Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Пермский национальный исследовательский политехнический университет»
Электротехнический факультет
Кафедра «Информационные технологии и автоматизированные системы»

Дисциплина: «Защита информации»

Профиль: «Автоматизированные системы обработки информации и управления»

Семестр 5

ОТЧЕТ

по лабораторной работе №6

Тема: «Методы сжатия информации»

Выполнил: студент группы РИС-19-1б
Миннахметов Э.Ю. _____
Проверил: доцент кафедры ИТАС
Шереметьев В. Г. _____
Дата _____

Пермь, 2021

ЦЕЛЬ РАБОТЫ

Получить практические навыки по применению различных методов сжатия информации. Получить сравнительную характеристику сжатия информации, используя различные комбинации методов. Сделать вывод.

ЗАДАНИЕ

Вариант №14. Выполнить первое сжатие, используя алгоритм Хаффмана, и повторное методом арифметического кодирования.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Сжатие информации - проблема, имеющая достаточно давнюю историю, гораздо более давнюю, нежели история развития вычислительной техники, которая (история) обычно шла параллельно с историей развития проблемы кодирования и шифровки информации.

Все алгоритмы сжатия оперируют входным потоком информации, минимальной единицей которой является бит, а максимальной - несколько бит, байт или несколько байт.

Целью процесса сжатия, как правило, есть получение более компактного выходного потока информационных единиц из некоторого изначально некомпактного входного потока при помощи некоторого их преобразования.

Основными техническими характеристиками процессов сжатия и результатов их работы являются:

- степень сжатия (compress rating) или отношение (ratio) объемов исходного и результирующего потоков;
- скорость сжатия - время, затрачиваемое на сжатие некоторого объема информации входного потока, до получения из него эквивалентного выходного потока;
- качество сжатия - величина, показывающая на сколько сильно упакован выходной поток, при помощи применения к нему повторного сжатия по этому же или иному алгоритму.

Все способы сжатия можно разделить на две категории: обратимое и необратимое сжатие.

Под необратимым сжатием подразумевают такое преобразование входного потока данных, при котором выходной поток, основанный на определенном формате информации, представляет, с некоторой точки зрения, достаточно похожий по внешним характеристикам на входной поток объект, однако отличается от него объемом.

Степень сходства входного и выходного потоков определяется степенью соответствия некоторых свойств объекта (т.е. сжатой и несжатой информации в соответствии с некоторым определенным форматом данных), представляемого данным потоком информации.

Такие подходы и алгоритмы используются для сжатия, например данных растровых графических файлов с низкой степенью повторяемости байтов в потоке. При таком подходе используется свойство структуры формата графического файла и возможность представить графическую картинку приблизительно схожую по качеству отображения (для восприятия человеческим глазом) несколькими (а точнее n) способами. Поэтому, кроме степени или величины сжатия, в таких алгоритмах возникает понятие качества, т.к. исходное изображение в процессе сжатия изменяется, то под качеством можно понимать степень соответствия исходного и результирующего изображения, оцениваемая субъективно, исходя из формата информации. Для графических файлов такое соответствие определяется визуально, хотя имеются и соответствующие интеллектуальные алгоритмы и программы. Необратимое сжатие невозможно применять в областях, в которых необходимо иметь точное соответствие информационной структуры входного и выходного потоков. Данный подход реализован в популярных форматах представления видео и фото информации, известных как JPEG и JFIF алгоритмы и JPG и JIF форматы файлов.

Обратимое сжатие всегда приводит к снижению объема выходного потока информации без изменения его информативности, т.е. - без потери информационной структуры.

Более того, из выходного потока, при помощи восстанавливающего или декомпрессирующего алгоритма, можно получить входной, а процесс восстановления называется декомпрессией или распаковкой и только после процесса распаковки данные пригодны для обработки в соответствии с их внутренним форматом.

Перейдем теперь непосредственно к алгоритмическим особенностям обратимых алгоритмов и рассмотрим важнейшие теоретические подходы к сжатию данных, связанные с реализацией кодирующих систем и способы сжатия информации.

Сжатие способом кодирования серий (RLE)

Наиболее известный простой подход и алгоритм сжатия информации обратимым путем - это кодирование серий последовательностей (Run Length Encoding - RLE).

Суть методов данного подхода состоит в замене цепочек или серий повторяющихся байтов или их последовательностей на один кодирующий байт и счетчик числа их повторений.

Например:

44 44 44 11 11 11 11 01 33 FF 22 22 - исходная последовательность

03 44 04 11 00 03 01 03 FF 02 22 - сжатая последовательность

Первый байт указывает сколько раз нужно повторить следующий байт

Если первый байт равен 00, то затем идет счетчик, показывающий сколько за ним следует неповторяющихся данных.

Данные методы, как правило, достаточно эффективны для сжатия растровых графических изображений (BMP, PCX, TIF, GIF), т.к. последние содержат достаточно много длинных серий повторяющихся последовательностей байтов.

Недостатком метода RLE является достаточно низкая степень сжатия.

Арифметическое кодирование

Совершенно иное решение предлагает т.н. арифметическое кодирование. Арифметическое кодирование является методом, позволяющим упаковывать символы входного алфавита без потерь при условии, что известно распределение частот этих символов и является наиболее оптимальным, т.к. достигается теоретическая граница степени сжатия.

Предполагаемая требуемая последовательность символов, при сжатии методом арифметического кодирования рассматривается как некоторая двоичная дробь из интервала [0, 1). Результат сжатия представляется как последовательность двоичных цифр из записи этой дроби.

Идея метода состоит в следующем: исходный текст рассматривается как запись этой дроби, где каждый входной символ является «цифрой» с весом, пропорциональным вероятности его появления. Этим объясняется интервал, соответствующий минимальной и максимальной вероятностям появления символа в потоке.

Пример.

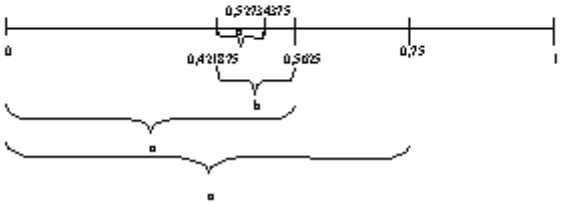
Пусть алфавит состоит из двух символов: a и b с вероятностями соответственно 0,75 и 0,25.

Рассмотрим наш интервал вероятностей [0, 1). Разобьем его на части, длина которых пропорциональна вероятностям символов. В нашем случае это [0; 0,75) и [0,75; 1). Суть алгоритма в следующем: каждому слову во входном алфавите соответствует некоторый подинтервал из интервала [0, 1) а пустому слову соответствует весь интервал

[0, 1). После получения каждого следующего символа интервал уменьшается с выбором той его части, которая соответствует новому символу. Кодом цепочки является интервал, выделенный после обработки всех ее символов, точнее, двоичная запись любой точки из этого интервала, а длина полученного интервала пропорциональна вероятности появления кодируемой цепочки.

Применим данный алгоритм для цепочки "aaba":

Шаг	Просмотренная цепочка	Интервал
0	Нет	[0; 1)
1	A	[0; 0,75)
2	Aa	[0; 0,5625)
3	Aab	[0,421875; 0,5625)
4	Aaba	[0,421875; 0,52734375)



Границы интервала вычисляются так берется расстояние внутри интервала (0,5625-0,421875=0,140625), делится на частоты [0; 0,10546875) и [0,10546875; 1) и находятся новые границы [0,421875; 0,52734375) и [0,52734375; 0,5625).

В качестве кода можно взять любое число из интервала, полученного на шаге 4, например, 0,43.

Алгоритм декодирования работает аналогично кодирующему. На входе 0,43 и идет разбиение интервала.

Продолжая этот процесс, мы однозначно декодируем все четыре символа. Для того, чтобы декодирующий алгоритм мог определить конец цепочки, мы можем либо передавать ее длину отдельно, либо добавить к алфавиту дополнительный уникальный символ – «конец цепочки».

Алгоритм Лемпеля-Зива-Велча (Lempel-Ziv-Welch - LZW)

Данный алгоритм отличают высокая скорость работы как при упаковке, так и при распаковке, достаточно скромные требования к памяти и простая аппаратная реализация.

Недостаток - низкая степень сжатия по сравнению со схемой двухступенчатого кодирования.

Предположим, что у нас имеется словарь, хранящий строки текста и содержащий порядка от 2-х до 8-ми тысяч пронумерованных гнезд. Запишем в первые 256 гнезд строки, состоящие из одного символа, номер которого равен номеру гнезда.

Алгоритм просматривает входной поток, разбивая его на подстроки и добавляя новые гнезда в конец словаря. Прочитаем несколько символов в строку s и найдем в словаре строку t - самый длинный префикс s.

Пусть он найден в гнезде с номером n. Выведем число n в выходной поток, переместим указатель входного потока на length(t) символов вперед и добавим в словарь новое гнездо, содержащее строку t+c, где c - очередной символ на входе (сразу после t). Алгоритм преобразует поток символов на входе в поток индексов ячеек словаря на выходе.

При практической реализации этого алгоритма следует учесть, что любое гнездо словаря, кроме самых первых, содержащих одно-символьные цепочки, хранит копию некоторого другого гнезда, к которой в конец приписан один символ. Вследствие этого можно обойтись простой списочной структурой с одной связью.

Пример: ABCABCABCABCABCABC - 1 2 3 4 6 5 7 7 7

1	A
2	B
3	C
4	AB
5	BC
6	CA
7	ABC
8	CAB
9	BCA

Двухступенчатое кодирование. Алгоритм Лемпеля-Зива

Гораздо большей степени сжатия можно добиться при выделении из входного потока повторяющихся цепочек - блоков, и кодирования ссылок на эти цепочки с построением хеш таблиц от первого до n-го уровня.

Метод, о котором и пойдет речь, принадлежит Лемпелю и Зиву и обычно называется LZ-compression.

Суть его состоит в следующем: упаковщик постоянно хранит некоторое количество последних обработанных символов в буфере. По мере обработки входного потока вновь поступившие символы попадают в конец буфера, сдвигая предшествующие символы и вытесняя самые старые.

Размеры этого буфера, называемого также скользящим словарем (sliding dictionary), варьируются в разных реализациях кодирующих систем.

Экспериментальным путем установлено, что программа LHarc использует 4-килобайтный буфер, LHA и PKZIP - 8-ми, а ARJ - 16-килобайтный.

Затем, после построения хеш таблиц алгоритм выделяет (путем поиска в словаре) самую длинную начальную подстроку входного потока, совпадающую с одной из подстрок в словаре, и выдает на выход пару (length, distance), где length - длина найденной в словаре подстроки, а distance - расстояние от нее до входной подстроки (то есть фактически индекс подстроки в буфере, вычтенный из его размера).

В случае, если такая подстрока не найдена, в выходной поток просто копируется очередной символ входного потока.

В первоначальной версии алгоритма предлагалось использовать простейший поиск по всему словарю. Однако, в дальнейшем, было предложено использовать двоичное дерево и хеширование для быстрого поиска в словаре, что позволило на порядок поднять скорость работы алгоритма.

Таким образом, алгоритм Лемпеля-Зива преобразует один поток исходных символов в два параллельных потока длин и индексов в таблице (length + distance).

Очевидно, что эти потоки являются потоками символов с двумя новыми алфавитами, и к ним можно применить один из упоминавшихся выше методов (RLE, кодирование Хаффмена или арифметическое кодирование).

Так мы приходим к схеме двухступенчатого кодирования - наиболее эффективной из практически используемых в настоящее время. При реализации этого метода необходимо добиться согласованного вывода обоих потоков в один файл. Эта проблема обычно решается путем поочередной записи кодов символов из обоих потоков.

Пример:

Первая ступень

abcabcabcabcabc - 1 a 1 b 1 c 3 3 6 3 9 3 12 3

Вторая ступень - исключение большой группы повторяющихся последовательностей

1 a 1 b 1 c 12 3

и сжатие RLE, кодирование Хаффмена , арифметическое кодирование

Алгоритм Хаффмана

Сжимая файл по алгоритму Хаффмана первое что мы должны сделать - это необходимо прочитать файл полностью и подсчитать сколько раз встречается каждый символ из расширенного набора ASCII.

Если мы будем учитывать все 256 символов, то для нас не будет разницы в сжатии текстового и EXE файла.

После подсчета частоты вхождения каждого символа, необходимо просмотреть таблицу кодов ASCII и сформировать бинарное дерево.

Пример:

Мы имеем файл длинной в 100 байт и имеющий 6 различных символов в себе . Мы подсчитали вхождение каждого из символов в файл и получили следующее :

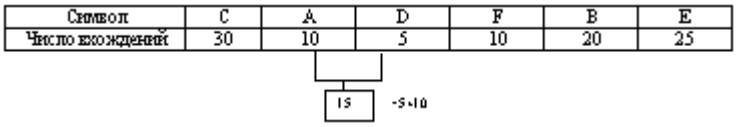
Символ	A	B	C	D	E	F
Число вхождений	10	20	30	5	25	10

Теперь мы берем эти числа и будем называть их частотой вхождения для каждого символа.

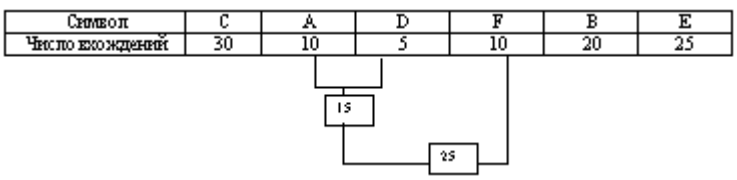
Символ	C	E	B	F	A	D
Число вхождений	30	25	20	10	10	5

Мы возьмем из последней таблицы 2 символа с наименьшей частотой. В нашем случае это D (5) и какой либо символ из F или A (10), можно взять любой из них например A.

Сформируем из "узлов" D и A новый "узел", частота вхождения для которого будет равна сумме частот D и A :

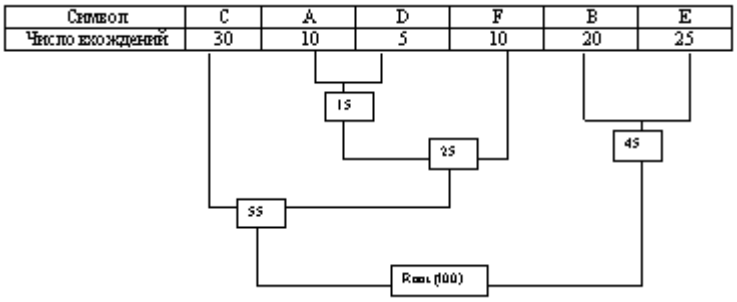


Номер в рамке - сумма частот символов D и A. Теперь мы снова ищем два символа с самыми низкими частотами вхождения. Исключая из просмотра D и A и рассматривая вместо них новый "узел" с суммарной частотой вхождения. Самая низкая частота теперь у F и нового "узла". Снова сделаем операцию слияния узлов :



Рассматриваем таблицу снова для следующих двух символов (B и E).

Мы продолжаем в этот режим пока все "дерево" не сформировано, т.е. пока все не сведется к одному узлу.



Теперь когда наше дерево создано, мы можем кодировать файл . Мы должны всегда начинать из корня (Root). Кодируя первый символ (лист дерева C) Мы прослеживаем вверх по дереву все повороты ветвей и если мы делаем левый поворот, то запоминаем 0-й бит, и аналогично 1-й бит для правого поворота. Так для C, мы будем идти влево к 55 (и запомним 0), затем снова влево (0) к самому символу . Код Хаффмана для нашего символа C - 00. Для следующего символа (A) у нас получается - лево,право,лево,лево , что выливается в последовательность 0100. Выполнив выше сказанное для всех символов получим

C = 00 (2 бита)

A = 0100 (4 бита)

D = 0101 (4 бита)

F = 011 (3 бита)

B = 10 (2 бита)

E = 11 (2 бита)

При кодировании заменяем символы на данные последовательности.

ХОД РАБОТЫ

На рисунке 1 представлена главная форма программы.

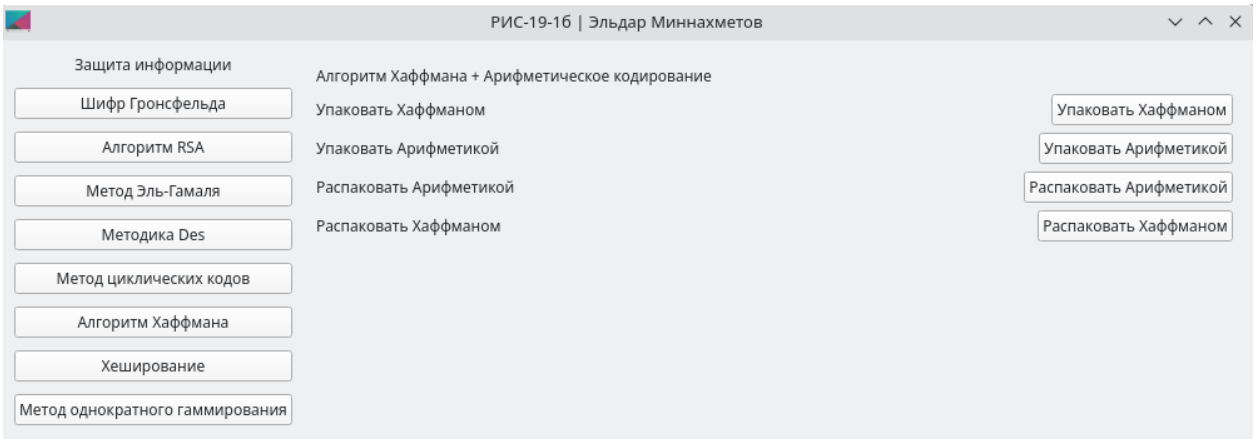


Рисунок 1 – Главная форма программы.

Пример работы программы представлен на рисунке 2.

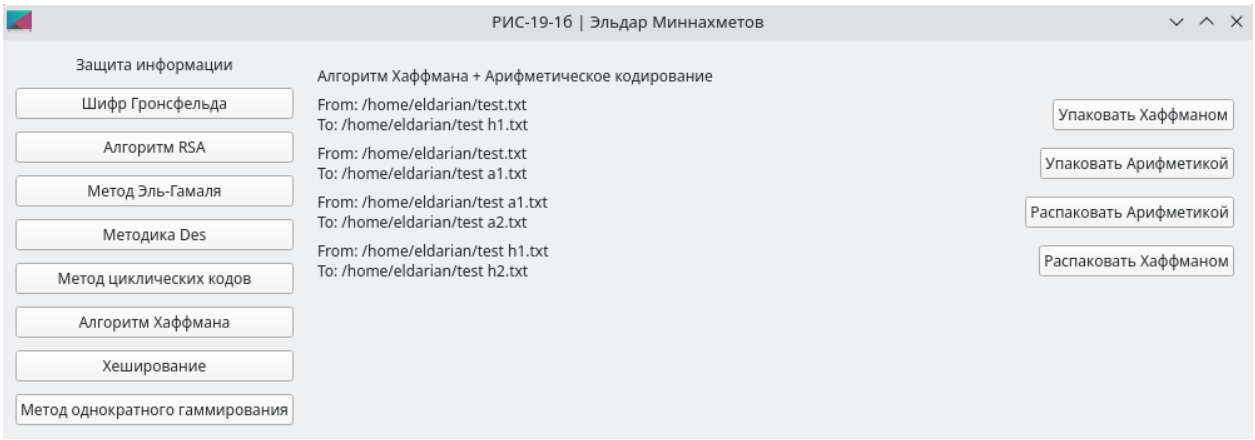


Рисунок 2 – Пример работы программы.

Текст исходного сообщения представлен на рисунке 3.

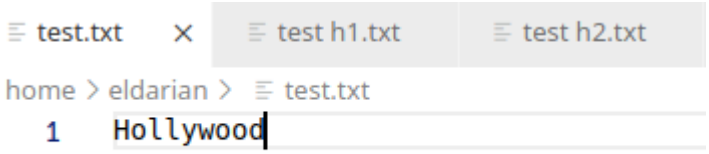


Рисунок 3 – Текст исходного файла.

Текст сообщения, сжатого по алгоритму Хаффмана, представлен на рисунке 4.

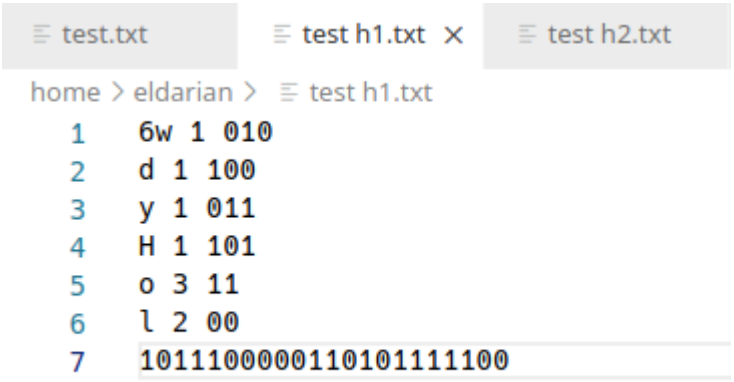


Рисунок 4 – Результат работы алгоритма Хаффмана.

Результат распаковки сообщения, сжатого по алгоритму Хаффмана, представлен на рисунке 5.

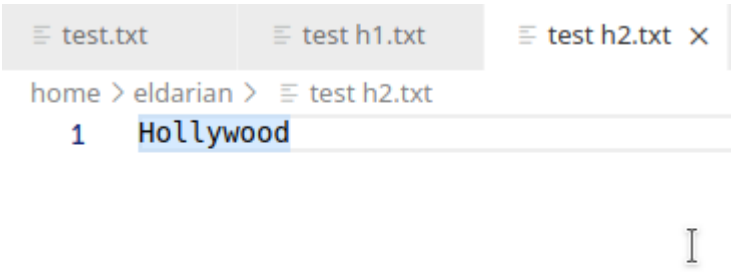


Рисунок 5 – Результат распаковки по алгоритму Хаффмана.

Текст сообщения, сжатого Арифметическим кодированием, представлен на рисунке 6.

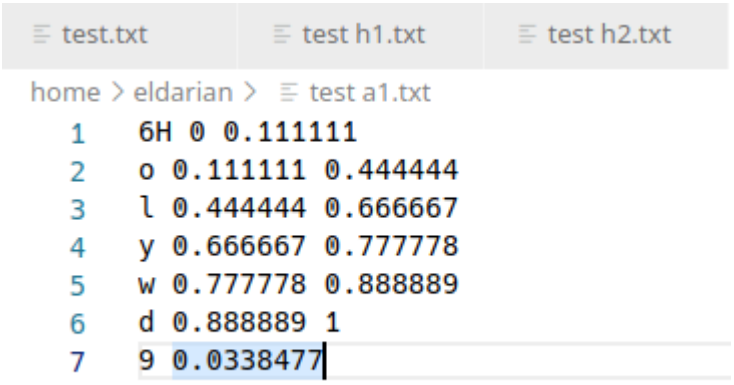


Рисунок 6 – Результат работы Арифметического кодирования

Результат распаковки сообщения, сжатого Арифметическим кодированием, представлен на рисунке 7.

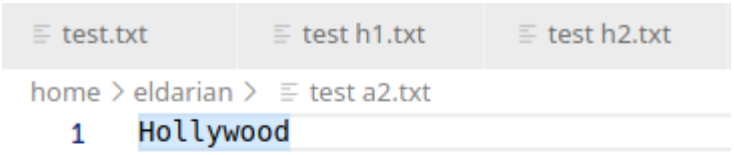


Рисунок 7 – Результат распаковки Арифметическим кодированием

Вывод, дабы наглядно показать, как работают алгоритмы сжатия, было принято решение упаковывать в текстовые файлы, а не в бинарные, чтобы действительно добиться эффекта сжатия. Кроме того, упаковка Арифметическим кодированием задействует работу с числами с плавающей точкой и при сжатии больших файлов – это чревато потерей точности, после чего последует некорректная распаковка. Именно поэтому, в примере работы программы после упаковки алгоритмом Хаффмана не следует упаковка сжатого файла методом Арифметического кодирования – результат упаковки алгоритмом Хаффмана слишком большой для корректности сжатия Арифметического кодирования.

ПРИЛОЖЕНИЕ А

Листинг файла Huffman.h

```
#pragma once

#include <QHash>
#include <QString>
#include <QTextStream>

#include <vector>
#include <queue>
using namespace std;

namespace Huffman {
  struct Node {
    QChar ch;
    int freq;
    Node *left, *right;

    bool operator()(Node *l, Node *r);

  };

  Node *node(QChar ch, int freq, Node *left, Node *right);
  Node *tree(QHash<QChar, int> freq);

  void encode(Node *root, QString text, QHash<QChar, QString> &huffmanCode);
  QString decode(QString text, Node *root);
  QChar decode(Node *root, int &index, QString str);

  class Encoder {
  private:
    QString text;
  public:
    friend QTextStream &operator>>(QTextStream &in, Encoder &encoder);
    friend QTextStream &operator<<(QTextStream &out, const Encoder &encoder);

  };

  class Decoder {
  private:
    QString text;
  public:
    friend QTextStream &operator>>(QTextStream &in, Decoder &decoder);
    friend QTextStream &operator<<(QTextStream &out, const Decoder &decoder);
  };

}
```

ПРИЛОЖЕНИЕ Б

Листинг файла Huffman.cpp

```
#include "Huffman.h"

#include <QHash>
#include <QBitArray>

#include <iostream>

Huffman::Node* Huffman::node(QChar ch, int freq, Node* left, Node* right) {
  Node* node = new Node;
  node->ch = ch;
  node->freq = freq;
  node->left = left;
  node->right = right;
  return node;
}

Huffman::Node* Huffman::tree(QHash<QChar, int> freq) {
  priority_queue<Node*, vector<Node*>, Node> pq;
  for (auto pair = freq.begin(); pair != freq.end(); ++pair) {
    pq.push(node(pair.key(), pair.value(), nullptr, nullptr));
  }
  while (pq.size() != 1) {
    Node *left = pq.top(); pq.pop();
    Node *right = pq.top();          pq.pop();
    int sum = left->freq + right->freq;
    pq.push(node('\0', sum, left, right));
  }
  return pq.top();
}

bool Huffman::Node::operator()(Node* l, Node* r) {
  return l->freq > r->freq;
}

void Huffman::encode(Node* root, QString text, QHash<QChar, QString> &huffmanCode) {
  if (root == nullptr) {
    return;
  }
  if (!root->left && !root->right) {
    huffmanCode[root->ch] = text;
  }
  encode(root->left, text + "0", huffmanCode);
  encode(root->right, text + "1", huffmanCode);
}

QString Huffman::decode(QString text, Node* root) {
  QString str = "";
  int index = -1;
  while (index < (int)text.size() - 2) {
    str += decode(root, index, text);
  }
  return str;
}

QChar Huffman::decode(Node* root, int &index, QString str) {
  if (root == nullptr) {
```

```
        return '\0';
    }
    if (!root->left && !root->right) {
        return root->ch;
    }
    index++;
    if (str[index] == '0') {
        return decode(root->left, index, str);
    } else {
        return decode(root->right, index, str);
    }
}

QTextStream& Huffman::operator >> (QTextStream &in, Encoder& encoder) {
    encoder.text = in.readAll();
    return in;
}

QTextStream& Huffman::operator << (QTextStream &out, const Encoder& encoder) {
    QHash<QChar, int> freq;
    for (QChar ch: encoder.text) {
        freq[ch]++;
    }
    Node* root = tree(freq);
    QHash<QChar, QString> huffmanCode;
    encode(root, "", huffmanCode);
    out << huffmanCode.size();
    for (auto pair = freq.begin(); pair != freq.end(); ++pair) {
        out << pair.key() << " " << pair.value() << " " << huffmanCode[pair.key()] << "\n";
    }
    for (QChar ch: encoder.text) {
        out << huffmanCode[ch];
    }
    return out;
}

QTextStream& Huffman::operator >> (QTextStream &in, Decoder& decoder) {
    int n;
    in >> n;
    QHash<QChar, int> freq;
    QHash<QChar, QString> huffmanCode;
    for (int i = 0; i < n; ++i) {
        QChar key, space, nl;
        int fq;
        QString code;
        in >> key >> space >> fq >> space >> code >> nl;
        freq[key] = fq;
        huffmanCode[key] = code;
    }

    QString packed;
    in >> packed;
    decoder.text = decode(packed, tree(freq));
    return in;
}

QTextStream& Huffman::operator << (QTextStream &out, const Decoder& decoder) {
    out << decoder.text;
    return out;
}
```

ПРИЛОЖЕНИЕ В

Листинг файла Arithmetic.h

```
#pragma once

#include <QString>
#include <QTextStream>

namespace Arithmetic {
  struct Freq {
    QChar symbol;
    int count;
  };

  struct Range {
    double lower, upper;
  };

  struct Symbol {
    QChar symbol;
    Range range;
  };

  struct Encoded {
    QList<Symbol> list;
    int count;
    double code;
  };

  Encoded encode(QString text);
  QString decode(Encoded encoded);

  int &count(QList<Freq> &list, const QChar &symbol);
  Range findRange(QList<Symbol> &list, const QChar &symbol);
  Range calculate(const Range &last, const Range &next);
  QList<Symbol> symbols(QString text);
  Symbol findBetween(const QList<Symbol> &list, const double &code);

  class Encoder {
  private:
    QString text;
  public:
    friend QTextStream &operator>>(QTextStream &in, Encoder &encoder);
    friend QTextStream &operator<<(QTextStream &out, const Encoder &encoder);

  };

  class Decoder {
  private:
    QString text;
  public:
    friend QTextStream &operator>>(QTextStream &in, Decoder &decoder);
    friend QTextStream &operator<<(QTextStream &out, const Decoder &decoder);

  };

}
```

ПРИЛОЖЕНИЕ Г

Листинг файла Arithmetic.cpp

```
#include "Arithmetic.h"

int& Arithmetic::count(QList<Freq>& list, const QChar& symbol) {
  for(auto& item : list) {
    if(item.symbol == symbol) {
      return item.count;
    }
  }
  list.append({symbol, 0});
  return list.back().count;
}

Arithmetic::Range Arithmetic::findRange(QList<Symbol>& list, const QChar& symbol) {
  for(auto& item : list) {
    if(item.symbol == symbol) {
      return item.range;
    }
  }
  return {0, 1};
}

Arithmetic::Range Arithmetic::calculate(const Range& last, const Range& next) {
  auto calc = [&last](double d) -> double {
    return last.lower + (last.upper - last.lower) * d;
  };
  return {calc(next.lower), calc(next.upper)};
}

QList<Arithmetic::Symbol> Arithmetic::symbols(QString text) {
  int n = text.size();
  QList<Freq> freq;
  QList<Symbol> result;
  for(auto ch : text) {
    ++count(freq, ch);
  }
  double lastLower = 0.;
  for(auto& item : freq) {
    double fc = (double)item.count / n;
    result.append({item.symbol, {lastLower, lastLower + fc}});
    lastLower += fc;
  }
  return result;
}

Arithmetic::Encoded Arithmetic::encode(QString text) {
  Encoded result;
  result.list = symbols(text);
  result.count = text.count();
  auto &list = result.list;
  Range range = {0, 1};
  for(auto ch : text) {
    range = calculate(range, findRange(list, ch));
  }
  result.code = (range.lower + range.upper) / 2;
  return result;
}
```

```
Arithmetic::Symbol Arithmetic::findBetween(const QList<Symbol>& list, const double& code) {
    for(auto& item : list) {
        if(item.range.lower <= code && code < item.range.upper) {
            return item;
        }
    }
    return {'a', {0, 1}};
}

QString Arithmetic::decode(Arithmetic::Encoded encoded) {
    QString result;
    double code = encoded.code;
    for(int i = 0; i < encoded.count; ++i) {
        auto symbol = findBetween(encoded.list, code);
        result += symbol.symbol;
        code = (code - symbol.range.lower) / (symbol.range.upper - symbol.range.lower);
    }
    return result;
}

QTextStream& Arithmetic::operator >> (QTextStream &in, Arithmetic::Encoder& encoder) {
    encoder.text = in.readAll();
    return in;
}

QTextStream& Arithmetic::operator << (QTextStream &out, const Arithmetic::Encoder& encoder) {
    auto ec = Arithmetic::encode(encoder.text);
    out << ec.list.size();
    for(auto item : ec.list) {
        out << item.symbol << " " << item.range.lower << " " << item.range.upper << "\n";
    }
    out << ec.count << " " << ec.code;
    return out;
}

QTextStream& Arithmetic::operator >> (QTextStream &in, Arithmetic::Decoder& decoder) {
    Arithmetic::Encoded ec;
    int n;
    in >> n;
    for(int i = 0; i < n; ++i) {
        QChar symbol, space, nl;
        double lower, upper;
        in >> symbol >> space >> lower >> space >> upper >> nl;
        ec.list.append({symbol, {lower, upper}});
    }
    in >> ec.count;
    in >> ec.code;
    decoder.text = Arithmetic::decode(ec);
    return in;
}

QTextStream& Arithmetic::operator << (QTextStream &out, const Arithmetic::Decoder& decoder) {
    out << decoder.text;
    return out;
}
```

ПРИЛОЖЕНИЕ Д

Листинг файла HuffmanTask.h

```
#pragma once

#include <QWidget>
#include <QLabel>

#include "Task.h"

class QVBoxLayout;
class QHBoxLayout;
class QPushButton;

class HuffmanTask: public QObject, public Task {
Q_OBJECT

private:
   QVBoxLayout *lytMain;
   QHBoxLayout *lytPackHuff;
   QHBoxLayout *lytUnpackHuff;
   QHBoxLayout *lytPackArif;
   QHBoxLayout *lytUnpackArif;

   QLabel *lblName;
   QLabel *lblPackHuff;
   QLabel *lblUnpackHuff;
   QLabel *lblPackArif;
   QLabel *lblUnpackArif;

   QPushButton *btnPackHuff;
   QPushButton *btnUnpackHuff;
   QPushButton *btnPackArif;
   QPushButton *btnUnpackArif;

public:
   HuffmanTask(): Task("Алгоритм Хаффмана") {}
   void initWidget(QWidget *wgt) override;

private slots:
   void packHuff();
   void unpackHuff();
   void packArif();
   void unpackArif();

private:
   template<class TCipher>
   void doAnything(QLabel *lbl);

   template<class TIn, class TOut>
   void doAnything(QLabel *lbl, void (*fun)(TIn&, TOut&));

};
```

ПРИЛОЖЕНИЕ Е

Листинг файла HuffmanTask.cpp

```
#include "HuffmanTask.h"
#include "Huffman.h"
#include "Arithmetic.h"

#include <QLabel>
#include <QLineEdit>
#include <QTextStream>
#include <QHBoxLayout>
#include <QPushButton>
#include <QFileDialog>
#include <QMessageBox>

void HuffmanTask::initWidget(QWidget *wgt) {
    lytMain = new QVBoxLayout;
    lytPackHuff = new QHBoxLayout;
    lytUnpackHuff = new QHBoxLayout;
    lytPackArif = new QHBoxLayout;
    lytUnpackArif = new QHBoxLayout;

    lblName = new QLabel("Алгоритм Хаффмана + Арифметическое кодирование");
    lblPackHuff = new QLabel("Упаковать Хаффманом");
    lblUnpackHuff = new QLabel("Распаковать Хаффманом");
    lblPackArif = new QLabel("Упаковать Арифметикой");
    lblUnpackArif = new QLabel("Распаковать Арифметикой");

    btnPackHuff = new QPushButton("Упаковать Хаффманом");
    btnUnpackHuff = new QPushButton("Распаковать Хаффманом");
    btnPackArif = new QPushButton("Упаковать Арифметикой");
    btnUnpackArif = new QPushButton("Распаковать Арифметикой");

    wgt->setLayout(lytMain);
    lytMain->addWidget(lblName);
    lytMain->addLayout(lytPackHuff);
    lytMain->addLayout(lytPackArif);
    lytMain->addLayout(lytUnpackArif);
    lytMain->addLayout(lytUnpackHuff);

    lytPackHuff->addWidget(lblPackHuff, 4);
    lytPackHuff->addWidget(btnPackHuff, 1);
    lytPackArif->addWidget(lblPackArif, 4);
    lytPackArif->addWidget(btnPackArif, 1);
    lytUnpackHuff->addWidget(lblUnpackHuff, 4);
    lytUnpackHuff->addWidget(btnUnpackHuff, 1);
    lytUnpackArif->addWidget(lblUnpackArif, 4);
    lytUnpackArif->addWidget(btnUnpackArif, 1);

    lytMain->setAlignment(Qt::Alignment::enum_type::AlignTop);

    connect(btnPackHuff, SIGNAL(released()), this, SLOT(packHuff()));
    connect(btnUnpackHuff, SIGNAL(released()), this, SLOT(unpackHuff()));
    connect(btnPackArif, SIGNAL(released()), this, SLOT(packArif()));
    connect(btnUnpackArif, SIGNAL(released()), this, SLOT(unpackArif()));
}

void HuffmanTask::packHuff() {
    doAnything<Huffman::Encoder>(lblPackHuff);
}
```

```
void HuffmanTask::unpackHuff() {
    doAnything<Huffman::Decoder>(lblUnpackHuff);
}

void HuffmanTask::packArif() {
    doAnything<Arithmetic::Encoder>(lblPackArif);
}

void HuffmanTask::unpackArif() {
    doAnything<Arithmetic::Decoder>(lblUnpackArif);
}

template<class TCipher>
void HuffmanTask::doAnything(QLabel *lbl) {
    doAnything<QTextStream, QTextStream>(lbl, [](QTextStream &in, QTextStream &out) -> void {
        TCipher decoder;
        in >> decoder;
        out << decoder;
    });
}

template<class TIn, class TOut>
void HuffmanTask::doAnything(QLabel *lbl, void (*fun)(TIn&, TOut&)) {
    QFile source(QFileDialog::getOpenFileName());
    QFile target(QFileDialog::getSaveFileName());
    lbl->setText(QString::asprintf(
            "From: %s\nTo: %s",
            source.fileName().toStdString().c_str(),
            target.fileName().toStdString().c_str()));
    if(!source.open(QIODevice::ReadOnly) || !target.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(nullptr, "Предупреждение", "Исходный файл не выбран или не
существует");
        return;
    }
    TIn in(&source);
    TOut out(&target);
    fun(in, out);
    source.close();
    target.close();
}
```