



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЁТ ПО ПРАКТИЧЕСКОМУ ЗАДАНИЮ №7
по дисциплине «Структуры и алгоритмы обработки данных»

Тема: «Кодирование и сжатие данных методами без потерь.»

Отчет представлен к
рассмотрению:

Студентка группы ИНБО-01-20 «7» декабря 2021 г.

(подпись)

Тульцова А.Д.

Преподаватель

«7» декабря 2021 г.

(подпись)

Сорокин А.В.

Москва, 2021 г.

СОДЕРЖАНИЕ

| | |
|--|-----------|
| Цель работы | 3 |
| Задание 1. Применение алгоритма группового сжатия текста..... | 3 |
| Постановка задачи..... | 3 |
| 1.1. Описание метода «RLE»..... | 3 |
| 1.2. Стандартное сжатие текста методом «RLE»..... | 3 |
| 1.3. Эффективное сжатие текста методом «RLE». | 4 |
| Задание 2. Исследование алгоритмов сжатия Лемпеля –Зива («LZ77», «LZ78») на примерах. | 6 |
| Постановка задачи..... | 6 |
| 2.1. Сжатие данных методами «LZ77», «LZ78»..... | 6 |
| 2.2. Восстановление сжатых методами «LZ77», «LZ78» данных. | 12 |
| Задание 3. Разработать программы сжатия и восстановления текста методами Шеннона—Фано и Хаффмана..... | 16 |
| Постановка задачи..... | 16 |
| 3.1. Кодирование текста по методу Шеннона—Фано и его декодирование. | 17 |
| 3.2. Кодирование текста по методу Хаффмана и его декодирование..... | 21 |
| 3.3. Сравнение коэффициентов сжатия разработанного алгоритма Хаффмана и архиватора..... | 26 |
| Вывод | 28 |
| Список информационных источников | 29 |

Цель работы

Получение практических навыков и знаний по выполнению сжатия данных без потерь рассматриваемыми методами.

Задание 1. Применение алгоритма группового сжатия текста.

Постановка задачи.

Сжать текст, используя метод «RLE» (run length encoding / кодирование длин серий / групповое кодирование).

1.1 Описать процесс сжатия алгоритмом «RLE».

1.2 Придумать текст, в котором есть длинные (в разумных пределах) серии из повторяющихся символов. Выполнить сжатие текста. Рассчитать коэффициент сжатия.

1.3 Придумать текст, в котором много неповторяющихся символов и между ними могут быть серии. Выполнить групповое сжатие, показать коэффициент сжатия. Применить алгоритм разделения текста при групповом кодировании, позволяющий повысить эффективность сжатия этого текста. Рассчитать коэффициент сжатия после применения алгоритма.

1.1. Описание метода «RLE».

Кодирование длин серий (англ. *run-length encoding*, **RLE**) или **кодирование повторов** — алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов.

Серия — последовательность, состоящая из нескольких одинаковых символов. При кодировании (упаковке, сжатии) строка одинаковых символов, составляющих серию, заменяется строкой, содержащей сам повторяющийся символ и количество его повторов.

1.2. Стандартное сжатие текста методом «RLE».

Пусть дан следующий текст для кодирования, в котором есть длинные (в разумных пределах) серии из повторяющихся символов:

«aaaaaaannnnnaaaaaaassssssssstttttttaaaaaaasssssiiiiiaaaaaaaaaaaaaa».

Для данного текста стандартное сжатие методом «RLE» будет происходить следующим образом: «<кол-во идущих подряд элементов 'a'>a<кол-во идущих подряд элементов 'n'>n<кол-во следующих идущих подряд элементов 'a'>a<кол-во идущих подряд элементов 's'>s<кол-во идущих подряд элементов 't'>t<кол-во очередных идущих подряд элементов 'a'>a<кол-во следующих идущих подряд элементов 's'>s<кол-во идущих подряд элементов 'i'>i<кол-во последних идущих подряд элементов 'a'>a».

В результате закодированный текст будет выглядеть так:

«7a5n7a10s9t8a5s12i15a».

Длина исходного текста: $L(\text{text}) = 78$.

Длина закодированного текста: $L(\text{code}) = 21$.

Коэффициент сжатия: $k = L(\text{code}) / L(\text{text}) = 21 / 78 \approx 0.269$.

Таким образом, методом «RLE» удалось выполнить сжатие исходного текста в ~ 3.714 раз.

1.3. Эффективное сжатие текста методом «RLE».

Пусть дан следующий текст, в котором много неповторяющихся символов, а между ними встречаются серии из символов:

«abcdefghijklmnopqrstuvwxyz».

В результате стандартного сжатия методом «RLE» закодированный текст будет выглядеть так:

«1a1b1c1d1e1f1g1h2i2j1k1l1m1n1o1p2q1r1s1t1u1v1w1x1y1z».

Длина исходного текста: $L(\text{text}) = 29$.

Длина закодированного текста: $L(\text{code}_1) = 52$.

Коэффициент сжатия: $k_1 = L(\text{code}_1) / L(\text{text}) = 52 / 29 \approx 1.793$.

Длина закодированного текста превышает длину исходного текста в ~ 1.793 раз: это означает, что такой метод кодирования увеличивает объём исходного текста.

Теперь рассмотрим алгоритм разделения текста при групповом кодировании, позволяющий повысить эффективность сжатия этого текста методом «RLE».

Алфавит целых чисел, в котором записаны длины серий, необходимо разделить на две части: положительные и отрицательные числа. Положительные числа будут использоваться для записи количества повторов одного символа, а отрицательные — для записи количества неодинаковых символов, следующих друг за другом.

Тогда для данного текста эффективное сжатие методом «RLE» будет происходить следующим образом: «—<кол-во символов в первом блоке неповторяющихся символов>abcdefgh2i2j—<кол-во символов во втором блоке неповторяющихся символов>klmnop2q—<кол-во символов в последнем блоке неповторяющихся символов>rstuvwxyz».

В результате эффективного сжатия методом «RLE» закодированный текст будет выглядеть так:

«—8abcdefgh2i2j—6klmnop2q—9rstuvwxyz».

Длина исходного текста: $L(\text{text}) = 29$.

Длина закодированного текста: $L(\text{code}_2) = 35$.

Коэффициент сжатия: $k_2 = L(\text{code}_2) / L(\text{text}) = 35 / 29 \approx 1.207$.

Длина закодированного текста снова превышает длину исходного текста, но уже в ~ 1.207 раз — для данного случая этот алгоритм работает эффективнее предыдущего в $k_1 / k_2 = 1.793 / 1.207 \approx 1.486$ раз, однако, из-за того что в тексте слишком короткие серии повторяющихся символов и при этом достаточно длинные блоки неповторяющихся символов, метод «RLE» оказывается в целом неэффективным алгоритмом сжатия для данного случая.

Задание 2. Исследование алгоритмов сжатия Лемпеля –Зива («LZ77», «LZ78») на примерах.

Постановка задачи.

2.1. Выполнить каждую задачу варианта (см. таблицу 1), представив алгоритм решения в виде таблицы и указав результат сжатия.

2.2. Описать процесс восстановления сжатого текста.

Вариант 8.

Таблица 1 — Задания варианта на алгоритмы Лемпеля — Зива.

| | |
|---|---|
| Сжатие данных по методу Лемпеля — Зива «LZ77». Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу: | Закодировать следующую фразу, используя код «LZ78»: |
| 010110110110100010001 | sarsalsarsanlasanl 33 |

2.1. Сжатие данных методами «LZ77», «LZ78».

Дана последовательность битов для сжатия методом «LZ77»: «010110110110100010001».

Сначала (перед записью LZ-кода) необходимо разбить исходную последовательность следующим образом:

- выбрать первый элемент последовательности и отделить его от всей последовательности;
- отделить группу из двух элементов, следующую после первого (уже отделённого от последовательности) элемента;
- далее необходимо последовательно формировать группы (из двух и более элементов) из элементов последовательности, идущих друг за другом, до тех пор, пока не образуется ещё не встречавшаяся комбинация (группа) элементов;
- предыдущий шаг необходимо повторять до тех пор, пока таким образом не будут пройдены все элементы последовательности.

После выполнения данного алгоритма исходная последовательность будет представлена как «0 10 11 01 101 1010 00 100 01».

Для дальнейшего рассмотрения процесса кодирования определим два понятия. Будем называть тривиальной такую комбинацию (группу), которая состоит из одного элемента («0» или «1»). Соответственно, комбинации, состоящие из двух и более элементов, будем считать нетривиальными.

Теперь необходимо последовательно назначить уникальный код для каждой группы по следующим правилам:

- для тривиальной комбинации назначенный код совпадает (т.е. для первого элемента назначенный код совпадает с самим элементом);
- для следующей за первым элементом группы из двух элементов назначенный код имеет значение «10» (совпадает с количеством уже использованных кодов для тривиальных комбинаций — всего их два: «0» и «1»);
- далее для каждой следующей нетривиальной комбинации назначенный код увеличивается на единицу, т.е. для любой группы, начиная с третьей, назначенный код больше на единицу от назначенного кода для предыдущей группы.

Таким образом для каждой выделенной группы данной последовательности были назначены уникальные коды: для элемента «0» назначенный код — «0», для комбинации «10» назначенный код — «10», для комбинации «11» назначенный код — «11», для следующей комбинации «01» назначенный код — «100» и т.д. Так как в данном примере всего 9 выделенных комбинаций, очевидно, что для последней группы («01») назначенным кодом будет код «1001».

Теперь осталось сформировать LZ-код для заданной последовательности. Сам процесс кодирования можно описать в виде следующего алгоритма:

- последовательно рассматриваются группы элементов исходной последовательности, причём проход по элементам каждой комбинации происходит слева направо;
- в процессе прохода элементов комбинации слева направо осуществляется попытка найти группу из элементов (одну из предыдущих комбинаций), которая бы совпадала со всеми элементами текущей рассматриваемой комбинации за исключением, быть может, последнего элемента; при успешном поиске такой группы совпавший с ней блок элементов рассматриваемой комбинации кодируется последним назначенным кодом для найденной группы; в таком случае оставшийся элемент рассматриваемой комбинации, если он существует, кодируется своим же значением (по правилу для тривиальной комбинации) — то же происходит и в случае неуспешного поиска: если не найдена такая группа элементов, то вся рассматриваемая комбинация кодируется своим же значением;
- при составлении LZ-кода в памяти отдельно сохраняется разрядность последней полученной группы элементов LZ-кода; правило кодирования дополняется тем, что разрядность очередной группы элементов LZ-кода не может быть меньше разрядности предыдущей комбинации элементов LZ-кода; в случае, если она меньше, добавляются незначащие нули до тех пор, пока разрядности этих двух комбинаций не станут равны.

Рассмотрим процесс формирования LZ-кода для исходной последовательности битов.

1. Очевидно, что для первого элемента «0» LZ-код — также «0».
2. Для групп «10», «11», «01» LZ-код будет соответственно совпадать, поскольку не существует таких предыдущих комбинаций, которые бы совпадали со всеми элементами групп за исключением, быть может,

последних элементов. Вообще говоря, можно утверждать, что для группы «01» существует предыдущая комбинация из символа «0», которая совпадает со всеми элементами данной группы за исключением последнего. Назначенный код для этой комбинации — «0», поэтому LZ-код составляется по принципу: «0» (исх.) → «0» (назн. код) → «0» (LZ-код), «1» (исх.) → «1» (LZ-код). Однако для простоты тривиальную комбинацию можно не рассматривать в качестве одной из предыдущих — на результат кодирования это никак не повлияет, поскольку для неё всегда выполняется правило совпадения назначенного кода.

3. Для группы «101» существует ранее встречавшаяся комбинация «10», которая совпадает со всеми элементами данной группы за исключением последнего. В свою очередь для комбинации «10» назначенный код — «10», поэтому LZ-код составляется по принципу: «10» (исх.) → «10» (назн. код) → «10» (LZ-код), «1» (исх.) → «1» (LZ-код).
4. Для следующей группы «1010» существует ранее встречавшаяся комбинация «101», которая совпадает со всеми элементами данной группы за исключением последнего. В свою очередь для комбинации «101» назначенный код — «101», поэтому LZ-код составляется по принципу: «101» (исх.) → «101» (назн. код) → «101» (LZ-код), «0» (исх.) → «0» (LZ-код).
5. Для группы «00» LZ-код составляется по принципу: «0» (исх.) → «0» (LZ-код), «0» (исх.) → «0» (LZ-код). Однако поскольку предыдущий LZ-код имеет 4 разряда, то для полученного LZ-кода («00») необходимо добавить 2 незначащих нуля для совпадения разрядностей.
6. Для группы «100» LZ-код составляется аналогично LZ-коду для группы «101»: «10» (исх.) → «10» (назн. код) → «10» (LZ-код). Отличием будет лишь последний символ: «0» (исх.) → «0» (LZ-код). Разрядность полученного LZ-кода равна трём, следовательно, необходимо добавить 1 незначащий ноль.

7. Для последней группы «01» LZ-код составляется по принципу: «01» (исх.) → «100» (назн. код) → «100» (LZ-код). Разрядность полученного LZ-кода равна трём, следовательно, необходимо добавить 1 незначащий ноль. Заметим, что полученный LZ-код ранее уже встречался, однако это никак не повлияет на процесс декодирования — однозначность декодирования исходной последовательности всё так же сохраняется (см. раздел 2.2).

Пошаговая схема сжатия исходной битовой последовательности методом «LZ77» представлена на рисунке 1.

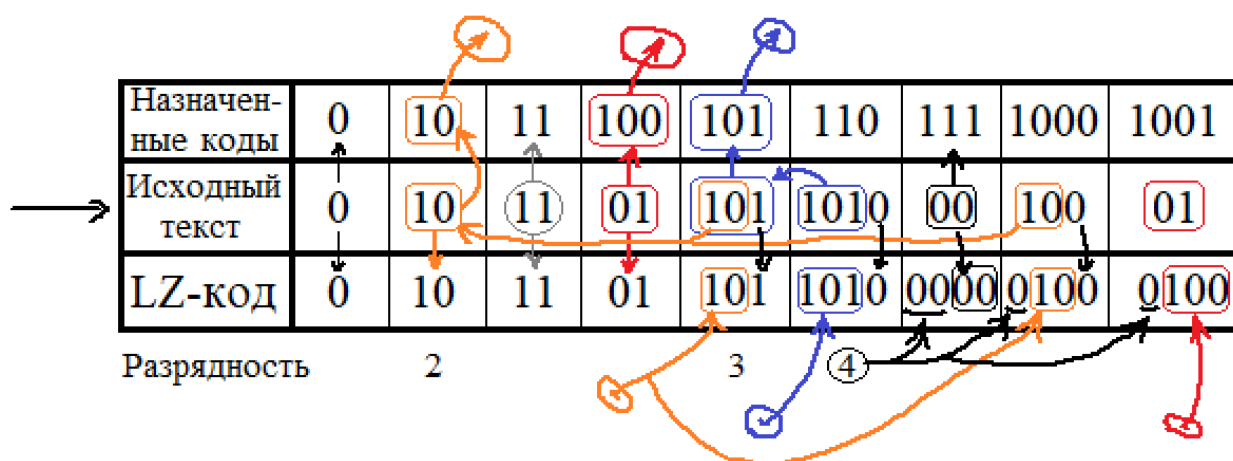


Рисунок 1 — Сжатие текста методом «LZ77»

«LZ78» ориентируется на данные, которые только будут получены (не используется скользящее окно); для данного метода составляется словарь из уже просмотренных фраз. Алгоритм считывает символы сообщения до тех пор, пока накапливаемая подстрока входит целиком в одну из фраз словаря. Как только эта строка перестанет соответствовать хотя бы одной фразе словаря, алгоритм генерирует код, состоящий из индекса строки в словаре, которая до последнего введенного символа содержала входную строку, и символа, нарушившего совпадение. Затем в словарь добавляется введенная подстрока.

Дана строка для сжатия методом «LZ78»: «sarsalsarsanlasanl 33».

Сначала необходимо её разбить на уникальные фразы по вышеописанному правилу. После разбиения фразы будут представлены следующим образом: «s», «a», «r», «sa», «l», «sar», «san», «la», «sanl», « », «3», «3(EOF)».

Поскольку последняя фраза повторяется с предыдущей, для обеспечения однозначного декодирования добавляется ключевая фраза «(EOF)», обозначающая дословно «Конец_Файла» (англ. «End_Of_File»), а именно — конец исходной строки.

Теперь можно производить кодирование. Весь процесс кодирования представлен в таблице 2.

Таблица 2 — Составление словаря для кодирования методом «LZ78».

| Ключ в словаре: ссылка на просмотренную фразу | Значение в словаре: просмотренная фраза | Код для просмотренной фразы |
|---|--|--------------------------------|
| 1 | «s» | <0: «s»> |
| 2 | «a» | <0: «a»> |
| 3 | «r» | <0: «r»> |
| 4 | «sa» | <1: «a»> |
| 5 | «l» | <0: «l»> |
| 6 | «sar» | <4: «r»> |
| 7 | «san» | <4: «n»> |
| 8 | «la» | <5: «a»> |
| 9 | «sanl» | <7: «l»> |
| 10 | « » | <0: « »> |
| 11 | «3» | <0: «3»> |
| 12 | «3(EOF)» | <11: «(EOF)»> |

В результате остаётся из полученных кодов для просмотренных фраз составить LZ-код: «0s0a0r1a0l4r4n5a7l0 0311(EOF)».

2.2. Восстановление сжатых методами «LZ77», «LZ78» данных.

Известно, что некоторая последовательность битов была закодирована методом «LZ77»: «0 10 11 01 101 1010 0000 0100 0100». Необходимо произвести декодирование.

Рассмотрим процесс декодирования.

1. Очевидно, что для первого элемента «0» LZ-кода исходный элемент — также «0» и назначенный код — тоже «0».
2. Для групп «10», «11», «01» LZ-кода исходные комбинации будут соответственно совпадать, поскольку не существует таких предыдущих назначенных кодов, которые бы совпадали с данными группами LZ-кода за исключением, быть может, последнего элемента. Вообще говоря, можно утверждать, что для группы «01» LZ-кода существует предыдущий назначенный код из символа «0», который совпадает со всеми элементами данной группы LZ-кода за исключением последнего. Исходный символ для этого назначенного кода — «0», поэтому исходная комбинация для текущей группы LZ-кода составляется по принципу: «0» (LZ-код) → «0» (пред. назн. код) → «0» (пред. исх.) → «0» (исх.), «1» (LZ-код) → «1» (исх.). Однако для простоты назначенный код тривиальной комбинации можно не рассматривать в качестве одного из предыдущих — на результат декодирования это никак не повлияет, поскольку для него всегда выполняется правило совпадения с тривиальной комбинацией.
3. Для группы «101» LZ-кода существует ранее встречавшийся назначенный код «10», который совпадает со всеми элементами данной группы LZ-кода за исключением последнего. В свою очередь для назначенного кода «10» исходная комбинация — «10», поэтому исходная комбинация для текущей группы LZ-кода составляется по принципу: «10» (LZ-код) → «10» (пред. назн. код) → «10» (пред. исх.) → «10» (исх.), «1» (LZ-код) → «1» (исх.).

4. Для следующей группы «1010» LZ-кода существует ранее встречавшийся назначенный код «101», который совпадает со всеми элементами данной группы LZ-кода за исключением последнего. В свою очередь для назначенного кода «101» исходная комбинация — «101», поэтому исходная комбинация для текущей группы LZ-кода составляется по принципу: «101» (LZ-код) → «101» (пред. назн. код) → «101» (пред. исх.) → «101» (исх.), «0» (LZ-код) → «0» (исх.).
5. Для группы «0000» LZ-кода исходная комбинация составляется по принципу: «0» (LZ-код) → «0» (исх.), «0» (LZ-код) → «0» (исх.). Первые два нуля — незначащие, поскольку иначе, во-первых, нарушалось бы правило при разбиении исходной последовательности для кодирования, во-вторых, нарушалось бы правило разрядности для данной группы LZ-кода (не может быть меньше разрядности предыдущей группы LZ-кода).
6. Для группы «0100» LZ-кода исходная комбинация составляется аналогично исходной комбинации для группы «101» LZ-кода: «10» (LZ-код) → «10» (пред. назн. код) → «10» (пред. исх.) → «10» (исх.). Одним из отличий будет последний символ: «0» (LZ-код) → «0» (исх.). Второе отличие — наличие незначащего нуля. Первый ноль — незначащий, поскольку иначе, во-первых, нарушалось бы правило при разбиении исходной последовательности для кодирования, во-вторых, нарушалось бы правило разрядности для данной группы LZ-кода (не может быть меньше разрядности предыдущей группы LZ-кода).
7. Для последней группы «0100» LZ-кода существует ранее встречавшийся назначенный код «100», который совпадает со всеми элементами данной группы LZ-кода за исключением первого элемента. В свою очередь для назначенного кода «100» исходная комбинация — «01», поэтому исходная комбинация для текущей группы LZ-кода составляется по принципу: «100» (LZ-код) → «100» (пред. назн. код) → «01» (пред. исх.) → «01» (исх.). Первый ноль является незначащим, так как предыдущая группа LZ-кода имеет разрядность 4.

Пошаговая схема декодирования битовой последовательности, сжатой методом «LZ77» представлена на рисунке 2.

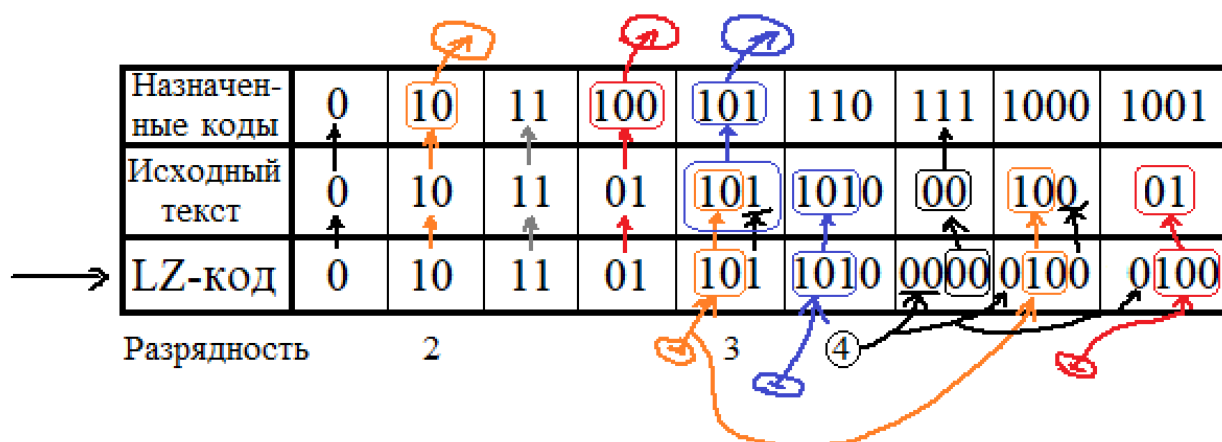


Рисунок 2 — Восстановление сжатого методом «LZ77» текста

Известно, что некоторая фраза была закодирована методом «LZ78»: «0s0a0r1a0l4r4n5a7l0 0311(EOF)». Необходимо произвести декодирование.

Для этого из закодированной строки последовательно выделяются коды, по которым создаются элементы словаря.

Весь процесс декодирования представлен в таблице 3.

Таблица 3 — Составление словаря для декодирования методом «LZ78».

| Код для просмотренной фразы | Ключ в словаре: ссылка на просмотренную фразу | Значение в словаре: просмотренная фраза |
|-----------------------------|---|--|
| <0: «s»> | 1 | «s» |
| <0: «a»> | 2 | «a» |
| <0: «r»> | 3 | «r» |
| <1: «a»> | 4 | «sa» |
| <0: «l»> | 5 | «l» |
| <4: «r»> | 6 | «sar» |
| <4: «n»> | 7 | «san» |
| <5: «a»> | 8 | «la» |
| <7: «l»> | 9 | «sanl» |
| <0: « »> | 10 | « » |
| <0: «3»> | 11 | «3» |
| <11: «(EOF)»> | 12 | «3(EOF)» |

Таким образом в результате декодирования была получена исходная строка «sarsalsarsanlasanl 33».

Задание 3. Разработать программы сжатия и восстановления текста методами Шеннона—Фано и Хаффмана.

Постановка задачи.

Сформировать отчёт по разработке каждого алгоритма в соответствии с требованиями.

3.1 Выполнить кодирование текста, заданного в варианте, по методу Шеннона—Фано, а также его декодирование.

1) Привести постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования.

2) Представить таблицу формирования кода.

3) Изобразить префиксное дерево.

4) Рассчитать коэффициент сжатия.

3.2 Выполнить кодирование текста «Тульцова Анастасия Денисовна» по методу Хаффмана, а также его декодирование.

1) Привести постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования.

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

3) Изобразить префиксное дерево Хаффмана.

4) Провести кодирование исходной строки по аналогии с примером (см. рисунок 3).

| | | | | | | | | | | | | | | |
|------|-------|------|-------|----|-------|-----|-------|-------|-------|-------|-----|----|-------|---|
| п | у | п | к | и | н | « | » | в | а | с | и | л | и | й |
| 1010 | 11000 | 1010 | 1011 | 00 | 11001 | 010 | 011 | 11010 | 11011 | 00 | 100 | 00 | 11100 | |
| « | к | и | р | и | л | л | о | в | и | ч | | | | |
| 010 | 1011 | 00 | 11101 | 00 | 100 | 100 | 11110 | 011 | 00 | 11111 | | | | |

Рисунок 3 — Пример кодирования исходной строки методом Хаффмана

5) Рассчитать коэффициент сжатия относительно кодировки «ASCII».

6) Рассчитать среднюю длину полученного кода и его дисперсию.

7) По результатам выполненной работы сделать выводы и сформировать отчёт. Отобразить результаты выполнения всех требований, предъявленных в задании и оформить разработку программы: постановка, подход к решению, код, результаты тестирования.

3.3. Реализовать и отладить программу. Применить алгоритм Хаффмана для архивации данных текстового файла. Провести архивацию этого же файла любым архиватором. Сравнить коэффициенты сжатия разработанного алгоритма и архиватора.

3.1. Кодирование текста по методу Шеннона—Фано и его декодирование.

Задание варианта представлено в таблице 4.

Вариант 8.

Таблица 4 — Задание варианта.

| |
|---|
| Закодировать фразу методом Шеннона—Фано. |
| Мой котёнок очень странный, Он не хочет есть сметану, К молоку не прикасался И от рыбки отказался. |

1) Алгоритм основан на частоте повторения символов. Так, часто встречающийся символ кодируется кодом меньшей длины, а редко встречающийся — кодом большей длины. В свою очередь, коды, полученные при кодировании, префиксные. Это и позволяет однозначно декодировать любую последовательность кодовых слов.

Алгоритм Шеннона-Фано работает следующим образом:

- на вход приходят упорядоченные по невозрастанию частоты данных;

- находится «середина», которая делит алфавит на две примерно равные части (суммы частот алфавита): для левой части присваивается «1», для правой — «0»;
- предыдущий шаг повторяется до тех пор, пока не будет получен единственный элемент последовательности, т.е. «лист» дерева.

Для кодирования текста из файла методом Шеннона—Фано, декодирования закодированного методом Шеннона—Фано текста в файл, а также для расчёта коэффициента сжатия данных была разработана программа, представленная на рисунках 4 – 6.

```
# функция для кодирования по алгоритму Шеннона - Фано.
def sf_encode(arr, current_result):
    quantity = sum(map(lambda x: x[1], arr)) # Общее количество символов.
    half = 0 # Количество символов в первой группе (первой половине).
    index = 0 # Индекс элемента массива arr, добавляемого в первую группу в последнюю очередь (последнего в группе).
    for arr_element_index, arr_element in enumerate(arr): # Для каждого индекса и элемента массива arr:
        half += arr_element[1] # Добавление кол-ва повторений текущего символа в кол-во символов в первой группе.
        if half * 2 >= quantity: # Если удвоенное кол-во символов в первой группе не меньше общего кол-ва символов:
            # Если разница между удвоенным кол-вом символов в первой группе и общим кол-вом символов меньше
            # разницы между удвоенным кол-вом символов в первой группе без кол-ва повторений последнего рассм. элемента:
            # следующий элемент после последнего рассматриваемого включается в первую группу, иначе:
            # последний рассматриваемый элемент остаётся последним в первой группе.
            index = arr_element_index + (abs(2 * half - quantity) < abs(2 * (half - arr_element[1]) - quantity))
            break # Индекс последнего элемента первой группы определён — выход из цикла.
    arr0, arr1 = [], [] # Массивы из элементов первой и второй групп соответственно.
    for arr_element in arr[:index]: # Для каждого элемента массива до элемента с индексом index:
        arr_element[2] += '0' # При кодировании элементов первой группы будет добавляться '0'.
        arr0.append(arr_element) # Добавление элемента в первую группу.
    for arr_element in arr[index:]: # Для каждого элемента массива до элемента с индексом index:
        arr_element[2] += '1' # При кодировании элементов второй группы будет добавляться '1'.
        arr1.append(arr_element) # Добавление элемента во вторую группу.
    if len(arr1) == 1: # Если во второй группе один элемент:
        current_result.append(arr1) # Добавить вторую группу в массив с текущим результатом кодирования.
    else: # Иначе (если во второй группе несколько элементов, то нужно продолжать разбиение):
        sf_encode(arr1, current_result) # Рекурсивный запуск функции для второй группы и массива с текущим результатом.
    if len(arr0) == 1: # Если в первой группе один элемент:
        current_result.append(arr0) # Добавить первую группу в массив с текущим результатом кодирования.
    else: # Иначе (если в первой группе несколько элементов, то нужно продолжать разбиение):
        sf_encode(arr0, current_result) # Рекурсивный запуск функции для первой группы и массива с текущим результатом.
    return current_result # Возврат массива с результатом кодирования символов.
```

Рисунок 4 — Функция для кодирования по алгоритму Шеннона—Фано

```

# Главная функция.
def main():

    # Кодирование текста из файла.

    result = [] # Объявление массива с результатом кодирования символов.
    file = open("file.txt", encoding='utf8') # Создание файла "file.txt" с кодировкой 'utf8' и его открытие на чтение.
    # Запись всех символов из файла в переменную text.
    text = file.read()
    file.close() # Закрытие файла file.
    dictionary = {} # Объявление словаря dictionary.
    for ch in text: # Для каждого символа ch в тексте text:
        # Увеличение элемента словаря dictionary с ключом ch на 1 при наличии ключа ch.
        # Присвоение элементу словаря dictionary с ключом ch значения 0 при предварительном отсутствии ключа ch:
        dictionary[ch] = dictionary.get(ch, 0) + 1
    ch_quantity = len(text)
    print(f"Общее количество символов в тексте: {ch_quantity}.")
    # Сортировка словаря по кол-ву повторений элементов в тексте.
    dictionary = sorted(dictionary.items(), key=lambda x: x[1], reverse=True)
    array = [] # Объявление массива для дальнейшей передачи его в функцию для кодирования.
    for element in dictionary: # Для каждого элемента element словаря dictionary:
        array.append(list(element) + ['']) # Добавление в конец массива array списка из [element] и [''].
    result = sf_encode(array, result)[::-1] # Запись в массив с результатом перевёрнутого массива от функции sf_encode.
    print("Кодирование символов методом Шеннона-фано:")
    for i in range(len(result)): # Для i от 0 до длины массива result:
        result[i] = result[i][0] # Удаление вложенности массивов.
        print(result[i]) # Вывод элемента массива result с индексом i.
    # Создание файла "file_code.txt" с кодировкой 'utf8' и его открытие на запись.
    file_code = open("file_code.txt", "w", encoding='utf8')
    for ch in text: # Для каждого символа ch в тексте text:
        for element in result: # Для каждого элемента element массива result:
            if element[0] == ch: # Если текущий кодируемый символ совпадает с текущим символом в тексте:
                file_code.write(element[2]) # Запись кода данного символа в файл file_code.
    file_code.close() # Закрытие файла file_code.
    print('Кодирование текста методом Шеннона-фано завершено. Результат сохранён в файл "file_code.txt".')

```

Рисунок 5 — Фрагмент «главной функции» для кодирования текста

```

# Оценка сжатия текста.

print(f"Объём незакодированной фразы: V(text) = {ch_quantity} * 8 бит = {ch_quantity * 8} бит.")
volume_code = 0 # Объём закодированной фразы.
for element in result: # Для каждого элемента element массива result:
    volume_code += len(element[2]) * element[1] # Кол-во символов в закод. символе * на кол-во повторений символа.
print(f"Объём закодированной фразы: V(code) = {volume_code} бит.")
k = volume_code / (ch_quantity * 8) # Коэффициент сжатия.
k = float('%.3f').format(k) # Запись k с тремя знаками после запятой.
print(f"Коэффициент сжатия: k = {volume_code} бит / {ch_quantity * 8} бит ≈ {k}.")

# Декодирование текста из файла.

# Создание файла "file_code.txt" с кодировкой 'utf8' и его открытие на чтение.
file_code = open("file_code.txt", encoding='utf8')
code = file_code.read() # Запись всех символов из файла file_code в переменную code.
file_code.close() # Закрытие файла file_code.
# Создание файла "file_decode.txt" с кодировкой 'utf8' и его открытие на запись.
file_decode = open("file_decode.txt", "w", encoding='utf8')
code_piece = "" # Текущий рассматриваемый фрагмент закодированного текста.
for ch in code: # Для каждого символа ch в закодированном тексте code:
    code_piece += ch # Добавление символа ch в code_piece.
    for element in result: # Для каждого элемента element массива result:
        if code_piece == element[2]: # Если code_piece совпадает с кодом текущего кодируемого символа:
            file_decode.write(element[0]) # Запись декодированного символа в файл file_decode.
            code_piece = "" # Очистка текущего рассматриваемого фрагмента закодированного текста.
file_decode.close() # Закрытие файла file_decode.
print('Декодирование текста методом Шеннона-фано завершено. Результат сохранён в файл "file_decode.txt".')

if __name__ == "__main__":
    main()

```

Рисунок 6 — Фрагмент «главной функции» для оценки сжатия и декодирования текста

Закодированные данные были сохранены в файл, представленный на рисунке 10.

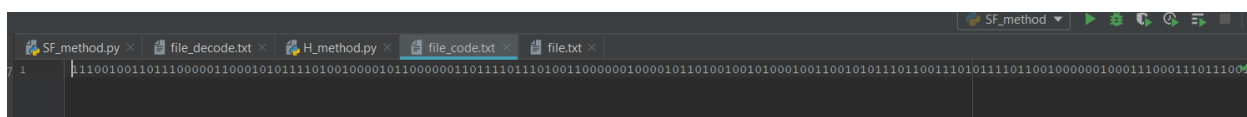


Рисунок 10 — Закодированный методом Шеннона—Фано текст

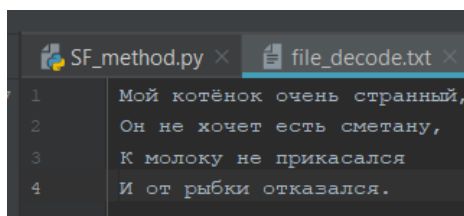
4) Расчёт коэффициента сжатия при кодировании методом Шеннона—Фано представлен на рисунке 11.

```
Объём незакодированной фразы:  $V(\text{text}) = 98 * 8 \text{ бит} = 784 \text{ бит}.$   
Объём закодированной фразы:  $V(\text{code}) = 437 \text{ бит}.$   
Коэффициент сжатия:  $k = 437 \text{ бит} / 784 \text{ бит} \approx 0.557.$ 
```

Рисунок 11 — Расчёт коэффициента сжатия методом Шеннона—Фано

Таким образом, кодирование методом Шеннона—Фано позволило уменьшить объём исходных данных в ~ 1.79 раз.

Далее было произведено декодирование, в результате которого из закодированного методом Шеннона-Фано текста был получен исходный текст без потери данных (см. рисунок 12).

A screenshot of a code editor with two tabs: 'SF_method.py' and 'file_decode.txt'. The 'file_decode.txt' tab is active and shows four lines of Russian text:

```
1 Мой котёнок очень странный,  
2 Он не хочет есть сметану,  
3 К молоку не прикасался  
4 И от рыбки отказался.
```

Рисунок 12 — Результат декодирования

3.2. Кодирование текста по методу Хаффмана и его декодирование.

1) **Код Хаффмана** — это особый тип оптимального префиксного кода, который обычно используется для сжатия данных без потерь.

Кодовый символ — это наименьшая единица данных, подлежащая сжатию.

Кодовое слово — это последовательность кодовых символов из алфавита кода.

Префиксный код — это код, в котором никакое кодовое слово не является префиксом любого другого кодового слова.

Оптимальный префиксный код — это префиксный код, имеющий минимальную среднюю длину.

Кодовое дерево (дерево кодирования Хаффмана, Н-дерево) — это бинарное дерево, у которого «листья» помечены символами, для которых разрабатывается кодировка, узлы (в том числе корень) помечены суммой вероятностей появления всех символов, соответствующих «листьям» поддерева, корнем которого является соответствующий узел.

Для кодирования текста из файла методом Хаффмана, декодирования закодированного методом Хаффмана текста в файл, а также для расчёта коэффициента сжатия данных, среднего значения, квадрата среднего значения, среднего квадратичного значения и дисперсии была разработана программа, представленная на рисунках 13 – 17.

```
import heapq # Импорт модуля heapq для работы с мин. кучей.
from collections import Counter # Импорт словаря с поддержкой счётчика для каждого объекта.
from collections import namedtuple # Импорт кортежа с индексируемыми и итерируемыми полями для поиска по атрибутам.

# Класс для "ветвей" дерева (внутренних узлов).
class Node(namedtuple("Node", ["left", "right"])):
    # функция для обхода дерева.
    def walk(self, code, pre):
        self.left.walk(code, pre + "0") # Переход к левому потомку с добавлением "0" к префиксу.
        self.right.walk(code, pre + "1") # Переход к правому потомку с добавлением "1" к префиксу.

# Класс для "листьев" дерева.
class Leaf(namedtuple("Leaf", ["char"])):
    def walk(self, code, pre): # Построенный код для данного символа.
        code[self.char] = pre or "0" # Если строка длиной 1, то pre = "", для этого случая — pre = "0".
```

Рисунок 13 — Классы узлов и «листьев» Н-дерева

```

# функция кодирования строки методом Хаффмана.
def huffman_encode(s):
    h = [] # Объявление очереди с приоритетами и её инициализация пустой.
    print("Таблица частот символов:")
    print('<символ>: <частота> (<кол-во повторов>')
    # Построение очереди циклом с уникальным для всех листьев счётчиком.
    for ch, freq in Counter(s).items():
        print("{}: {}".format(ch, freq / len(s)), end=' ')
        print(f"({freq})")
        # Очередь будет представлена частотой символа, счётчиком и самим символом.
        h.append((freq, len(h), Leaf(ch)))
    heapq.heapify(h) # Построение очереди с приоритетами.
    count = len(h) # Объявление счётчика и его инициализация со значением длины очереди.
    while len(h) > 1: # Пока в очереди больше 1 элемента:
        # Элемент с минимальной частотой — левый узел.
        freq1, _count1, left = heapq.heappop(h)
        # Следующий элемент с минимальной частотой — правый узел.
        freq2, _count2, right = heapq.heappop(h)
        # Вставка в очередь нового элемента, у которого частота равна сумме частот полученных из очереди элементов.
        heapq.heappush(h, (freq1 + freq2, count, Node(left, right))) # Добавление нового узла с потомками left, right.
        count += 1 # Увеличение значения счётчика на 1 (при добавлении нового элемента дерева).
    code = {} # Объявление словаря кодов символов и его инициализация пустым.
    # Если строка пустая, то очередь будет пустая — обхода не будет.
    if h: # Если в очереди 1 элемент:
        # В очереди 1 элемент, приоритет которого не важен, а сам элемент — корень дерева.
        [(_freq, _count, root)] = h
        root.walk(code, "") # Обход дерева от корня и заполнение словаря для получения кодирования Хаффмана.
    return code # Возврат словаря символов и соответствующих им кодов Хаффмана.

```

Рисунок 14 — Функция кодирования строки методом Хаффмана

```

# функция декодирования строки по кодам Хаффмана.
def huffman_decode(encoded, code):
    decoded = [] # Объявление массива символов расшифрованной строки и его инициализация пустым.
    enc_ch = "" # Объявление переменной значения закодированного символа и её инициализация пустой строкой.
    # Обход закодированной строки по символам.
    for ch in encoded: # Для каждого символа ch в строке encoded.
        enc_ch += ch # Добавление текущего символа ch к строке закодированного символа enc_ch.
        # Нахождение закодированного символа в словаре кодов.
        for dec_ch in code: # Для каждого ключа dec_ch в словаре code.
            if code.get(dec_ch) == enc_ch: # Если закодированный символ найден:
                decoded.append(dec_ch) # Добавление в конец массива decoded значения расшифрованного символа.
                enc_ch = "" # Обнуление значения закодированного символа.
                break # Закодированный символ найден — выход из цикла.
    return "".join(decoded) # Возврат значения расшифрованной строки.

```

Рисунок 15 — Функция декодирования строки методом Хаффмана

```

# Главная функция.
def main():
    # Кодирование текста из файла.

    file = open("file.txt", encoding='utf8') # Создание файла "file.txt" с кодировкой 'utf8' и его открытие на чтение.
    # Запись всех символов из файла в переменную text.
    text = file.read()
    file.close() # Закрытие файла file.
    ch_quantity = len(text)
    print(f"Общее количество символов в тексте: {ch_quantity}.")
    code = huffman_encode(text) # Запуск функции кодирования текста методом Хаффмана.
    # Отображение закодированного текста путём отображения каждого символа соответств. кодом и объединением результата.
    encoded = "".join(code[ch] for ch in text)
    print("Кодирование символов методом Хаффмана:")
    # Обход символов в словаре в алфавитном порядке с помощью функции sorted().
    for ch in sorted(code): # Для каждого символа ch в отсортированном словаре code:
        print("{}: {}".format(ch, code[ch])) # Вывод символа, кол-ва его повторов и соответствующего ему кода.
    # Создание файла "file_code.txt" с кодировкой 'utf8' и его открытие на запись.
    file_code = open("file_code.txt", "w", encoding='utf8')
    for ch in encoded: # Для каждого символа ch в закодированном тексте encoded:
        file_code.write(ch) # Запись данного символа в файл file_code.
    file_code.close() # Закрытие файла file_code.
    print('Кодирование текста методом Хаффмана завершено. Результат сохранён в файл "file_code.txt".')

```

Рисунок 16 — Фрагмент «главной функции» для кодирования текста

```

# Оценка сжатия текста.

print(f"Объем незакодированной фразы: V(text) = L(ASCII) = {ch_quantity} * 8 бит = {ch_quantity * 8} бит.")
volume_code = 0 # Объем закодированной фразы.
for ch, freq in Counter(text).items(): # Для каждого символа ch и кол-ва его повторений freq в тексте text:
    volume_code += len(code[ch]) * freq # Кол-во символов в закод. символе * на кол-во повторений символа.
print(f"Объем закодированной фразы: V(code) = L(Huff) = {volume_code} бит.")
k = volume_code / (ch_quantity * 8) # Коэффициент сжатия.
k = float('{:.3f}'.format(k)) # Запись k с тремя знаками после запятой.
print(f"Коэффициент сжатия: k = V(code) / V(text) = L(Huff) / L(ASCII) = "
      f"{volume_code} бит / {ch_quantity * 8} бит ≈ {k}.")

# Нахождение средней величины и дисперсии.
average_value = 0 # Средняя величина.
average_quadratic_value = 0 # Средняя квадратичная величина.
for ch, freq in Counter(text).items(): # Для каждого символа ch и кол-ва его повторений freq в тексте text:
    average_value += freq / len(text) * freq # <вероятность> * <кол-во повторений символа>.
    average_quadratic_value += freq / len(text) * freq**2 # <вероятность> * <кол-во повторений символа>^(2).
variance = average_quadratic_value - average_value**2 # Дисперсия.
print(f"Средняя величина: M[X] ≈ '{:.3f}'.format(average_value).")
print(f"Квадрат средней величины: M²[X] ≈ '{:.3f}'.format(average_value**2).")
print(f"Средняя квадратичная величина: M[X²] ≈ '{:.3f}'.format(average_quadratic_value).")
print(f"Дисперсия: D[X] = M[X²] - M²[X] ≈ '{:.3f}'.format(variance).")

# Декодирование текста.

file_decode = open("file_decode.txt", "w", encoding='utf8')
decoded = huffman_decode(encoded, code) # Запуск функции декодирования текста.
for ch in decoded: # Для каждого символа ch в расшифрованном тексте decoded:
    file_decode.write(ch) # Запись данного символа в файл file_decode.
file_decode.close() # Закрытие файла file_decode.
print('Декодирование текста методом Хаффмана завершено. Результат сохранен в файл "file_decode.txt".')

```

Рисунок 17 — Фрагмент «главной функции» для оценки сжатия, нахождения средней величины и дисперсии, декодирования текста

2) Перед запуском программы был создан исходный файл с текстом «Тульцова Анастасия Денисовна» (см. рисунок 18).

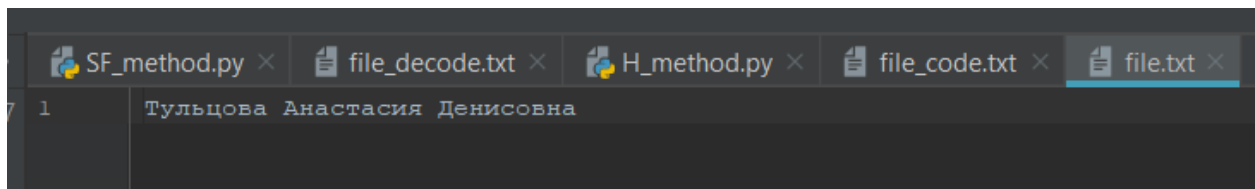


Рисунок 18 — Файл с исходным текстом для дальнейшего кодирования

После запуска программы были составлены таблицы частот символов (см. рисунок 19) и формирования кода (см. рисунок 20).


```

Общее количество символов в тексте: 28.
Таблица частот символов:
'<символ>': <частота> (<кол-во повторений>)
'T': 0.036 (1)
'y': 0.036 (1)
'л': 0.036 (1)
'ь': 0.036 (1)
'ц': 0.036 (1)
'o': 0.071 (2)
'в': 0.071 (2)
'a': 0.143 (4)
' ': 0.071 (2)
'A': 0.036 (1)
'н': 0.107 (3)
'с': 0.107 (3)
'т': 0.036 (1)
'и': 0.071 (2)
'я': 0.036 (1)
'д': 0.036 (1)
'е': 0.036 (1)

```

Рисунок 19 — Таблица частот символов

```

Кодирование символов методом Хаффмана:
' ': 1010
'A': 11101
'д': 0000
'T': 11000
'a': 011
'в': 1001
'е': 0001
'и': 1011
'л': 11010
'н': 001
'o': 1000
'с': 010
'т': 11110
'y': 11001
'ц': 11100
'ь': 11011
'я': 11111

```

Рисунок 20 — Таблица формирования кода Хаффмана

3) Тогда построенное префиксное дерево Хаффмана выглядит следующим образом (см. рисунок 21):

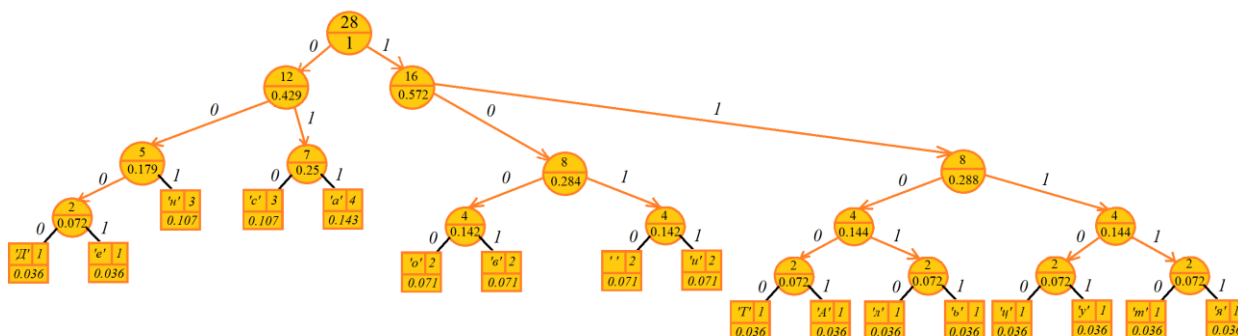


Рисунок 21 — Префиксное дерево кода Хаффмана

Закодированные данные были сохранены в файл (см. рисунок 22).

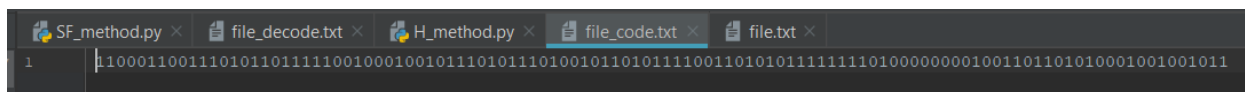


Рисунок 22 — Закодированный методом Хаффмана текст

4) См. ранее (см. п. 2 раздела 3.2).

5) Расчёт коэффициента сжатия относительно кодировки ASCII при кодировании методом Хаффмана представлен на рисунке 23.

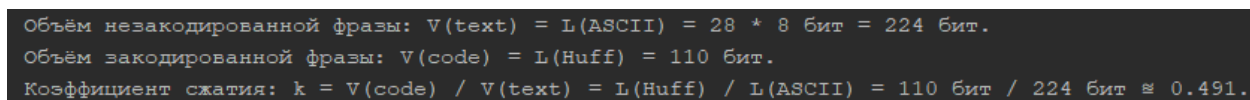


Рисунок 23 — Расчёт коэффициента сжатия методом Хаффмана

Таким образом, кодирование методом Хаффмана позволило уменьшить объём исходных данных в ~2.04 раза.

6) Далее была определена средняя величина, квадрат средней величины, средняя квадратичная величина и дисперсия (см. рисунок 24).

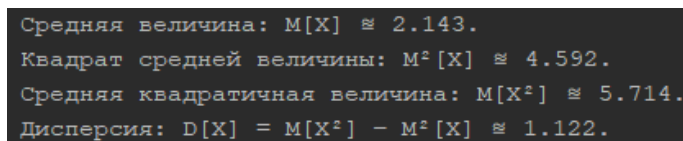


Рисунок 24 — Средняя величина и дисперсия

После этого было произведено декодирование, в результате которого из закодированного методом Хаффмана текста был получен исходный текст без потери данных (см. рисунок 25).

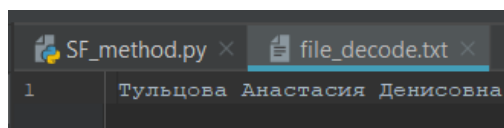


Рисунок 25 — Результат декодирования

7) См. ранее (см. раздел 3.2).

3.3. Сравнение коэффициентов сжатия разработанного алгоритма Хаффмана и архиватора.

Для сравнения коэффициентов сжатия разработанного алгоритма Хаффмана и архиватора был выбран архиватор «WinRAR», а также создан

текстовый файл (см. рисунок 26) с данными из предыдущего задания (см. раздел 3.1).

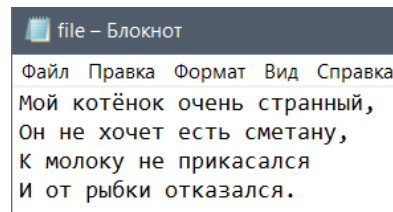


Рисунок 26 — Файл с исходными данными

Рассмотрим коэффициент сжатия разработанного алгоритма Хаффмана для данного файла (см. рисунок 27).

```
Кодирование текста методом Хаффмана завершено. Результат сохранён в файл "file_code.txt".
Объём незакодированной фразы:  $V(\text{text}) = L(\text{ASCII}) = 98 * 8 \text{ бит} = 784 \text{ бит}$ .
Объём закодированной фразы:  $V(\text{code}) = L(\text{Huff}) = 437 \text{ бит}$ .
Коэффициент сжатия:  $k = V(\text{code}) / V(\text{text}) = L(\text{Huff}) / L(\text{ASCII}) = 437 \text{ бит} / 784 \text{ бит} \approx 0.557$ .
```

Рисунок 27 — Расчёт коэффициента сжатия методом Хаффмана

Далее файл был заархивирован с помощью программы-архиватора «WinRAR» (см. рисунок 28).

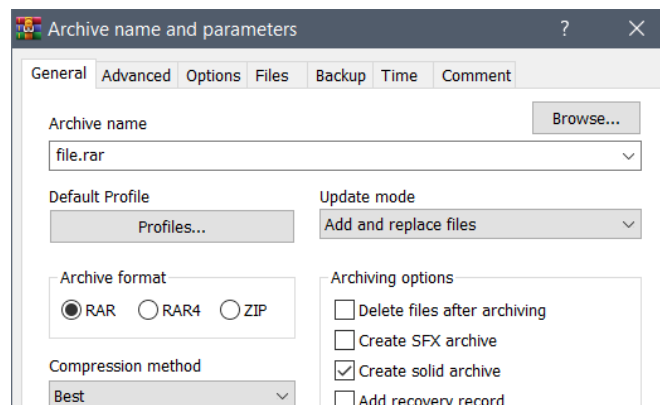


Рисунок 28 — Архивация файла с помощью программы «WinRAR»

По результату архивации (см. рисунок 29) был определён коэффициент сжатия файла при использовании программы «WinRAR»: $k_{\text{rar}} = 0.75$.

| Name | Size | Packed |
|----------|------|--------|
| .. | | |
| file.txt | 180 | 135 |

Рисунок 29 — Результат архивации

Таким образом, разработанный алгоритм Хаффмана превосходит по степени сжатия выбранный метод архивации в программе «WinRAR» в ~ 1.35 раз: $k_{\text{rar}} / k \approx 1.35$.

Вывод

В ходе работы были получены практические навыки по выполнению сжатия данных следующими методами без потерь: методом «RLE», методами «LZ77», «LZ78», методом Шеннона—Фано и методом Хаффмана.

Список информационных источников

1. Лекции по дисциплине «Структуры и алгоритмы обработки данных» / Л. А. Скворцова, МИРЭА – Российский технологический университет, 2021.