Реализация самих алгоритмов, изменяющих текст, и их сравнение представлены в блокноте sjatie\_DZ3.ipynb. Итоговая реализация выполнена в качестве CLI-инструмента, ее демонстрация также представлена в кратком отчете.

**Задание 1-2. Для проверки используется простой пример.**

**BWT**

"#" символ конца текста, он будет замене на символ "\x00" (байтовое значение равно 0), который гарантированно не встречается, при текстовом выводе на месте данного символа будет пропуск

def bwt\_suffix\_arr(text: str) -> tuple[str, int]:

    # добавление специального символа конца текста # либо /x00

    if not text:

        return "", -1

    # if not text.endswith('#'):

    #     text += '#'

    text += '\x00'

    byte\_array = (text).encode('utf-8')

    n\_bytes = len(byte\_array)

    sa = pydivsufsort.divsufsort(np.frombuffer(byte\_array, dtype=np.uint8).copy())

    bwt\_bytes = np.empty(n\_bytes, dtype=np.uint8)

    original\_row\_index = -1

    for i in range(n\_bytes):

        suffix\_start\_index = sa[i]

        if suffix\_start\_index == 0:

            original\_row\_index = i

            bwt\_bytes[i] = byte\_array[n\_bytes - 1]

        else:

            bwt\_bytes[i] = byte\_array[suffix\_start\_index - 1]

    if original\_row\_index == -1:

        raise RuntimeError("Original index not found.")

    return bwt\_bytes.tobytes().decode('latin-1'), original\_row\_index

Обратное преобразование

На вход: сама строка BWT и номер строки исходного текста в M(T) (в лекции было Номер i нужной строки — это позиция # в bwt(T), т.е. позиция символа конца текста (оригинального))

def bwt\_decode(bwt\_string: str, original\_row\_index: int) -> str:

    # декодирование BWT, с учетом наличия спец символа в конце

    if not bwt\_string:

        return ""

    bwt\_array = np.frombuffer(bwt\_string.encode('latin-1'), dtype=np.uint8)

    n = len(bwt\_array)

    char\_counts = np.zeros(256, dtype=np.int32)

    for byte\_val in bwt\_array:

        char\_counts[byte\_val] += 1

    char\_start\_positions = np.zeros(256, dtype=np.int32)

    for i in range(1, 256):

        char\_start\_positions[i] = char\_start\_positions[i-1] + char\_counts[i-1]

    lf\_mapping = np.zeros(n, dtype=np.int32)

    for i in range(n):

        char = bwt\_array[i]

        lf\_mapping[i] = char\_start\_positions[char]

        char\_start\_positions[char] += 1

    decoded\_array = np.empty(n, dtype=np.uint8)

    current\_index = original\_row\_index

    for i in range(n - 1, -1, -1):

        decoded\_array[i] = bwt\_array[current\_index]

        current\_index = lf\_mapping[current\_index]

    decoded\_bytes = decoded\_array.tobytes()

    # Удаляем — символ конца строки

    hash\_byte\_value = ord('\x00')

    if decoded\_bytes[-1] != hash\_byte\_value:

        raise ValueError("Decoded string does not end with expected null byte.")

    return decoded\_bytes[:-1].decode('utf-8')

**MTF**

(MTF) читает входную строку слева направо и посимвольно меняет ее по следующим правилам:

\* Алфавит упорядочен, все символы адресуются своими номерами в списке

(первый элемент имеет номер 0)

\* Очередной символ a заменяется на номер a в текущем списке, после чего

a перемещается в начало списка (получает номер 0)

\* у каждого символа с номером меньшим номера a номер увеличивается на 1

Алфавит заранее согласуем, передавать его не надо (учитывая что это предназначено для кодирования текста состоящего из цифр и маленьких букв, можно будет урезать диапазон наверное или же вернуться к созданию оптимального алфавита, а потом передавать его декдеру)

def mtf\_encode(text: str) -> list[int]:

    # используется фиксированный, заранее определенный алфавит (все байты от 0 до 255)

    alphabet = [chr(i) for i in range(256)]

    encoded\_output = []

    for char in text:

        try:

            rank = alphabet.index(char)

        except ValueError:

            # символы не из диапазона

            raise ValueError(f"символ '{char}' не найден в фиксированном алфавите")

        encoded\_output.append(rank)

        # Перемещаем символ в начало списка

        char\_to\_move = alphabet.pop(rank)

        alphabet.insert(0, char\_to\_move)

    return encoded\_output

def mtf\_decode(encoded\_data: list[int]) -> str:

    # для декодирования нам нужен точно такой же начальный алфавит он заранее известен

    alphabet = sorted(list(set(chr(i) for i in range(256))))

    decoded\_chars = []

    for rank in encoded\_data:

        # символ по его номеру (индексу)

        char = alphabet[rank]

        decoded\_chars.append(char)

        # реремещаем этот символ в начало алфавита

        char\_to\_move = alphabet.pop(rank)

        alphabet.insert(0, char\_to\_move)

    return "".join(decoded\_chars)

**RLE и ZLE**

На книге проверим, RLE или ZLE даст результат лучше. (Мое предположение на данном этапе: после применения MTF похожие контексты рядом превратятся в нули, в последовательности будет много НЕдлинных серий нулей, остальные повторы будут встречаться реже, скорее ввсего ZLE справится лучше). В итоге был оставлен **ZLE**

def zle\_encode(data):

    result = []

    i = 0

    while i < len(data):

        if data[i] != 0:

            if data[i] in [254, 255]:

                result.extend([0, data[i]])  # Экранируем 254 и 255

            else:

                result.append(data[i])

            i += 1

        else:

            # Подсчет длины последовательности нулей

            zero\_count = 0

            while i < len(data) and data[i] == 0:

                zero\_count += 1

                i += 1

            # Кодируем число zero\_count + 1 в двоичном виде (без первой 1)

            binary = bin(zero\_count + 1)[3:]  # Пропускаем первую 1

            for b in binary:

                result.append(254 if b == '0' else 255)

    return result

def zle\_decode(data):

    result = []

    i = 0

    while i < len(data):

        if data[i] == 0:

            if i + 1 >= len(data):

                raise ValueError("Неправильное экранирование 254/255")

            result.append(data[i + 1])  # Восстанавливаем 254 или 255

            i += 2

        elif data[i] in [254, 255]:

            # Читаем последовательность битов

            bits = ""

            while i < len(data) and data[i] in [254, 255]:

                bits += '0' if data[i] == 254 else '1'

                i += 1

            n = int("1" + bits, 2) - 1  # Восстанавливаем длину нулей

            result.extend([0] \* n)

        else:

            result.append(data[i])

            i += 1

    return result

**ARI и HUF**

Сравним их. Их реализация была взята из ДЗ 2, но рамках этого ДЗ были написаны декодеры к ним. **Код обоих алгоритмов представлен в блокноте**. Были учтены ошибки из прошлого дз и теперь сравниваем не только размеры закодированных строк, но и размер данных, которые нуджны декодеру. **Результат будет представлен в следующем пункте**, т.к. сравнение на простом примере в этомм разделе имеет мало смысла, сравним их на книге.

**Задание 3. Проверка на книге**

Декапитализированный текст

with open('rousseau-confessions-lowercase.txt', 'r', encoding='utf-8') as file:

    lower\_text = file.read()

Размер исх текста: 1480.73 КБ

BWT (смотрим на разумность времени):

Время выполнения кодирования: 0.2840 секунд.

Время выполнения ДЕкодирования: 0.7430 секунд.

Декодированный текст совпадает с исходным: True

BWT + MTF: текст не изменился, как и ожидалось (это же не сжимающее преобразование)

RLE:

Получаем два списка значения *rle\_values* и частоты *rle\_freqs :*

Размер rle\_values: 668.16 КБ

Размер rle\_freqs: 668.16 КБ

Средняя длина повторов: 1.75

ZLE:

Размер zle\_encoded: 668.16 КБ

Таким образом после применения ZLE получаем последовательность zle\_encoded весом 668.16 КБ. Далее будем использоватть его.

ARI:

Однако для декодирования арифметического кода нужно будет передать алфавит, bit\_precision\_config

Размер сжатых данных: 367.3887 КБ

Размер алфавита: 0.08 КБ

Финальный вес текста после применения: BWT + MTF + ZLE + ARI = 367.3887 KB + 0.08 КБ = 367,4687 КБ

Хаффман. Кодируем последовательность после ZLE:

Размер сжатых данных: 371.90 КБ

Размер сжатых данных codebook\_freqs: 0.25 КБ

Финальный вес текста после применения: BWT + MTF + ZLE + HUF = 372.15 KB

Таким образом остановимся на BWT + MTF + ZLE + ARI.

Далее в блокноте представлена демонстрация сжатия-декомпрессии с использованием написанных алгоритмов. Проверено, что текст, прошедший через сжатие и декомпрессию совпадает с исходным.

В итоге после сжатия нужно будет хранить следующую информацию, чтобы ее успешно декодировать:

\* закодированный текст (после арифметического кодирования)

\* bit\_precision\_config = 32

\* alphabet (для арифметического декодирования)

\* eof\_pos (для BWT)

**Задание 3. СLI обертка**

Все реализованные алгоритмы раскидаем по файлам и будем вызывать их в файле – архиваторе. Будем использовать утилиту argparse для чтения аргументов из командной строки.

Сам файл архиватора (он лежит в папке sjatie\_DZ3 с файлами, где реализованы алгоритмы):

import argparse

import json

import os

import struct  # Для упаковки длины заголовка в байты

from bwt import bwt\_suffix\_arr, bwt\_decode

from mtf import mtf\_encode, mtf\_decode

from zle import zle\_encode, zle\_decode

from arithmetic import din\_arithmetic\_compression, decompress

#from tqdm import tqdm

# константа для точности ставим 32 бита, хотя можно было бы передавать ее в качестве параметра

BIT\_PRECISION\_CONFIG = 32

# заголовок в 8 байт (unsigned long long)

HEADER\_LENGTH\_BYTES = 8

def bits\_to\_bytes(bit\_string: str) -> tuple[bytes, int]:

    padding\_needed = (8 - len(bit\_string) % 8) % 8

    padded\_bit\_string = bit\_string + '0' \* padding\_needed

    byte\_array = bytearray()

    for i in range(0, len(padded\_bit\_string), 8):

        byte\_chunk = padded\_bit\_string[i:i+8]

        byte\_array.append(int(byte\_chunk, 2))

    return bytes(byte\_array), padding\_needed

def bytes\_to\_bits(byte\_data: bytes, padding\_bits: int) -> str:

    bit\_string = ''.join(format(byte, '08b') for byte in byte\_data)

    if padding\_bits > 0:

        return bit\_string[:-padding\_bits]

    return bit\_string

def compress\_file(input\_path, output\_path):

    print(f"[\*] Сжатие файла: {input\_path}")

    try:

        with open(input\_path, 'r', encoding='utf-8') as f:

            text = f.read()

    except FileNotFoundError:

        print(f"[!] Ошибка: Файл не найден {input\_path}")

        return

    original\_size = os.path.getsize(input\_path)

    print(f"[\*] Размер исходного файла: {original\_size} байт")

    if not text:

        print("[!] Входной файл пуст")

        with open(output\_path, 'wb') as f:

            pass

        return

    # сжатие

    print("[\*] BWT...")

    bwt\_result, eof\_pos = bwt\_suffix\_arr(text)

    print("[\*] MTF...")

    mtf\_result = mtf\_encode(bwt\_result)

    print("[\*] ZLE...")

    zle\_encoded = zle\_encode(mtf\_result)

    print("[\*] ARI...может занять немного времени (минут)")

    zle\_encoded\_ari\_bit\_string = din\_arithmetic\_compression(zle\_encoded, BIT\_PRECISION\_CONFIG)

    packed\_data, padding\_bits = bits\_to\_bytes(zle\_encoded\_ari\_bit\_string)

    # алфавит

    alphabet = list(dict.fromkeys(zle\_encoded))

    header\_data = {

        "eof\_pos": eof\_pos,

        "alphabet": alphabet,

        "original\_len": len(zle\_encoded), # L

        "bit\_precision": BIT\_PRECISION\_CONFIG,

        "padding\_bits": padding\_bits

    }

    # ням

    header\_json = json.dumps(header\_data)

    header\_bytes = header\_json.encode('utf-8')

    # выходной файл

    with open(output\_path, 'wb') as f:

        # длина заголовка (8 байт)

        f.write(struct.pack('!Q', len(header\_bytes)))

        # заголовок

        f.write(header\_bytes)

        # Записываем сжатые данные

        f.write(packed\_data)

    compressed\_size = os.path.getsize(output\_path)

    compression\_ratio = original\_size / compressed\_size if compressed\_size > 0 else 0

    print(f"[+] Сжатие завершено")

    print(f"[+] Выходной файл: {output\_path}")

    print(f"[+] Размер сжатого файла: {compressed\_size} байт")

    print(f"[+] Коэффициент сжатия: {compression\_ratio:.2f}x")

def decompress\_file(input\_path, output\_path):

    print(f"[\*] Распаковка файла: {input\_path}")

    try:

        with open(input\_path, 'rb') as f:

            # длина заголовка

            packed\_header\_len = f.read(HEADER\_LENGTH\_BYTES)

            if not packed\_header\_len:

                print("[!] Ошибка: Архив поврежден.")

                return

            header\_len = struct.unpack('!Q', packed\_header\_len)[0]

            # ням десериализация

            header\_bytes = f.read(header\_len)

            header\_data = json.loads(header\_bytes.decode('utf-8'))

            # метаданные

            eof\_pos = header\_data["eof\_pos"]

            alphabet = header\_data["alphabet"]

            original\_len = header\_data["original\_len"] # Наша L

            bit\_precision = header\_data["bit\_precision"]

            padding\_bits = header\_data["padding\_bits"]

            # данные

            compressed\_data = f.read()

            bit\_string\_to\_decompress = bytes\_to\_bits(compressed\_data, padding\_bits)

    except FileNotFoundError:

        print(f"[!] Ошибка: Файл не найден по пути {input\_path}")

        return

    except (struct.error, json.JSONDecodeError):

        print(f"[!] Ошибка: Формат архива некорректен или файл поврежден")

        return

    # декодирование

    print("[\*] ARI DEC...может занять немного времени (минут)")

    zle\_decoded = decompress(bit\_string\_to\_decompress, bit\_precision, alphabet, original\_len)

    print("[\*] ZLE DEC...")

    mtf\_decoded = zle\_decode(zle\_decoded)

    print("[\*] MTF DEC...")

    bwt\_decoded = mtf\_decode(mtf\_decoded)

    print("[\*] BWT DEC...")

    decoded\_text = bwt\_decode(bwt\_decoded, eof\_pos)

    # запись результата в файл

    with open(output\_path, 'w', encoding='utf-8') as f:

        f.write(decoded\_text)

    print(f"[+] Распаковка завершена")

    print(f"[+] Исходный текст сохранен в: {output\_path}")

def main():

    parser = argparse.ArgumentParser(description="Простой BWT-архиватор на Python.")

    subparsers = parser.add\_subparsers(dest="command", required=True, help="Доступные команды")

    # сжатия

    parser\_compress = subparsers.add\_parser("compress", help="Сжать файл.")

    parser\_compress.add\_argument("input", type=str, help="Путь к исходному файлу.")

    parser\_compress.add\_argument("output", type=str, help="Путь к сжатому файлу.")

    # распаковка

    parser\_decompress = subparsers.add\_parser("decompress", help="Распаковать файл.")

    parser\_decompress.add\_argument("input", type=str, help="Путь к сжатому файлу.")

    parser\_decompress.add\_argument("output", type=str, help="Путь к распакованному файлу.")

    args = parser.parse\_args()

    if args.command == "compress":

        compress\_file(args.input, args.output)

    elif args.command == "decompress":

        decompress\_file(args.input, args.output)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

В него добавлено 2 функции, которые позволяют корректно обработать результат арифметического кодирования (выход алгоритма это строка, нам нужно корректно преобразовать ее в байты, чтобы не допустить ситуацию когда каждый элемент строки (он же 0 или 1, бит) будет занимать целый байт после записи.

Запуск:

1. В папке с файлом archiver.py и другими, которые он использует, лежит файл с текстом. Чтобы сжать файл, выполним команду

python archiver.py compress rousseau-confessions-lowercase.txt rousseau.compressed

где rousseau-confessions-lowercase.txt – исходный текст

rousseau.compressed – сжатый файл

compress – функция сжатия

(можно не класть файл в папку с кодом, просто указать путь до него)

1. Распаковка

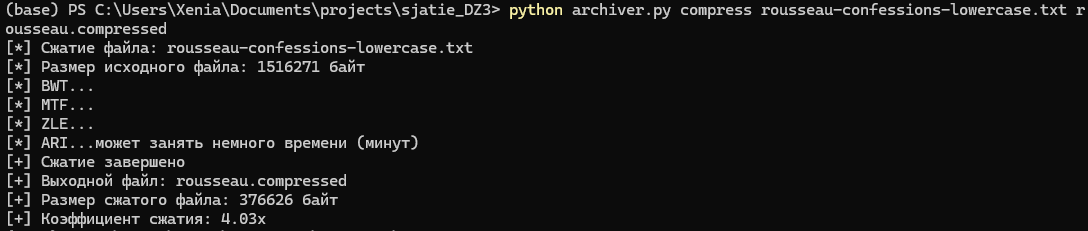
python archiver.py decompress rousseau.compressed decoded\_rousseau.txt

rousseau.compressed – сжатый файл

decoded\_rousseau.txt – куда хотим записать результат

decompress – команда разархивирования

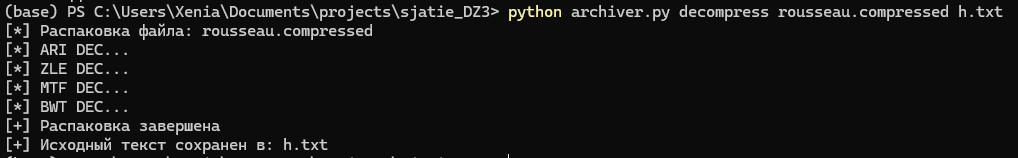
Демонстрация



Вот что получили



Разархивируем



Вот что получили





Разархивированный файл совпадает с исходным

**Зависимости:**

import argparse

import json

import os

import struct

import pydivsufsort

import numpy as np

from math import ceil

from os.path import commonprefix

from collections import Counter

import math

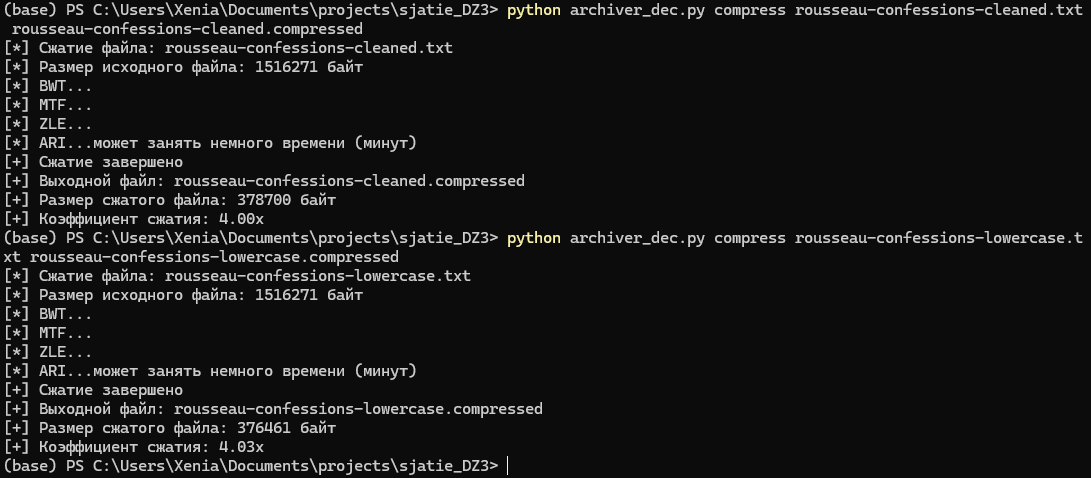
Версия python 3.12.7

Добавим в файл с зависимостями то, что не входит в стандартные пакеты, чтобы можно было сделать   
*pip install -r requirements.txt*

**Задание 3. Сравнение с исходным текстом**

Сначала проверим как архиватор работает с исходным текстом (где есть символы разного регистра)

Сравним размер сжатого исходного текста и сжатого декапитализироованного текста rousseau-confessions-cleaned.txt

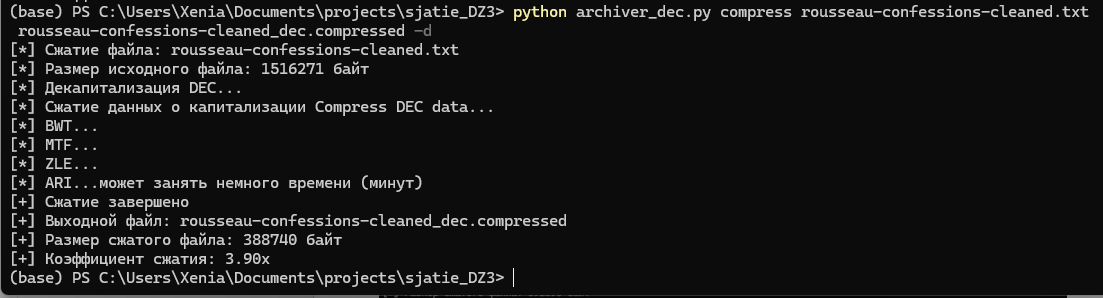


Можно задуматься о том, чтобы при сжатии текста сначала декапитализировать его, а потом сжимать и посмотреть имеет ли это смысл. Может получиться так, что информация необходимая для капитализации, которую нужно также передавать и сжимать, будет иметь размер больше, чем выигрыш от сжатия текста в нижнем регистре. Также заменим JSON на бинарный формат, чтобы немного уменьшить оверхед.

**Архиватор был переписан и сохранен в файл archiver\_dec.py**

**Когда хотим сначала декапитализировать текст, а потом сжимать, добавляем флаг -d**

python archiver\_dec.py compress rousseau-confessions-cleaned.txt rousseau\_dec.compressed -d



Сжатие без декапитализации (файл с разным регистром): 378 700 байт

Сжатие с дкапитализацией (флагом -d): 388 740 байт

Сжатие lowercase-файла (rousseau-confessions-lowercase.txt): 376 461 байт

Размер несжатого файла: 1 516 271 байт

**Результат на большом датасете (фрагменте английской википедии)**

Данные отсюда <https://mattmahoney.net/dc/textdata.html>

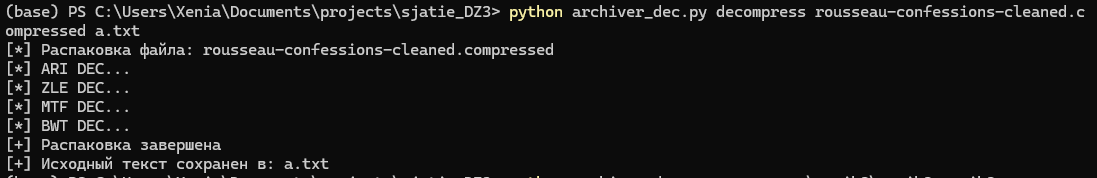
**Enwik8**

Ставим процессу высокий приоритет, ждем минут 50 (+-5 минут). Архивируем



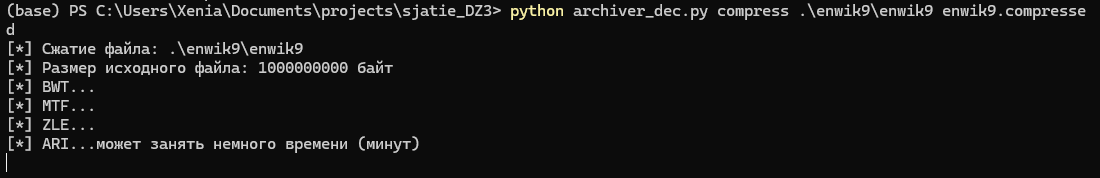


Теперь распаковываем



Все успешно распаковалось

*Попробуем сжать 1000 мб*



К сожалению это заняло достаточно много времени на этапе арифметического кодирования.

**Использование библиотечного ARI**

Для ускорения работы архиватора будем использовать библиотечную реализацию. Будем использовать реализацию <https://github.com/nayuki/Reference-arithmetic-coding/blob/master/python/arithmeticcoding.py>

Зменим модуль arithmetic.py, чтобы архиватор мог использовать арифметическое кодирование из arithmeticcoding.

import io

import arithmeticcoding

# библиотека работает с потоками байтов

def bits\_to\_bytes(bit\_string: str) -> bytes:

    """Преобразует строку битов в байты."""

    # Дополняем строку нулями до длины, кратной 8

    padding\_needed = (8 - len(bit\_string) % 8) % 8

    padded\_bit\_string = bit\_string + '0' \* padding\_needed

    # Создаем байтовый массив

    byte\_array = bytearray()

    for i in range(0, len(padded\_bit\_string), 8):

        byte\_chunk = padded\_bit\_string[i:i+8]

        byte\_array.append(int(byte\_chunk, 2))

    return bytes(byte\_array)

def bytes\_to\_bits(byte\_data: bytes) -> str:

    return ''.join(format(byte, '08b') for byte in byte\_data)

def din\_arithmetic\_compression(source\_message: list, bit\_precision\_config: int) -> str:

    max\_symbol = max(source\_message) if source\_message else -1

    if max\_symbol < 0:

        return ""

    init\_freqs = arithmeticcoding.FlatFrequencyTable(max\_symbol + 1)

    freqs = arithmeticcoding.SimpleFrequencyTable(init\_freqs)

    buffer = io.BytesIO()

    bitout = arithmeticcoding.BitOutputStream(buffer)

    enc = arithmeticcoding.ArithmeticEncoder(bit\_precision\_config, bitout)

    for symbol in source\_message:

        enc.write(freqs, symbol)

        freqs.increment(symbol)

    enc.finish()

    while bitout.numbitsfilled != 0:

        bitout.write(0)

    compressed\_bytes = buffer.getvalue()

    buffer.close()

    return bytes\_to\_bits(compressed\_bytes)

def decompress(encoded\_sequence: str, N: int, ordered\_alphabet: list, message\_len: int) -> list:

    if message\_len == 0:

        return []

    compressed\_bytes = bits\_to\_bytes(encoded\_sequence)

    alphabet\_size = len(ordered\_alphabet)

    init\_freqs = arithmeticcoding.FlatFrequencyTable(alphabet\_size)

    freqs = arithmeticcoding.SimpleFrequencyTable(init\_freqs)

    decoded\_message = [] # символы из плотного алфавита

    buffer = io.BytesIO(compressed\_bytes)

    bitin = arithmeticcoding.BitInputStream(buffer)

    dec = arithmeticcoding.ArithmeticDecoder(N, bitin)

    for \_ in range(message\_len):

        symbol = dec.read(freqs)

        decoded\_message.append(symbol)

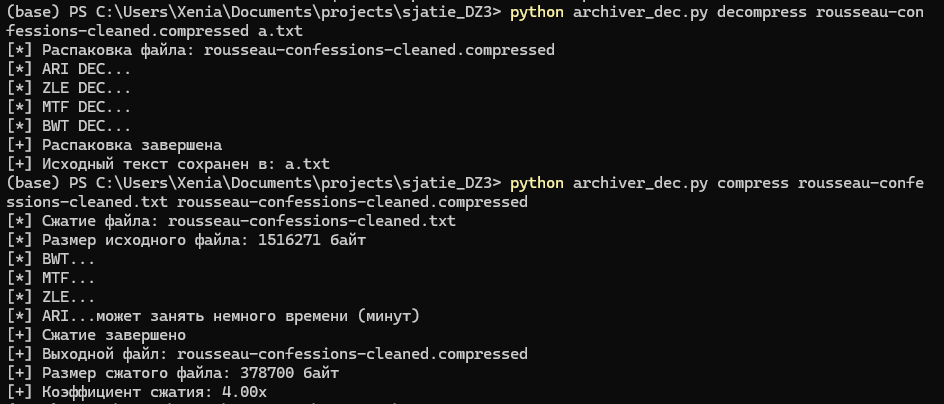
        freqs.increment(symbol)

    bitin.close()

    return decoded\_message

Проверим работу архиватора на книге:

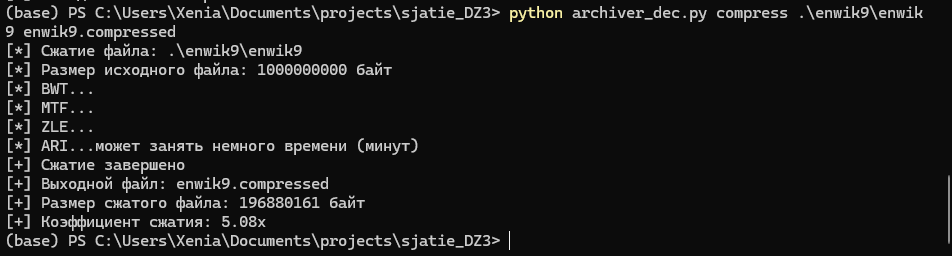




Работает корректно, коэффициент сжатия не изменился

**Теперь проверим на Wiki**

**Попробуем сжать 1000 мб (enwik9)**

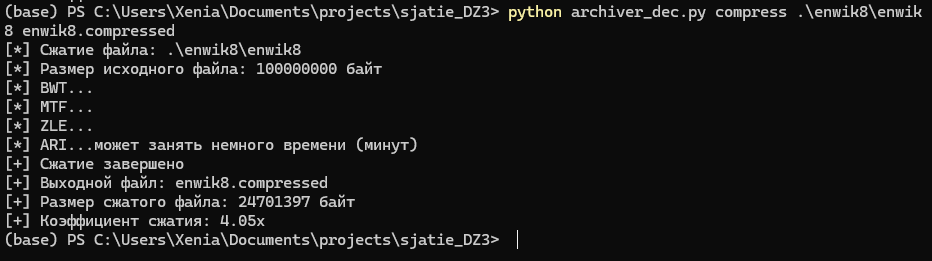


* Сжатие заняло около 1.5 часа (+- 15 минут)

Размер сжатого файла: 192 265,78222 Кб

**Также сожмем 100 мб (enwik8)**

Сжатие заняло около 12-15 минут минут



Проверим декомпрессию. Выполняется примерно за то же время:

