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 № 9

# СЖАТИЕ/РАСПАКОВКА ДАННЫХ НА ОСНОВЕ СТАТИСТИЧЕСКИХ МЕТОДОВ

**Цель**: приобретение практических навыков использования статистических методов Шеннона − Фано и Хаффмана (ShannonFano and Huffman coding) для сжатия/распаковки данных.

**Задачи**:

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

# Теоретические сведения

Статистические методы сжатия данных, такие как алгоритмы Шеннона–Фано и Хаффмана, основываются на фундаментальном принципе использования вероятностных характеристик символов алфавита. Основная идея заключается в том, чтобы символы, встречающиеся в сообщении чаще, кодировались более короткими последовательностями битов, а редко встречающиеся – более длинными. Это позволяет значительно уменьшить общий размер данных. Такие алгоритмы относятся к префиксным кодам, что означает, что ни один код не является началом (префиксом) другого, обеспечивая однозначное декодирование. Важно отметить, что эти методы реализуют сжатие без потерь информации, то есть исходное сообщение может быть точно восстановлено из сжатого. Различие между методами Шеннона – Фано и Хаффмана заключается, главным образом, в способе построения таблицы бинарных кодов.

Метод Шеннона – Фано, хотя и не всегда является оптимальным, предлагает следующий подход к построению кодов. Сначала подсчитываются вероятностные параметры (частоты появления) каждого символа алфавита. Затем символы сортируются по убыванию их вероятностей. После этого отсортированное множество символов рекурсивно делится на две группы таким образом, чтобы суммарные вероятности символов в каждой группе были приблизительно равны. Одной группе присваивается бит '0', другой – '1' в качестве первого (или очередного) бита кода. Этот процесс повторяется для каждой полученной подгруппы, пока в ней не останется только один символ, которому и будет соответствовать сформированный код. При декодировании анализируется последовательность битов, начиная с минимальной возможной длины кода в таблице, и постепенно увеличивая её до максимальной, пока не будет найдено совпадение с одним из кодов символов.

Метод Хаффмана, в свою очередь, является оптимальным для посимвольного кодирования при известном распределении вероятностей символов. Его суть заключается в построении специального бинарного дерева. Процесс начинается с сортировки символов по их вероятностям (обычно по возрастанию, чтобы легко выбирать наименее вероятные). Затем два символа (или узла дерева) с наименьшими вероятностями объединяются в один новый узел, ветви к которому от этого узла помечаются '0' и '1'. Этот новый узел рассматривается как один «виртуальный» символ с суммарной вероятностью своих «потомков». Процесс объединения двух наименее вероятных узлов/символов повторяется до тех пор, пока все символы не будут включены в одно дерево, имеющее единственный корень. Кодом каждого исходного символа является последовательность битов, представляющая путь от корня дерева до соответствующего листа.

# Практическое задание

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

# Метод коридорки Шеннона-Фано

Рекурсивное присвоение кодов (AssignCodesRecursive): Это сердце алгоритма Шеннона-Фано, где происходит разделение списка символов и присвоение префиксов '0' и '1'.

|  |
| --- |
| private void AssignCodesRecursive(List<SymbolInfo> symbols, string currentCode)  {  if (!symbols.Any()) return;  // Console.WriteLine($"\nРекурсия: [{string.Join(", ", symbols.Select(s => $"'{s.Symbol}'({s.Probability:F3})"))}], префикс: '{currentCode}'");  if (symbols.Count == 1)  {  symbols[0].Code = currentCode;  var originalSymbol = \_symbolInfos.FirstOrDefault(s => s.Symbol == symbols[0].Symbol);  if (originalSymbol != null) originalSymbol.Code = currentCode;  // Console.WriteLine($" Лист: '{symbols[0].Symbol}' -> '{currentCode}'");  return;  }  if (symbols.Count == 0) return;  int splitIndex = FindSplitPoint(symbols);  // Console.WriteLine($" Разделение по индексу: {splitIndex}");  List<SymbolInfo> group1 = new List<SymbolInfo>();  List<SymbolInfo> group2 = new List<SymbolInfo>();  for (int i = 0; i < symbols.Count; i++)  {  if (i <= splitIndex) group1.Add(symbols[i]);  else group2.Add(symbols[i]);  }  // Обработка случая, когда разделение неэффективно (например, все символы в одной группе)  if (!group1.Any() || !group2.Any())  {  // Console.WriteLine($" Предупреждение: Неэффективное разделение для [{string.Join(", ", symbols.Select(s => $"'{s.Symbol}'"))}]. Аварийное присвоение.");  // Аварийное присвоение: просто добавляем '0' или '1' последовательно  // Это может быть неоптимально, но предотвращает бесконечную рекурсию или ошибки  if (symbols.Count > 0)  {  for (int i = 0; i < symbols.Count; ++i)  {  // Простой способ аварийного присвоения, чтобы избежать зацикливания  // Можно просто добавлять биты по порядку или более сложную логику  string emergencyCode = currentCode + (i % 2 == 0 ? "0" : "1"); // Простейший вариант  if (i == symbols.Count -1 && symbols.Count % 2 != 0 && symbols.Count >1 ) { // чтобы последний не получил такой же код как предпоследний если их нечетное количество  emergencyCode = currentCode + "1";  }  if (symbols.Count == 1) emergencyCode = currentCode; // Если остался один  symbols[i].Code = emergencyCode;  var originalSymbol = \_symbolInfos.FirstOrDefault(s => s.Symbol == symbols[i].Symbol);  if (originalSymbol != null) originalSymbol.Code = symbols[i].Code;  // Console.WriteLine($" Аварийно: '{symbols[i].Symbol}' -> '{symbols[i].Code}'");  }  }  return;  }  // Console.WriteLine($" Группа 1 (код + '0'): [{string.Join(", ", group1.Select(s => $"'{s.Symbol}'"))}]");  AssignCodesRecursive(group1, currentCode + "0");  // Console.WriteLine($" Группа 2 (код + '1'): [{string.Join(", ", group2.Select(s => $"'{s.Symbol}'"))}]");  AssignCodesRecursive(group2, currentCode + "1");  } |

Листинг 1.1 – Рекурсивное присвоение кодов

Данная функция рекурсивно строит коды. На каждом шаге список символов (отсортированный по убыванию вероятности) делится на две подгруппы. Символам первой подгруппы к текущему коду добавляется '0', второй – '1'. Рекурсия продолжается до тех пор, пока в подгруппе не останется один символ. Особое внимание уделяется обработке случаев неэффективного разделения для предотвращения ошибок.

Поиск точки разделения (FindSplitPoint): Эта функция определяет, как именно разделить список символов на две группы, чтобы суммы их вероятностей были максимально близки.

|  |
| --- |
| private int FindSplitPoint(List<SymbolInfo> symbols)  {  if (symbols.Count <= 1) return 0; // Нечего разделять или всего один символ  double totalProbability = symbols.Sum(s => s.Probability);  double minDifference = double.MaxValue;  int bestSplitIndex = 0;  // Итерируемся по всем возможным точкам разделения (кроме последней)  for (int i = 0; i < symbols.Count - 1; i++)  {  double sumGroup1 = 0;  // Суммируем вероятности первой группы  for (int j = 0; j <= i; j++) sumGroup1 += symbols[j].Probability;  // Вероятность второй группы  double sumGroup2 = totalProbability - sumGroup1;  double difference = Math.Abs(sumGroup1 - sumGroup2);  if (difference < minDifference)  {  minDifference = difference;  bestSplitIndex = i;  }  }  return bestSplitIndex;  } |

Листинг 1.2 – Поиск точки разделения

Функция FindSplitPoint перебирает все возможные точки разделения отсортированного списка символов. Для каждой точки вычисляется сумма вероятностей символов в первой и второй предполагаемых подгруппах. Индекс, при котором разница этих сумм минимальна, возвращается как оптимальная точка разделения.

# Метод коридорки Хаффмана

Основная публичная функция для построения таблицы кодов Хаффмана. Включает расчет вероятностей, создание начальных узлов дерева, итеративное построение дерева Хаффмана и запуск генерации кодов из построенного дерева.

|  |
| --- |
| public void BuildCodes(List<SymbolInfo> symbolInfosInput)  {  \_symbolInfosStorage = symbolInfosInput.Select(si => new SymbolInfo(si.Symbol, si.Frequency)).ToList();  if (!\_symbolInfosStorage.Any()) return;  foreach (var si in \_symbolInfosStorage)  {  if (si.Frequency == 0) si.Frequency = 1;  }  double totalFrequency = \_symbolInfosStorage.Sum(s => s.Frequency);  if (totalFrequency > 0)  {  foreach (var si in \_symbolInfosStorage)  {  si.Probability = si.Frequency / totalFrequency;  }  }  else if (\_symbolInfosStorage.Count == 1)  {  \_symbolInfosStorage.First().Probability = 1.0;  }  Console.WriteLine("\nСимволы (вероятность) для Хаффмана:");  \_symbolInfosStorage.OrderBy(s => s.Probability).ThenBy(s => s.Symbol) // Сортируем для начального отображения как в Хаффмане  .ToList().ForEach(s => Console.WriteLine($" '{s.Symbol}': {s.Probability:F5} (Частота: {s.Frequency})"));  Codes.Clear();  \_reverseCodes.Clear();  // 1. Создание листовых узлов  var nodes = \_symbolInfosStorage.Select(si => new HuffmanNode(si.Symbol, si.Probability)).ToList();  if (nodes.Count == 0) return;  Console.WriteLine("\n--- Построение дерева Хаффмана ---");  // 2. Особый случай: один символ в алфавите  if (nodes.Count == 1)  {  var singleSymbolNode = nodes.First();  string code = "0";  Console.WriteLine($" Единственный узел: '{singleSymbolNode.Symbol}'({singleSymbolNode.Probability:F3}). Присваиваем код '{code}'.");  Codes.Add(singleSymbolNode.Symbol, code);  \_reverseCodes.Add(code, singleSymbolNode.Symbol);  UpdateSymbolInfoCodeInStorage(singleSymbolNode.Symbol, code);  }  else  {  // 3. Построение дерева Хаффмана  int step = 1;  // Используем PriorityQueue для эффективности, но для простоты и совместимости с List.Sort() оставим List  // List<HuffmanNode> priorityQueue = new List<HuffmanNode>(nodes);  while (nodes.Count > 1)  {  nodes.Sort(); // Сортируем узлы по вероятности (и символу для стабильности)  HuffmanNode left = nodes[0];  HuffmanNode right = nodes[1];  Console.WriteLine($"\n Шаг {step++}:");  Console.WriteLine($" Выбираем два узла с наименьшей вероятностью:");  Console.WriteLine($" Левый: {(left.IsLeaf() ? $"'{left.Symbol}'" : "Внутр. узел")} ({left.Probability:F5})");  Console.WriteLine($" Правый: {(right.IsLeaf() ? $"'{right.Symbol}'" : "Внутр. узел")} ({right.Probability:F5})");  HuffmanNode parent = new HuffmanNode(left, right);  Console.WriteLine($" Создаем родительский узел с вероятностью: {parent.Probability:F5}");  nodes.Remove(left);  nodes.Remove(right);  nodes.Add(parent);  if (nodes.Count > 1) // Выводим текущее состояние очереди узлов, если это не последний шаг  {  Console.WriteLine(" Текущие узлы в очереди (отсортированы):");  foreach (var node in nodes.OrderBy(n => n.Probability)) // Показываем в отсортированном виде  {  Console.WriteLine($" - {(node.IsLeaf() ? $"'{node.Symbol}'" : "Внутр. узел")} ({node.Probability:F5})");  }  }  }  HuffmanNode root = nodes.First();  Console.WriteLine("\n Дерево Хаффмана построено. Корень имеет вероятность: " + root.Probability.ToString("F5"));  Console.WriteLine("\n--- Генерация кодов из дерева Хаффмана ---");  // 4. Генерация кодов из дерева  GenerateCodesRecursive(root, "");  }  Console.WriteLine("\nИтоговые коды (Хаффман):");  foreach (var codeEntry in Codes.OrderBy(kvp => kvp.Key))  {  Console.WriteLine($" '{codeEntry.Key}': {codeEntry.Value}");  }  } |

Листинг 1.3 – Построения таблицы кодов Хаффмана

Рекурсивно обходит построенное дерево Хаффмана для присвоения бинарных кодов каждому символу (листовому узлу).

|  |
| --- |
| private void GenerateCodesRecursive(HuffmanNode node, string currentCode)  {  if (node == null) return;  string nodeDescription = node.IsLeaf() ? $"Листовой узел '{node.Symbol}' ({node.Probability:F5})" : $"Внутренний узел ({node.Probability:F5})";  Console.WriteLine($" Обход дерева: {nodeDescription}, Текущий префикс: '{currentCode}'");  if (node.IsLeaf())  {  // Для единственного символа в алфавите, код должен быть "0"  string finalCode = (\_symbolInfosStorage.Count == 1 && string.IsNullOrEmpty(currentCode)) ? "0" : currentCode;  Codes[node.Symbol] = finalCode;  \_reverseCodes[finalCode] = node.Symbol;  UpdateSymbolInfoCodeInStorage(node.Symbol, finalCode);  Console.WriteLine($" └─ Символу '{node.Symbol}' присвоен код: '{finalCode}'");  return;  }  if (node.LeftChild != null)  {  Console.WriteLine($" ├─ Переход к левому потомку (добавляем '0' к префиксу)");  GenerateCodesRecursive(node.LeftChild, currentCode + "0");  }  if (node.RightChild != null)  {  Console.WriteLine($" └─ Переход к правому потомку (добавляем '1' к префиксу)");  GenerateCodesRecursive(node.RightChild, currentCode + "1");  }  } |

Листинг 1.4 – Рекурсивный обход

1. **С помощью приложения выполнить прямое и обратное преобразования сообщения, состоящего из собственных имени и фамилии.**

# Метод коридорки Шеннона-Фано

Можно использовать любой из известных методов сортировки символов массива. Метод кодировки (Шеннона − Фано, Хаффмана) использовать по указанию преподавателя.

При этом таблица отсортированных символов строится:

а) на основе данных, полученных в лабораторной работе № 2;

б) динамически, на основе анализа сжимаемого сообщения.

Оригинал: «лопатнюк полина» (Длина: 15)

Записываем все символы из входящей строки в порядке убывания вероятностей.

Символы (вероятность):

* 'а': 0,33684
* 'н': 0,17444
* 'т': 0,16842
* 'л': 0,13173
* 'о': 0,06195
* 'к': 0,05654
* 'и': 0,04090
* 'п': 0,02887
* 'ю': 0,00030

Далее начинаем рекурсивное построение двоичного дерева, где каждому левому переходу соответствует '0', а правому – '1'

**Рекурсия**: ['а'(0,337), 'н'(0,174), 'т'(0,168), 'л'(0,132), 'о'(0,062), 'к'(0,057), 'и'(0,041), 'п'(0,029), 'ю'(0,000)], префикс: ''

* Разделение по индексу: 1
* Группа 1 (код + '0'): ['а', 'н']

**Рекурсия**: ['а'(0,337), 'н'(0,174)], префикс: '0'

* Разделение по индексу: 0
* Группа 1 (код + '0'): ['а']

**Рекурсия**: ['а'(0,337)], префикс: '00'

* Лист: 'а' -> '00'
* Группа 2 (код + '1'): ['н']

**Рекурсия**: ['н'(0,174)], префикс: '01'

* Лист: 'н' -> '01'
* Группа 2 (код + '1'): ['т', 'л', 'о', 'к', 'и', 'п', 'ю']

**Рекурсия**: ['т'(0,168), 'л'(0,132), 'о'(0,062), 'к'(0,057), 'и'(0,041), 'п'(0,029), 'ю'(0,000)], префикс: '1'

* Разделение по индексу: 1
* Группа 1 (код + '0'): ['т', 'л']

**Рекурсия**: ['т'(0,168), 'л'(0,132)], префикс: '10'

* Разделение по индексу: 0
* Группа 1 (код + '0'): ['т']

**Рекурсия**: ['т'(0,168)], префикс: '100'

* Лист: 'т' -> '100'
* Группа 2 (код + '1'): ['л']

**Рекурсия**: ['л'(0,132)], префикс: '101'

* Лист: 'л' -> '101'
* Группа 2 (код + '1'): ['о', 'к', 'и', 'п', 'ю']

**Рекурсия**: ['о'(0,062), 'к'(0,057), 'и'(0,041), 'п'(0,029), 'ю'(0,000)], префикс: '11'

* Разделение по индексу: 1
* Группа 1 (код + '0'): ['о', 'к']

**Рекурсия**: ['о'(0,062), 'к'(0,057)], префикс: '110'

* Разделение по индексу: 0
* Группа 1 (код + '0'): ['о']

**Рекурсия**: ['о'(0,062)], префикс: '1100'

* Лист: 'о' -> '1100'
* Группа 2 (код + '1'): ['к']

**Рекурсия**: ['к'(0,057)], префикс: '1101'

* Лист: 'к' -> '1101'
* Группа 2 (код + '1'): ['и', 'п', 'ю']

**Рекурсия**: ['и'(0,041), 'п'(0,029), 'ю'(0,000)], префикс: '111'

* Разделение по индексу: 0
* Группа 1 (код + '0'): ['и']

**Рекурсия**: ['и'(0,041)], префикс: '1110'

* Лист: 'и' -> '1110'
* Группа 2 (код + '1'): ['п', 'ю']

**Рекурсия**: ['п'(0,029), 'ю'(0,000)], префикс: '1111'

* Разделение по индексу: 0
* Группа 1 (код + '0'): ['п']

**Рекурсия**: ['п'(0,029)], префикс: '11110'

* Лист: 'п' -> '11110'
* Группа 2 (код + '1'): ['ю']

**Рекурсия**: ['ю'(0,000)], префикс: '11111'

* Лист: 'ю' -> '11111'

**Итоговые коды:**

* 'а': 00
* 'и': 1110
* 'к': 1101
* 'л': 101
* 'н': 01
* 'о': 1100
* 'п': 11110
* 'т': 100
* 'ю': 11111

Так мы получаем коды Шеннона-Фано для данной последовательности символов и их вероятностей. Как видно, символы с большей вероятностью ('а', 'н') получили более короткие коды, а символы с меньшей вероятностью ('п', 'ю') – более длинные, что и является целью алгоритма для достижения сжатия.

**Декодирование**: читаем биты последовательно, пока не найдем совпадение в нашей таблице кодов:

101110011110001000111111110111110110010111100100

* Текущий код: 1 (поиск)
* Текущий код: 10 (поиск)
* **Текущий код: 101 -> Символ: 'л'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 110 (поиск)
* **Текущий код: 1100 -> Символ: 'о'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 111 (поиск)
* Текущий код: 1111 (поиск)
* **Текущий код: 11110 -> Символ: 'п'**
* Текущий код: 0 (поиск)
* **Текущий код: 00 -> Символ: 'а'**
* Текущий код: 1 (поиск)
* Текущий код: 10 (поиск)
* **Текущий код: 100 -> Символ: 'т'**
* Текущий код: 0 (поиск)
* **Текущий код: 01 -> Символ: 'н'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 111 (поиск)
* Текущий код: 1111 (поиск)
* **Текущий код: 11111 -> Символ: 'ю'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 110 (поиск)
* **Текущий код: 1101 -> Символ: 'к'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 111 (поиск)
* Текущий код: 1111 (поиск)
* **Текущий код: 11110 -> Символ: 'п'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 110 (поиск)
* **Текущий код: 1100 -> Символ: 'о'**
* Текущий код: 1 (поиск)
* Текущий код: 10 (поиск)
* **Текущий код: 101 -> Символ: 'л'**
* Текущий код: 1 (поиск)
* Текущий код: 11 (поиск)
* Текущий код: 111 (поиск)
* **Текущий код: 1110 -> Символ: 'и'**
* Текущий код: 0 (поиск)
* **Текущий код: 01 -> Символ: 'н'**
* Текущий код: 0 (поиск)
* **Текущий код: 00 -> Символ: 'а'**

**Результат декодирования**: «лопатнюкполина»

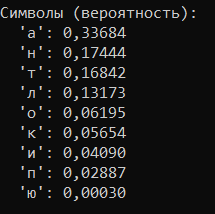


Рисунок 2.1 – Вероятность символов (статическая)

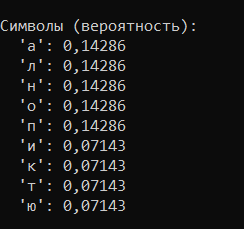


Рисунок 2.2 – Вероятность символов (динамическая)

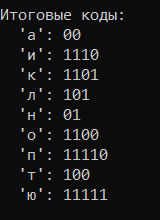


Рисунок 2.3 – Итоговая таблица кодов

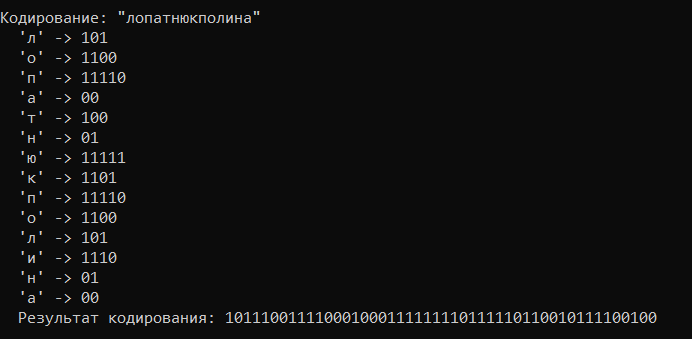


Рисунок 2.4 – Процесс кодирования

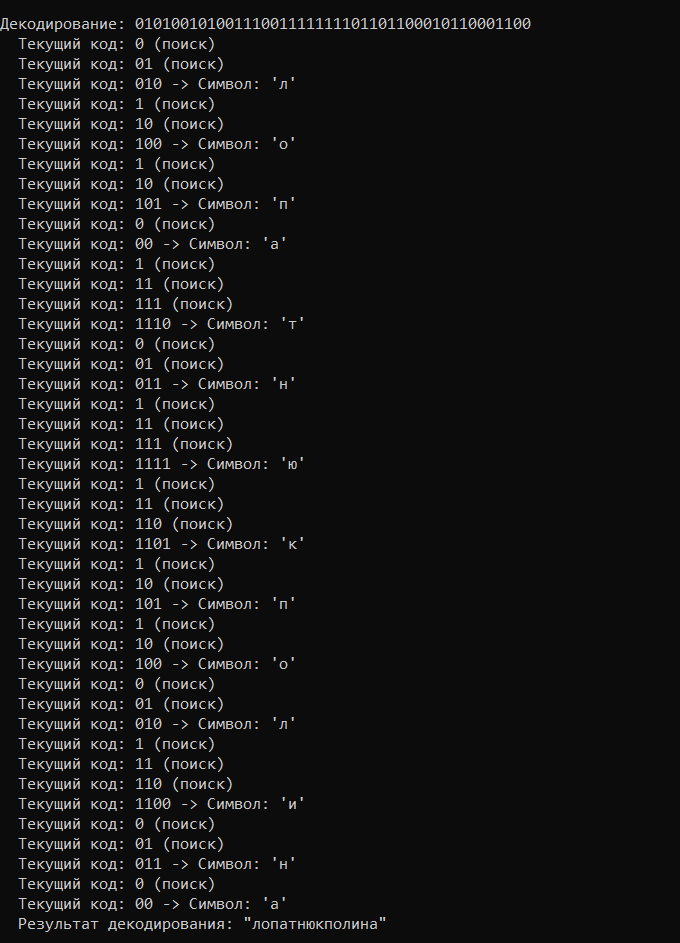


Рисунок 2.5 – Процесс декодирования

# Метод коридорки Хаффмана

**Режим C: Хаффман (Статическая вероятность)**

* Символы (вероятность) для Хаффмана:
* 'ю': 0,00030 (Частота: 1)
* 'п': 0,02887 (Частота: 96)
* 'и': 0,04090 (Частота: 136)
* 'к': 0,05654 (Частота: 188)
* 'о': 0,06195 (Частота: 206)
* 'л': 0,13173 (Частота: 438)
* 'т': 0,16842 (Частота: 560)
* 'н': 0,17444 (Частота: 580)
* 'а': 0,33684 (Частота: 1120)

**Построение дерева Хаффмана:**

**Шаг 1:Выбираем два узла с наименьшей вероятностью:**

* Левый: 'ю' (0,00030)
* Правый: 'п' (0,02887)

**Создаем родительский узел с вероятностью: 0,02917**

**Текущие узлы в очереди (отсортированы):**

* Внутр. узел (0,02917)
* 'и' (0,04090)
* 'к' (0,05654)
* 'о' (0,06195)
* 'л' (0,13173)
* 'т' (0,16842)
* 'н' (0,17444)
* 'а' (0,33684)

**Шаг 2: Выбираем два узла с наименьшей вероятностью:**

* Левый: Внутр. узел (0,02917)
* Правый: 'и' (0,04090)

**Создаем родительский узел с вероятностью: 0,07008**

**Текущие узлы в очереди (отсортированы):**

* 'к' (0,05654)
* 'о' (0,06195)
* Внутр. узел (0,07008)
* 'л' (0,13173)
* 'т' (0,16842)
* 'н' (0,17444)
* 'а' (0,33684)

**Шаг 3: Выбираем два узла с наименьшей вероятностью:**

* Левый: 'к' (0,05654)
* Правый: 'о' (0,06195)

**Создаем родительский узел с вероятностью: 0,11850**

**Текущие узлы в очереди (отсортированы):**

* Внутр. узел (0,07008)
* Внутр. узел (0,11850)
* 'л' (0,13173)
* 'т' (0,16842)
* 'н' (0,17444)
* 'а' (0,33684)

**Шаг 4: Выбираем два узла с наименьшей вероятностью:**

* Левый: Внутр. узел (0,07008)
* Правый: Внутр. узел (0,11850)

**Создаем родительский узел с вероятностью: 0,18857**

**Текущие узлы в очереди (отсортированы):**

* 'л' (0,13173)
* 'т' (0,16842)
* 'н' (0,17444)
* Внутр. узел (0,18857)
* 'а' (0,33684)

**Шаг 5: Выбираем два узла с наименьшей вероятностью:**

* Левый: 'л' (0,13173)
* Правый: 'т' (0,16842)

**Создаем родительский узел с вероятностью: 0,30015**

**Текущие узлы в очереди (отсортированы):**

* 'н' (0,17444)
* Внутр. узел (0,18857)
* Внутр. узел (0,30015)
* 'а' (0,33684)

**Шаг 6: Выбираем два узла с наименьшей вероятностью:**

* Левый: 'н' (0,17444)
* Правый: Внутр. узел (0,18857)

**Создаем родительский узел с вероятностью: 0,36301**

**Текущие узлы в очереди (отсортированы):**

* Внутр. узел (0,30015)
* 'а' (0,33684)
* Внутр. узел (0,36301)

**Шаг 7: Выбираем два узла с наименьшей вероятностью:**

* Левый: Внутр. узел (0,30015)
* Правый: 'а' (0,33684)

**Создаем родительский узел с вероятностью: 0,63699**

**Текущие узлы в очереди (отсортированы):**

* Внутр. узел (0,36301)
* Внутр. узел (0,63699)

**Шаг 8: Выбираем два узла с наименьшей вероятностью:**

* Левый: Внутр. узел (0,36301)
* Правый: Внутр. узел (0,63699)

**Создаем родительский узел с вероятностью: 1,00000**

**Определение кодов из дерева Хаффмана:**

1. Обход дерева: Внутренний узел (1,00000), Текущий префикс: ''

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Внутренний узел (0,36301), Текущий префикс: '0'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Листовой узел 'н' (0,17444), Текущий префикс: '00'

* Символу 'н' присвоен код: '00'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Внутренний узел (0,18857), Текущий префикс: '01'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Внутренний узел (0,07008), Текущий префикс: '010'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Внутренний узел (0,02917), Текущий префикс: '0100'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Листовой узел 'ю' (0,00030), Текущий префикс: '01000'

* Символу 'ю' присвоен код: '01000'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Листовой узел 'п' (0,02887), Текущий префикс: '01001'

* Символу 'п' присвоен код: '01001'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Листовой узел 'и' (0,04090), Текущий префикс: '0101'

* Символу 'и' присвоен код: '0101'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Внутренний узел (0,11850), Текущий префикс: '011'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Листовой узел 'к' (0,05654), Текущий префикс: '0110'

* Символу 'к' присвоен код: '0110'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Листовой узел 'о' (0,06195), Текущий префикс: '0111'

* Символу 'о' присвоен код: '0111'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Внутренний узел (0,63699), Текущий префикс: '1'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Внутренний узел (0,30015), Текущий префикс: '10'

* Переход к левому потомку (добавляем '0' к префиксу)

1. Обход дерева: Листовой узел 'л' (0,13173), Текущий префикс: '100'

* Символу 'л' присвоен код: '100'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Листовой узел 'т' (0,16842), Текущий префикс: '101'

* Символу 'т' присвоен код: '101'
* Переход к правому потомку (добавляем '1' к префиксу)

1. Обход дерева: Листовой узел 'а' (0,33684), Текущий префикс: '11'

* Символу 'а' присвоен код: '11'

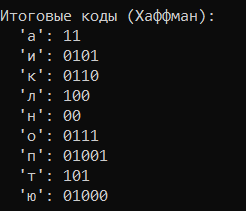


Рисунок 2.6 – Итоговая таблица кодов

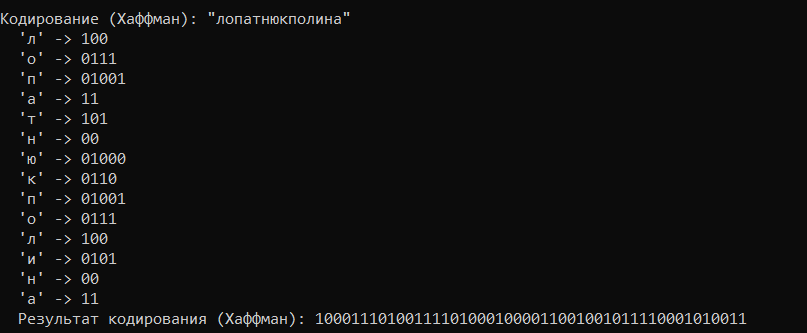


Рисунок 2.7 – Результат кодирования

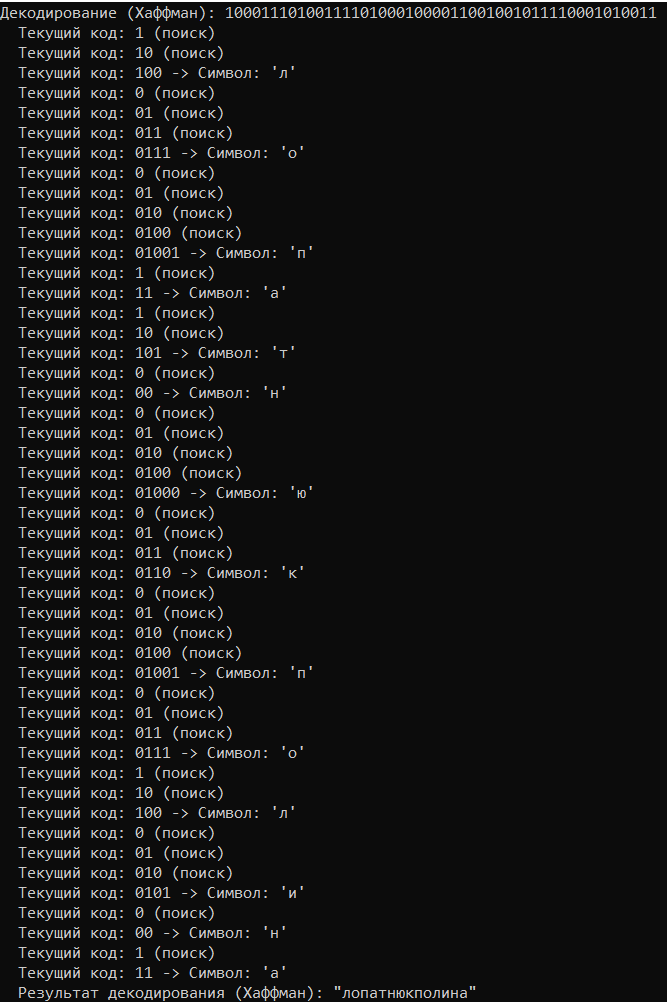


Рисунок 2.8 – Результат декодирования

1. **Определить эффективность (в сравнении с кодами ASCII) сжатия сообщения.**

# Метод коридорки Шеннона-Фано

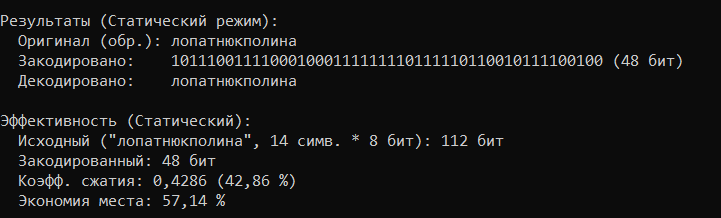


Рисунок 3.1 – Подсчёт эффективности для статического

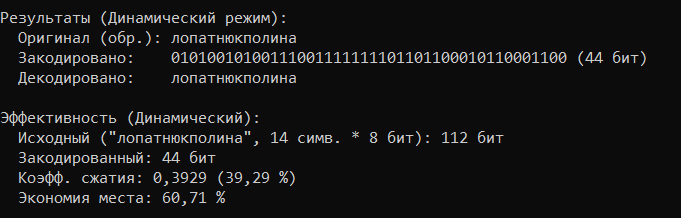


Рисунок 3.2 – Подсчёт эффективности для динамического

# Метод коридорки Хаффмана

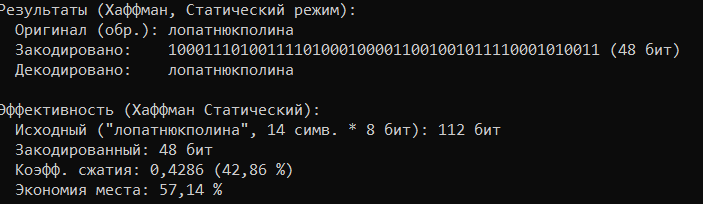


Рисунок 3.3 – Подсчёт эффективности для статического

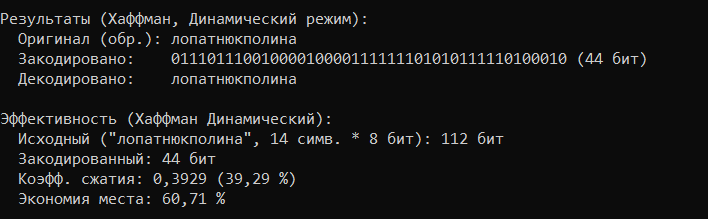


Рисунок 3.4 – Подсчёт эффективности для динамического

# Вывод

Была разработана программа, позволяющая кодировать и декодировать текстовую информацию с использованием указанного алгоритма.

В качестве исходных данных использовалась строка «лопатнюк полина», которая для непосредственного кодирования была предварительно обработана: удалены пробелы и все символы приведены к нижнему регистру, в результате чего получена строка «лопатнюкполина».

Работа алгоритма была продемонстрирована в двух режимах:

**Режим A (Статическая вероятность):** Вероятности символов определялись на основе усредненных частот для казахского алфавита. Для анализируемой строки были построены коды Шеннона-Фано, итоговая таблица которых показала присвоение более коротких кодов чаще встречающимся (согласно статической модели) символам. Эффективность сжатия в данном режиме составила: коэффициент сжатия 42,86%, экономия места 57,14% (по сравнению с 8-битным представлением каждого из 14 символов).

**Режим B (Динамическая вероятность):** Вероятности символов вычислялись непосредственно на основе частот их появления в обрабатываемой строке «лопатнюкполина». Аналогично режиму А, была построена таблица кодов и выполнены процессы кодирования и декодирования. Декодирование также прошло успешно. Эффективность сжатия в динамическом режиме: коэффициент сжатия 39,29%, экономия места 60,71%.

**Режим C (Статическая вероятность):** Аналогично режиму A для Шеннона-Фано, вероятности символов были взяты из статической модели казахского алфавита. На основе этих вероятностей было построено бинарное дерево Хаффмана, и для каждого символа сгенерирован код. Процесс кодирования и последующего декодирования строки «лопатнюкполина» прошел успешно. Эффективность сжатия в данном режиме составила: коэффициент сжатия 42,86%, экономия места 57,14%.

**Режим D (Динамическая вероятность):**Подобно режиму B для Шеннона-Фано, вероятности символов рассчитывались динамически на основе самой строки «лопатнюкполина». Построение дерева Хаффмана и генерация кодов на этих данных позволили достичь более высокой степени сжатия. Кодирование и декодирование также были выполнены корректно. Эффективность сжатия в динамическом режиме: коэффициент сжатия 39,29%, экономия места 60,71%.

Проведенные эксперименты наглядно демонстрируют принцип работы алгоритма Шеннона-Фано, заключающийся в рекурсивном делении множества символов и присвоении префиксных кодов. Сравнение двух режимов показало, что адаптация вероятностной модели под конкретный текст (динамический режим) позволяет достичь лучшей степени сжатия по сравнению с использованием усредненной статической модели. Оба режима обеспечили корректное кодирование и полное восстановление информации.

В ходе работы были закреплены теоретические знания о статистических методах сжатия и получены практические навыки их реализации.