1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных наук и кибербезопасности
5. **Высшая школа кибербезопасности**

**ЛАБОРАТОРНАЯ РАБОТА № 7**

**«Кодирование и упаковка данных»**

1. по дисциплине «Основы информационной безопасности»
2. Выполнили
3. студенты гр. 5131001/40003 Веденеев А. С.
4. <*подпись*>  
     
    Поляков Д. Т.
   1. <*подпись*>
5. Преподаватель
6. ст. преподаватель Вагисаров В. Б.

<*подпись*>

1. Санкт-Петербург
2. 2025

# Цель и задачи

**Цель:** приобретение навыков по защите информации с

помощью методов кодирования, получение прикладных знаний в области

исследования и реализации алгоритмов упаковки данных.

**Задачи:**

1. Согласовать с преподавателем два разных алгоритма кодирования
2. Создать рабочий файл, содержащий информацию, предназначенную для кодирования. Использовать алфавит из 3...5 символов. Размер файла - более 100 000 символов.
3. Разработать две утилиты, реализующие процедуры кодирования и декодирования для обоих алгоритмов. Ключ, определяющий режим работы утилиты (кодирование или декодирование), пути к входному и выходному файлам должны задаваться в командной строке запуска утилит. Программы должны содержать функции подсчета длительности кодирования/декодирования и коэффициента сжатия (отношения размера закодированного файла к размеру исходного).
4. Закодировать входной файл при помощи одной утилиты. Зафиксировать длительность кодирования и степень сжатия для первого алгоритма.
5. Закодировать тот же входной файл при помощи второй утилиты. Зафиксировать длительность кодирования и степень сжатия для второго алгоритма.
6. Декодировать файл, полученный на выходе первой утилиты. Зафиксировать длительность декодирования для первого алгоритма.
7. Декодировать файл, полученный на выходе второй утилиты. Зафиксировать длительность декодирования для второго алгоритма.
8. Полученный на выходе первой утилиты закодированный файл направить на вход второй утилиты кодирования и закодировать его.
9. Сначала с помощью второй, а затем первой утилиты декодировать полученный в п. 8 файл. Сравнить декодированный файл с исходным, убедиться в их идентичности.
10. Повторить выполнение пп. 4-9 для произвольного бинарного файла (например, файла в формате Word, Adobe) размером более 2 Мб.
11. Создать новый рабочий файл из одного символа - ноль в шестнадцатеричной кодировке. Повторить выполнение пп. 4-9.
12. Построив гистограммы, сопоставить характеристики двух исследованных алгоритмов (длительность кодирования, длительность декодирования, коэффициент сжатия) для текстового файла малого алфавита, для бинарного файла, для файла с одним символом. Обосновать полученные показатели. Отметить замечания и предложить рекомендации по оптимизации работы кодировщиков и декодировщиков.

# ход работы

* 1. Перед началом работы были выбраны два алгоритма кодирования: алгоритм Хаффмена и алгоритм LZW.
  2. С использованием языка Python и библиотеки random был создан файл “text.txt”, содержащий в себе 121220 символ из алфавита из 4 символов: [“a”, “b”, “c”, “d”]. Символы были расставлены в случайном порядке с использованием функций shuffle и choice из библиотеки random.
  3. На языке программирования Python были разработаны две утилиты “new\_Huffman.py” и “LZW.py”, реализующие кодирование и декодирование алгоритмов Хаффмена и LZW. На вход каждой утилите приходит команда формата: “python <имя файла.py> <encode/decode> <путь\_к\_файлу\_для кодирования/декодирования> <путь\_к\_файлу\_в который кодируют/декодируют>”. В результате работы программа выводит статистику со следующим содержанием: размер первого и второго файла в байтах, коэффициент сжатия и время работы программы в секундах.
  4. При помощи утилиты “LZW.py” был закодирован файл “text.txt” в файл “LZW\_encode.txt” (рисунок 1).

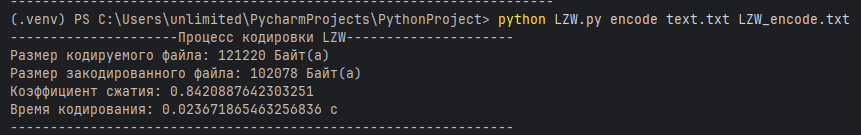


Рисунок 1. Результат кодирования утилиты “LZW.py”.

* 1. При помощи утилиты “new\_Huffman.py” был закодирован файл “text.txt” в файл “Huffman\_encode.txt” (рисунок 2).

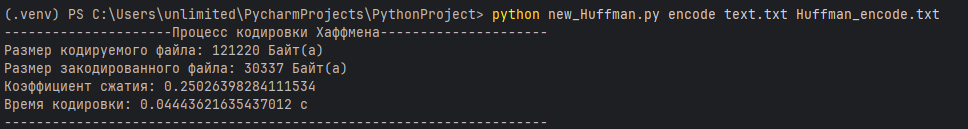


Рисунок 2. Результат кодирования утилиты “ new\_Huffman.py”.

* 1. При помощи утилиты “LZW.py” был декодирован файл “LZW\_encode.txt” в файл “LZW\_decode.txt” (рисунок 3).

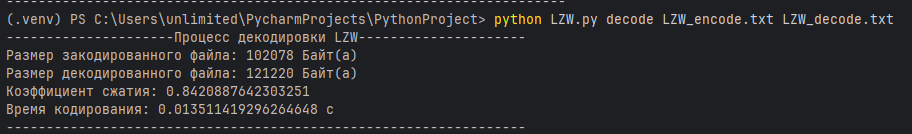


Рисунок 3. Результат декодирования утилиты “LZW.py”.

* 1. При помощи утилиты “new\_Huffman.py” был декодирован файл “Huffman\_encode.txt” в файл “Huffman\_decode.txt” (рисунок 4).

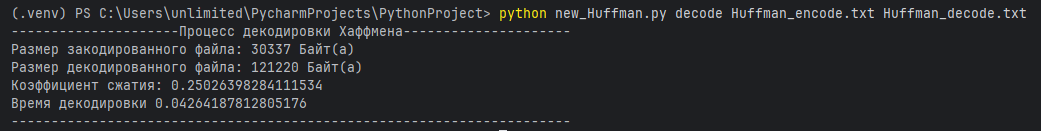


Рисунок 4. Результат декодирования утилиты “ new\_Huffman.py”.

* 1. Полученный на выходе утилиты “LZW.py” файл “LZW\_encode.txt” был направлен на вход утилите “new\_Huffman.py” и записан в файл “Huffman\_encode.txt” (рисунок 5).

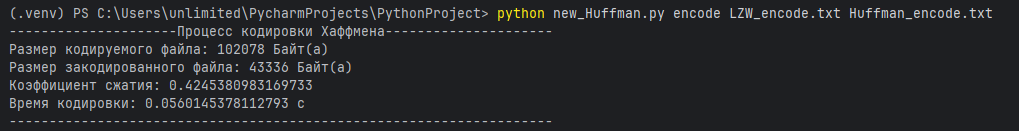


Рисунок 5. Результат кодирования файла “LZW\_encode.txt” утилитой “new\_Huffman.py”

* 1. Файл “Huffman\_encode.txt” был декодирован в файл “LZW\_encode.txt” утилитой “new\_Huffman.py” (рисунок 6), а после декодирован утилитой “LZW.py” в файл “LZW\_decode.txt” (рисунок 7). Для сравнения идентичности была написана утилита “files\_compare.py”, сравнивающая каждый соответственный символ в двух файлах, утилита принимает на вход команда формата: “ python files\_compare.py <путь\_к\_первому\_файлу> <путь\_ко\_второму\_файлу>”. В результате выполнения утилиты для файлов “text.txt” и “LZW\_decode.txt” была доказана их идентичность (рисунок 8).

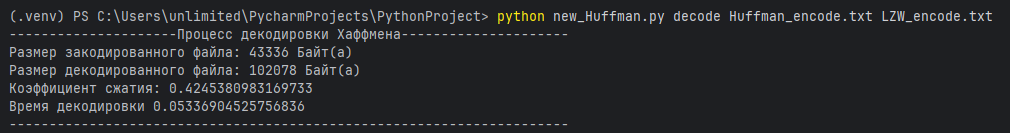


Рисунок 6. Результат декодирования файла “Huffman\_encode.txt” утилитой “new\_Huffman.py”.

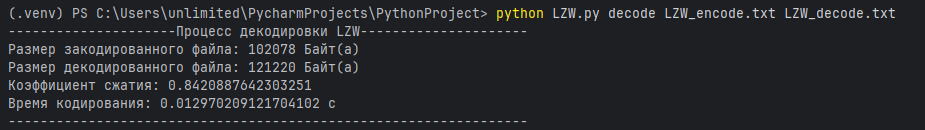


Рисунок 7. Результат декодирования файла “LZW\_encode.txt” утилитой “LZW.py”.



Рисунок 8. Результат сравнения файлов “text.txt” и “LZW\_decode.txt” утилитой “files\_compare.py”.

* 1. Для выполнения пункта был создан файл “text.docx”, содержащий в себе 12122005 символов из алфавита [“0”, “1”], размером примерно 11,5 МБ. После этого были заново выполнены пункты 4-9 для файла “text.docx” с использованием утилит “LZW.py” и “new\_Huffman.py” а также файлов “LZW\_encode.docx”, “LZW\_decode.docx”, “Huffman\_encode.docx”, “Huffman\_decode.docx”. Результаты выполнения представлены в Приложении 1.
  2. Для выполнения пункта был создан файл “0\_in\_16.txt” с единственным содержимым – 0 в шестнадцатеричной записи. После этого были заново выполнены пункты 4-9 для файла “0\_in\_16.txt” с использованием утилит “LZW.py” и “new\_Huffman.py” а также файлов “LZW\_encode.docx”, “LZW\_decode.docx”, “Huffman\_encode.docx”, “Huffman\_decode.docx”. Результаты выполнения представлены в Приложении 2.
  3. Были построены таблицы и гистограммы сравнения двух алгоритмов. Сравнив характеристики можно сделать вывод, что с задачей кодировки и декодировки быстрее всего справляется LZW, хотя сжимает он намного хуже, чем алгоритм Хаффмана. Особого внимания заслуживает случай работы с файлом, состоящим из одного символа. Здесь оба алгоритма показали крайне низкую эффективность: размер выходного файла значительно превысил исходный. Это связано с тем, что служебная информация (таблицы кодов, дерево Хаффмана и пр.) существенно превышает полезные данные. Подобные ситуации демонстрируют ограниченность исследованных методов при работе с сильно однотипными или слишком короткими данными. Таким образом, можно сделать вывод, что выбор алгоритма зависит от поставленной задачи: если приоритетом является скорость обработки данных, предпочтительным будет LZW, а если требуется максимальное сжатие при работе с файлами разнообразной структуры — лучше использовать кодирование Хаффмана.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Алфавит из 4 символов | Бинарный файл | 0 в шестнадцатеричной |  |
| Алгоритм LZW | 0,027с | 2,50с | 0с | Время кодирования |
| 0,014с | 0,71с | 0с | Время декодирования |
| 0,84 | 0,4 | 2 | Коэффициент сжатия |
| Алгоритм Хаффмана | 0,044с | 2,79с | 0с | Время кодирования |
| 0,043с | 2,76с | 0с | Время декодирования |
| 0,25 | 0,125 | 7 | Коэффициент сжатия |

# ответы на контольные вопросы

1. В чем заключаются преимущества и недостатки адаптивных методов кодирования по сравнению со статическими? - Адаптивные методы кодирования изменяют свои параметры в процессе обработки данных: вероятности появления символов или структура кодов обновляются «на лету». Это позволяет получать хорошие результаты сжатия без предварительного анализа всего файла. Преимущество таких методов заключается в возможности обработки потока данных неизвестной структуры и отсутствии необходимости хранить дополнительную служебную информацию (например, таблицы частот). Недостатком является сравнительно высокая вычислительная сложность и замедление работы, так как пересчёт вероятностей и перестроение кодов выполняются динамически. Статические методы, напротив, проще и быстрее, но требуют анализа всего файла перед началом кодирования и передачи в заголовке вспомогательной информации.
2. Опишите алгоритм получения арифметического кода? - Кодирование реализуется путем сужения исходного интервала в зависимости от приходящей буквы. При получении первой буквы сообщения кодер определяет, какой ей соответствует интервал, и сужает исходный интервал [0; 1) до него. При получении следующей буквы кодер определяет ее интервал и ставит этому интервалу в соответствие новый, внутри суженного на предыдущем шаге. Новый интервал в пропорции к суженному будет таким же, как интервал буквы в пропорции к исходному интервалу [0; 1). При получении следующих символов интервал снова сужается. После сужения его в последний раз любое число, лежащее внутри полученного интервала, будет являться кодом сообщения.
3. Почему текстовые и бинарные файлы имеют разную степень сжатия? - Текстовые файлы обычно имеют небольшой алфавит и высокую статистическую неравномерность частот символов (например, в русском или английском языке чаще встречаются гласные, пробелы и определённые буквы). Это позволяет алгоритмам эффективно кодировать текст. Бинарные файлы, напротив, часто содержат данные, близкие к случайным, где равномерное распределение символов лишает алгоритмы возможности извлечь избыточность. Поэтому такие файлы сжимаются хуже или даже могут увеличиться в размере.
4. В каких случаях кодирование не имеет смысла? - Кодирование не имеет смысла, если при использовании алгоритма кодирования закодированное сообщение становится по объему больше, чем исходное. Также, это бессмысленно если сообщение небольшого размера.
5. В чем заключается доработка алгоритма LZW по сравнению с алгоритмом LZ? - Алгоритм LZW - усовершенствованная версия алгоритма LZ. Исходные символы получают свои кодовые номера, например, ASCII-коды. Эту информацию не надо передавать декодеру, так как она общеизвестна. Встретившиеся в тексте цепочки символов заносятся в кодовую таблицу для дальнейшего использования.

# Выводы

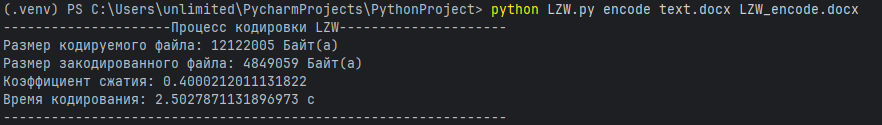
В ходе выполнения лабораторной работы были изучены и реализованы два алгоритма упаковки данных — кодирование Хаффмана и алгоритм LZW. Реализация была выполнена в виде двух самостоятельных утилит, поддерживающих режимы кодирования и декодирования, а также подсчёт длительности обработки и коэффициента сжатия.

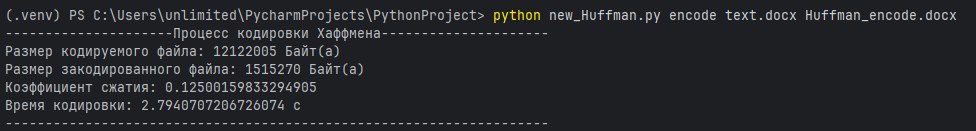
Экспериментальная часть включала работу с текстовым файлом малого алфавита, бинарным файлом значительного объёма и файлом, содержащим один символ. Для каждого случая были проведены замеры времени кодирования, декодирования и вычислен коэффициент сжатия. Результаты показали различия в характеристиках двух методов: LZW демонстрировал более высокую скорость работы, но уступал по эффективности сжатия, тогда как алгоритм Хаффмана обеспечивал лучшее уменьшение размера данных, но требовал больше времени на обработку. При этом в случае с односимвольным файлом оба алгоритма оказались неэффективны, что объясняется значительным объёмом служебной информации по сравнению с полезными данными.

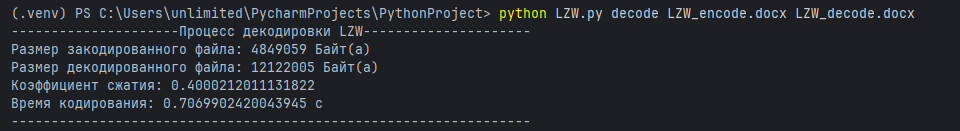
Таким образом, цель работы была достигнута: были получены прикладные навыки защиты информации с использованием методов кодирования и проведено сравнение их характеристик на практике. Задачи, поставленные в рамках лабораторной работы, были выполнены — разработаны утилиты кодирования и декодирования, проведены эксперименты с различными типами файлов, построены и проанализированы зависимости времени и степени сжатия.

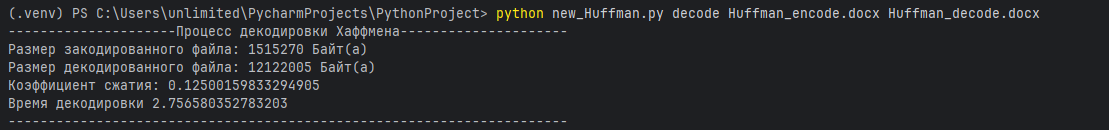
# Приложения

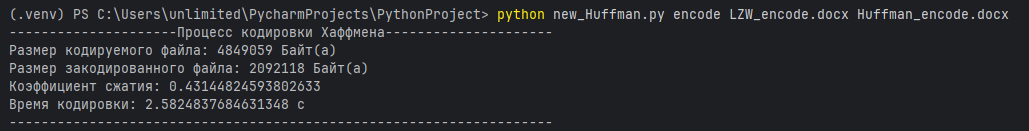
Приложение 1. Результаты выполнения пп. 4-9 для файла “text.docx”.

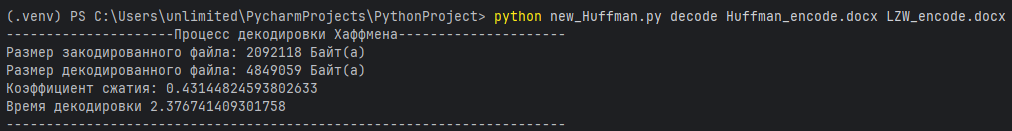


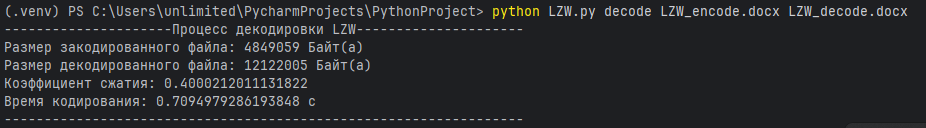






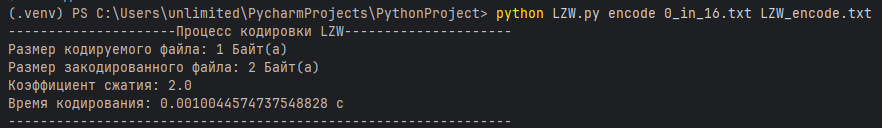


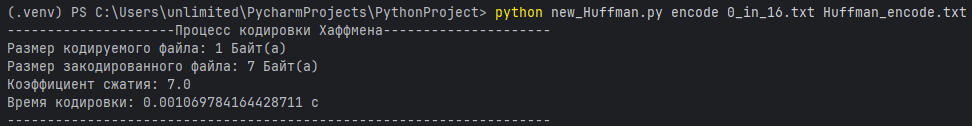


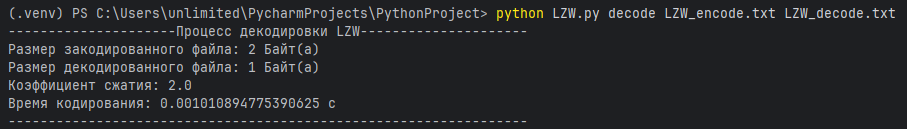


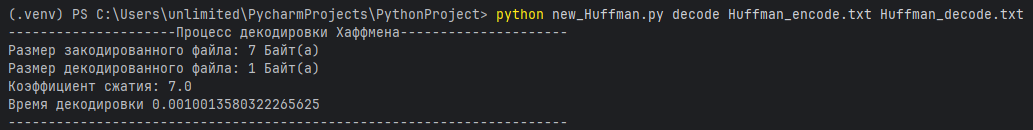


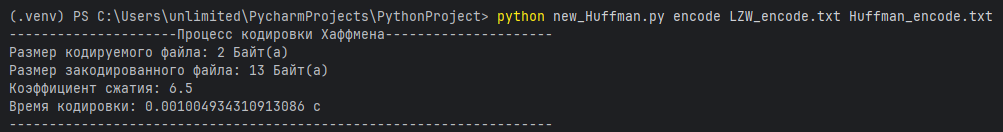
Приложение 2. Результаты выполнения пп. 4-9 для файла “0\_in\_16.txt”.

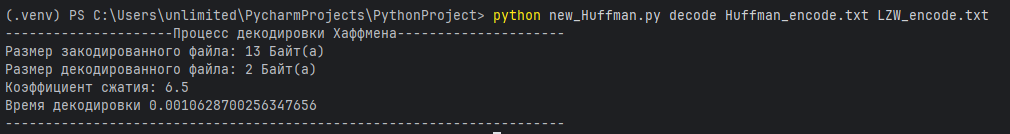


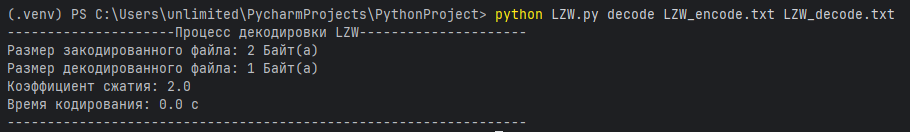














Приложение 3. Гистограммы сравнения алгоритмов кодирования.

Приложение 4. Листинг утилиты «LZW.py»

import time  
import os  
import sys  
  
# для запуска кодирования: python LZW.py encode text.txt LZW\_encode.txt  
# для запуска декодирования: python LZW.py decode LZW\_encode.txt LZW\_decode.txt  
  
def encode(input\_file, output\_file):  
 start = time.time()  
 print("---------------------Процесс кодировки LZW---------------------")  
 input\_file\_size = os.path.getsize(input\_file)  
 print("Размер кодируемого файла:", input\_file\_size, "Байт(а)")  
  
 read\_file = open(input\_file, 'r')  
 write\_file = open(output\_file, 'w')  
  
 code\_dict = dict()  
 message = read\_file.read()  
 temp\_str = message[0]  
 i = 256  
 # процесс кодировки по методичке  
 for char in message[1::]:  
 if (temp\_str + char) in code\_dict:  
 temp\_str += char  
 else:  
 if (len(temp\_str) > 1): # если строка - один символ, записываем ее через получение кода ASCII (ord(char)), иначе как индекс в словаре  
 write\_file.write(str(code\_dict[temp\_str]) + ";")  
 else:  
 write\_file.write(str(ord(temp\_str)) + ";")  
 code\_dict[temp\_str + char] = i  
 temp\_str = char  
 i+=1  
 if (len(temp\_str) > 1):  
 write\_file.write(str(code\_dict[temp\_str]) + ";")  
 else:  
 write\_file.write(str(ord(temp\_str)) + ";")  
  
 read\_file.close()  
 write\_file.close()  
 output\_file\_size = os.path.getsize(output\_file)  
 print("Размер закодированного файла:", output\_file\_size, "Байт(а)")  
 k = output\_file\_size / input\_file\_size  
 print("Коэффициент сжатия:", k)  
 print("Время кодирования:", time.time() - start, "с")  
 print("---------------------------------------------------------------")  
  
def decode(input\_file, output\_file):  
 start = time.time()  
 print("---------------------Процесс декодировки LZW---------------------")  
 input\_file\_size = os.path.getsize(input\_file)  
 print("Размер закодированного файла:", input\_file\_size, "Байт(а)")  
  
 read\_file = open(input\_file, 'r')  
 write\_file = open(output\_file, 'w')  
  
 message = (read\_file.readline().split(";")) # последний символ - пустой, так что сразу почистим его с помощью pop - возвращает последний символ списка, если ничего не указывать  
 message.pop()  
 codes = {i: chr(i) for i in range(256)}  
 i = 256  
 # процесс декодировки по методичке, обозначим за now\_code - текущий код, char - символ, соответствующий текущему коду, temp\_str - строка, next\_code - следующий код  
 now\_code = int(message[0]) # читать текущий код  
 temp\_str = codes[now\_code]  
 write\_file.write(temp\_str) # вывести в выходной поток текущий код  
 char = temp\_str[0]  
 for next\_code in message[1::]: # из-за того что мы уже считали первый символ, продолжаем со второго  
 next\_code = int(next\_code)  
 if next\_code in codes: # если следующего кода нет в словаре  
 temp\_str = codes[next\_code] # то строка соответствует текущему коду в словаре  
 else:  
 temp\_str = codes[now\_code] + char # строка = строка, соответствующая следующему коду в словаре  
 write\_file.write(temp\_str) # выводим строку в файл  
 char = temp\_str[0] # символ - первый символ выведенной строки  
 codes[i] = codes[now\_code] + temp\_str[0] # добавляем в словарь код + символ  
 i+=1  
 char = temp\_str[0]  
 now\_code = next\_code # переходим к следующему символу, изменяя текущий код на следующий  
  
 read\_file.close()  
 write\_file.close()  
 output\_file\_size = os.path.getsize(output\_file)  
 print("Размер декодированного файла:", output\_file\_size, "Байт(а)")  
 k = input\_file\_size / output\_file\_size  
 print("Коэффициент сжатия:", k)  
 print("Время кодирования:", time.time() - start, "с")  
 print("-----------------------------------------------------------------")  
  
if len(sys.argv) != 4:  
 print("Для запуска: python LZW.py encode/decode input\_file output\_file")  
 sys.exit(1)  
  
mode = sys.argv[1]  
input\_file = sys.argv[2]  
output\_file = sys.argv[3]  
  
if mode == "encode":  
 encode(input\_file, output\_file)  
elif mode == "decode":  
 decode(input\_file, output\_file)

Приложение 5. Листинг утилиты «new\_Huffman.py»

import os, time, sys  
  
#для запуска в консоли Win: .venv\Scripts\activate уже после можно делать следующие команды  
#для запуска кодирования: python new\_Huffman.py encode text.txt Huffman\_encode.txt  
#для запуска декодирования: python new\_Huffman.py decode Huffman\_encode.txt Huffman\_decode.txt  
  
def encode(input\_file, output\_file):  
 start = time.time()  
 print("---------------------Процесс кодировки Хаффмена---------------------")  
 input\_file\_size = os.path.getsize(input\_file)  
 print("Размер кодируемого файла:", input\_file\_size, "Байт(а)")  
  
 # читаем символы как байты  
 with open(input\_file, "rb") as read\_file:  
 message = read\_file.read()  
  
 # считаем частоты по байтам (ключ — int 0..255)  
 count = {}  
 length = 0  
 for b in message:  
 count[b] = count.get(b, 0) + 1 # .get возвращает или count[b] или 0  
 length += 1  
  
 # обработка случая если файл пустой  
 if length == 0:  
 with open(output\_file, "wb") as wf:  
 wf.write(b"0\n\n")  
 print("Пустой файл — готово.")  
 return  
  
 # nodes: список кортежей (freq, uid, node)  
 # node: (symbol\_or\_None, left\_node, right\_node)  
 nodes = []  
 uid = 0  
 for symbol, freq in count.items():  
 nodes.append((freq, uid, (symbol, None, None)))  
 uid += 1  
  
 # если всего 1 уникальный символ — даём ему код '0'  
 if len(nodes) == 1:  
 \_, \_, single = nodes[0] # так как символ лишь один то для него мы не уточняем uid и вероятность - нам нужно изменить лишь символ  
 codes = { single[0]: "0" } # приписываем код для символа  
 else:  
 # процесс объединения узлов по их вероятностям  
 while len(nodes) > 1:  
 # сортируем по вероятности появления freq и uid, первым идет freq так что uid не потеряется  
 nodes.sort(key=lambda x: (x[0], x[1]))  
 f1, uid1, n1 = nodes.pop(0) # тут n1 и n2 - кортеж (symbol, ссылки на самих себя)  
 f2, uid2, n2 = nodes.pop(0)  
 new\_node = (None, n1, n2) # создаем кортеж, где однако не сохраняем символ, потому что он должен храниться только у листьев - то есть одиночных символов, слитым символам такой параметр не нужен, он нужен пользователю возможно, но не компьютеру  
 nodes.append((f1 + f2, uid, new\_node)) # вносим новый узел - слияние n1 и n2  
 uid += 1  
  
 # корень дерева  
 \_, \_, root = nodes[0]  
  
 # обходим полученное дерево и заполняем словарь  
 codes = {}  
 def build\_codes(node, prefix):  
 symbol, left, right = node  
 if symbol is not None:  
 # ситуация листа: если prefix пустой (единичный символ), даём '0'  
 codes[symbol] = prefix if prefix != "" else "0"  
 return  
 # совершаем рекурсивный обход дерева, влево - 0, вправо - 1  
 build\_codes(left, prefix + "0")  
 build\_codes(right, prefix + "1")  
  
 build\_codes(root, "")  
  
 # ---- запись: заголовок + упакованные байты ----  
 with open(output\_file, "wb") as write\_file:  
 # 1) длина (кол-во байт) и перевод строки  
 write\_file.write(str(length).encode("utf-8"))  
 write\_file.write(b"\n")  
 # 2) коды (ключи — целые числа)  
 str\_codes = ";".join(f"{key}:{value}" for key, value in codes.items())  
 write\_file.write(str\_codes.encode("utf-8"))  
 write\_file.write(b"\n")  
 # 3) упакуем биты в байты по ходу  
 buffer = 0  
 bits\_entered = 0  
 for b in message:  
 code = codes[b]  
 for bit\_symbol in code:  
 buffer = (buffer << 1) | int(bit\_symbol)  
 bits\_entered += 1  
 if bits\_entered == 8:  
 write\_file.write(bytes([buffer]))  
 buffer = 0  
 bits\_entered = 0  
 # дописываем последний неполный байт, заполняя если что нулями справа  
 if bits\_entered > 0:  
 buffer = buffer << (8 - bits\_entered)  
 write\_file.write(bytes([buffer]))  
  
 output\_file\_size = os.path.getsize(output\_file)  
 k = output\_file\_size / input\_file\_size  
 print("Размер закодированного файла:", output\_file\_size, "Байт(а)")  
 print("Коэффициент сжатия:", k)  
 print("Время кодировки:", time.time() - start, "с")  
 print("--------------------------------------------------------------------")  
  
def decode(input\_file, output\_file):  
 start = time.time()  
 print("---------------------Процесс декодировки Хаффмена---------------------")  
 encode\_file\_size = os.path.getsize(input\_file)  
 print("Размер закодированного файла:", encode\_file\_size, "Байт(а)")  
  
 with open(input\_file, "rb") as encode\_file, open(output\_file, "wb") as decode\_file:  
 length\_bytes = b""  
 while True:  
 ch = encode\_file.read(1)  
 if not ch or ch == b'\n':  
 break  
 length\_bytes += ch  
 length = int(length\_bytes.decode("utf-8"))  
  
 codes\_str = b""  
 while True:  
 ch = encode\_file.read(1)  
 if ch == b'\n':  
 break  
 codes\_str += ch  
 codes\_str = (codes\_str.decode("utf-8"))  
  
 codes = dict(item.split(":") for item in codes\_str.split(";")) # сначала разделяет по ",", так как цикл запускается первым, после делит на пары ключ:значение по запятой  
 codes = {v: int(k) for k, v in codes.items()} # разворачиваем словарь - меняем местами ключи и значения чтобы правильно декодировать  
  
 encoded = encode\_file.read()  
 bits = "".join(format(byte, "08b") for byte in encoded)  
 # 4) читаем оставшиеся байты и декодируем  
 count = 0  
 temp\_str = ""  
 for char in bits:  
 if (count == length):  
 break  
 temp\_str = temp\_str + char  
 if (temp\_str in codes):  
 decode\_file.write(bytes([codes[temp\_str]]))  
 count += 1  
 temp\_str = ""  
  
 decode\_file\_size = os.path.getsize(output\_file)  
 print("Размер декодированного файла:", decode\_file\_size, "Байт(а)")  
 k = encode\_file\_size / decode\_file\_size if decode\_file\_size > 0 else 0  
 print("Коэффициент сжатия:", k)  
 print("Время декодировки", time.time() - start)  
 print("----------------------------------------------------------------------")  
  
if len(sys.argv) != 4:  
 print("Для запуска: python new\_Huffman.py encode/decode input\_file output\_file")  
 sys.exit(1)  
  
mode = sys.argv[1]  
input\_file = sys.argv[2]  
output\_file = sys.argv[3]  
  
if mode == "encode":  
 encode(input\_file, output\_file)  
elif mode == "decode":  
 decode(input\_file, output\_file)

Приложение 6. Листинг программы «files\_compare.py»

import os  
import sys  
  
# Для запуска: python files\_compare.py text.txt Huffman\_decode.txt/LZW\_decode.txt  
  
def compare(file1, file2):  
 f\_1 = open(file1, "r")  
 f\_2 = open(file2, "r")  
 str1 = f\_1.read()  
 str2 = f\_2.read()  
 for ch1, ch2 in zip(str1, str2):  
 if ch1 != ch2:  
 print("Файлы не идентичны!")  
 exit(1)  
 print("Файлы одинаковы")  
  
if len(sys.argv) != 3:  
 print("Для запуска: python files\_compare.py 1st\_file 2nd\_file")  
 print(sys.argv)  
 exit(1)  
  
file\_1 = sys.argv[1]  
file\_2 = sys.argv[2]  
  
compare(file\_1, file\_2)