

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 заменен 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)
        # перемещаем 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 всего 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

    # Кодировем 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 нужны декодеру. **Результат будет представлен в следующем пункте,** т.к. сравнение на простом примере в этом 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 использовать 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: $BWT + MTF + ZLE + ARI = 367.3887 \text{ KB} + 0.08 \text{ KB} = 367,4687 \text{ KB}$

Хаффман. Кодировем последовательность после ZLE:

Размер сжатых данных: 371.90 КБ

Размер сжатых данных *codebook_freqs*: 0.25 КБ

Финальный вес текста после применения: $BWT + MTF + ZLE + HUF = 372.15 \text{ KB}$

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. CLI 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)

2. 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

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver.py compress rousseau-confessions-lowercase.txt r
rousseau.compressed
[*] Сжатие файла: rousseau-confessions-lowercase.txt
[*] Размер исходного файла: 1516271 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: rousseau.compressed
[+] Размер сжатого файла: 376626 байт
[+] Коэффициент сжатия: 4.03x
```



Вот что получили

 rousseau.compressed	17.06.2025 16:28	Файл "COMPRESS..."	368 КБ
 rousseau-confessions-lowercase.txt	03.06.2025 0:02	Текстовый докум...	1 481 КБ

Разархивируем

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver.py decompress rousseau.compressed h.txt
[*] Распаковка файла: rousseau.compressed
[*] ARI DEC...
[*] ZLE DEC...
[*] MTF DEC...
[*] BWT DEC...
[+] Распаковка завершена
[+] Исходный текст сохранен в: h.txt
```

Вот что получили

 h.txt	17.06.2025 16:31	Текстовый докум...	1 481 КБ
 rousseau.compressed	17.06.2025 16:28	Файл "COMPRESS..."	368 КБ

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 декапитализированного текста **rousseau-confessions-cleaned.txt**

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress rousseau-confessions-cleaned.txt
rousseau-confessions-cleaned.compressed
[*] Сжатие файла: rousseau-confessions-cleaned.txt
[*] Размер исходного файла: 1516271 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: rousseau-confessions-cleaned.compressed
[+] Размер сжатого файла: 378700 байт
[+] Коэффициент сжатия: 4.00x
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress rousseau-confessions-lowercase.t
xt rousseau-confessions-lowercase.compressed
[*] Сжатие файла: rousseau-confessions-lowercase.txt
[*] Размер исходного файла: 1516271 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: rousseau-confessions-lowercase.compressed
[+] Размер сжатого файла: 376461 байт
[+] Коэффициент сжатия: 4.03x
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> |
```

Можно задуматься о том, чтобы при сжатии текста сначала декапитализировать его, а потом сжимать и посмотреть имеет ли это смысл. Может получится 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
```

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress rousseau-confessions-cleaned.txt
rousseau-confessions-cleaned_dec.compressed -d
[*] Сжатие файла: rousseau-confessions-cleaned.txt
[*] Размер исходного файла: 1516271 байт
[*] Декапитализация DEC...
[*] Сжатие данных о капитализации Compress DEC data...
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: rousseau-confessions-cleaned_dec.compressed
[+] Размер сжатого файла: 388740 байт
[+] Коэффициент сжатия: 3.90x
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> |
```

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

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress .\enwik8\enwik8 enwik8.compressed
[*] Сжатие файла: .\enwik8\enwik8
[*] Размер исходного файла: 100000000 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: enwik8.compressed
[+] Размер сжатого файла: 24701397 байт
[+] Коэффициент сжатия: 4.05x
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> |
```

 enwik8.compressed	19.06.2025 23:48	Файл "COMPRESS..."	24 123 КБ
---	------------------	--------------------	-----------

Теперь распаковываем

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py decompress rousseau-confessions-cleaned.compressed a.txt
[*] Распаковка файла: rousseau-confessions-cleaned.compressed
[*] ARI DEC...
[*] ZLE DEC...
[*] MTF DEC...
[*] BWT DEC...
[+] Распаковка завершена
[+] Исходный текст сохранен в: a.txt
```

Все успешно распаковалось

Попробуем сжать 1000 мб

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress .\enwik9\enwik9 enwik9.compressed
[*] Сжатие файла: .\enwik9\enwik9
[*] Размер исходного файла: 1000000000 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
```

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:

	rousseau-confessions-cleaned.compressed...	21.06.2025 19:09	Файл "COMPRESS..."	370 КБ
	rousseau-confessions-cleaned.txt	04.06.2025 23:09	Текстовый докум...	1 481 КБ

```

(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py decompress rousseau-con
fessions-cleaned.compressed a.txt
[*] Распаковка файла: rousseau-confessions-cleaned.compressed
[*] ARI DEC...
[*] ZLE DEC...
[*] MTF DEC...
[*] BWT DEC...
[+] Распаковка завершена
[+] Исходный текст сохранен в: a.txt
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress rousseau-confe
ssions-cleaned.txt rousseau-confessions-cleaned.compressed
[*] Сжатие файла: rousseau-confessions-cleaned.txt
[*] Размер исходного файла: 1516271 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: rousseau-confessions-cleaned.compressed
[+] Размер сжатого файла: 378700 байт
[+] Коэффициент сжатия: 4.00x

```

Работает корректно, коэффициент сжатия не изменился

Теперь проверим на Wiki

Попробуем сжать 1000 мб (enwik9)

```

(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress .\enwik9\enwik
9 enwik9.compressed
[*] Сжатие файла: .\enwik9\enwik9
[*] Размер исходного файла: 1000000000 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: enwik9.compressed
[+] Размер сжатого файла: 196880161 байт
[+] Коэффициент сжатия: 5.08x
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> |

```

- 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

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py compress .\enwik8\enwik8 enwik8.compressed
[*] Сжатие файла: .\enwik8\enwik8
[*] Размер исходного файла: 100000000 байт
[*] BWT...
[*] MTF...
[*] ZLE...
[*] ARI...может занять немного времени (минут)
[+] Сжатие завершено
[+] Выходной файл: enwik8.compressed
[+] Размер сжатого файла: 24701397 байт
[+] Коэффициент сжатия: 4.05x
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> |
```

Проверим декомпрессию. Выполняется примерно за то же время:

```
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> python archiver_dec.py decompress enwik8.compressed enwik8.txt
[*] Распаковка файла: enwik8.compressed
[*] ARI DEC...
[*] ZLE DEC...
[*] MTF DEC...
[*] BWT DEC...
[+] Распаковка завершена
[+] Исходный текст сохранен в: enwik8.txt
(base) PS C:\Users\Xenia\Documents\projects\sjatie_DZ3> |
```