МИНОБРНАУКИ РОССИИ САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

**ОТЧЕТ**

по лабораторной работе № 1

дисциплина «Алгоритмы и Структуры данных»

Тема: «Алгоритмы сжатия без потерь»

Студент гр. 3351 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Фабер К.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Пестерев Д.О.

Санкт-Петербург

2025

**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.**

**HA.**

Алгоритм Хаффмана – алгоритм, который на основе таблицы частотности символов сопоставляет каждому байту уникальный код путем построения дерева, причем, чем чаще встречается определенный байт, тем меньше его код.

**BWT.**

Алгоритм BWT – алгоритм, который преобразует исходную байтовую строку в перестановку этой байтовой строки, такую что, одинаковые байты чаще всего оказываются рядом.

**MTF.**

Алгоритм MTF – алгоритм, который преобразует данные, заменяя каждый символ на его текущий индекс в динамическом списке. После обработки символа он перемещается в начало списка.

**RLE.**

Алгоритм RLE - алгоритм, который кодирует часто повторяющиеся байты в байт (или несколько байт) количества и сам повторяющийся байт, причем, для оптимизации можно считать последовательность байтов, которые не повторяются, путем использования старшего бита в качестве флага.

**LZ77.**

Алгоритм LZ77 – алгоритм, который сжимает данные при помощи скользящего окна, состоящего из уже прочитанных данных и еще не прочитаных. На каждом байте происходит проверка, есть ли в уже прочитанных данных данный байт или последовательность байт, в случае, если такая последовательность была найдена, в закодированную строку добавляется ссылка на вхождение и следующий байт. Ссылка состоит из трех байт: первый отвечает за отступ найденного байта от буфера, второй – за длину последовательности, а третий – следующий байт в буфере.

**LZ78.**

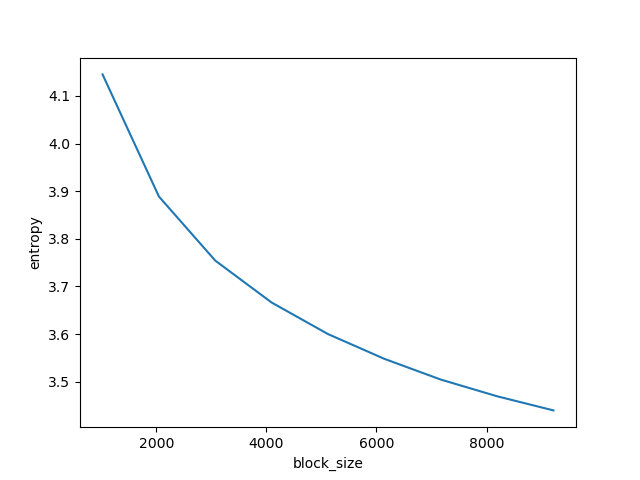
Алгоритм LZ78 - алгоритм, который сжимает данные, строя словарь встречающихся байтов. Если байт/байты уже есть в словаре, он/они заменяется на ссылку (индекс в словаре + новый символ).

**Сложности алгоритмов.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм | Временная сложность кодирования | Временная сложность декодирования | Пространственная сложность кодирования | Пространственная сложность декодирования |
| HA |  |  |  |  |
| BWT |  |  |  |  |
| MTF |  |  |  |  |
| RLE |  |  |  |  |
| LZ77 | ,  w – размер окна буфера  l – размер буфера |  |  |  |
| LZ78 |  |  | ,  d – размер словаря |  |

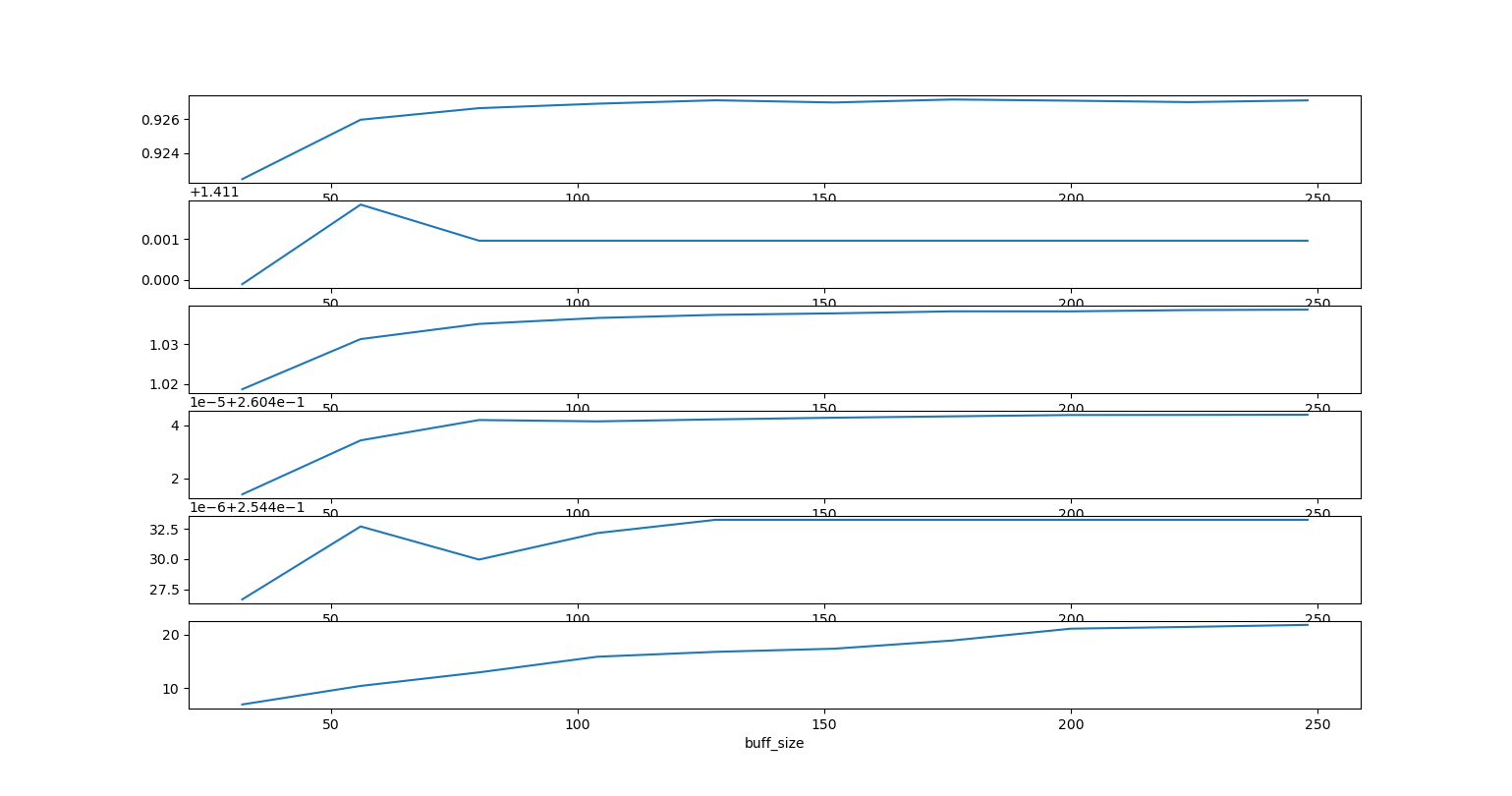
**Практическая часть.**

**Исследование зависимости энтропии от размера блоков, на который разбивается текст enwik7 для BWT + MTF.**



С увеличением размера блоков, энтропия уменьшается. Однако энтропия меняется незначительно, в отличии от времени и памяти. В качестве оптимального размера блока можно взять значение 10 Кб.

**Исследование зависимости коэффициента сжатия от размера буфера LZ77.**

****

Можно заметить, что для каждого файла, коэффициент сжатия увеличивается по мере увеличения размера буфера, однако, с увеличением размера буфера увеличивается и время сжатия, а также требуется больше памяти при сжатии, самый оптимальный размер = 90 байт, т.к. на большей части графиков, такой размер обеспечивает коэффициент сжатия, незначительно меньше максимального.

**Таблица эффективности компрессоров.**

****

**Ссылки.**

Репозиторий GitHub: <https://github.com/KIRILLFABER/Compress-alg>

**Код.**

**bwt.py**

def compress(data): # time complexity: T(n) = 2n + 1 + O(nlogn) + O(n) + 2n + 2 = 2n + O(nlogn) + O(n) + 3 = O(nlogn)

                    # space complexity: S(n) = n \* n + 4 + n = O(n^2)

    matrix = [data[i:] + data[0:i] for i in range(len(data))]

    matrix.sort()

    index = matrix.index(data)

    last\_col = bytes(list(map(lambda x: x[-1], matrix)))

    return index.to\_bytes(2, 'big') + last\_col

def decompress(last\_column\_BWM): # time compltxity: T(n) = 1 + 3 + 1 + 1 + O(n) + 2 + 3n = O(n)

                                # space complexity: S(n) = 2 + n - 2 + 4 + n + 4 + 4 + n = O(n)

    if not last\_column\_BWM:

        return b""

    S\_index = int.from\_bytes(last\_column\_BWM[:2], 'big')

    last\_column\_BWM = last\_column\_BWM[2:]

    N = len(last\_column\_BWM)

    P\_inverse = counting\_sort\_arg\_bytes(last\_column\_BWM)

    reconstructed = bytearray()

    j = S\_index

    for \_ in range(N):

        if j >= len(P\_inverse):

            break

        j = P\_inverse[j]

        reconstructed.append(last\_column\_BWM[j])

    return bytes(reconstructed)

def counting\_sort\_arg\_bytes(S):

    N = len(S)

    if N == 0:

        return []

    M = 256

    counts = [0] \* M

    for b in S:

        counts[b] += 1

    pos = [0] \* M

    for i in range(1, M):

        pos[i] = pos[i-1] + counts[i-1]

    P\_inverse = [0] \* N

    temp\_pos = pos.copy()

    for i, b in enumerate(S):

        P\_inverse[temp\_pos[b]] = i

        temp\_pos[b] += 1

    return P\_inverse

    # banana

    # aaabnn

# def counting\_sort(data):

#     if not data:

#         return b""

#     counts = [0] \* 256

#     for byte in data:

#         counts[byte] += 1

#     sorted\_data = bytearray()

#     for byte in range(256):

#         sorted\_data.extend([byte] \* counts[byte])

#     return bytes(sorted\_data)

**haffman.py**

from collections import defaultdict

import heapq

import struct

class Node:

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

        self.symbol = symbol

        self.counter = counter

        self.left = left

        self.right = right

    def \_\_lt\_\_(self, other):

        return self.counter < other.counter

def build\_tree(text):

    symbols = defaultdict(int)

    for byte in text:

        symbols[byte] += 1

    heap = []

    for byte, freq in symbols.items():

        node = Node(symbol=byte, counter=freq)

        heapq.heappush(heap, node)

    while len(heap) > 1:

        left = heapq.heappop(heap)

        right = heapq.heappop(heap)

        parent = Node(counter=left.counter + right.counter, left=left, right=right)

        heapq.heappush(heap, parent)

    return heapq.heappop(heap)

def generate\_codes(node, code="", code\_dict=None):

    if code\_dict is None:

        code\_dict = {}

    if node is not None:

        if node.symbol is not None:

            code\_dict[node.symbol] = code

        generate\_codes(node.left, code + "0", code\_dict)

        generate\_codes(node.right, code + "1", code\_dict)

    return code\_dict

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) + serialize\_tree(node.right)

def deserialize\_tree(data):

    def helper():

        nonlocal index

        if index >= len(data):

            return None

        flag = data[index]

        index += 1

        if flag == 1:

            symbol = data[index]

            index += 1

            return Node(symbol=symbol)

        elif flag == 0:

            left = helper()

            right = helper()

            return Node(left=left, right=right)

    index = 0

    return helper()

def compress(text):

    if not text:

        return b""

    root = build\_tree(text)

    codes = generate\_codes(root)

    encoded\_bits = "".join([codes[byte] for byte in text])

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

    encoded\_bits += "0" \* padding

    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(root)

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

    return packed\_data

def decompress(packed\_data):

    if not packed\_data:

        return b""

    padding = packed\_data[0]

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

    tree\_bytes = packed\_data[5:5 + tree\_length]

    root = deserialize\_tree(tree\_bytes)

    encoded\_bytes = packed\_data[5 + tree\_length:]

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

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

    decoded\_text = bytearray()

    current\_node = root

    for bit in encoded\_bits:

        if bit == "0":

            current\_node = current\_node.left

        else:

            current\_node = current\_node.right

        if current\_node.symbol is not None:

            decoded\_text.append(current\_node.symbol)

            current\_node = root

    return bytes(decoded\_text)

**mtf.py**

def compress(data): # time complexity: T(n) = 1 + 1 + 4n = O(n)

                    # space complexity: S(n) = 256 \* 4 + 4 + 4n = O(n)

    alphabet = list(range(256))

    result = bytearray()

    for byte in data:

        idx = alphabet.index(byte)

        result.append(idx)

        alphabet.pop(idx)

        alphabet.insert(0, byte)

    return bytes(result)

def decompress(data):

                        # time complexity: T(n) = O(n)

                        # space complexity: S(n) = O(n)

    alphabet = list(range(256))

    result = bytearray()

    for idx in data:

        byte = alphabet[idx]

        result.append(byte)

        alphabet.pop(idx)

        alphabet.insert(0, byte)

    return bytes(result)

**rle.py**

def compress(data, byte\_count=2):

    if not data:

        return b""

    max\_count = (1 << (8 \* byte\_count)) - 1

    comp\_data = bytearray()

    buffer = bytearray()

    prev\_byte = data[0]

    count = 1

    for byte in data[1:]:

        if byte == prev\_byte and count < max\_count:

            count += 1

        else:

            if count > 1:

                if buffer:

                    comp\_data.extend((len(buffer) | (1 << (8 \* byte\_count - 1))).to\_bytes(byte\_count, 'big'))

                    comp\_data.extend(buffer)

                    buffer.clear()

                comp\_data.extend(count.to\_bytes(byte\_count, 'big'))

                comp\_data.append(prev\_byte)

            else:

                buffer.append(prev\_byte)

                if len(buffer) >= max\_count:

                    comp\_data.extend((max\_count | (1 << (8 \* byte\_count - 1))).to\_bytes(byte\_count, 'big'))

                    comp\_data.extend(buffer[:max\_count])

                    buffer = buffer[max\_count:]

            count = 1

            prev\_byte = byte

    if count > 1:

        if buffer:

            comp\_data.extend((len(buffer) | (1 << (8 \* byte\_count - 1))).to\_bytes(byte\_count, 'big'))

            comp\_data.extend(buffer)

        comp\_data.extend(count.to\_bytes(byte\_count, 'big'))

        comp\_data.append(prev\_byte)

    else:

        buffer.append(prev\_byte)

        if buffer:

            comp\_data.extend((len(buffer) | (1 << (8 \* byte\_count - 1))).to\_bytes(byte\_count, 'big'))

            comp\_data.extend(buffer)

    return bytes(comp\_data)

def decompress(data, byte\_count=2):

    if not data:

        return b""

    decomp\_data = bytearray()

    i = 0

    n = len(data)

    mask = 1 << (8 \* byte\_count - 1)

    while i < n:

        if i + byte\_count > n:

            raise ValueError("Invalid compressed data")

        count = int.from\_bytes(data[i:i+byte\_count], 'big')

        i += byte\_count

        if count & mask:

            true\_count = count & ~mask

            if i + true\_count > n:

                raise ValueError("Invalid compressed data")

            decomp\_data.extend(data[i:i+true\_count])

            i += true\_count

        else:

            byte = data[i]

            decomp\_data.extend([byte] \* count)

            i += 1

    return bytes(decomp\_data)

**lz77.py**

import struct

from collections import defaultdict

def compress(data, window\_size=8192, lookahead\_size=90):

    compressed = bytearray()

    pos = 0

    length = len(data)

    hash\_table = defaultdict(list)

    while pos < length:

        best\_offset = 0

        best\_len = 0

        remaining = length - pos

        current\_lookahead = min(lookahead\_size, remaining)

        if current\_lookahead > 0:

            max\_offset = min(window\_size, pos)

            if remaining >= 3:

                key = data[pos:pos+3]

                for candidate in hash\_table.get(key, []):

                    if pos - candidate > max\_offset:

                        continue

                    match\_len = 0

                    while (match\_len < current\_lookahead and

                           data[candidate + match\_len] == data[pos + match\_len]):

                        match\_len += 1

                    if match\_len > best\_len:

                        best\_offset = pos - candidate

                        best\_len = match\_len

                        if best\_len == current\_lookahead:

                            break

            if remaining >= 3:

                hash\_table[data[pos:pos+3]].append(pos)

                if len(hash\_table[key]) > 32:

                    hash\_table[key] = hash\_table[key][-16:]

        next\_char\_pos = pos + best\_len

        next\_char = data[next\_char\_pos] if next\_char\_pos < length else 0

        compressed.extend(struct.pack('>HB', best\_offset, best\_len))

        if best\_len < remaining:

            compressed.append(next\_char)

        pos += best\_len + 1 if best\_len < remaining else best\_len

    return bytes(compressed)

def decompress(compressed):

    decompressed = bytearray()

    pos = 0

    length = len(compressed)

    while pos + 2 < length:

        offset = compressed[pos] << 8 | compressed[pos+1]

        ln = compressed[pos+2]

        pos += 3

        if offset > 0:

            start = len(decompressed) - offset

            if start >= 0 and ln > 0:

                end = start + ln

                if end <= len(decompressed):

                    decompressed.extend(decompressed[start:end])

                else:

                    repeat = ln // offset

                    remainder = ln % offset

                    for \_ in range(repeat):

                        decompressed.extend(decompressed[start:start+offset])

                    if remainder:

                        decompressed.extend(decompressed[start:start+remainder])

        if pos < length:

            decompressed.append(compressed[pos])

            pos += 1

    return bytes(decompressed)

**lz78.py**

import struct

from collections import defaultdict

def compress(data, max\_dict\_size=65535):

    dictionary = {b'': 0}

    next\_code = 1

    compressed = bytearray()

    w = b''

    for byte in data:

        c = bytes([byte])

        wc = w + c

        if wc in dictionary:

            w = wc

        else:

            compressed.extend(struct.pack('>H', dictionary[w]))

            compressed.append(byte)

            if next\_code < max\_dict\_size:

                dictionary[wc] = next\_code

                next\_code += 1

            w = b''

    if w:

        last\_byte = w[-1:]

        prefix = w[:-1]

        compressed.extend(struct.pack('>H', dictionary[prefix]))

        compressed.append(ord(last\_byte))

    return bytes(compressed)

def decompress(compressed, max\_dict\_size=65535):

    dictionary = {0: b''}

    next\_code = 1

    decompressed = bytearray()

    pos = 0

    while pos + 2 < len(compressed):

        code = (compressed[pos] << 8) | compressed[pos+1]

        byte = compressed[pos+2]

        pos += 3

        phrase = dictionary[code] + bytes([byte])

        decompressed.extend(phrase)

        if next\_code < max\_dict\_size:

            dictionary[next\_code] = phrase

            next\_code += 1

    if pos < len(compressed):

        decompressed.extend(compressed[pos:])

    return bytes(decompressed)

**compressors.py**

import haffman, bwt, lz77, lz78, mtf, rle, metrics

# 1. HA

def HA(data):

    return haffman.compress(data)

# 2. Run-length encoding (RLE)

def RLE(data):

    return rle.compress(data)

# 3. BWT + RLE

def BWT\_RLE(data, block\_size = 1024 \* 4):

    bwt\_result = b''

    for block in [data[i:i + block\_size] for i in range(0, len(data), block\_size)]:

        bwt\_result += bwt.compress(block)

    return rle.compress(bwt\_result)

# 4. BWT + MTF + HA

def BWT\_MTF\_HA(data, block\_size = 1024 \* 4):

    bwt\_result = b''

    for block in [data[i:i + block\_size] for i in range(0, len(data), block\_size)]:

        bwt\_result += bwt.compress(block)

    mtf\_result = mtf.compress(bwt\_result)

    ha\_result = haffman.compress(mtf\_result)

    return ha\_result

# 5. BWT + MTF + RLE + HA

def BWT\_MTF\_RLE\_HA(data, block\_size = 1024 \* 4):

    bwt\_result = b''

    for block in [data[i:i + block\_size] for i in range(0, len(data), block\_size)]:

        bwt\_result += bwt.compress(block)

    mtf\_result = mtf.compress(bwt\_result)

    rle\_result = rle.compress(mtf\_result)

    ha\_result = haffman.compress(rle\_result)

    return ha\_result

# 6. LZ77

def LZ77(data):

    return lz77.compress(data)

# 7. LZ77 + HA

def LZ77\_HA(data):

    lz77\_result = lz77.compress(data)

    return haffman.compress(lz77\_result)

# 8. LZ78

def LZ78(data):

    return lz78.compress(data)

# 9. LZ78 + HA

def LZ78\_HA(data):

    lz78\_result = lz78.compress(data)

    return haffman.compress(lz78\_result)

**decompressors.py**

import haffman, bwt, lz77, lz78, mtf, rle, metrics

# 1. HA

def HA(data):

    return haffman.decompress(data)

# 2. Run-length encoding (RLE)

def RLE(data):

    return rle.decompress(data)

# 3. BWT + RLE

def BWT\_RLE(data, block\_size = 1024 \* 4 + 2):

    data = rle.decompress(data)

    bwt\_result = b''

    for block in [data[i:i + block\_size] for i in range(0, len(data), block\_size)]:

        bwt\_result += bwt.decompress(block)

    return bwt\_result

# 4. BWT + MTF + HA

def BWT\_MTF\_HA(data, block\_size = 1024 \* 4 + 2):

    ha\_decomp = haffman.decompress(data)

    mtf\_decomp = mtf.decompress(ha\_decomp)

    data = mtf\_decomp

    bwt\_decomp = b''

    for block in [data[i:i + block\_size] for i in range(0, len(data), block\_size)]:

        bwt\_decomp += bwt.decompress(block)

    return bwt\_decomp

# 5. BWT + MTF + RLE + HA

def BWT\_MTF\_RLE\_HA(data, block\_size = 1024 \* 4 + 2):

    ha\_decomp = haffman.decompress(data)

    rle\_decomp = rle.decompress(ha\_decomp)

    mtf\_decomp = mtf.decompress(rle\_decomp)

    data = mtf\_decomp

    bwt\_decomp = b''

    for block in [data[i:i + block\_size] for i in range(0, len(data), block\_size)]:

        bwt\_decomp += bwt.decompress(block)

    return bwt\_decomp

# 6. LZ77

def LZ77(data):

    return lz77.decompress(data)

# 7. LZ77 + HA

def LZ77\_HA(data):

    ha\_decomp = haffman.decompress(data)

    return lz77.decompress(ha\_decomp)

# 8. LZ78

def LZ78(data):

    return lz78.decompress(data)

# 9. LZ78 + HA

def LZ78\_HA(data):

    ha\_decomp = haffman.decompress(data)

    return lz78.decompress(ha\_decomp)

**count\_block\_size.py**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import bwt, mtf

import metrics

f\_input = open('./data/enwik7', 'rb')

data = f\_input.read()

f\_input.close()

x = []

y = []

for block\_size in range(1024 \* 1, 1024 \* 10, 1024):

    bwt\_result = b''

    for i in range(0, len(data), block\_size):

        bwt\_result += bwt.compress(data[i: i + block\_size])

    mtf\_result = mtf.compress(bwt\_result)

    print(block\_size)

    x.append(block\_size)

    y.append(metrics.entropy(mtf\_result))

fig, ax = plt.subplots()

ax.plot(np.array(x), np.array(y))

ax.set\_xlabel('block\_size')

ax.set\_ylabel('entropy')

fig.savefig('./block\_size\_BWT\_MTF.png')

**count\_size\_buff.py**

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import metrics, lz77

df = pd.DataFrame(columns=['buffer\_size', 'data\_file', 'compress\_factor', 'correct\_decompress'])

data\_path = './data/'

data\_files = []

data\_files\_name = ['enwik7', 'ru\_text.txt', 'exe\_file.exe', 'img.CR2', 'grey\_img.raw', 'bw\_img.raw']

for name in data\_files\_name:

        f\_input = open(data\_path + name, 'rb')

        file = f\_input.read()

        f\_input.close()

        data\_files.append(file)

        print('file - ', name)

k = 3

for size in range(2 \*\* 5, 2 \*\* 8, 8 \* k):

    for i in range(len(data\_files)):

        compress\_data = lz77.compress(data\_files[i], lookahead\_size=size)

        decompress\_data = lz77.decompress(compress\_data)

        df.loc[len(df)] = [size, data\_files\_name[i], metrics.compressFactor(compress\_data, data\_files[i]), decompress\_data == data\_files[i]]

        print(size)

print(df)

print('===============')

fig, ax = plt.subplots(len(data\_files), 1, figsize = (100, 100))

for i in range(len(data\_files)):

     x = df[df['data\_file'] == data\_files\_name[i]]['buffer\_size'].values

     y = df[df['data\_file'] == data\_files\_name[i]]['compress\_factor'].values

     ax[i].plot(x, y, label=data\_files\_name[i])

     ax[i].set\_xlabel("buff\_size")

fig.show()

fig.savefig('./buffer\_size\_lz77.png')

f\_input.close()

**metrics.py**

import numpy as np

import numpy as np

def prob(data):

    counts = np.zeros(256, dtype=np.float64)

    for byte in data:

        counts[byte] += 1

    total = len(data)

    return counts / total

def entropy(data):

    P = prob(data)

    P = P[P != 0]

    if len(P) == 0:

        return 0.0

    return -np.sum(P \* np.log2(P))

def compressFactor(compress\_data, data):

    return len(data) / len(compress\_data)

# def prob(seq):

#     uniq = list(set(seq))

#     amount = []

#     probs = []

#     for i in uniq:

#         c = 0

#         for j in seq:

#             if j == i:

#                 c+= 1

#         amount.append(c)

#     print(amount)

#     for i in range(len(uniq)):

#         probs.append(amount[i] / len(seq))

#     return probs

**fillTable.py**

import pandas as pd

import numpy as np

import compressors, decompressors, metrics

def fillTable():

    table = pd.DataFrame(columns=['Compressor', 'Data\_File', 'Size\_before\_compression', 'Size\_after\_compression', 'Size\_after\_decompression', 'Compress\_factor', 'Correct\_decompress'])

    compressors\_list = [compressors.HA, compressors.RLE, compressors.BWT\_RLE, compressors.BWT\_MTF\_HA, compressors.BWT\_MTF\_RLE\_HA, compressors.LZ77, compressors.LZ77\_HA, compressors.LZ78, compressors.LZ78\_HA]

    decompressors\_list = [decompressors.HA, decompressors.RLE, decompressors.BWT\_RLE, decompressors.BWT\_MTF\_HA, decompressors.BWT\_MTF\_RLE\_HA, decompressors.LZ77, decompressors.LZ77\_HA, decompressors.LZ78, decompressors.LZ78\_HA]

    compressors\_name = ['HA', 'RLE', 'BWT + RLE', 'BWT + MTF + HA', 'BWT + MTF + RLE + HA', 'LZ77', 'LZ77 + HA', 'LZ78', 'LZ78 + HA']

    data\_files = []

    data\_files\_name = ['enwik7', 'ru\_text.txt', 'exe\_file.exe', 'img.CR2', 'grey\_img.raw', 'bw\_img.raw']

    size = int(5e3)

    data\_path = './data/'

    comp\_path = './compress\_data/'

    decomp\_path = './decompress\_data/'

    for name in data\_files\_name:

        f\_input = open(data\_path + name, 'rb')

        file = f\_input.read()

        f\_input.close()

        data\_files.append(file)

        print('file - ', name)

    for i in range(len(compressors\_list)):

        if True:

        #if (compressors\_name[i] != 'LZ78' and compressors\_name[i] != 'LZ78 + HA'):

        #if (int(input(f'{compressors\_name[i]} - ')) == 1):

            for j in range(len(data\_files)):

                print('compressor - ', compressors\_name[i])

                print('file - ', data\_files\_name[j])

                compress\_file = compressors\_list[i](data\_files[j])

                decompress\_file = decompressors\_list[i](compress\_file)

                table.loc[len(table)] = [compressors\_name[i], data\_files\_name[j], len(data\_files[j]), len(compress\_file), len(decompress\_file), metrics.compressFactor(compress\_file, data\_files[j]), data\_files[j] == decompress\_file]

                f\_comp = open(comp\_path + compressors\_name[i] + '\_comp\_' + data\_files\_name[j], 'wb')

                f\_decomp = open(decomp\_path + compressors\_name[i] + '\_decomp\_' + data\_files\_name[j], 'wb')

                f\_comp.write(compress\_file)

                f\_decomp.write(decompress\_file)

                f\_comp.close()

                f\_decomp.close()

    table.to\_excel('table.xlsx', index=False)

**main.py**

import haffman, bwt, lz77, lz78, mtf, rle, metrics, compressors, decompressors

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import fillTable

# import tmp ###

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

    match(int(input("Заполнить таблицу? 1 - Да, остальное - нет: "))):

        case 1:

            fillTable.fillTable()

        case \_:

            pass