**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ЛАБОРАТОРНАЯ РАБОТА**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Алгоритмы сжатия**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3351 |  | Игнатков И.Д. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

Реализовать алгоритмы: HA, BWT, MTF, RLE, LZ77, LZ78. При желании заменить LZ77 на LZSS, L78 на LZW.

Исследовать зависимость энтропии от размера блоков, на которые разбивается текст, подаваемый на вход BWT+MTF, для enwik7. Вывести полученные результаты на график и сделать вывод об оптимальном размере блока.

Исследовать зависимость коэффициента сжатия от размера буфера для алгоритма LZ77 (LZSS). Вывести полученные результаты на график и сделать вывод об оптимальном размере буфера.

Собрать на основе реализованных алгоритмов следующие компрессоры:

1. HA

2. Run-length encoding (RLE)

3. BWT + RLE

4. BWT + MTF + HA

5. BWT + MTF + RLE + HA

6. LZ77

7. LZ77 + HA

8. LZ78

9. LZ78 + HA

Удостовериться в корректной компрессии и декомпрессии данных.

Исследовать эффективность компрессоров для всех тестовых данных.

Свести результаты в виде коэффициента сжатия для каждого компрессора и всех тестовых данных в таблицы. В таблице указать размер до компрессии, после компрессии и после декомпрессии в байтах, коэффициент сжатия. Коэффициент сжатия округлить до трех цифр после запятой.

**ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

Описание алгоритмов:

RLE — это алгоритм, который сжимает данные, заменяя повторяющиеся символы на пару: количество повторений и сам символ. Например, вместо "AAAAААА" он запишет "7A".

BWT — это алгоритм, который переставляет исходные данные так, чтобы одинаковые символы оказались рядом друг с другом. В результате, после такой перестановки, повторяющиеся символы группируются вместе, что помогает лучше сжимать данные.

HA — это такой алгоритм, который использует таблицу частотности символов, чтобы каждому байту присвоить уникальный код. Он строит дерево, где часто встречающиеся символы получают более короткие коды, а редкие — длинные. В итоге, данные сжимаются, так как частые символы занимают меньше места.

MTF – это агоритм, который меняет каждый символ на его текущий номер в списке. После того как символ обработан, он перемещается в начало этого списка. Таким образом, часто встречающиеся символы оказываются ближе к началу, что помогает улучшить сжатие данных.

LZ77 — это алгоритм, который использует скользящее окно, разделенное на две части: уже прочитанные данные и еще не прочитанные. На каждом шаге он проверяет, есть ли текущий байт или последовательность байт в уже прочитанных данных. Если находит совпадение, то в сжатую строку добавляется ссылка на это совпадение и следующий байт. Ссылка состоит из трех частей: отступ (сколько байт назад найдено совпадение), длина (сколько байт подряд совпадают) и следующий байт в буфере.

LZ78 — это алгоритм, который использует словарь для хранения встречающихся байтов или последовательностей. Если байт или последовательность уже есть в словаре, они заменяются на ссылку, состоящую из индекса в словаре и нового символа. Таким образом, часто повторяющиеся данные занимают меньше места.

**ТАБЛИЦА СЛОЖНОСТЕЙ АЛГОРИТМОВ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм | Временная сложность кодирования | Временная сложность декодирования | Пространственная сложность кодирования | Пространственная сложность декодирования |
| HA |  |  |  |  |
| BWT |  |  |  |  |
| MTF |  |  |  |  |
| RLE |  |  |  |  |
| LZ77 |  |  |  |  |
| LZ78 |  |  |  |  |

Где:

w – размер окна буфера для LZ77

l – размер самого буфера для LZ77

d – размер словаря для LZ78

**ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ**

Таблица информации об эффективности компрессоров, основанных на алгоритмах сжатия (см. Рис 1–3).

Изображение выглядит как текст, число, снимок экрана, меню

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1 – Первая часть таблица

Изображение выглядит как текст, число, меню

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2 – Вторая часть таблицы

Изображение выглядит как текст, снимок экрана, число, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 3 – Третья часть таблицы

Требовалось найти значения зависимости энтропии от размера блока для алгоритма сжатия BWT+MTF, для файла enwik7.txt и построить график (см. Рис 4).

Изображение выглядит как линия, График, диаграмма, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 4 – график зависимости энтропии от размера блока для алгоритма BWT+MTF

При увеличении размера блока энтропия снижается, но не так сильно, как растет время и потребление памяти. Оптимальный размер блока 1024 байта.

Требовалось найти значения коэффициента сжатия от размера буфера для алгоритма сжатия LZ77, для файла enwik7.txt и построить график (см. Рис 5).

**Изображение выглядит как линия, График, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.**

Рисунок 5 – График зависимости коэффициента сжатия от размера буфера для алгоритма LZ77

Для каждого файла коэффициент сжатия растет с увеличением размера буфера. Однако, большой буфер увеличивает время сжатия и требует больше памяти.

**КОД АЛГОРИТМОВ СЖАТИЯ**

Ссылка на репозиторий в Github:

<https://github.com/IgorIgnatkov/ALG_lab1_sem4>

Excel: для хранения данных

Python: для написания алгоритмов

main.py:

from compressors.compress\_BWT\_MTF import compressor\_bwt\_mtf

from compressors.compress\_HA import compressor\_ha

from compressors.compressor\_BWT import compressor\_bwt

from compressors.compressor\_BWT\_MTF\_HA import compressor\_bwt\_mtf\_ha

from compressors.compressor\_BWT\_MTF\_RLE\_HA import compressor\_bwt\_mtf\_rle\_ha

from compressors.compressor\_BWT\_RLE import compressor\_bwt\_rle

from compressors.compressor\_LZ77 import compressor\_lz77

from compressors.compressor\_LZ77\_HA import compressor\_lz77\_ha

from compressors.compressor\_LZ78 import compressor\_lz78

from compressors.compressor\_LZ78\_HA import compressor\_lz78\_ha

from compressors.compressor\_MTF import compressor\_mtf

from compressors.compressor\_RLE import compressor\_rle

def main():

input\_files = [

r"common/original/enwik7.txt",

r"common/original/Отцы\_и\_дети.txt",

r"common/original/grey\_image.raw",

r"common/original/color\_image.raw",

r"common/original/black\_white\_image.raw",

r"common/original/file.exe",

]

compressor\_ha(input\_files[0], r"common/compressed/HA\_compressed.txt", r"common/decompressed/HA\_compressed.txt", False)

compressor\_bwt(input\_files[0], r"common/compressed/BWT\_compressed.txt", r"common/decompressed/BWT\_compressed.txt", False)

compressor\_rle(input\_files[0], r"common/compressed/RLE\_compressed.txt", r"common/decompressed/RLE\_compressed.txt", False)

compressor\_lz77(input\_files[0], r"common/compressed/LZ77\_compressed.txt", r"common/decompressed/LZ77\_compressed.txt", False)

compressor\_lz78(input\_files[0], r"common/compressed/LZ78\_compressed.txt", r"common/decompressed/LZ78\_compressed.txt", False)

compressor\_mtf(input\_files[0], r"common/compressed/MTF\_compressed.txt",r"common/decompressed/MTF\_compressed.txt", False)

compressor\_bwt\_rle(input\_files[0], r"common/compressed/BWT\_RLE\_compressed.txt", r"common/decompressed/BWT\_RLE\_compressed.txt", True)

compressor\_bwt\_mtf\_ha(input\_files[0], r"common/compressed/BWT\_MTF\_HA\_compressed.txt", r"common/decompressed/BWT\_MTF\_HA\_compressed.txt", True)

compressor\_bwt\_mtf\_rle\_ha(input\_files[0], r"common/compressed/BWT\_MTF\_RLE\_HA\_compressed.txt", r"common/decompressed/BWT\_MTF\_RLE\_HA\_compressed.txt", True)

compressor\_lz77\_ha(input\_files[0], r"common/compressed/LZ77\_HA\_compressed.txt",r"common/decompressed/LZ77\_HA\_compressed.txt", True)

compressor\_lz78\_ha(input\_files[0], r"common/compressed/LZ78\_HA\_compressed.txt",r"common/decompressed/LZ78\_HA\_compressed.txt", True)

compressor\_bwt\_mtf(input\_files[0], r"common/compressed/BWT\_MTF\_compressed.txt",r"common/decompressed/BWT\_MTF\_compressed.txt", True)

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

main()

base\_algorithms/RLE.py:

def rle\_compress(data: bytes) -> bytes:

compressed\_output = []

repeat\_count = 1

for current\_index in range(1, len(data)):

if data[current\_index] == data[current\_index - 1] and repeat\_count < 127:

repeat\_count += 1

else:

while repeat\_count > 127:

compressed\_output.append(0x80 | 127)

compressed\_output.append(data[current\_index - 1])

repeat\_count -= 127

compressed\_output.append(0x80 | repeat\_count)

compressed\_output.append(data[current\_index - 1])

repeat\_count = 1

while repeat\_count > 127:

compressed\_output.append(0x80 | 127)

compressed\_output.append(data[-1])

repeat\_count -= 127

compressed\_output.append(0x80 | repeat\_count)

compressed\_output.append(data[-1])

return bytes(compressed\_output)

def rle\_decompress(encoded\_data: bytes) -> bytes:

decompressed\_output = bytearray()

current\_position = 0

data\_length = len(encoded\_data)

while current\_position < data\_length:

count = encoded\_data[current\_position]

current\_position += 1

if count & 0x80:

count &= 0x7F

byte\_value = encoded\_data[current\_position]

current\_position += 1

decompressed\_output.extend([byte\_value] \* count)

else:

decompressed\_output.append(encoded\_data[current\_position])

current\_position += 1

return bytes(decompressed\_output)

base\_algorithms/BWT.py:

import struct

CHUNK\_SIZE = 1000

END\_OF\_BLOCK\_MARKER = b'\x01'

def create\_suffix\_array(data: bytes):

return sorted(

range(len(data)),

key=lambda index: data[index:]

)

def bwt\_compress(input\_data: bytes) -> bytes:

input\_data += END\_OF\_BLOCK\_MARKER

suffix\_array = create\_suffix\_array(input\_data)

original\_index = suffix\_array.index(0)

bwt\_transformed = bytearray(input\_data[i - 1] if i > 0 else input\_data[-1] for i in suffix\_array)

compressed\_data = struct.pack('I', original\_index) + bwt\_transformed

return compressed\_data

def bwt\_decompress(compressed\_data: bytes) -> bytes:

if len(compressed\_data) < 5:

raise ValueError("Некорректные данные: слишком короткий блок")

original\_index = struct.unpack('I', compressed\_data[:4])[0]

bwt\_transformed = compressed\_data[4:]

data\_length = len(bwt\_transformed)

if original\_index >= data\_length:

raise ValueError(f"Ошибка индекса: original\_index {original\_index} >= {data\_length}")

byte\_count = [0] \* 256

for byte in bwt\_transformed:

byte\_count[byte] += 1

total = 0

first\_column\_positions = [0] \* 256

for i in range(256):

first\_column\_positions[i] = total

total += byte\_count[i]

rank = [0] \* data\_length

seen = [0] \* 256

for i in range(data\_length):

rank[i] = first\_column\_positions[bwt\_transformed[i]] + seen[bwt\_transformed[i]]

seen[bwt\_transformed[i]] += 1

original\_data = bytearray(data\_length)

current\_index = original\_index

for i in range(data\_length - 1, -1, -1):

if current\_index >= data\_length:

raise ValueError(f"Ошибка ранга: current\_index {current\_index} >= {data\_length} (i={i})")

original\_data[i] = bwt\_transformed[current\_index]

current\_index = rank[current\_index]

decompressed\_data = bytes(original\_data).rstrip(END\_OF\_BLOCK\_MARKER)

return decompressed\_data

base\_algorithms/HA.py:

from collections import defaultdict

import heapq

import struct

class TreeNode:

def \_\_init\_\_(self, symbol=None, frequency=None, left\_child=None, right\_child=None):

self.symbol = symbol

self.frequency = frequency

self.left\_child = left\_child

self.right\_child = right\_child

def \_\_lt\_\_(self, other):

return self.frequency < other.frequency

def build\_huffman\_tree(data: bytes):

frequency\_table = defaultdict(int)

for byte in data:

frequency\_table[byte] += 1

priority\_queue = []

for byte, freq in frequency\_table.items():

node = TreeNode(symbol=byte, frequency=freq)

heapq.heappush(priority\_queue, node)

while len(priority\_queue) > 1:

left\_node = heapq.heappop(priority\_queue)

right\_node = heapq.heappop(priority\_queue)

merged\_node = TreeNode(frequency=left\_node.frequency + right\_node.frequency, left\_child=left\_node, right\_child=right\_node)

heapq.heappush(priority\_queue, merged\_node)

return heapq.heappop(priority\_queue)

def generate\_huffman\_codes(node, current\_code="", code\_table=None):

if code\_table is None:

code\_table = {}

if node is not None:

if node.symbol is not None:

code\_table[node.symbol] = current\_code

generate\_huffman\_codes(node.left\_child, current\_code + "0", code\_table)

generate\_huffman\_codes(node.right\_child, current\_code + "1", code\_table)

return code\_table

def serialize\_tree(node):

if node is None:

return b""

if node.symbol is not None:

return b"\x01" + bytes([node.symbol])

return b"\x00" + serialize\_tree(node.left\_child) + serialize\_tree(node.right\_child)

def deserialize\_tree(data):

def deserialize\_helper():

nonlocal index

if index >= len(data):

return None

flag = data[index]

index += 1

if flag == 1:

symbol = data[index]

index += 1

return TreeNode(symbol=symbol)

elif flag == 0:

left\_child = deserialize\_helper()

right\_child = deserialize\_helper()

return TreeNode(left\_child=left\_child, right\_child=right\_child)

index = 0

return deserialize\_helper()

def ha\_compress(input\_data: bytes) -> bytes:

if not input\_data:

return b""

tree\_root = build\_huffman\_tree(input\_data)

code\_table = generate\_huffman\_codes(tree\_root)

encoded\_bits = "".join([code\_table[byte] for byte in input\_data])

padding\_amount = 8 - len(encoded\_bits) % 8

encoded\_bits += "0" \* padding\_amount

encoded\_bytes = bytearray()

for i in range(0, len(encoded\_bits), 8):

byte = encoded\_bits[i:i+8]

encoded\_bytes.append(int(byte, 2))

tree\_bytes = serialize\_tree(tree\_root)

compressed\_data = (bytes([padding\_amount]) + struct.pack(">I", len(tree\_bytes)) + tree\_bytes + encoded\_bytes)

return compressed\_data

def ha\_decompress(compressed\_data: bytes) -> bytes:

if not compressed\_data:

return b""

padding\_amount = compressed\_data[0]

tree\_size = struct.unpack(">I", compressed\_data[1:5])[0]

tree\_bytes = compressed\_data[5:5 + tree\_size]

tree\_root = deserialize\_tree(tree\_bytes)

encoded\_bytes = compressed\_data[5 + tree\_size:]

encoded\_bits = "".join(f"{byte:08b}" for byte in encoded\_bytes)

encoded\_bits = encoded\_bits[:-padding\_amount]

decompressed\_data = bytearray()

current\_node = tree\_root

for bit in encoded\_bits:

if bit == "0":

current\_node = current\_node.left\_child

else:

current\_node = current\_node.right\_child

if current\_node.symbol is not None:

decompressed\_data.append(current\_node.symbol)

current\_node = tree\_root

return bytes(decompressed\_data)

base\_algorithms/MTF.py:

def mtf\_compress(data: bytes) -> bytes:

symbol\_list = list(range(256))

compressed\_output = []

for byte in data:

index = symbol\_list.index(byte)

compressed\_output.append(index)

symbol\_list.pop(index)

symbol\_list.insert(0, byte)

return bytes(compressed\_output)

def mtf\_decompress(data: bytes) -> bytes:

symbol\_list = list(range(256))

decompressed\_output = []

for index in data:

symbol = symbol\_list[index]

decompressed\_output.append(symbol)

symbol\_list.pop(index)

symbol\_list.insert(0, symbol)

return bytes(decompressed\_output)

base\_algorithms/LZ77.py:

import struct

def lz77\_compress(input\_data: bytes, window\_size=4096, lookahead\_buffer\_size=18):

compressed\_data = bytearray()

current\_index = 0

while current\_index < len(input\_data):

offset, length\_of\_match = 0, 0

search\_start = max(0, current\_index - window\_size)

for match\_length in range(1, min(lookahead\_buffer\_size, len(input\_data) - current\_index) + 1):

sequence = input\_data[current\_index:current\_index + match\_length]

position = input\_data[search\_start:current\_index].rfind(sequence)

if position != -1:

offset = current\_index - (search\_start + position)

length\_of\_match = match\_length

else:

break

next\_char = input\_data[current\_index + length\_of\_match] if current\_index + length\_of\_match < len(input\_data) else 0

compressed\_data.extend(struct.pack(">HB", offset, length\_of\_match) + bytes([next\_char]))

current\_index += length\_of\_match + 1

return compressed\_data

def lz77\_decompress(compressed\_data: bytes):

decompressed\_data = bytearray()

current\_index = 0

while current\_index < len(compressed\_data):

offset, length\_of\_match = struct.unpack(">HB", compressed\_data[current\_index:current\_index+3])

next\_char = compressed\_data[current\_index+3]

current\_index += 4

start\_position = len(decompressed\_data) - offset

for j in range(length\_of\_match):

decompressed\_data.append(decompressed\_data[start\_position + j])

if next\_char:

decompressed\_data.append(next\_char)

return bytes(decompressed\_data)

base\_algorithms/LZ78.py:

import struct

def lz78\_compress(input\_data: bytes):

dictionary = {}

compressed\_data = bytearray()

current\_index, dictionary\_size = 0, 1

current\_buffer = bytearray()

while current\_index < len(input\_data):

current\_buffer.append(input\_data[current\_index])

if bytes(current\_buffer) not in dictionary:

dictionary[bytes(current\_buffer)] = dictionary\_size

prefix\_index = dictionary.get(bytes(current\_buffer[:-1]), 0)

compressed\_data.extend(struct.pack(">IB", prefix\_index, current\_buffer[-1]))

dictionary\_size += 1

current\_buffer.clear()

current\_index += 1

return compressed\_data

def lz78\_decompress(compressed\_data: bytes):

dictionary = {0: b""}

decompressed\_data = bytearray()

current\_index, dictionary\_size = 0, 1

while current\_index < len(compressed\_data):

prefix\_index, char = struct.unpack(">IB", compressed\_data[current\_index:current\_index+5])

current\_index += 5

new\_entry = dictionary[prefix\_index] + bytes([char])

dictionary[dictionary\_size] = new\_entry

dictionary\_size += 1

decompressed\_data.extend(new\_entry)

return bytes(decompressed\_data)

compressors/compress\_RLE.py:

import os

from base\_algorithms.RLE import rle\_compress, rle\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_rle(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

start\_compression\_decompression(input\_path, compressed\_path, rle\_compress)

start\_compression\_decompression(compressed\_path, decompressed\_path, rle\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

if not(part\_of\_another\_compressor):

print(f"Исходный размер RLE файла: {input\_size} байт")

print(f"Сжатый размер файла RLE: {compressed\_size} байт")

print(f"Разжатый размер файла RLE: {decompressed\_size} байт")

return compressed\_path

compressors/compress\_BWT.py:

import os

from base\_algorithms.BWT import bwt\_compress, bwt\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_bwt(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

start\_compression\_decompression(input\_path, compressed\_path, bwt\_compress)

start\_compression\_decompression(compressed\_path, decompressed\_path, bwt\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

if not(part\_of\_another\_compressor):

print(f"Исходный размер BWT файла: {input\_size} байт")

print(f"Сжатый размер файла BWT: {compressed\_size} байт")

print(f"Разжатый размер файла BWT: {decompressed\_size} байт")

return compressed\_path

compressors/compress\_HA.py:

import os

from base\_algorithms.HA import ha\_compress, ha\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_ha(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

start\_compression\_decompression(input\_path, compressed\_path, ha\_compress)

start\_compression\_decompression(compressed\_path, decompressed\_path, ha\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

if not(part\_of\_another\_compressor):

print(f"Исходный размер HA файла: {input\_size} байт")

print(f"Сжатый размер файла HA: {compressed\_size} байт")

print(f"Разжатый размер файла HA: {decompressed\_size} байт")

return compressed\_path

compressors/compress\_MTF.py:

import os

from base\_algorithms.MTF import mtf\_compress, mtf\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_mtf(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

start\_compression\_decompression(input\_path, compressed\_path, mtf\_compress)

start\_compression\_decompression(compressed\_path, decompressed\_path, mtf\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

if not(part\_of\_another\_compressor):

print(f"Исходный размер MTF файла: {input\_size} байт")

print(f"Сжатый размер файла MTF: {compressed\_size} байт")

print(f"Разжатый размер файла MTF: {decompressed\_size} байт")

return compressed\_path

compressors/compress\_LZ77.py:

import os

from base\_algorithms.LZ77 import lz77\_compress, lz77\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_lz77(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

start\_compression\_decompression(input\_path, compressed\_path, lz77\_compress)

start\_compression\_decompression(compressed\_path, decompressed\_path, lz77\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

if not(part\_of\_another\_compressor):

print(f"Исходный размер LZ77 файла: {input\_size} байт")

print(f"Сжатый размер файла LZ77: {compressed\_size} байт")

print(f"Разжатый размер файла LZ77: {decompressed\_size} байт")

return compressed\_path

compressors/compress\_LZ78.py:

import os

from base\_algorithms.LZ78 import lz78\_compress, lz78\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_lz78(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

start\_compression\_decompression(input\_path, compressed\_path, lz78\_compress)

start\_compression\_decompression(compressed\_path, decompressed\_path, lz78\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

if not(part\_of\_another\_compressor):

print(f"Исходный размер LZ78 файла: {input\_size} байт")

print(f"Сжатый размер файла LZ78: {compressed\_size} байт")

print(f"Разжатый размер файла LZ78: {decompressed\_size} байт")

return compressed\_path

compressors/compress\_LZ77\_HA.py:

import os

from base\_algorithms.HA import ha\_compress, ha\_decompress

from base\_algorithms.LZ77 import lz77\_compress, lz77\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_lz77\_ha(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

lz77\_compress\_path = start\_compression\_decompression(input\_path, compressed\_path, lz77\_compress)

lz77\_ha\_compress\_path = start\_compression\_decompression(lz77\_compress\_path, compressed\_path, ha\_compress)

path\_lz77\_decompress = start\_compression\_decompression(lz77\_ha\_compress\_path, decompressed\_path, ha\_decompress)

path\_lz77\_ha\_decompress = start\_compression\_decompression(path\_lz77\_decompress, decompressed\_path, lz77\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

print(f"Исходный размер LZ77\_HA файла: {input\_size} байт")

print(f"Сжатый размер файла LZ77\_HA: {compressed\_size} байт")

print(f"Разжатый размер файла LZ77\_HA: {decompressed\_size} байт")

return lz77\_ha\_compress\_path

compressors/compress\_LZ78\_HA.py:

import os

from base\_algorithms.HA import ha\_compress, ha\_decompress

from base\_algorithms.LZ78 import lz78\_compress, lz78\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_lz78\_ha(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

lz78\_compress\_path = start\_compression\_decompression(input\_path, compressed\_path, lz78\_compress)

lz78\_ha\_compress\_path = start\_compression\_decompression(lz78\_compress\_path, compressed\_path, ha\_compress)

path\_lz78\_decompress = start\_compression\_decompression(lz78\_ha\_compress\_path, decompressed\_path, ha\_decompress)

path\_lz78\_ha\_decompress = start\_compression\_decompression(path\_lz78\_decompress, decompressed\_path, lz78\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

print(f"Исходный размер LZ78\_HA файла: {input\_size} байт")

print(f"Сжатый размер файла LZ78\_HA: {compressed\_size} байт")

print(f"Разжатый размер файла LZ78\_HA: {decompressed\_size} байт")

return lz78\_ha\_compress\_path

compressors/compress\_BWT\_RLE.py

import os

from base\_algorithms.BWT import bwt\_compress, bwt\_decompress

from base\_algorithms.RLE import rle\_compress, rle\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_bwt\_rle(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

bwt\_compress\_path = start\_compression\_decompression(input\_path, compressed\_path, bwt\_compress)

bwt\_rle\_compress\_path = start\_compression\_decompression(bwt\_compress\_path, compressed\_path, rle\_compress)

path\_rle\_decompress = start\_compression\_decompression(bwt\_rle\_compress\_path, decompressed\_path, rle\_decompress)

path\_bwt\_rle\_decompress = start\_compression\_decompression(path\_rle\_decompress, decompressed\_path, bwt\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

print(f"Исходный размер BWT\_RLE файла: {input\_size} байт")

print(f"Сжатый размер файла BWT\_RLE: {compressed\_size} байт")

print(f"Разжатый размер файла BWT\_RLE: {decompressed\_size} байт")

return bwt\_rle\_compress\_path

compressors/compress\_BWT\_MTF\_HA.py

import os

from base\_algorithms.BWT import bwt\_compress, bwt\_decompress

from base\_algorithms.HA import ha\_compress, ha\_decompress

from base\_algorithms.MTF import mtf\_compress, mtf\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_bwt\_mtf\_ha(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

bwt\_compress\_path = start\_compression\_decompression(input\_path, compressed\_path, bwt\_compress)

bwt\_mtf\_compress\_path = start\_compression\_decompression(bwt\_compress\_path, compressed\_path, mtf\_compress)

bwt\_mtf\_ha\_compress\_path = start\_compression\_decompression(bwt\_mtf\_compress\_path, compressed\_path, ha\_compress)

bwt\_decompress\_path = start\_compression\_decompression(bwt\_mtf\_ha\_compress\_path, decompressed\_path, bwt\_decompress)

bwt\_mtf\_decompress\_path = start\_compression\_decompression(bwt\_decompress\_path, decompressed\_path, mtf\_decompress)

bwt\_mtf\_ha\_decompress\_path = start\_compression\_decompression(bwt\_mtf\_decompress\_path, decompressed\_path, ha\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

print(f"Исходный размер BWT\_MTF\_HA файла: {input\_size} байт")

print(f"Сжатый размер файла BWT\_MTF\_HA: {compressed\_size} байт")

print(f"Разжатый размер файла BWT\_MTF\_HA: {decompressed\_size} байт")

return bwt\_mtf\_ha\_compress\_path

compressors/compress\_BWT\_MTF\_RLE\_HA.py

import os

from base\_algorithms.BWT import bwt\_compress, bwt\_decompress

from base\_algorithms.HA import ha\_compress, ha\_decompress

from base\_algorithms.MTF import mtf\_compress, mtf\_decompress

from base\_algorithms.RLE import rle\_compress, rle\_decompress

from lib.start\_compression\_decompression import start\_compression\_decompression

def compressor\_bwt\_mtf\_rle\_ha(input\_path: str, compressed\_path: str, decompressed\_path: str, part\_of\_another\_compressor: bool):

bwt\_compress\_path = start\_compression\_decompression(input\_path, compressed\_path, bwt\_compress)

bwt\_mtf\_compress\_path = start\_compression\_decompression(bwt\_compress\_path, compressed\_path, mtf\_compress)

bwt\_mtf\_rle\_compress\_path = start\_compression\_decompression(bwt\_mtf\_compress\_path, compressed\_path, rle\_compress)

bwt\_mtf\_rle\_ha\_compress\_path = start\_compression\_decompression(bwt\_mtf\_rle\_compress\_path, compressed\_path, ha\_compress)

bwt\_decompress\_path = start\_compression\_decompression(bwt\_mtf\_rle\_ha\_compress\_path, decompressed\_path, bwt\_decompress)

bwt\_mtf\_decompress\_path = start\_compression\_decompression(bwt\_decompress\_path, decompressed\_path, mtf\_decompress)

bwt\_mtf\_rle\_decompress\_path = start\_compression\_decompression(bwt\_mtf\_decompress\_path, decompressed\_path, rle\_decompress)

bwt\_mtf\_rle\_ha\_decompress\_path = start\_compression\_decompression(bwt\_mtf\_rle\_decompress\_path, decompressed\_path, ha\_decompress)

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

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

decompressed\_size = os.path.getsize(decompressed\_path)

print(f"Исходный размер BWT\_MTF\_RLE\_HA файла: {input\_size} байт")

print(f"Сжатый размер файла BWT\_MTF\_RLE\_HA: {compressed\_size} байт")

print(f"Разжатый размер файла BWT\_MTF\_RLE\_HA: {decompressed\_size} байт")

return bwt\_mtf\_rle\_ha\_compress\_path

lib/entropy.py

import math

from collections import Counter

def calculate\_entropy(input\_data: bytes) -> float:

if not input\_data:

return 0.0

frequency = Counter(input\_data)

total\_bytes = len(input\_data)

entropy\_value = -sum(

(count / total\_bytes) \* math.log2(count / total\_bytes)

for count in frequency.values()

)

return entropy\_value

lib/start\_compression\_decompression.py

CHUNK\_SIZE = 1024

def start\_compression\_decompression(input\_path: str, compressed\_decompressed\_path: str, compression\_function):

with open(input\_path, 'rb') as input\_file, open(compressed\_decompressed\_path, 'wb') as output\_file:

while chunk := input\_file.read(CHUNK\_SIZE):

processed\_chunk = compression\_function(chunk)

with open(input\_path, 'w'):

pass

output\_file.write(processed\_chunk)

return compressed\_decompressed\_path