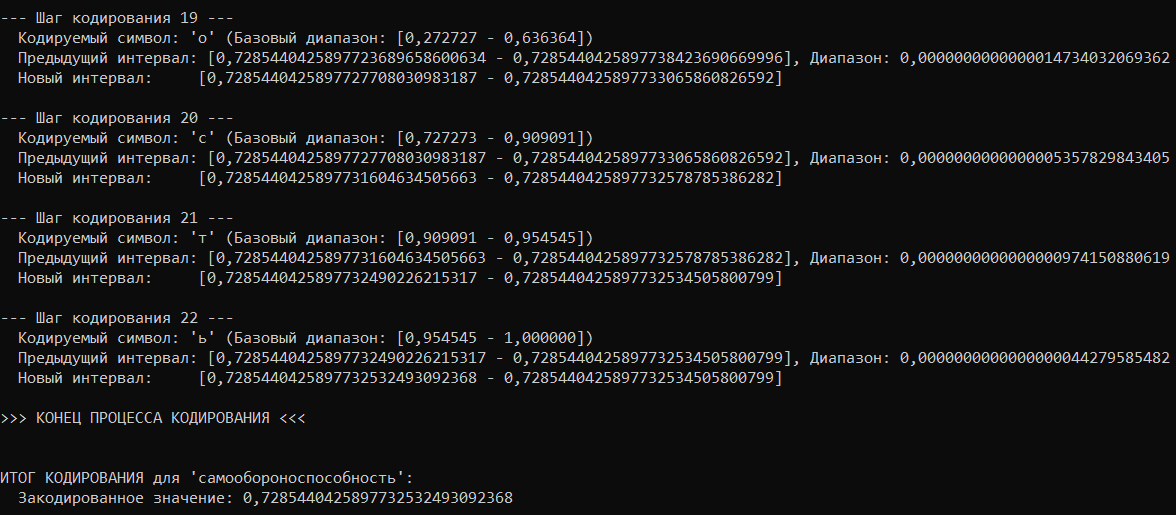
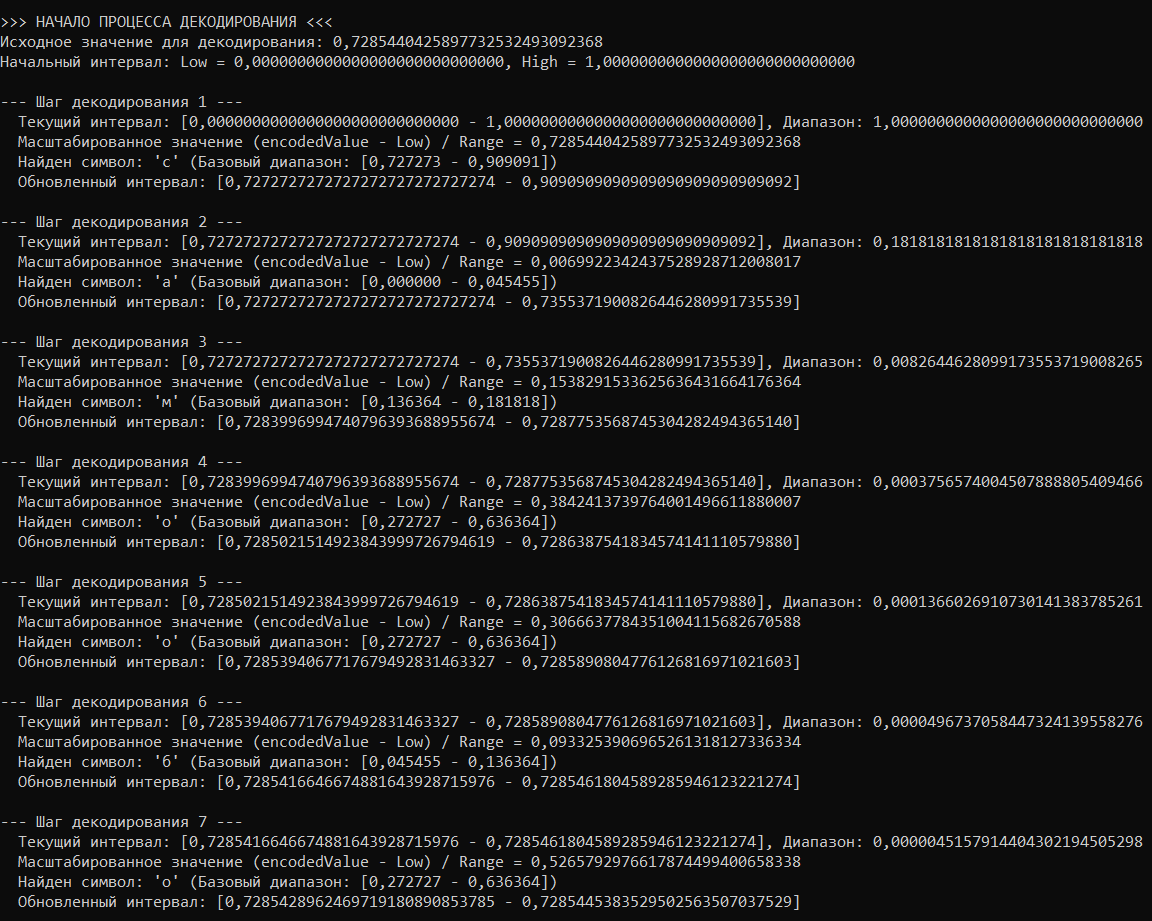
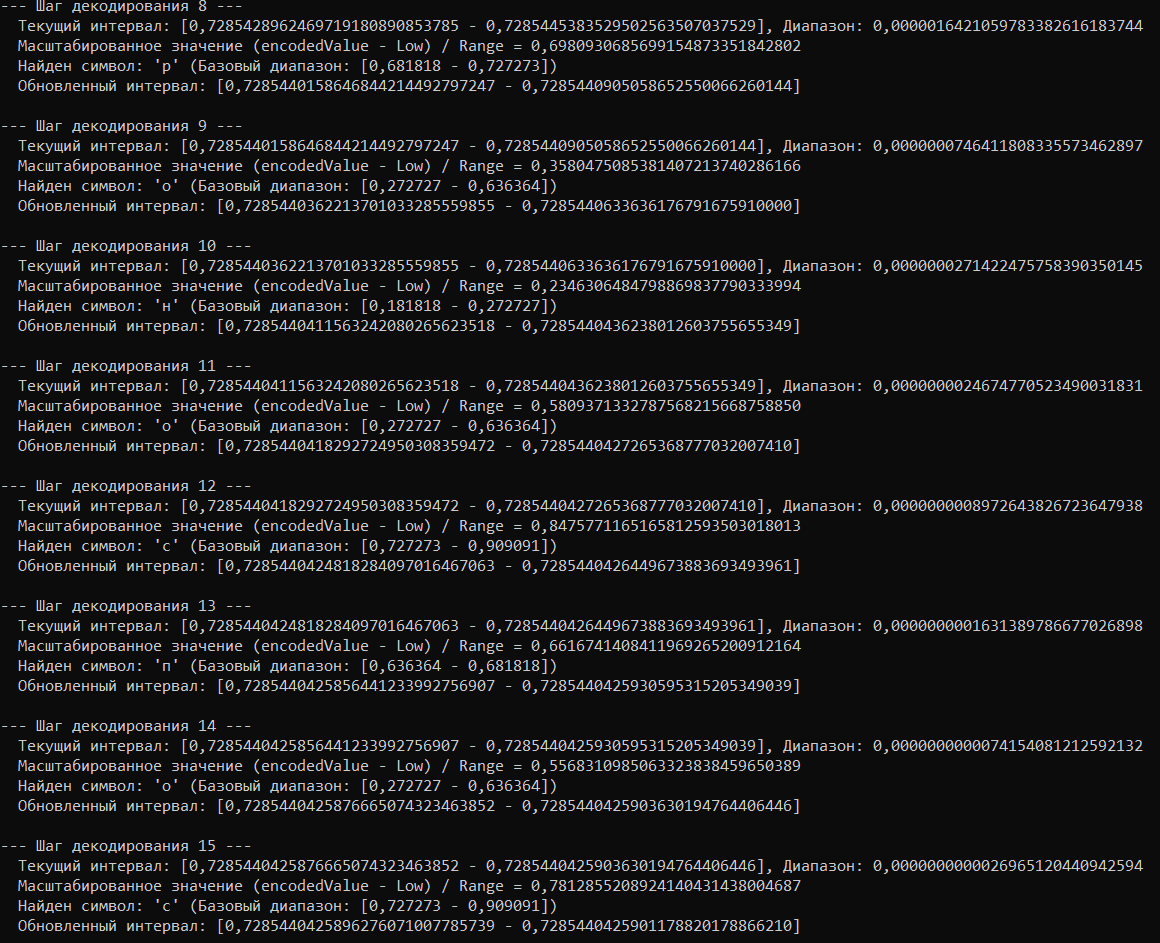


Рисунок 2.2 – Первая часть: кодирование





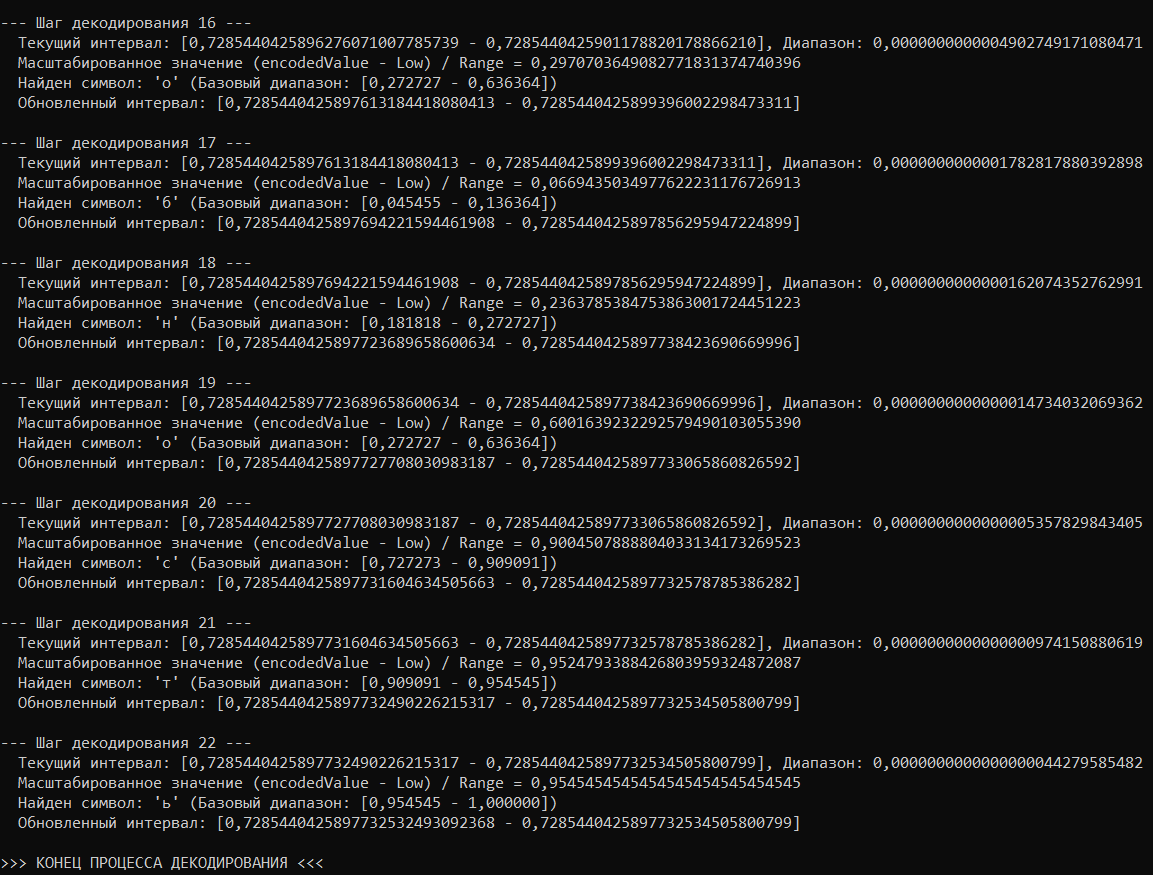


Рисунок 2.3 – Первая часть: декодирование

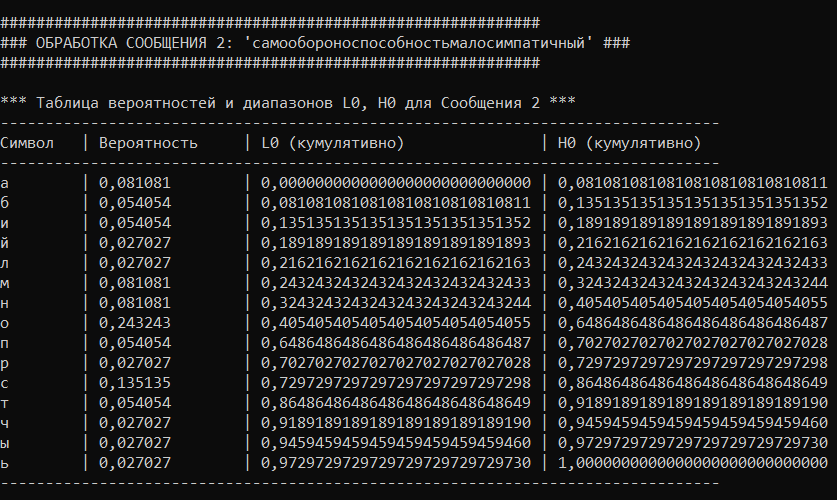
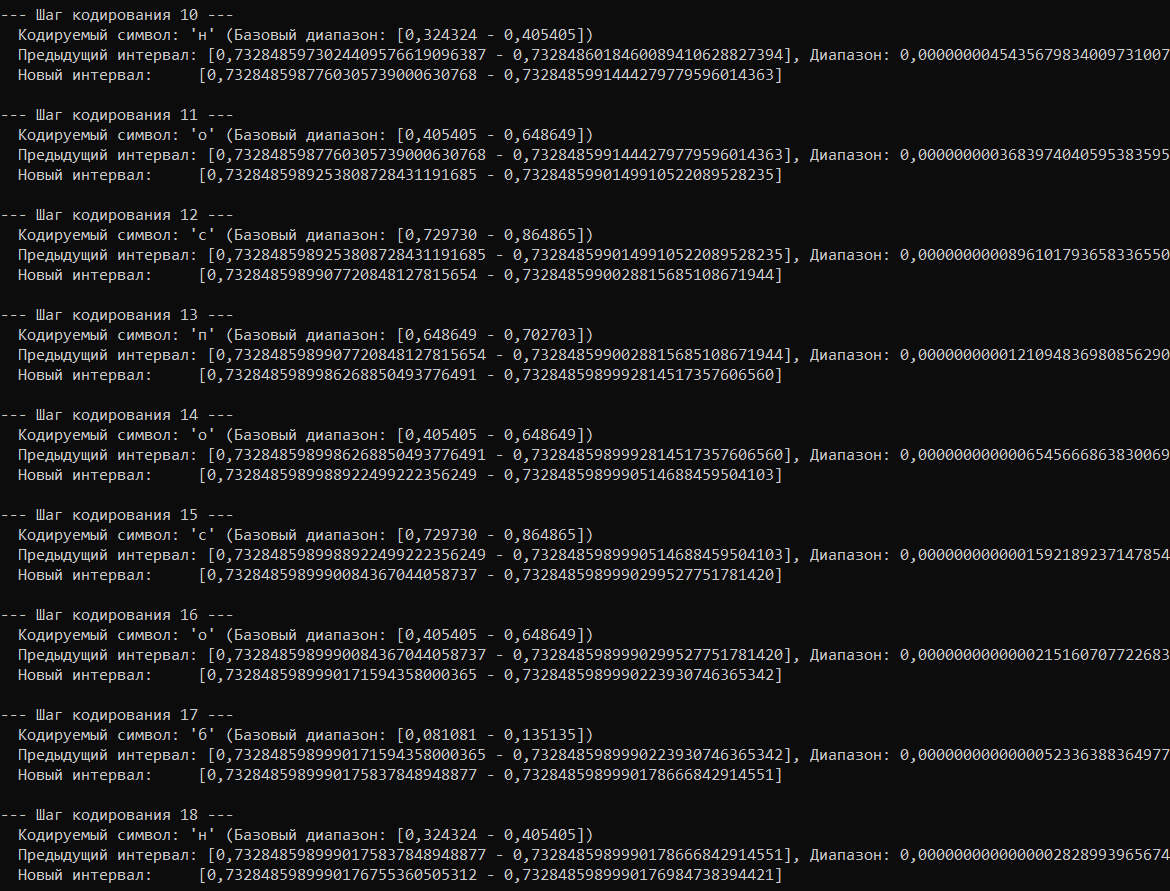
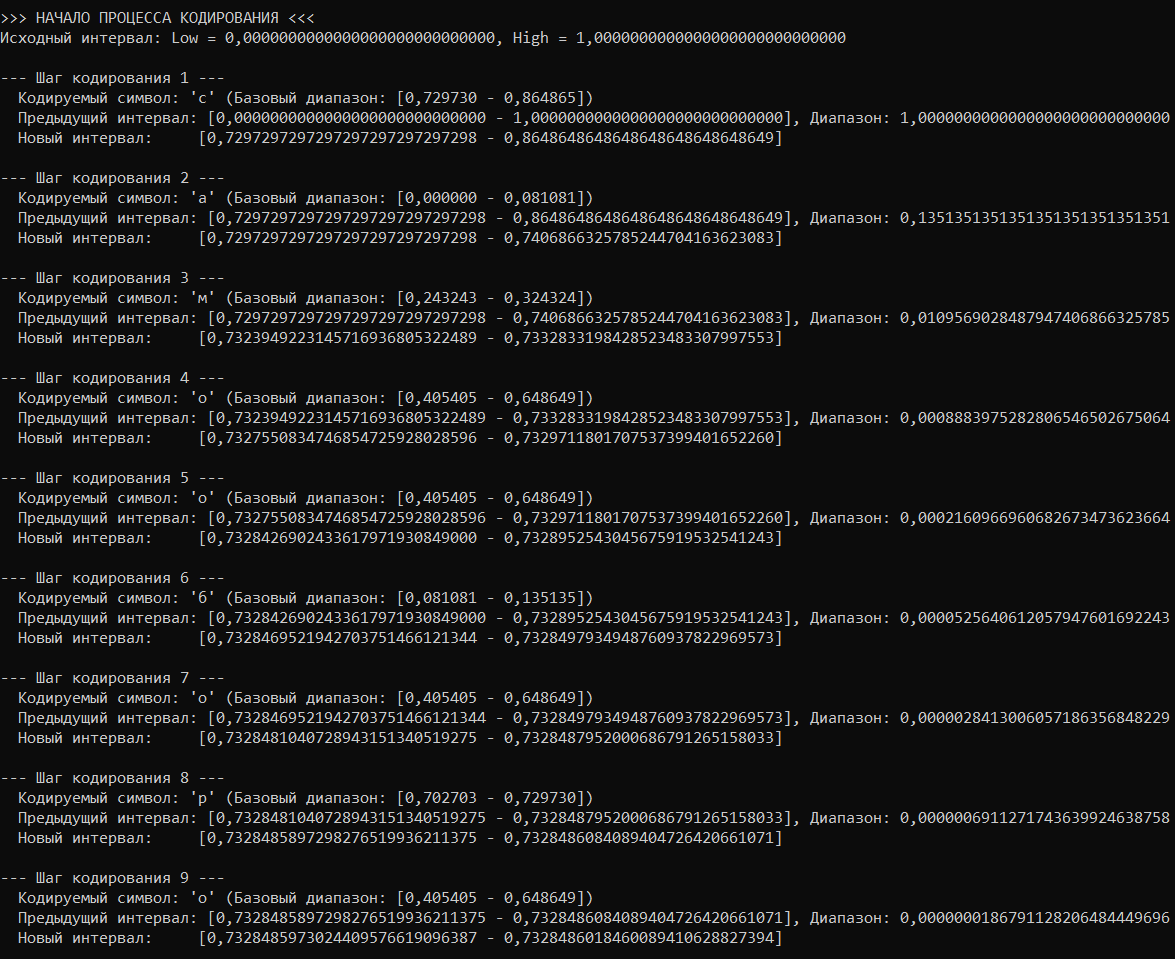


Рисунок 2.4 – Вторая часть: список интервалов



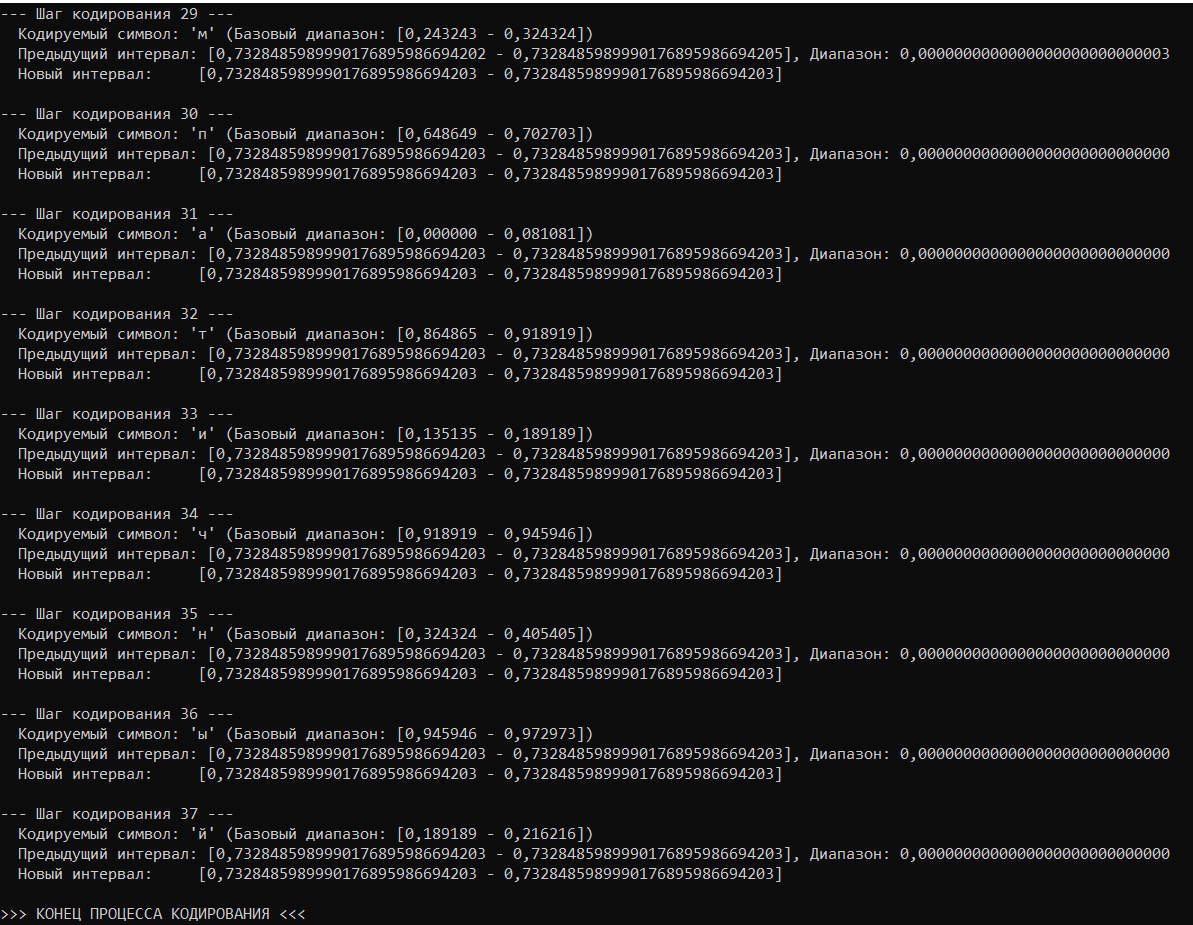
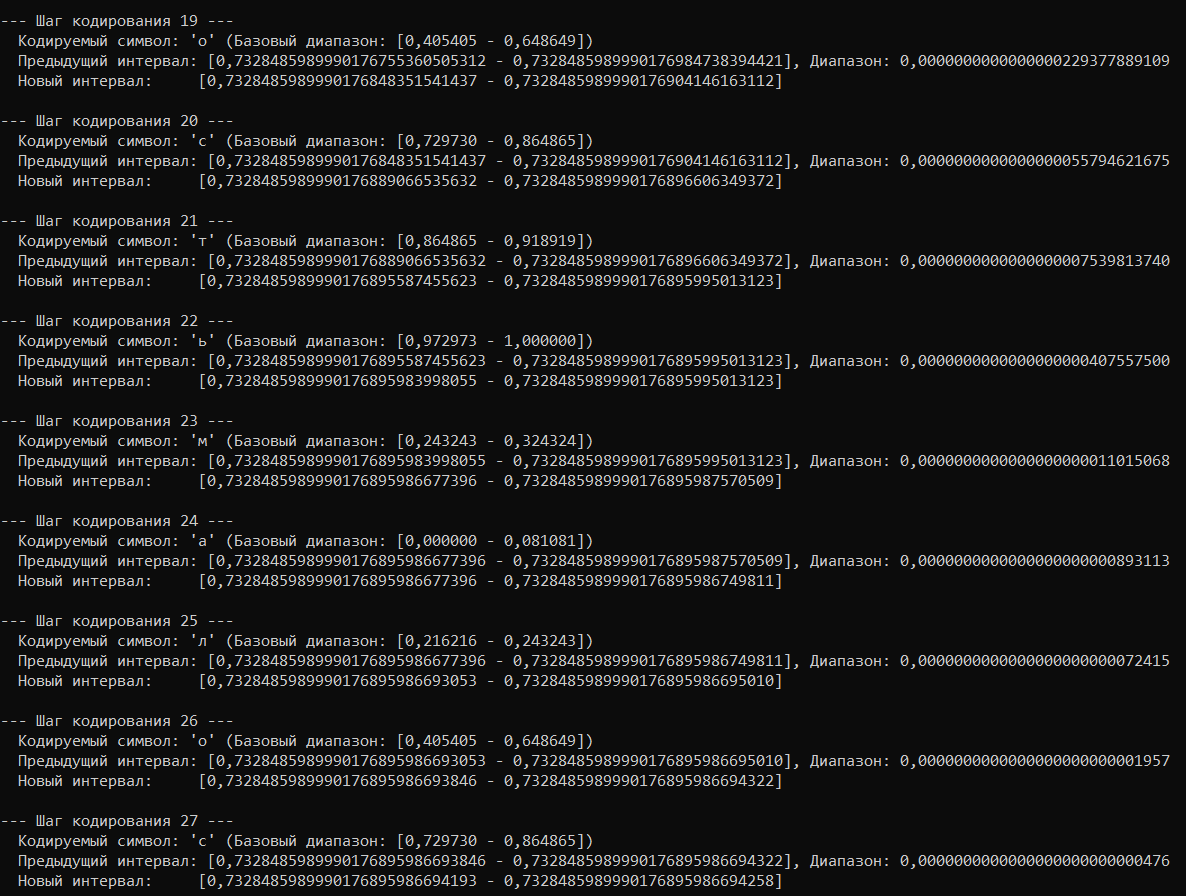
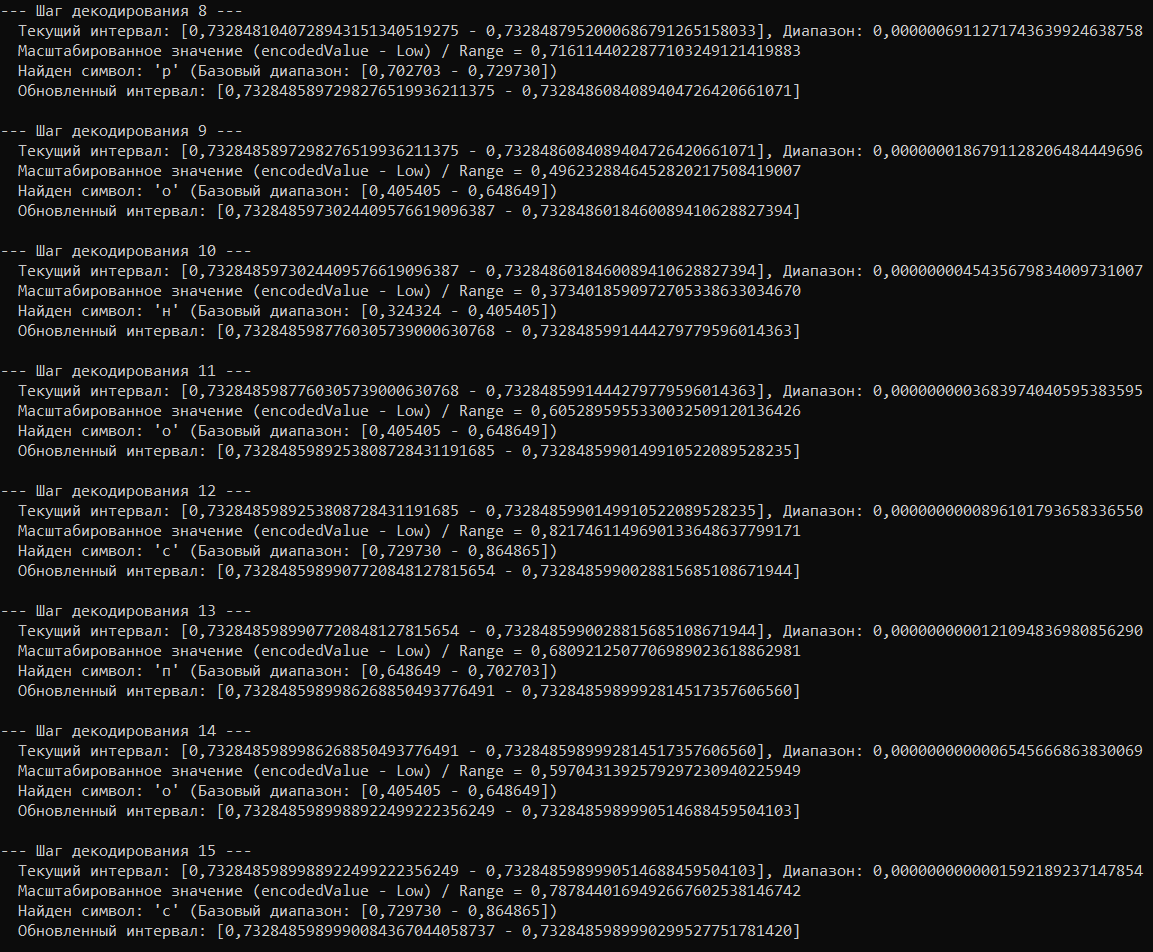
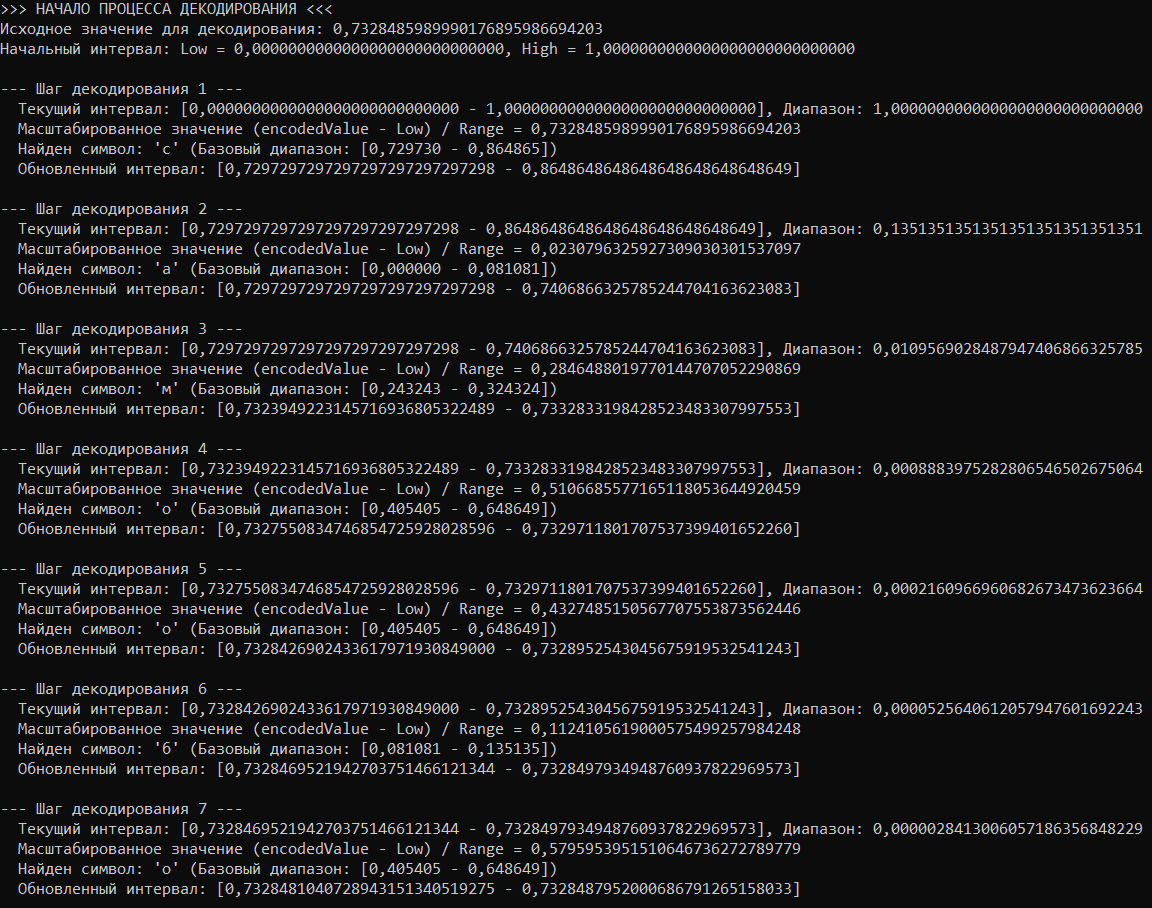


Рисунок 2.5 – Вторая часть: кодирование



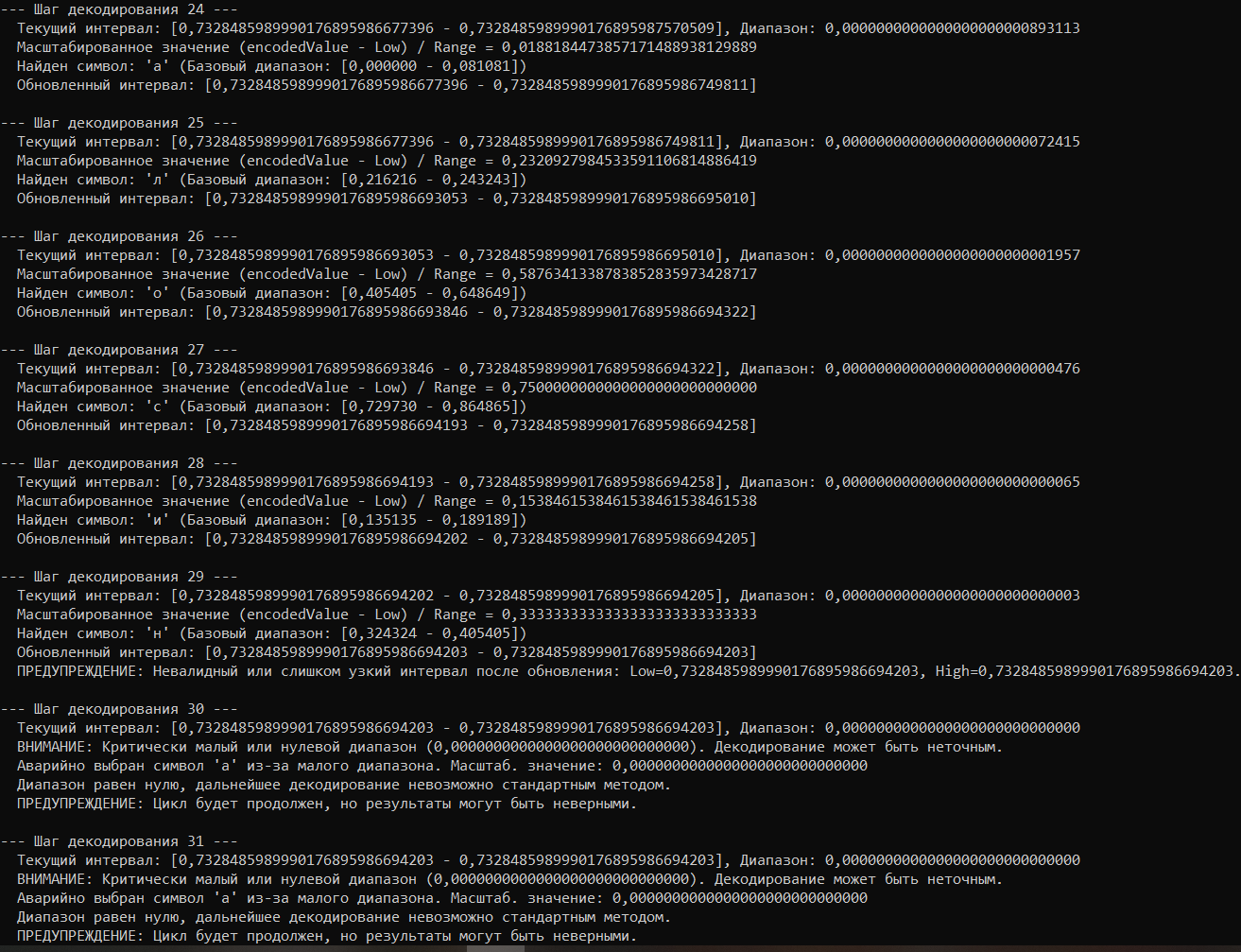
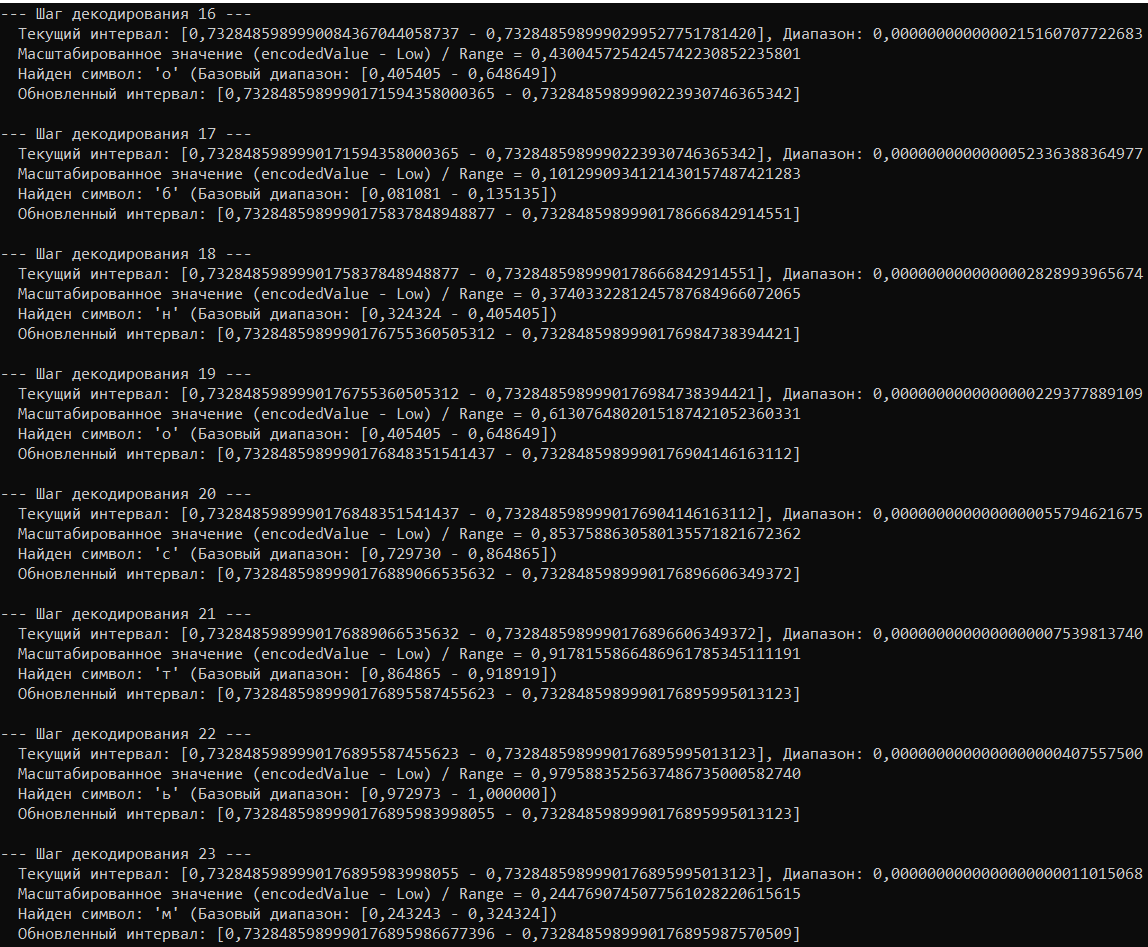


Рисунок 2.6 – Вторая часть: декодирование

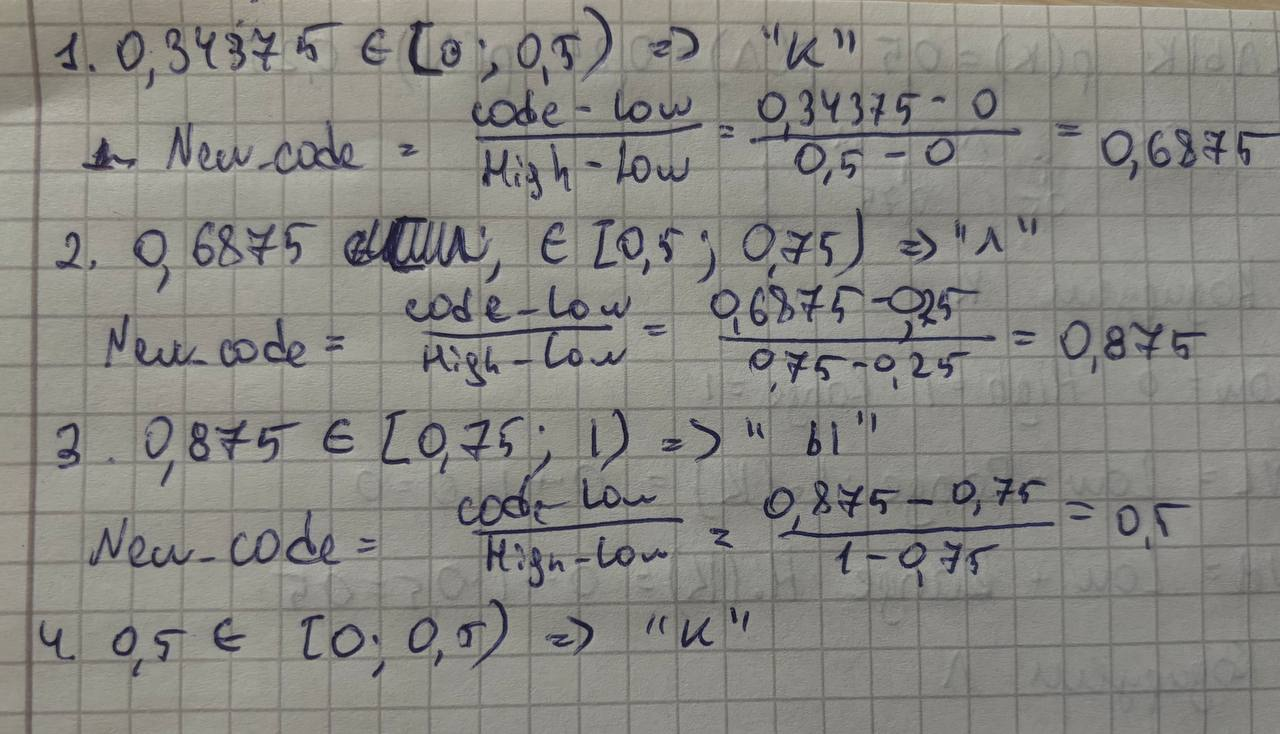
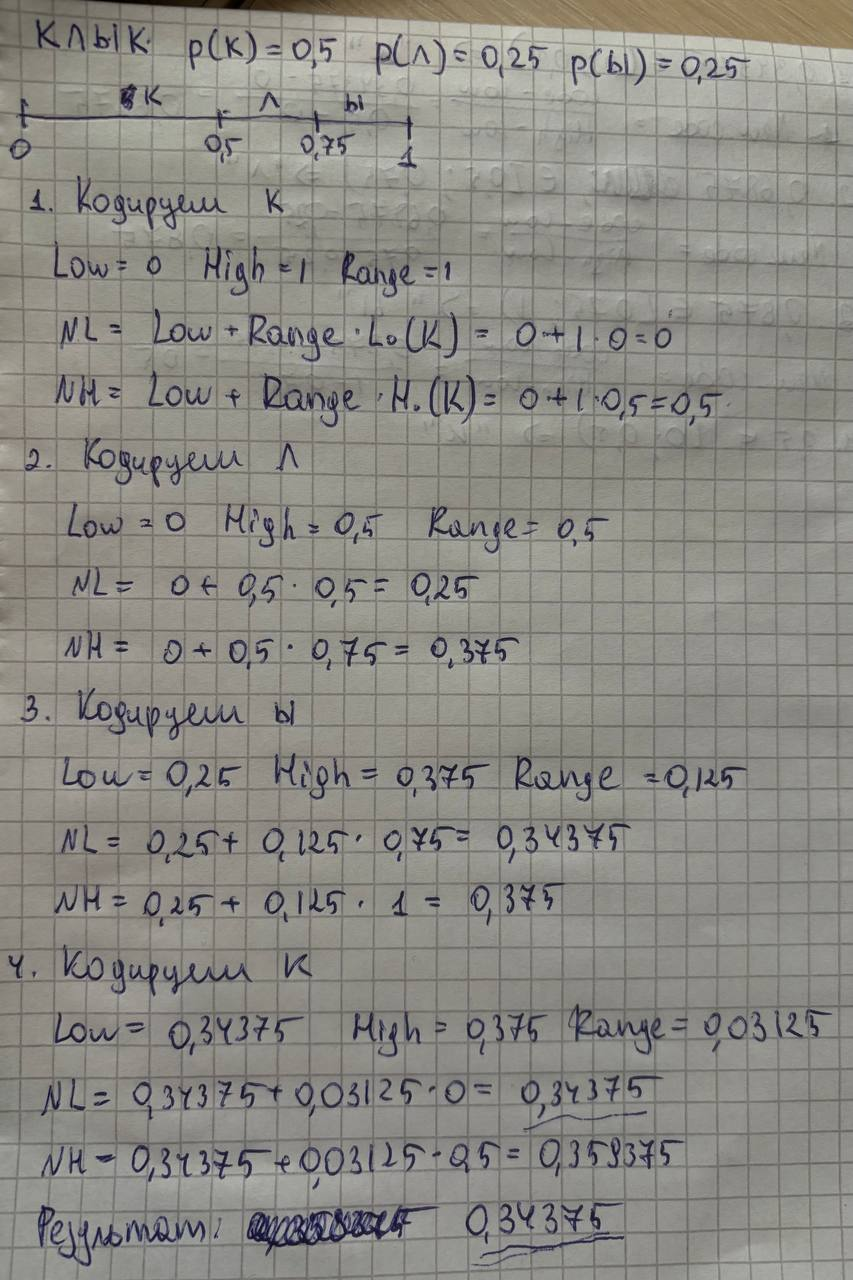


Рисунок 2.7 – Пример арифметического сжатия.

1. **Дать оценку возможности переполнения при выполнении вычислений.**

При использовании типа decimal для арифметического кодирования классическое переполнение (когда результат вычислений превышает максимально представимое значение для типа) маловероятно. Диапазон значений decimal (примерно от ±1.0 x 10⁻²⁸ до ±7.9 x 10²⁸) значительно превосходит значения, с которыми оперирует алгоритм (вероятности и границы интервалов всегда находятся в диапазоне [0, 1]).

Основная проблема, связанная с вычислениями в арифметическом кодировании, — это потеря точности. Это происходит, когда интервал [low, high) становится чрезвычайно узким.

Причина: После каждого шага кодирования (или декодирования) длина текущего интервала currentRange = high - low умножается на вероятность символа P(s\_i). Если вероятности малы и/или сообщение длинное, currentRange может стать настолько маленьким, что:

Он становится меньше, чем наименьшее приращение, которое может представить тип decimal (его эпсилон). В этом случае high - low может стать равным нулю, даже если теоретически они должны быть различны.

Различия между low и high или между sInfo.Low и sInfo.High для разных символов могут «стерться» из-за ограниченной точности (28-29 значащих цифр для decimal).

**Последствия:**

* Деление на currentRange (в декодере) может привести к ошибке деления на ноль или к очень большим ошибкам округления.
* Становится невозможно различить интервалы для разных символов, что приводит к неправильному декодированию.

**Смягчение в коде:**

* Использование типа decimal вместо double уже является значительным шагом для повышения точности.
* Введение константы EPSILON для «неточных» сравнений.
* Реализация «аварийных» веток выбора символа в декодере при критически малом диапазоне или когда символ не найден стандартным способом.

1. **Сравнить характеристики арифметического сжатия с вероятностными алгоритмами.**

Арифметическое сжатие само по себе является вероятностным (или энтропийным) алгоритмом сжатия, так как его эффективность напрямую зависит от точности вероятностной модели исходных данных. Сравним его с другими известными вероятностными алгоритмами, в первую очередь с кодированием Хаффмана.

**Эффективность сжатия (степень сжатия):**

* Арифметическое кодирование: Теоретически может достигать степени сжатия, очень близкой к энтропии источника. Это связано с тем, что оно может присваивать символам "дробное" количество бит (в среднем). Например, символу с вероятностью 0.9 может быть эффективно выделено ~0.15 бита.
* Кодирование Хаффмана: Присваивает каждому символу код, состоящий из целого числа бит. Это приводит к некоторой избыточности, особенно когда вероятности символов не являются степенями 1/2. Эффективность Хаффмана ограничена H < L < H + 1, где H – энтропия, L – средняя длина кода. Для алфавитов с малым числом символов или сильно смещенными вероятностями разница может быть существенной.

**Сложность реализации и вычислений:**

* Арифметическое кодирование: Более сложно в реализации. Требует выполнения операций с высокой точностью (умножение, сложение, деление чисел с плавающей/фиксированной точкой). Может быть медленнее из-за этих вычислений.
* Кодирование Хаффмана: Относительно просто в реализации. Операции в основном целочисленные и битовые. Обычно работает быстрее.

**Адаптивность:**

Оба алгоритма могут быть реализованы в адаптивном варианте, где вероятностная модель обновляется по мере обработки данных (однопроходное сжатие). Адаптивное арифметическое кодирование часто проще концептуально, чем адаптивное Хаффмана, так как изменение вероятностей напрямую влияет на диапазоны без необходимости перестройки всего дерева кодов.

**Разделение моделирования и кодирования:**

* Арифметическое кодирование: Четко разделяет этап моделирования (определение вероятностей) и этап кодирования. Это позволяет легко заменять или улучшать вероятностную модель, не меняя сам кодер.
* Кодирование Хаффмана: Модель (дерево Хаффмана) тесно связана с процессом кодирования.

# Вывод

В ходе выполнения лабораторной работы было разработано приложение, реализующее алгоритм арифметического кодирования. Были изучены и практически применены основные этапы данного метода сжатия данных: вычисление вероятностей символов, построение таблицы диапазонов, а также процессы кодирования сообщения в числовой интервал и последующего декодирования.

Приложение продемонстрировало способность арифметического кодирования преобразовывать текстовую информацию в единое число и восстанавливать ее с высокой точностью при использовании типа данных decimal. Были проанализированы потенциальные проблемы, связанные с ограниченной точностью вычислений, и предприняты меры для их смягчения, такие как использование decimal и специальных проверок в процессе декодирования.

Сравнение с другими вероятностными методами, в частности с кодированием Хаффмана, показало, что арифметическое кодирование обладает потенциалом для достижения более высокой степени сжатия за счет возможности присвоения символам «дробного» числа бит. Однако это преимущество сопровождается большей вычислительной сложностью.

Работа позволила получить практические навыки в реализации сложных алгоритмов сжатия, а также глубже понять принципы энтропийного кодирования и проблемы, связанные с точностью представления чисел в вычислительных системах. Разработанное приложение успешно справляется с поставленной задачей кодирования и декодирования текстовых сообщений методом арифметического сжатия.