МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №6

Тема: Кодирование и сжатие данных методами без потерь

Дисциплина Структуры и алгоритмы обработки данных

Выполнил студент Анисимов Д.Н.

группа ИКБО-24-20

Москва 2021

Содержание

Цель работы

Получить знания алгоритмов кодирования и сжатия данных методами без потерь и навыки их применения.

Отчет по заданию.

Задание 1 Исследование алгоритмов сжатия на примерах

1) Выполнить каждую задачу варианта, представив алгоритм решения в виде таблицы и указав результат сжатия. Примеры оформления решения представлены в Приложении1 этого документа.
2) Описать процесс восстановления сжатого текста.
3) Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта.

Индивидуальный вариант:

Вариант	Закодировать фразу методами Шеннона–Фано	Сжатие данных по методу Лемпеля–Зива LZ77 Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу:	Закодировать следующую фразу, используя код LZ78
5	Прибавь к ослиной голове еще одну, получишь две. Но сколько б ни было ослов, они и двух не свяжут слов.	101000100101010001011	какатанекатанекатата

Алгоритм решения:

1. Для метода Шеннона-Фано:

Закодировать фразу «Прибавь к ослиной голове ещё одну, получишь две. Но сколько б ни было ослов, они и двух не свяжут слов.», используя метод Шеннона–Фано.

Таблица 1.

Символ	Кол-во	1-я цифра	2-я цифра	3-я цифра	4-я цифра	5-я цифра	6-я цифра	7-я цифра	Код	Кол-во бит
Пробел	19	0	0	0					000	57
о	14	0	0	1					001	42
в	7	0	1	0					010	21
л	7	0	1	1	0				0110	28
и	6	0	1	1	1				0111	24
н	6	1	0	0	0				1000	24
с	5	1	0	0	1				1001	20
е	4	1	0	1	0	0			10100	20
у	4	1	0	1	0	1			10101	20
б	3	1	0	1	1	0			10110	15
ь	3	1	0	1	1	1			10111	15
к	3	1	1	0	0	0			11000	15
д	3	1	1	0	0	1			11001	15
п	2	1	1	0	1	0			11010	10
,	2	1	1	0	1	1	0		110110	12
.	2	1	1	0	1	1	1		110111	12
р	1	1	1	1	0	0	0		111000	6
а	1	1	1	1	0	0	1	0	1110010	7
й	1	1	1	1	0	0	1	1	1110011	7
г	1	1	1	1	0	1	0		111010	6
щ	1	1	1	1	0	1	1	0	1110110	7
ё	1	1	1	1	0	1	1	1	1110111	7
ч	1	1	1	1	1	0	0		111100	6

ш	1	1	1	1	1	0	1	0	1111010	7
ы	1	1	1	1	1	0	1	1	1111011	7
х	1	1	1	1	1	1	0	0	1111100	7
я	1	1	1	1	1	1	0	1	1111101	7
ж	1	1	1	1	1	1	1	0	1111110	7
т	1	1	1	1	1	1	1	1	1111111	7
										438

Незакодированная фраза – 103*8 бит = 824 бит.

Закодированная фраза – 438 бит.

Каждый код символа уникален, значит раскодировать можно, зная коды символов и заменяя их обратно.

2. Для метода Лемпеля – Зива LZ77

Для сжатия двоичного кода:

10100010010100010011

Таблица 2.

Содержимое окна (сжимаемый текст)	Содержимое буфера	Код
10100010010100010011	1	<0,0,1>
10100010010100010011	0	<0,0,0>
10100010010100010011	100	<2,2,0>
10100010010100010011	01001	<4,4,1>
10100010010100001011	010100	<2,5,0>
10100010010100001011	01011	<6,4,1>

Результат сжатия: (<0,0,1>, <0,0,0>, <2,2,0>, <4,4,1>, <2,5,0>, <6,4,1>)

Чтобы восстановить строку, нужно последовательно пройтись по коду (<символов назад, сколько пройти, символ в конце прохода>).

3. Для метода Лемпеля – Зива LZ78

Для фразы: какатанекатанекатата

Таблица 3.

Содержимое словаря	Содержимое считаной строки	Код
	к	<0, к>
к	а	<0, а>
к, а	ка	<1, а>
к, а, ка	т	<0, т>
к, а, ка, т	ан	<2, н>
к, а, ка, т, ан	е	<0, е>
к, а, ка, т, ан, е	кат	<3, т>
к, а, ка, т, ан, е, кат	ане	<5, е>
к, а, ка, т, ан, е, кат, ане	ката	<6, а>
к, а, ка, т, ан, е, кат, ане, ката	та	<4, а>

Результат сжатия: (<0, к>, <0, а>, <1, а>, <0, т>, <2, н>, <0, е>, <3, т>, <5, е>, <6, а>, <4, а>)

Чтобы восстановить строку, проходимся по коду, строя словарь и беря из него значения.

Задание 2 Разработать программы сжатия и восстановления текста методами Хаффмана и Шеннона – Фано.

1) Реализовать и отладить программы.

2) Сформировать отчет по разработке каждой программы в соответствии с требованиями.

- По методу Шеннона-Фано привести: постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования, код и результаты тестирования. Рассчитать коэффициент сжатия. Сравнить результат сжатия вашим алгоритмом с результатом любого архиватора.
- по методу Хаффмана выполнить и отобразить результаты выполнения всех требований, предъявленных в задании и оформить разработку программы: постановка, подход к решению, код, результаты тестирования.

Для выполнения работы необходимо выполнить следующие действия:

2.1 Построить таблицу частот встречаемости символов в исходной строке символов для чего сформировать алфавит исходной строки и посчитать количество вхождений (частот) символов и их вероятности появления.

2.2 Отсортировать алфавит в порядке убывания частот появления символов.

2.3 Построить дерево кодирования Хаффмана.

2.4 Присвоить ветвям коды.

2.5 Определить коды символов.

2.6 Провести кодирование исходной строки.

2.7 Рассчитать коэффициенты сжатия относительно кодировки ASCII и относительно равномерного кода.

2.8 Рассчитать среднюю длину полученного кода и его дисперсию.

Кодирование строки “Фамилия Имя Отчество” по алгоритму Хаффмана.

2.1, 2.2 Таблица частот встречаемости символов.

Таблица 4.

Алфавит	и	а	в	л	н	о	< >
Кол. вх.	5	2	2	2	2	2	2
Вероятн.	0.232	0.077	0.077	0.077	0.077	0.077	0.077

Таблица 5.

Алфавит	А	Д	Н	е	к	м	с	ч
Кол. вх.	1	1	1	1	1	1	1	1
Вероятн.	0.038	0.038	0.038	0.038	0.038	0.038	0.038	0.038

2.3, 2.4 Дерево кодирования Хаффмана с кодами, присвоенными ветвям.

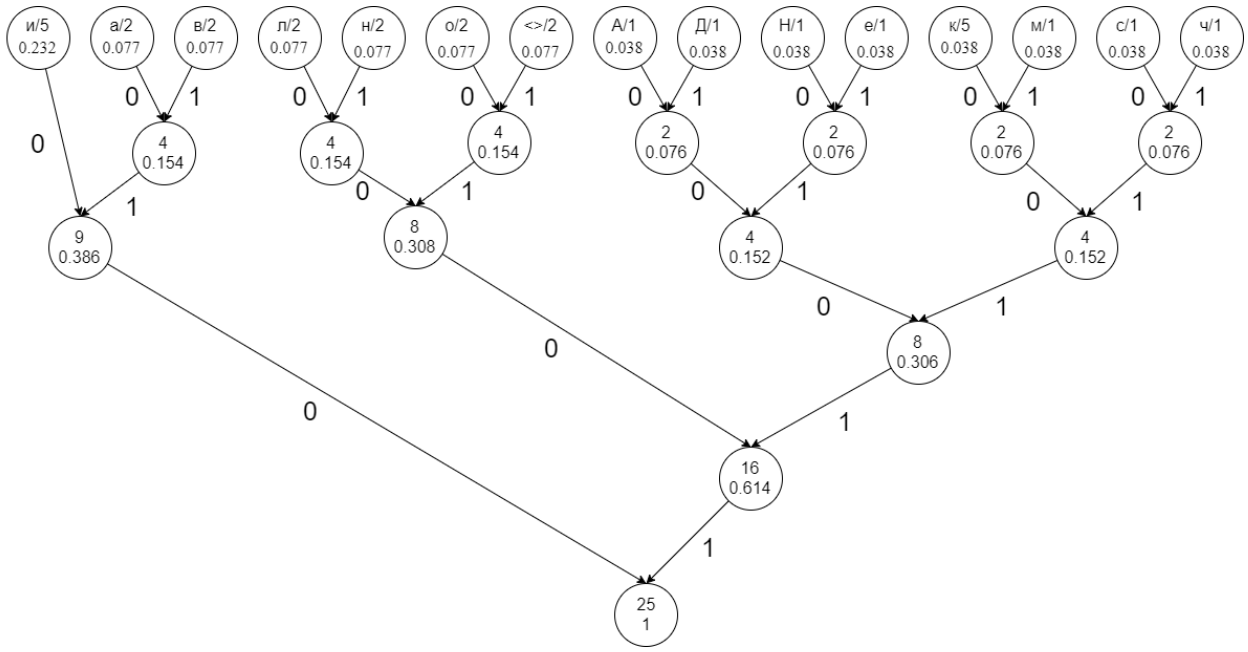

Рис. 1. Дерево Хаффмана для ФИО.

2.5 Коды символов:

и – 00, а – 010, в – 011, л – 1000, н – 1001, о – 1010, <> - 1011, А – 11000, Д – 11001, Н – 11010, е – 11011, к – 11100, м – 11101, с – 11110, ч – 11111.

2.6 Кодирование исходной строки:

11000 1001 00 11110 00 11101 1010 011 1011 11001 010 1001 00 00 1000 1011 11010 00 11100 1010 1000 010 11011 011 00 11111

2.7 Рассчитаем коэффициент сжатия относительно использования кодировки ASCII (8 бит/символ).

$L_{ASCII} = 8 * 26 = 208$ бит.

$L_{Huff} = 5 * 2 + 2 * 2 * 3 + 2 * 4 * 4 + 5 * 8 = 94$ бит.

Следовательно, коэффициент сжатия будет равен

$$К_{СЖ} = \frac{L_{ASCII}}{L_{Huff}} \approx 2{,}213$$

Коэффициент сжатия относительно равномерного кода (5 бит/символ, т. к. у нас всего 26 символов) будет равен

$$К_{СЖ} = \frac{5 * 26}{L_{Huff}} = \frac{130}{94} \approx 1{,}383$$

2.8 Рассчитаем среднюю длину полученного кода по формуле $l_{ср} = \sum_s p_s * l_s$, где s — множество символов алфавита; p_s — вероятность появления символа; l_s — количество бит в коде символа.
Для полученного кода средняя длина будет равна

$l_{ср} = 0{,}232 * 2 + 0{,}077 * 3 * 2 + 0{,}077 * 4 * 4 + 0{,}038 * 5 * 8 = 3{,}678$ бит/символ.

Дисперсия рассчитывается по формуле

$$\delta = \sum_s p_s (l_s - l_{ср})^2$$

Для полученного кода дисперсия будет равна

$\delta = 0{,}232 * (2 - 3{,}678)^2 + 0{,}077 * (3 - 3{,}678)^2 * 2 + 0{,}077 * (4 - 3{,}678)^2 * 4 + 0{,}038 * (5 - 3{,}678)^2 * 8 = 1{,}287261.$

Описание алгоритма:

Создание словарей частоты и вероятности символов:

Присваиваем каждому символу количество его вхождений в тексте для частоты и частоту, разделённую на количество символов в тексте для вероятности.

Создание префиксного дерева Шеннона-Фано:

Имеем узел-корень со словарём, отсортированным по убыванию с повторяющимися ключами: вероятность и значением равным соответствующему ей символу.

Разделяем словарь по вероятностям на две группы, так чтобы разделение было максимально ровным.

Создаём новые 2 узла и перемещаем словарь левой группы в первый узел и словарь правой группы во второй узел.

Создаём ссылки у узла, из которого мы брали словарь на созданные узлы, первый новый узел становится левым сыном, а второй новый узел становится правым сыном.

Повторяем разделение и создание ссылок у новых узлов, если в их словарях больше 1 элемента.

Создание префиксного дерева Хаффмана:

Присваиваем по отдельности элементы словаря ключей - вероятность и значений – символ, новым узлам и кладём эти узлы в приоритетную очередь, в которой всегда в начале узел с наименьшей суммой вероятностей элементов его словаря.

Пока не останется только один узел в очереди, который станет корнем:

- Вытаскиваем из очереди 2 узла.
- Объединяем их словари в один, создаём новый узел и присваиваем ему этот словарь, создаём ссылки у нового узла – левый сын — это первый вытащенный из очереди узел, а правый сын – второй вытащенный из очереди.
- Кладём этот новый узел в очередь.

Кодирование сообщения:

Получаем строку и корень дерева Шеннона-Фано или Хаффмана.

Рекурсивно проходим по дереву сверху вниз, переходя по ссылкам на сыновей и если мы прошли на ссылку левого сына, то добавляем в передаваемый код 1, а если на ссылку правого сына, то добавляем в передаваемый код 0.

Когда у узла не будет сыновей, то мы берём из словаря этого узла символ и тот код который получился в итоге рекурсивного обхода и заменяем все вхождения необходимого символа в исходной строке на этот код.

Декодирование сообщения:

Получаем закодированную строку и корень дерева Шеннона-Фано или Хаффмана.

Создаём словарь символ: код, и заполняем его рекурсивно обходя дерево сверху вниз, добавляя в передаваемый код 1, если мы прошли влево и добавляя в передаваемый код 0, если мы прошли вправо, когда мы доберёмся до узла без ссылок на сыновей, то создадим новый элемент в словаре – рекурсивно полученный код и символ из словаря узла.

Используя полученный словарь, декодируем сообщение, постепенно проходя по строке, создавая подстроки и сравнивая их с кодами в словаре и если подстрока равна коду из словаря, то заменяем её на нужный символ.

Код программы:

Листинг 1. Структура узла деревьев Шеннона-Фано и Хаффмана.

```
struct node
{
	multimap<float, char, greater<float>> probabilities; // Словарь с повтором ключей (вероятность : символ), всегда отсортирован по убыванию
	node* left = NULL; // Левое поддерево
	node* right = NULL; // Правое поддерево
};
```

Листинг 2. Создание словаря Символ:Частота.

```
map<char, int> build_frequency(string input) {
	map<char, int> m; // Словарь (символ : кол-во вхождений в тексте)
	for (auto symbol: input) // по всем символам строки
	{
		if (m.find(symbol) == m.end()) // если символа нет в словаре
		{
			m[symbol] = 0; // создаём элемент
		}
		m[symbol] += 1; // увеличиваем счётчик вхождений символа
	}
	return m;
}
```

Листинг 3. Создание словаря Вероятность:Символ, с возможностью повторения ключей и сортируемого по убыванию значений ключей.

```
multimap<float, char, greater<float>> build_probability(map<char, int> m, int length) {
	multimap<float, char, greater<float>> p; // Словарь с повтором ключей (вероятность : символ), всегда отсортирован по убыванию
```

```
        for (auto item: m) // по всем элементам словаря вхождений символа в тексте
        {
                p.insert(make_pair((float(item.second) / length), item.first)); // ввод
элемента словаря вероятностей
        }
        return p;
}
```

Листинг 4. Создание нижних узлов корня дерева Шеннона-Фано.

```
void shannon_leaf_creator(node* parent) {
        float prob1 = 0, prob2, prob1buf = 0, prob2buf = 0, diversitybuf = 1; //
переменные для расчёта групп вероятностей
        for (auto item : parent->probabilities) // по элементам словаря вероятностей узла
        {
                prob2buf += item.first; // увеличиваем вероятность правой группы
        }
        prob2 = prob2buf;
        for (auto item : parent->probabilities) // по элементам словаря вероятностей узла
        {
                prob1 += item.first; // увеличиваем группу вероятностей слева
                prob2 -= item.first; // уменьшаем группу вероятностей справа
                if (diversitybuf < abs(prob2 - prob1)) // если разделение групп
вероятностей
                {                                                                  // было
более ровным одну итерацию раньше
                        prob2 = prob2buf;                          // вернём разделение
обратно
                        prob1 = prob1buf;                          //
                        break;                                             // и завершим
рассмотрение
                }
                else
                {
                        prob1buf = prob1;                                  // запоминаем
разделение
                        prob2buf = prob2;                                  //
                        diversitybuf = abs(prob2buf - prob1buf); // и насколько оно было
ровным
                }
        }
        multimap<float, char, greater<float>> l; // словарь вероятностей символов с
повтором ключей для левого сына
        multimap<float, char, greater<float>> r; // словарь вероятностей символов с
повтором ключей для правого сына
        float pl = 0; // переменная для проверки что мы разделили словарь правильно
        for (auto item : parent->probabilities) // по элементам словаря вероятностей узла
        {
                if (pl != prob1) // пока у левого сына словарь не станет быть как левая
группа вероятностей
                {
                        l.insert(make_pair(item.first, item.second)); // добавляем элементы
левой группы в словарь левого сына
                        pl += item.first; // увеличиваем вероятность для проверки разделения
                }
                else
                {
                        r.insert(make_pair(item.first, item.second)); // добавляем элементы
правой группы в словарь правого сына
                }
        }
        node* left_node = new node;
        node* right_node = new node;
        left_node->probabilities = l;
        right_node->probabilities = r;
```

```
        parent->left = left_node;
        parent->right = right_node;
        parent->probabilities.clear(); // очищаем словарь узла-родителя
        if (l.size() > 1) // пока словарь для левого сына имеет больше 2 элементов
        {
                shannon_leaf_creator(parent->left); // разделяем ещё дальше словарь у
левого сына
        }
        if (r.size() > 1) // пока словарь для правого сына имеет больше 2 элементов
        {
                shannon_leaf_creator(parent->right); // разделяем ещё дальше словарь у
правого сына
        }
}
```

Листинг 5. Создание дерева Шеннона-Фано.

```
node* shannon_tree_creator(multimap<float, char, greater<float>> p) {
        node* tree = new node; // корень дерева
        tree->probabilities = p; // словарь вероятностей корня
        shannon_leaf_creator(tree); // строим поддеревья
        return tree;
}
```

Листинг 6. Компаратор для приоритетной очереди.

```
class Comparator
{
public:
        bool operator()(node* l, node* r) { //Сравниваем общую частоту в словарях узлов
                float probs1 = 0, probs2 = 0;
                for (auto item : l->probabilities)
                {
                        probs1 += item.first;
                }
                for (auto item : r->probabilities)
                {
                        probs2 += item.first;
                }
                return probs1 > probs2; // Узлы с наименьшей общей вероятностью в вершине
очереди
        }
};
```

Листинг 7. Создание дерева Хаффмана.

```
node* huffman_tree_creator(multimap<float, char, greater<float>> p) {
        priority_queue<node*, vector<node*>, Comparator> pq; // приоритетная очередь узлов
с сортировкой по возрастанию верояности
        for (auto item : p) // наполняем очередь узлами символов с вероятностью из словаря
        {
                node* current_node = new node;
                current_node->probabilities.insert(make_pair(item.first, item.second));
                pq.push(current_node);
        }
        while (pq.size() != 1) // пока не останется корень
        {
                node* leftNode = pq.top(); // элемент из начала очереди с минимальной общей
вероятностью
                pq.pop(); // удаляем его из очереди
                node* rightNode = pq.top(); // первый элемент из начала очереди с
минимальной общей вероятностью
                pq.pop(); // удаляем его из очереди
                multimap<float, char, greater<float>> sum_of_probs; //объединяем словари
меньших узлов в один
```

```
                for (auto item : leftNode->probabilities)
                {
                        sum_of_probs.insert(make_pair(item.first, item.second));
                }
                for (auto item : rightNode->probabilities)
                {
                        sum_of_probs.insert(make_pair(item.first, item.second));
                }
                node* newNode = new node; // новый узел - родитель объединённых
                newNode->probabilities = sum_of_probs;
                newNode->left = leftNode; // меньший из 2 объединённых узлов - левый сын у
родителя
                newNode->right = rightNode; // больший из 2 объединённых узлов - правый сын
у родителя
                pq.push(newNode); // кладём родителя в очередь
        }
        node* tree = pq.top(); // верхний узел в очереди - корень
        return tree;
}
```

Листинг 8. Замена вхождения строки на другую строку.

```
static void ReplaceAll(string& str, const string& from, const string& to) { // для замены
подстроки на строку
        size_t start_pos = 0;
        while ((start_pos = str.find(from, start_pos)) != string::npos) {
                str.replace(start_pos, from.length(), to);
                start_pos += to.length();
        }
}
```

Листинг 9. Рекурсивное кодирование исходной строки по дереву Шеннона-Фано или Хаффмана.

```
void unified_encode_part(string& input, node* node, string code) {
        if (node->left == NULL && node->right == NULL)
        {
                string to_replace(1, node->probabilities.begin()->second); // из символа в
строку
                ReplaceAll(input, to_replace, code); // заменяем все символы в строке на их
код
                cout << "Замена '" << to_replace << "' на " << code << "\n";
        }
        else if (node->left != NULL && node->right != NULL)
        {
                unified_encode_part(input, node->left, code + '1'); // кодируем символы
левого поддерева, добавляя 1 в код
                unified_encode_part(input, node->right, code + '0'); // кодируем символы
правого поддерева, добавляя 0 в код
        }
        else if (node->left != NULL && node->right == NULL)
        {
                unified_encode_part(input, node->left, code + '1');
        }
        else
        {
                unified_encode_part(input, node->right, code + '0');
        }
}
```

Листинг 10. Запуск кодирования.

```
string unified_encoder(string input, node* tree) {
        string result = input;
```

```
	unified_encode_part(result, tree, ""); // кодируем строку, читая узлы сверху вниз
	return result;
}
```

Листинг 11. Создание словаря для декодирования по дереву Шеннона-Фано или Хаффмана.

```
void create_dict_part(map<string, string>& dict, node* node, string code) {
	if (node->left == NULL && node->right == NULL)
	{
		string to_replace(1, node->probabilities.begin()->second); // из символа в
строку
		dict[code] = to_replace; // присваиваем коду значение символа
	}
	else if (node->left != NULL && node->right != NULL)
	{
		create_dict_part(dict, node->right, code + '0'); // в код для правых
поддеревьев дописывается 0
		create_dict_part(dict, node->left, code + '1'); // в код для левых
поддеревьев дописывается 1
	}
	else if (node->right != NULL && node->left == NULL)
	{
		create_dict_part(dict, node->right, code + '0');
	}
	else
	{
		create_dict_part(dict, node->left, code + '1');
	}
}
```

Листинг 12. Декодирование по дереву Шеннона-Фано или Хаффмана.

```
string unified_decoder(string input, node* tree) {
	string result = input;
	map<string, string> dict; // словарь (код : символ, который мы закодировали)
	create_dict_part(dict, tree, ""); // строим словарь
	for (size_t i = 0; i < input.length(); i++) // двигаем подстроку чтобы
	{                                                                    //
раскодировать сообщение правильно
		for (size_t j = 1; j < (input.length() / 2) - 1; j++)
		{
			if (result[i] == NULL) // конец строки
			{
				return result; // выход из циклов
			}
			string sub = result.substr(i, j); // подстрока
			if (dict.find(sub) != dict.end()) // подстрока есть как код в словаре
			{
				result.replace(result.find(sub), sub.size(), dict.find(sub)-
>second); // заменяем код на символ
				//cout << "Замена " << sub << " на " << dict.find(sub)->second
<< "\n";
				break;
			}
		}
	}
	return result;
}
```

Листинг 13. Чтение файла в одну строку.

```
string fileReader(string fileName) { // читает файл
	ifstream file(fileName);          // в кодировке ANSI
	stringstream ss;
	ss << file.rdbuf();
```

```
	return ss.str();
}
```

Листинг 14. Запись строки в файл.

```
void fileWriter(string fileName, string info) {
	ofstream file;                              // записывает в файл
	file.open(fileName);       // в кодировке UTF-8
	file << info;
	file.close();
}
```

Листинг 15. Функция main.

```
int main() {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	int option = 3, choice = 3;
	string str;
	while (choice != 0)
	{
		cout << "Что кодировать?\n" << "1. ФИО\n" << "2. input.txt\n";
		cin >> option;
		switch (option)
		{
		case 1:
		{
			str = "Анисимов Даниил Николаевич";
			break;
		}
		case 2:
		{
			str = fileReader("input.txt"); // читаем исходный файл
			break;
		}
		default:
			str = fileReader("input.txt"); // читаем исходный файл
			break;
		}
		map<char, int> f = build_frequency(str); // строим словарь частоты
упоминаний символов
		multimap<float, char, greater<float>> p = build_probability(f,
str.length()); // строим словарь вероятностей символов
		cout << "Содержимое исходной строки:\n\n" << str << "\n\n";
		cout << "Вероятность : символ" << "\n";
		for (auto item : p)
		{
			cout << item.first << " : " << item.second << "\n";
		}

		cout << "По какому методу кодировать и раскодировать?" << "\n";
		cout << "1. Шеннон-фано\n" << "2. Хаффман\n" << "0. Выход\n";
		cin >> choice;
		switch (choice)
		{
		case 1:
		{
			node* tree = shannon_tree_creator(p); // строим префиксное дерево
шеннона-фано для кодирования
			cout << "\n" << "Какие символы на какие коды заменяем:" << "\n";

			string encoded = unified_encoder(str, tree);
			cout << "\n" << "Закодированная информация:" << "\n" << encoded <<
"\n";
```

```
                        cout << "\n" << "Коэффициент сжатия: " << (float(float(str.length() *
8) / encoded.length())) << "\n";
                        fileWriter("output.txt", encoded); // записываем закодированный файл
                        string encoded_get = fileReader("output.txt");

                        string decoded = unified_decoder(encoded_get, tree);
                        cout << "\n" << "Декодированная информация:" << "\n" << decoded <<
"\n\n";
                        break;
                }
                case 2:
                {
                        node* tree = huffman_tree_creator(p); // строим префиксное дерево
хаффмана для кодирования
                        cout << "\n" << "Какие символы на какие коды заменяем:" << "\n";

                        string encoded = unified_encoder(str, tree);
                        cout << "\n" << "Закодированная информация:" << "\n" << encoded <<
"\n";
                        cout << "\n" << "Коэффициент сжатия: " << (float(float(str.length() *
8) / encoded.length())) << "\n";
                        fileWriter("output.txt", encoded); // записываем закодированный файл
                        string encoded_get = fileReader("output.txt");

                        string decoded = unified_decoder(encoded_get, tree);
                        cout << "\n" << "Декодированная информация:" << "\n" << decoded <<
"\n\n";
                        break;
                }
                case 0:
                        break;
                default:
                        break;
                }
        }
        return 0;
}
```

Результат тестирования:

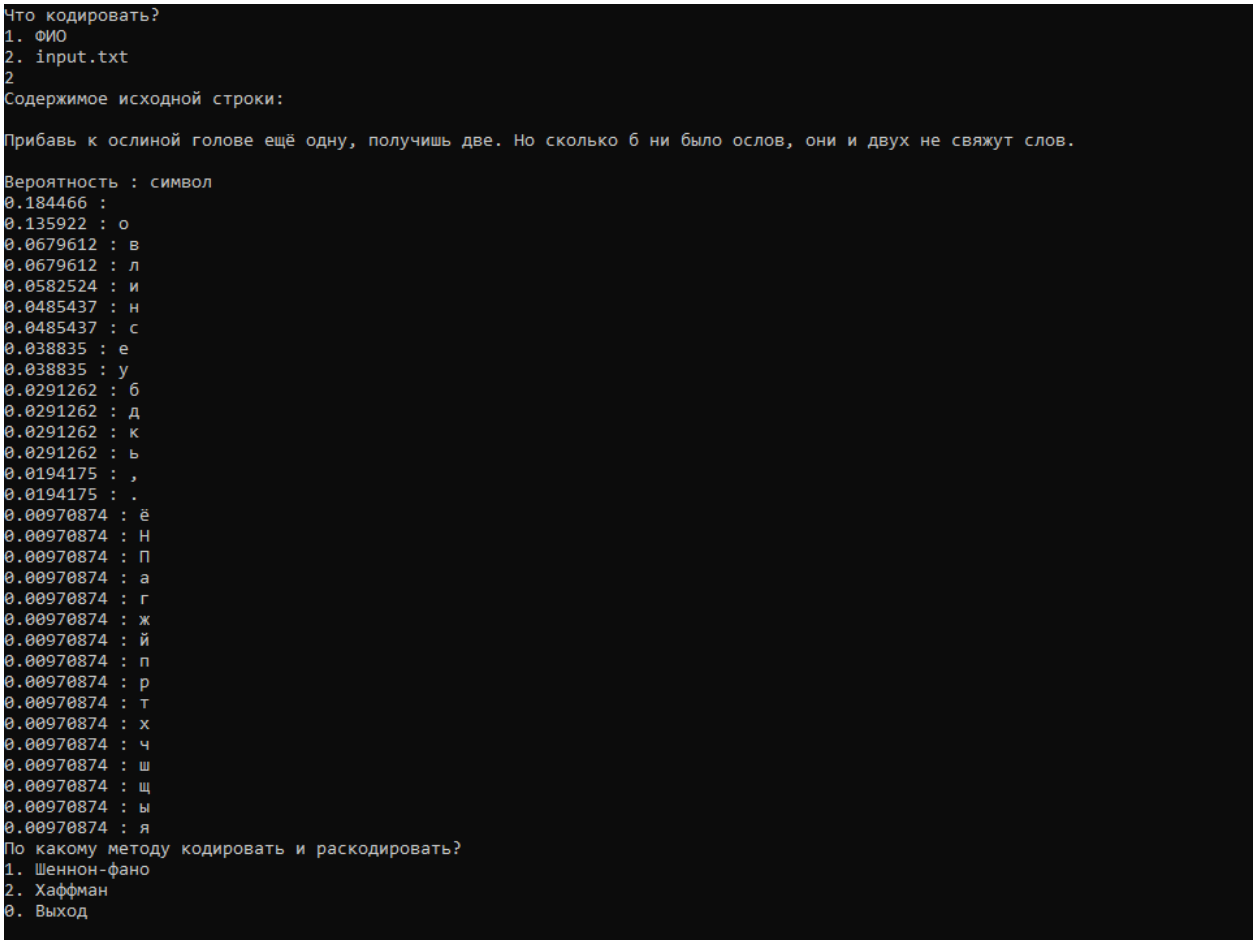

Рис. 2. Вывод таблицы вероятностей символов для текста из входного файла

Все вероятности вычислены правильно.

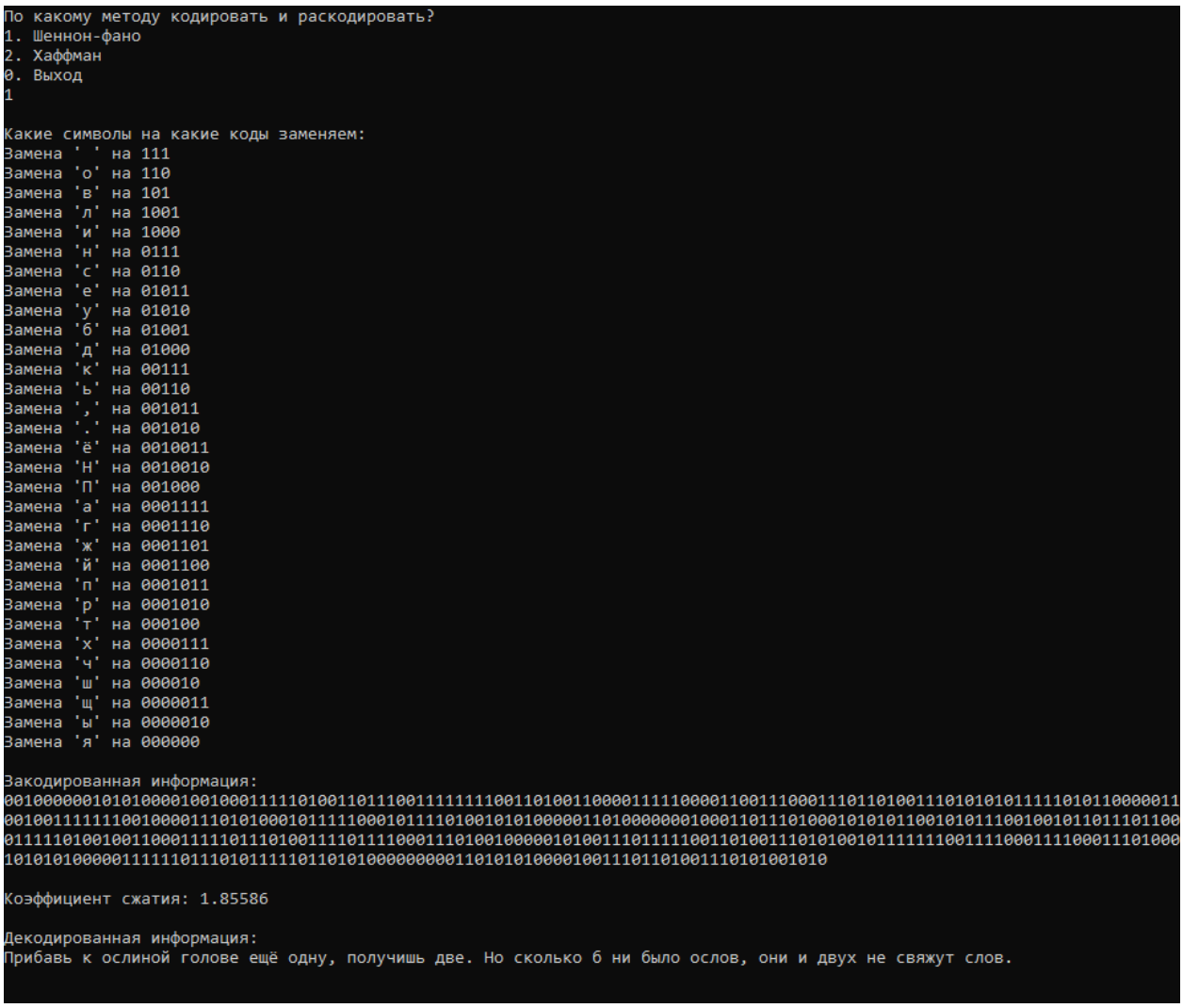

Рис. 3. Результат кодирования по Шеннону-Фано, декодирования и вычисления коэффициента сжатия для текста из входного файла

Кодирование прошло успешно, декодирование тоже прошло успешно.

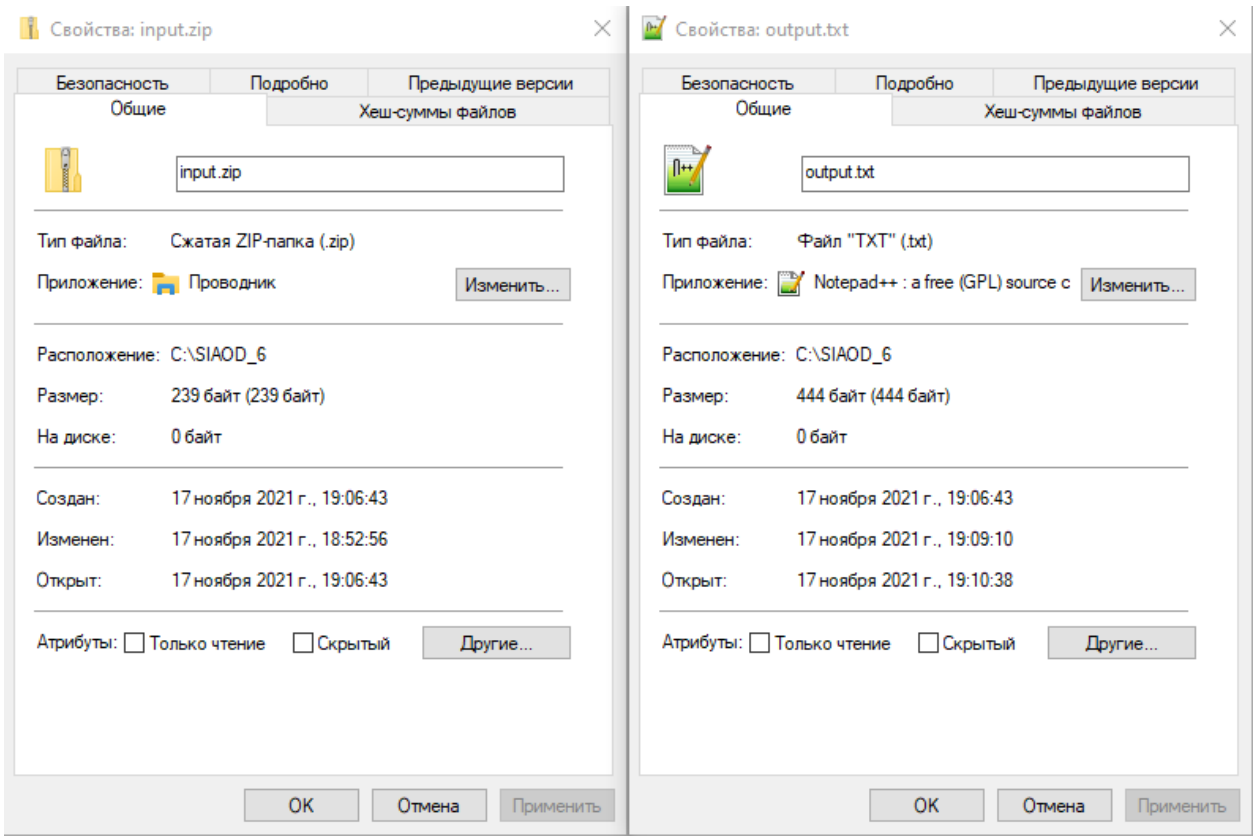

Рис. 4. Сравнение результата кодирования Шеннона-Фано с результатом архиватора по алгоритму Deflate

Сжатие архиватора по Deflate в 2 раза сильнее чем по алгоритму Шеннона-Фано.

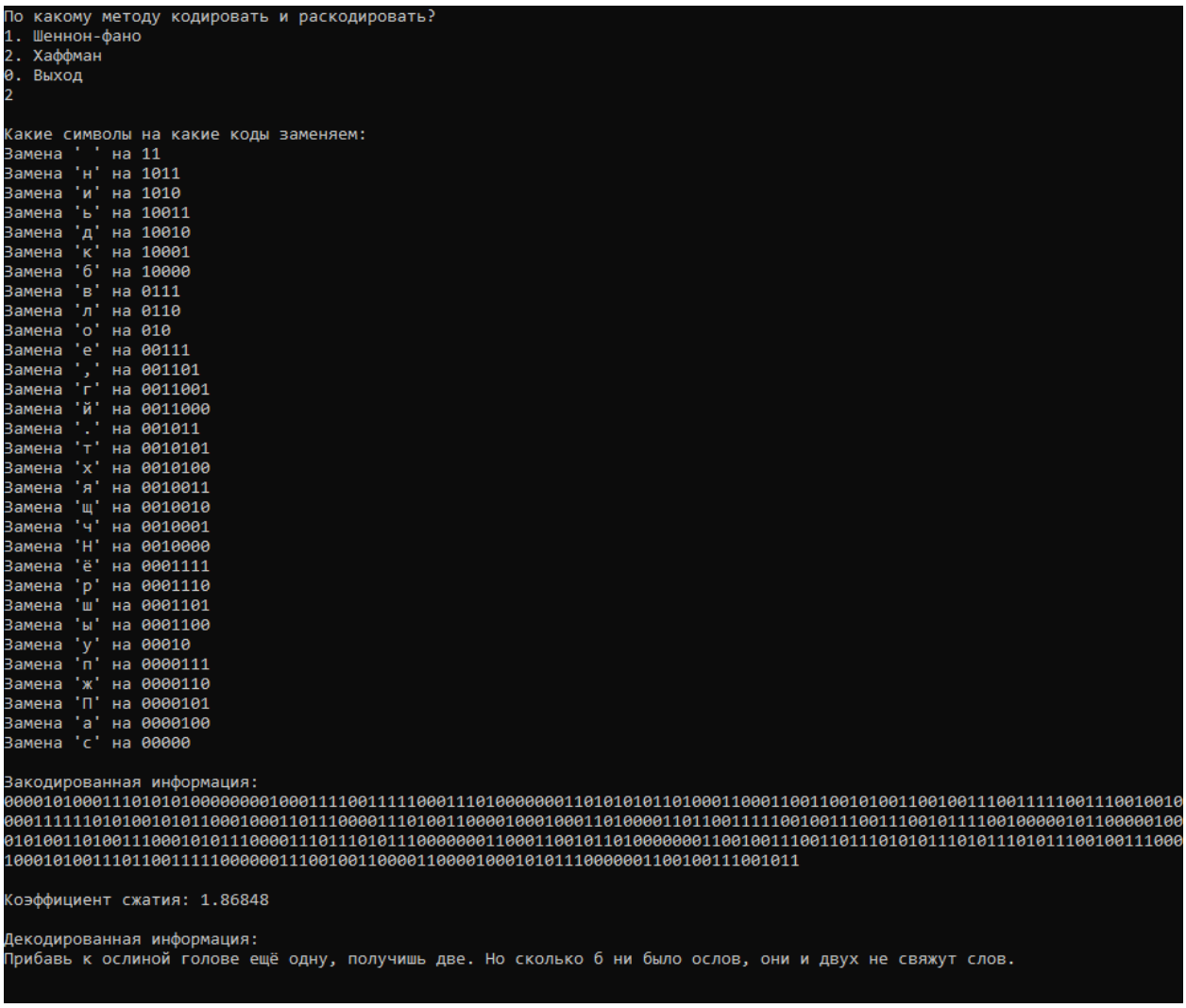

Рис. 5. Результат кодирования по Хаффману, декодирования и вычисления коэффициента сжатия для текста из входного файла

Кодирование прошло успешно, декодирование тоже прошло успешно.

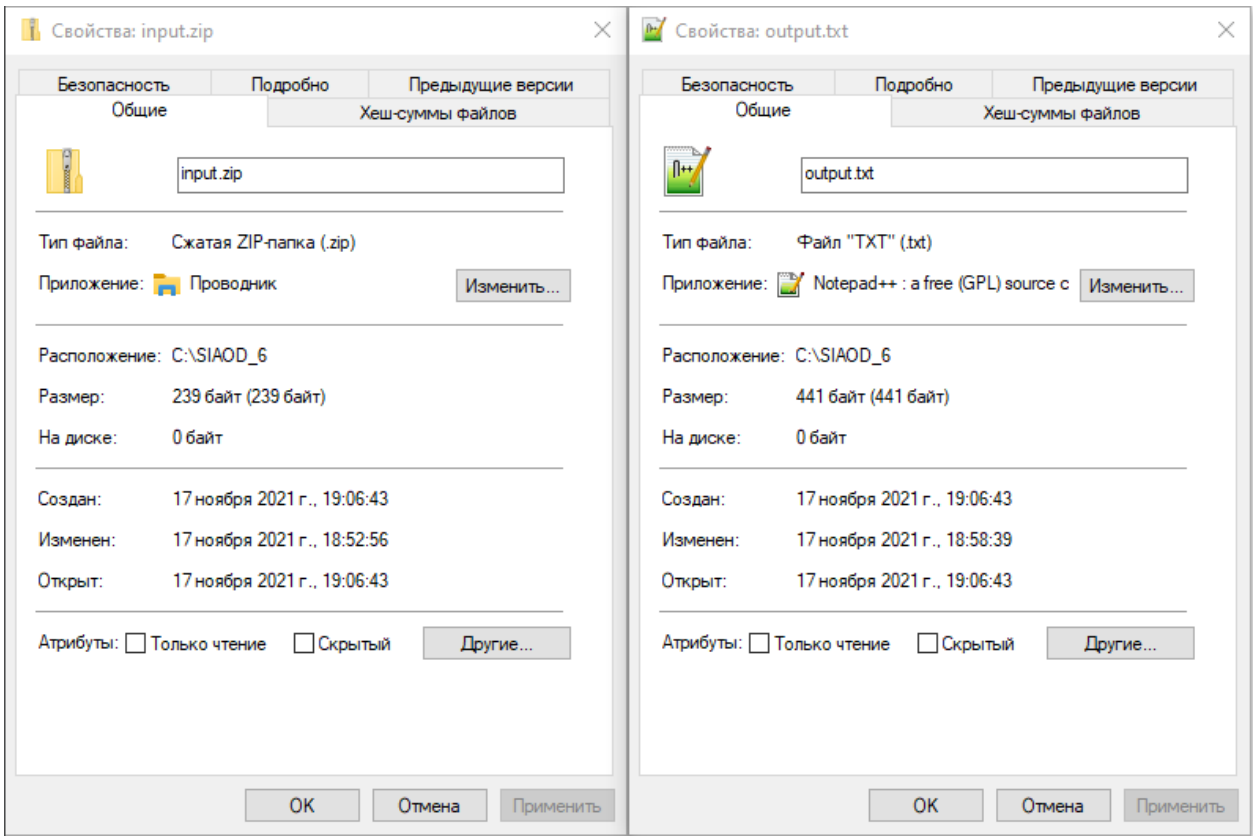

Рис. 6. Сравнение результата кодирования Хаффмана с результатом архиватора по алгоритму Deflate

Сжатие архиватора по Deflate опять в 2 раза сильнее чем по алгоритму Хаффмана.

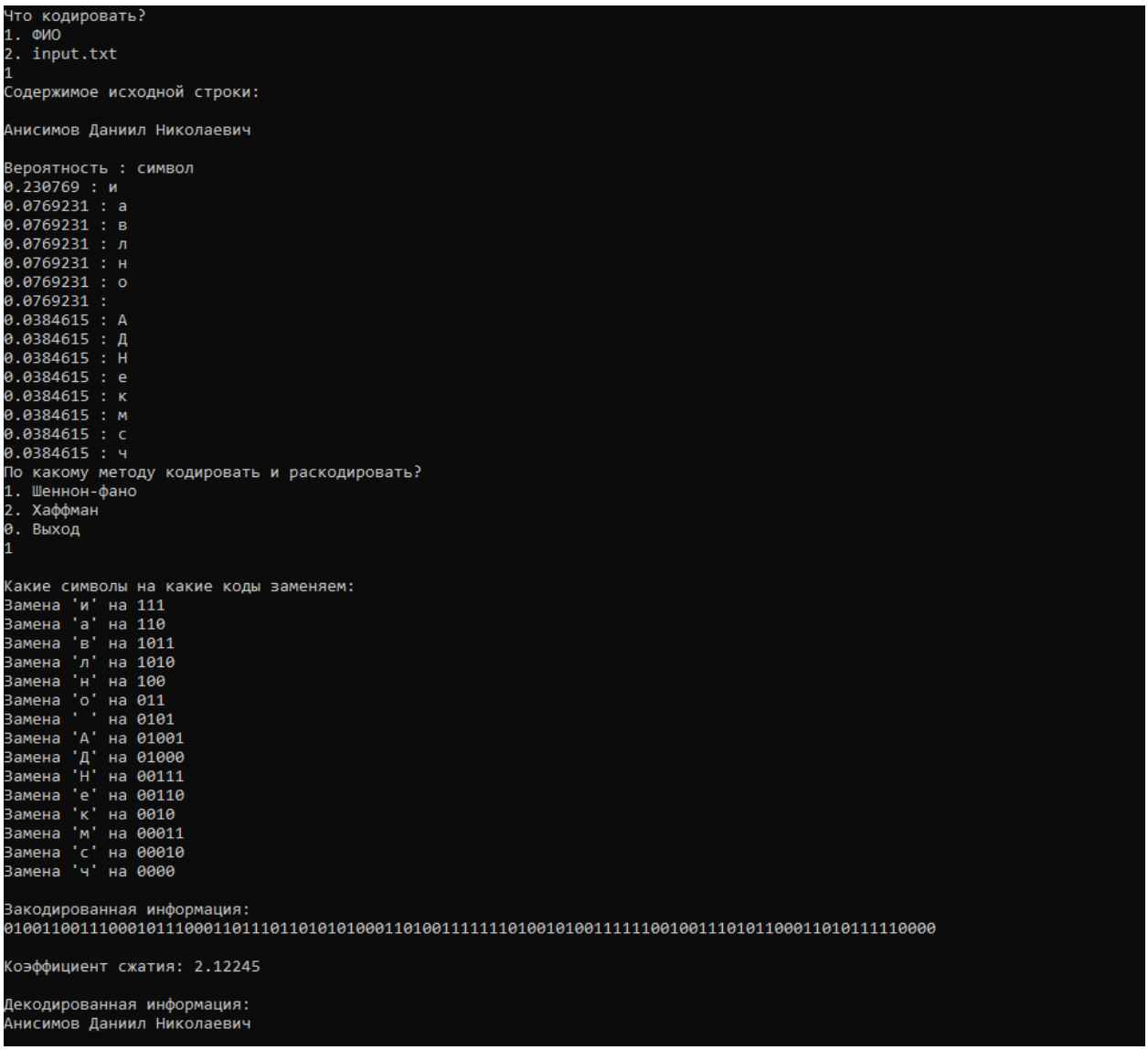

Рис. 7. Результат кодирования по Шеннону-Фано, декодирования и вычисления коэффициента сжатия для ФИО

Кодирование прошло успешно, декодирование тоже прошло успешно.

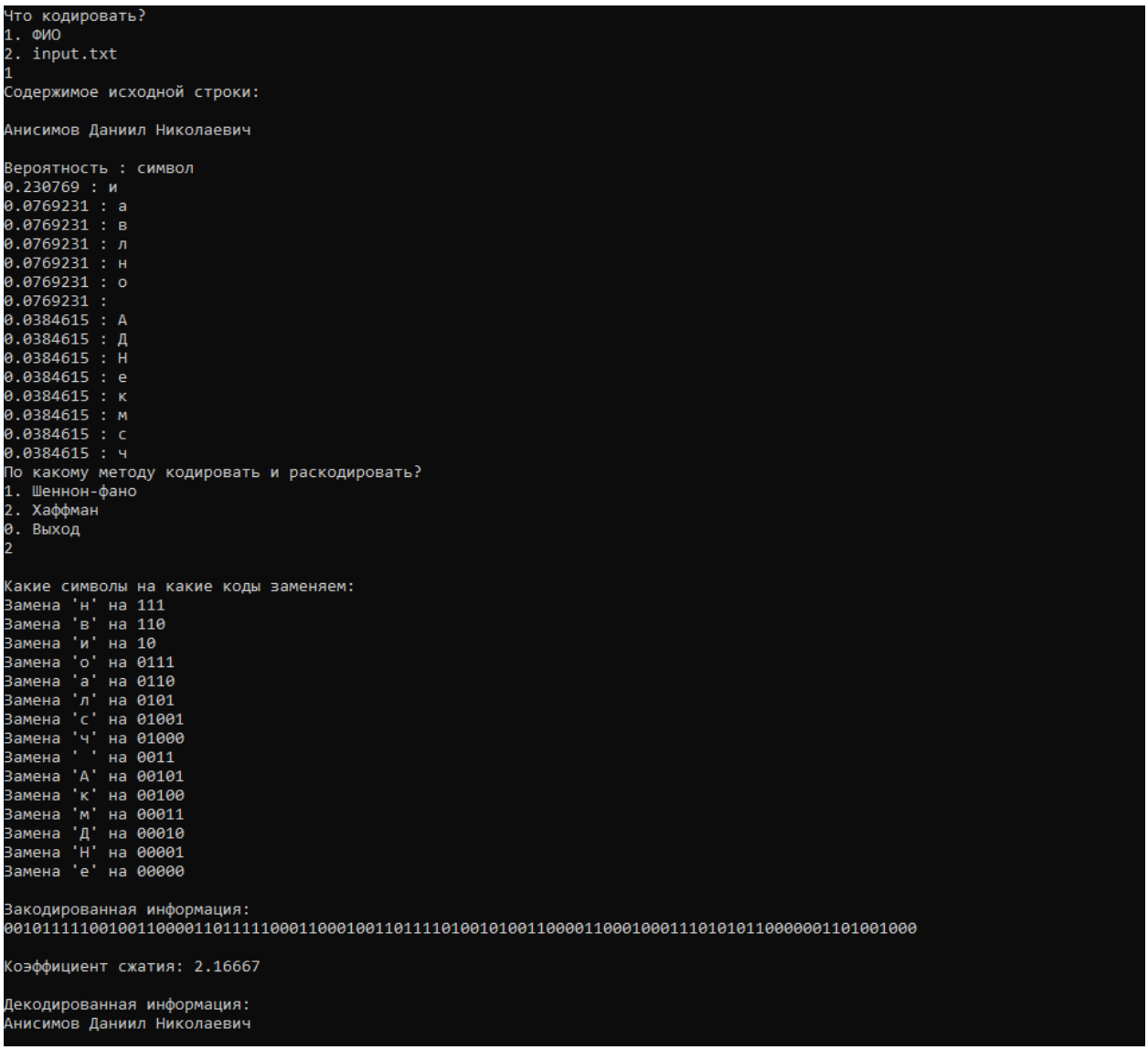

Рис. 8. Результат кодирования по Хаффману, декодирования и вычисления коэффициента сжатия для ФИО

Кодирование прошло успешно, декодирование тоже прошло успешно.

Общий вывод

Я получил знания алгоритмов кодирования и сжатия данных методами без потерь и навыки их применения.

Список информационных источников

1. Кораблин Ю.П. Структуры и алгоритмы обработки данных. Часть 1 [Электронный ресурс]: учебно методическое пособие / Ю.П.Кораблин, В.П.Сыромятников, Л.А. Скворцова – М.: РТУ МИРЭА, 2020. – 1 электрон. опт. диск (CD-ROM)