

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: М. А. Бронников
Преподаватель: А. А. Журавлёв
Группа: М8О-207Б
Дата:
Оценка:
Подпись:

Москва, 2019

Архиватор (Huffman + BWT + MTF + RLE)

Задача:

Необходимо реализовать четыре известных метода преобразования данных для сжатия одного файла.

Формат запуска должен быть аналогичен формату запуска программы gzip, должны быть поддержаны следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Ключи:

- **-c** - произвести вывод результата в стандартный вывод
- **-k** - не удалять входной файл после сжатия
- **-l** - вывести информацию о сжатом файле
- **-d** - разархивирование файла
- **-r** - рекурсивный проход по директориям с архивированием лежащих файлов
- **-t** - проверка целостности файла
- **-1** - быстрое сжатие
- **-9** - глубокое сжатие

Помимо ключей при запуске следует указать пути к архивируемым файлам. Если пути не указаны, программа должна считать данные из стандартного потока ввода.

1 Описание

Принцип работы программы:

Необходимо разработать программу, которая для сжатия заданного файла последовательно применяет следующие алгоритмы сжатия:

1. **Burrows-Wheeler transform**
2. **Move-To-Front**
3. **Run-Length Encoding**
4. **Код Хаффмана**

Такая последовательность обусловлена тем, что *преобразование Барроуза-Уилера* преобразует данные к виду, в котором часто встречаются последовательности повторяющихся символов. С такими последовательностями хорошо работают алгоритмы *MTF* и *RLE* для последующего преобразования алгоритмом Хаффмана, который после оптимизирует кодирование повторяющихся последовательностей, полученных после *BWT* что улучшает степень сжатия. [4]

Принцип работы программы:

Основной принцип работы заключается в следующей последовательности шагов:

1. Открытие файлового потока, который необходимо сжать
2. Создание временного файла для размещения в нем информации из потока
3. Последовательное применение алгоритмов сжатия к временному файлу. После применения каждого алгоритма преобразования на выходе получается новый временный файл с результатом работы алгоритма, после чего закрывается и удаляется старый временный файл.
4. После применения всех алгоритмов преобразования на основе полученного результата сжатия формируется результирующий файл, который будет содержать необходимую информацию и результат сжатия.

Преобразование Барроуза-Уилера:

«Преобразование Барроуза — Уилера (англ. Burrows-Wheeler transform) — алгоритм, используемый для предварительной обработки данных перед сжатием, разработанный для улучшения эффективности последующего кодирования. Преобразование Барроуза — Уилера меняет порядок символов во входной строке таким образом, что повторяющиеся подстроки образуют на выходе идущие подряд последовательности

одинаковых символов.» [3]

Преобразование выполняется в три этапа:

1. Составляется таблица всех циклических сдвигов входной строки.
2. Производится лексикографическая (в алфавитном порядке) сортировка строк таблицы.
3. В качестве выходной строки выбирается последний столбец таблицы преобразования и номер строки, совпадающей с исходной.

Наивная реализация первых двух этапов описанного алгоритма будет иметь как пространственную, так и временную сложность $O(n^2)$. Что непозволительно долго.

Поэтому я воспользовался идеей применения суффиксного массива, который по определению является перестановкой всех суффиксов строки в лексикографическом порядке. Существуют довольно сложные алгоритмы, например описанный в [6] *алгоритм Карккайнена-Сандерса*, производящие построение суффиксного массива за время $O(n)$, однако при этом встает нетривиальная задача преобразования суффиксного массива к перестановке циклических сдвигов.

Поэтому я решил пожертвовать асимптотикой и остановился на описанном в [1] алгоритме построения суффиксного массива за $O(n \log n)$, который обладает полезным свойством: он строит массив на основе сортировки циклических сдвигов путем добавления к алгоритму терминального символа, что делает этот метод идеально адаптируемым к нашей задаче, так как мы получим необходимый результат лишь пропустив шаг добавления терминала.

К приятным свойствам алгоритма можно отнести и то, что он требует $O(n)$ памяти.

Опишем алгоритм построения массива. Как сказано в [1]:

«На нулевой фазе мы должны отсортировать циклические подстроки длины 1, т.е. отдельные символы строки, и разделить их на классы эквивалентности (просто одинаковые символы должны быть отнесены к одному классу эквивалентности). Это можно сделать тривиально, например, сортировкой подсчётом. Для каждого символа посчитаем, сколько раз он встретился. Потом по этой информации восстановим отсортированный массив. После этого, проходом по массиву и сравнением символов, строится массив классов эквивалентности.

Далее, пусть мы выполнили $k - 1$ -ю фазу, теперь научимся за $O(n)$ выполнять следующую, k -ю, фазу. Поскольку фаз всего $O(\log n)$, это даст нам требуемый алгоритм с временем $O(n \log n)$. Для этого заметим, что циклическая подстрока длины 2^k состоит из двух подстрок длины 2^{k-1} , которые мы можем сравнивать между собой за $O(1)$, используя информацию с предыдущей фазы — номера классов эквивалентности. Таким образом, для подстроки длины 2^k , начинающейся в позиции i , вся необхо-

димая информация содержится в паре чисел классов эквивалентности для позиции i и $i + 2^{k-1}$.

Это даёт нам весьма простое решение: отсортировать подстроки длины 2^k просто по этим парам чисел, это и даст нам требуемый порядок.

Воспользуемся здесь приёмом, на котором основана так называемая цифровая сортировка: чтобы отсортировать пары, отсортируем их сначала по вторым элементам, а затем — по первым элементам (но уже обязательно стабильной сортировкой, т.е. не нарушающей относительного порядка элементов при равенстве).»

После построения массива сдвигов выполнить 3-ий шаг преобразования *BWT* - тривиальная задача.

При декодировании воспользуемся оптимизацией наивного алгоритма. Наивный алгоритм из [3]: «Выпишем в столбик нашу преобразованную последовательность символов. Запишем её как последний столбик предыдущей матрицы (при прямом преобразовании Барроуза — Уилера), при этом все предыдущие столбцы оставляем пустыми. Далее построчно отсортируем матрицу, затем в предыдущий столбец запишем преобразованную последовательность. Опять построчно отсортируем матрицу. Продолжая таким образом, можно восстановить полный список всех циклических сдвигов строки, которую нам надо найти. Выстроив полный отсортированный список сдвигов, выберем строку с номером, который нам был изначально дан. В итоге мы получим искомую строку.»

Оптимизация наивного алгоритма из [3]: «Заметим, что при каждом проявлении неизвестного столбца выполнялись одни и те же действия. К предыдущему приписывался новый столбец и имеющиеся данные сортировались. На каждом шаге к строке, которая находилась на i -ом месте, приписывался в начало i -ый элемент столбца входных данных. Пусть изначально известно, каким по порядку является приписанный в начало символ (то есть каким по порядку в столбце). Из предыдущего шага известно, какое место занимала строка без этого первого символа (i -ое). Тогда несложно заметить, что при выполнении такой операции строка с номером i всегда будет перемещаться на позицию с номером j .

Поскольку в нашем алгоритме новый столбец приписывается в начало, то мы из состояния i (левый столбец) переходим в состояние j (правый). Для того, чтобы восстановить строку, нам необходимо от последней такой цифры по пути из j в i восстановить строку. »

Описанный в [3] алгоритм декодирования даёт сложность $O(n + m)$, где m - размер алфавита.

Преобразование MTF:

«Преобразование MTF (англ. move-to-front, движение к началу) — алгоритм кодирования, используемый для предварительной обработки данных (обычно потока байтов) перед сжатием, разработанный для улучшения эффективности последующего кодирования.»[4]

Мы будем использовать наивный способ реализации алгоритма, описанного в [4]: «Изначально каждое возможное значение байта записывается в список (алфавит), в ячейку с номером, равным значению байта, т.е. (0, 1, 2, 3, ..., 255). В процессе обработки данных этот список изменяется. По мере поступления очередного символа на выход подается номер элемента, содержащего его значение. После чего этот символ перемещается в начало списка, смещая остальные элементы вправо.»

Изменяющийся алфавит реализуем при помощи структуры - список.

Декодирование будет производиться последовательно аналогично шагам кодирования.

Кодирование длин серий:

«Кодирование длин серий (англ. run-length encoding, RLE) или кодирование повторов — алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов. Серией называется последовательность, состоящая из нескольких одинаковых символов. При кодировании (упаковке, сжатии) строка одинаковых символов, составляющих серию, заменяется строкой, содержащей сам повторяющийся символ и количество его повторов.»[5]

При реализации будем последовательно считать либо количество подряд идущих неповторяющихся символов, либо неповторяющихся символов, записываем полученное число перед каждой серией: отрицательное если серия неповторяющихся символов и положительное если серия повторяющихся, после чего помещаем серию после отрицательного числа или один повторяющийся после положительного.

Интересным моментом является то, что мы кодируем подсчитываемое число одним байтом, что дает нам закодировать серии до 127 повторяющихся символов и до 128 неповторяющихся, после чего будет необходимо обнулить счетчик и рассматривать дальнейшую последовательность символов как новую серию.

Декодирование происходит тем же методом: считываем размер серии, и если он положительный выписываем нужное количество раз повторяющийся символ, иначе выписываем указанное количество символов после числа из входной последовательности.

Алгоритм Хаффмана:

«Идея, положенная в основу кодирования Хаффмана, основана на частоте появления символа в последовательности. Символ, который встречается в последовательности чаще всего, получает новый очень маленький код, а символ, который встречается реже всего, получает, наоборот, очень длинный код.»[2]

Важное свойство кодирования: каждый код символа не является префиксом для кода другого символа.

Алгоритм будет использовать очередь с приоритетом для построения дерева. Приоритетом будет выступать частота встречи символа в последовательности. Описание алгоритма на основе материала из [2]:

1. Для начала посчитаем частоты всех символов
2. После вычисления частот мы создадим узлы бинарного дерева для каждого знака и добавим их в очередь, используя частоту в качестве приоритета.
3. Теперь мы достаём два первых элемента из очереди и связываем их, создавая новый узел дерева, в котором они оба будут потомками, а приоритет нового узла будет равен сумме их приоритетов. После этого мы добавим получившийся новый узел обратно в очередь.
4. После того, как мы свяжем два последних элемента, получится итоговое дерево
5. Теперь, чтобы получить код для каждого символа, надо просто пройти по дереву, и для каждого перехода добавлять 0, если мы идём влево, и 1 — если направо

После построения дерева построим таблицу кодов при помощи обхода дерева для быстрого кодирования символа.

Дерево мы сохраним в файле для возможности декодирования последовательности.

При декодировании необходимо считать дерево из входной последовательности, после чего в зависимости от встреченного бита во входной последовательности будем двигаться влево или вправо пока не придем в лист. Если попадаем в лист выписываем полученный символ в выходную последовательность и возвращаемся в корень.

2 Демонстрация работы:

Продemonстрируем работу алгоритма на примере текстового файла, содержащего в себе строку «abrakadabra».

Исходный буфер после считывания из файла будет выглядеть следующим образом:

0	97 'a'
1	98 'b'
2	114 'r'
3	97 'a'
4	107 'k'
5	97 'a'
6	100 'd'
7	97 'a'
8	98 'b'
9	114 'r'
10	97 'a'

После работы алгоритма BWT исходная строка будет преобразована в следующую последовательность, содержащую в себе преобразованную строку и позицию исходной строки в таблице отсортированных циклических сдвигов:

0	114 'r'
1	100 'd'
2	97 'a'
3	107 'k'
4	114 'r'
5	97 'a'
6	97 'a'
7	97 'a'
8	97 'a'
9	98 'b'
10	98 'b'
11	0 '\0'
12	0 '\0'
13	0 '\0'
14	2 '\x2'

После работы в начало добавляется размер закодированного буфера и добавлен в выходной временный файл вместе с преобразованной таблицей.
























Считываем буфер из временного файла:

[0]	15 '\xf'
[1]	0 '\0'
[2]	0 '\0'
[3]	0 '\0'
[4]	114 'r'
[5]	100 'd'
[6]	97 'a'
[7]	107 'k'
[8]	114 'r'
[9]	97 'a'
[10]	97 'a'
[11]	97 'a'
[12]	97 'a'
[13]	98 'b'
[14]	98 'b'
[15]	0 '\0'
[16]	0 '\0'
[17]	0 '\0'
[18]	2 '\x2'

После преобразования алгоритмом MTF отметим, что размер выходной последовательности не поменялся:

[0]	15 '\xf'
[1]	1 '\x1'
[2]	0 '\0'
[3]	0 '\0'
[4]	114 'r'
[5]	101 'e'
[6]	99 'c'
[7]	108 'l'
[8]	3 '\x3'
[9]	2 '\x2'
[10]	0 '\0'
[11]	0 '\0'
[12]	0 '\0'
[13]	101 'e'
[14]	0 '\0'
[15]	5 '\x5'
[16]	0 '\0'
[17]	0 '\0'
[18]	8 '\b'

Снова считаем буффер из временного файла:

 [0]	19 '\x13'
 [1]	0 '\0'
 [2]	0 '\0'
 [3]	0 '\0'
 [4]	15 '\xf'
 [5]	1 '\x1'
 [6]	0 '\0'
 [7]	0 '\0'
 [8]	114 'r'
 [9]	101 'e'
 [10]	99 'c'
 [11]	108 'l'
 [12]	3 '\x3'
 [13]	2 '\x2'
 [14]	0 '\0'
 [15]	0 '\0'
 [16]	0 '\0'
 [17]	101 'e'
 [18]	0 '\0'
 [19]	5 '\x5'
 [20]	0 '\0'
 [21]	0 '\0'
 [22]	8 '\b'

После работы RLE у нас вся последовательность преобразовалась в вид, в котором идут сначала либо положительные, либо отрицательные числа, после которых идут либо один символ, если число положительное, либо количество неповторяющихся символов равное взятому абсолютному значению числа:

[0]	-1 'я'
[1]	19 '\x13'
[2]	3 '\x3'
[3]	0 '\0'
[4]	-2 '\x0'
[5]	15 '\xf'
[6]	1 '\x1'
[7]	2 '\x2'
[8]	0 '\0'
[9]	-6 '\x6'
[10]	114 '\x7'
[11]	101 '\xe'
[12]	99 '\xc'
[13]	108 '\x8'
[14]	3 '\x3'
[15]	2 '\x2'
[16]	3 '\x3'
[17]	0 '\0'
[18]	-3 '\x3'
[19]	101 '\xe'
[20]	0 '\0'
[21]	5 '\x5'
[22]	2 '\x2'
[23]	0 '\0'
[24]	1 '\x1'
[25]	8 '\b'

Считываем буффер из временного файла для последнего преобразования:

🔍 [0]	26 '\x1a'
🔍 [1]	0 '\0'
🔍 [2]	0 '\0'
🔍 [3]	0 '\0'
🔍 [4]	-1 'я'
🔍 [5]	19 '\x13'
🔍 [6]	3 '\x3'
🔍 [7]	0 '\0'
🔍 [8]	-2 'ю'
🔍 [9]	15 '\xf'
🔍 [10]	1 '\x1'
🔍 [11]	2 '\x2'
🔍 [12]	0 '\0'
🔍 [13]	-6 'ь'
🔍 [14]	114 'r'
🔍 [15]	101 'e'
🔍 [16]	99 'c'
🔍 [17]	108 'l'
🔍 [18]	3 '\x3'
🔍 [19]	2 '\x2'
🔍 [20]	3 '\x3'
🔍 [21]	0 '\0'
🔍 [22]	-3 'э'
🔍 [23]	101 'e'
🔍 [24]	0 '\0'
🔍 [25]	5 '\x5'
🔍 [26]	2 '\x2'
🔍 [27]	0 '\0'
🔍 [28]	1 '\x1'
🔍 [29]	8 '\b'

После преобразования получилась длинная последовательность, во много раз превышающая исходный файл. Это произошло потому, что дерево сохраняется не оптимально и занимает в данном примере 65 байта. Формат хранения дерева: сначала следует символ с вершины файла, после чего следует либо 1, либо 0 в зависимости от направления перехода при обходе дерева:

[52]	-6 'ъ'
[53]	48 '0'
[54]	-6 'ъ'
[55]	48 '0'
[56]	-6 'ъ'
[57]	49 '1'
[58]	-2 'ю'
[59]	49 '1'
[60]	5 '\x5'
[61]	48 '0'
[62]	5 '\x5'
[63]	49 '1'
[64]	108 'l'
[65]	50 '2'
[66]	0 '\0'
[67]	0 '\0'
[68]	0 '\0'
[69]	30 '\x1e'
[70]	109 'm'
[71]	79 'O'
[72]	97 'a'
[73]	-70 'e'
[74]	-19 'h'
[75]	23 '\x17'
[76]	22 '\x16'
[77]	-118 'Ъ'
[78]	-7 'щ'
[79]	6 '\x6'
[80]	-62 'B'
[81]	94 '^'
[82]	22 '\x16'
[83]	-28 'д'

[0]	2 '\x2'
[1]	48 '0'
[2]	2 '\x2'
[3]	48 '0'
[4]	2 '\x2'
[5]	48 '0'
[6]	2 '\x2'
[7]	49 '1'
[8]	3 '\x3'
[9]	49 '1'
[10]	101 'e'
[11]	48 '0'
[12]	101 'e'
[13]	48 '0'
[14]	101 'e'
[15]	49 '1'
[16]	99 'c'
[17]	48 '0'
[18]	99 'c'
[19]	49 '1'
[20]	114 'r'
[21]	49 '1'
[22]	19 '\x13'
[23]	48 '0'
[24]	19 '\x13'
[25]	48 '0'
[26]	19 '\x13'
[27]	49 '1'
[28]	26 '\x1a'
[29]	49 '1'
[30]	15 '\xf'
[31]	48 '0'
[32]	15 '\xf'
[33]	49 '1'

[19]	49 '1'
[20]	114 'r'
[21]	49 '1'
[22]	19 '\x13'
[23]	48 '0'
[24]	19 '\x13'
[25]	48 '0'
[26]	19 '\x13'
[27]	49 '1'
[28]	26 '\x1a'
[29]	49 '1'
[30]	15 '\xf'
[31]	48 '0'
[32]	15 '\xf'
[33]	49 '1'
[34]	-1 'я'
[35]	49 '1'
[36]	0 '\0'
[37]	48 '0'
[38]	0 '\0'
[39]	49 '1'
[40]	-3 'я'
[41]	48 '0'
[42]	-3 'я'
[43]	48 '0'
[44]	-3 'я'
[45]	48 '0'
[46]	-3 'я'
[47]	49 '1'
[48]	8 '\b'
[49]	49 '1'
[50]	1 '\x1'
[51]	49 '1'

Итоговая последовательность записывается в результирующий файл.

3 Исходный код

Структура программы построена таким образом, что каждый класс, инкапсулирующий в себе реализацию алгоритма преобразования наследуется от общего класса **Compressor**, переопределяя его метод *buffer_encode* и *buffer_decode*.

Определение класса **Compressor** в *Compressor.hpp*:

```
1 class Compressor{
2     public:
3         Compressor();
4         Compressor(istream& inpt, ostream& otpt);
5         Compressor(istream* inpt, ostream* otpt);
6         void set_input(istream& inpt);
7         void set_input(istream* inpt);
8         void set_output(ostream* otpt);
9         void set_output(ostream& otpt);
10        uint64_t get_until_size();
11        uint64_t get_after_size();
12        void encode();
13        void decode();
14        virtual ~Compressor(){};
15
16    protected:
17        virtual void buffer_encode() = 0;
18        virtual void buffer_decode() = 0;
19        uint64_t until_size = 0;
20        uint64_t after_size = 0;
21        istream* is;
22        ostream* os;
23        string in_buffer;
24        string out_buffer;
25        const uint32_t max_buffer_size = 4194304;
26};
```

Класс **RLE** объявлен в *RLE.hpp* следующим образом:

```
1 class RLE : public Compressor{
2     public:
3         RLE();
4         RLE(istream& inpt, ostream& otpt);
5         RLE(istream* inpt, ostream* otpt);
6     protected:
7         void buffer_encode() override;
8         void buffer_decode() override;
9};
```

```
10 || };
```

Класс **MTF** из *MTF.hpp* содержит в себе список *alfabet*, реализующий в себе алгоритм, а так же 2 функции позволяющие получить код для входного символа:

```
1 | class MTF : public Compressor{
2 |     public:
3 |         MTF();
4 |         MTF(istream& inpt, ostream& otpt);
5 |         MTF(istream* inpt, ostream* otpt);
6 |
7 |     protected:
8 |         void buffer_encode() override;
9 |         void buffer_decode() override;
10 |    private:
11 |        list<uint8_t> alfabet;
12 |        uint8_t move_to_front_encode(uint8_t c);
13 |        uint8_t move_to_front_decode(uint8_t c);
14 |};
```

Класс **Huffman** из *Huffman.hpp* имеет функции для подсчета вхождений символа в файл, создания дерева, его чтения и записи, а также несколько вспомогательных функций и структур для работы с деревом.

```
1 | class Huffman : public Compressor{
2 |     public:
3 |         Huffman();
4 |         Huffman(istream& inpt, ostream& otpt);
5 |         Huffman(istream* inpt, ostream* otpt);
6 |     protected:
7 |         void buffer_encode() override;
8 |         void buffer_decode() override;
9 |     private:
10 |         typedef struct Node{
11 |             int32_t left;
12 |             int32_t right;
13 |             uint32_t count;
14 |             char value; // has value only if list, else it rubbish
15 |         }node;
16 |
17 |         // comparator for priority queue
18 |         typedef struct functor{
19 |             bool operator()(const pair<int32_t, uint32_t> lhs, const pair<int32_t,
20 |                 uint32_t> rhs){
21 |                 return lhs.second > rhs.second;
22 |             }
23 |         }comparator;
```



```

23
24 // TREE (too lazy create new structure):
25 int32_t tree = -1; // root
26 vector<node> vertex; // vector of vertexes of tree
27 // AND OF TREE
28
29 void tree_create(); // create tree of encoding
30 void tree_from_input(uint32_t& i); // create tree from input for decoding
31 void tree_write(int32_t obj); // write tree in output
32 void count_table(map<char, uint32_t>& table_of_counts); // count symbols
33 void create_encode_table(map<char, vector<uint8_t>>& encode_table);
34 void depth_walker(int32_t obj, vector<uint8_t>& vec, map<char, vector<uint8_t
    >>& mp);
35 };

```

Класс **BWT** из *BWT.hpp* содержит в себе функцию построения суффиксного массива и позицию исходной строки в таблице сдвигов.

```

1 class BWT : public Compressor{
2     public:
3         BWT();
4         BWT(istream& inpt, ostream& otpt);
5         BWT(istream* inpt, ostream* otpt);
6     protected:
7         void buffer_encode() override;
8         void buffer_decode() override;
9     private:
10        uint32_t position; // position in sorted cyclic table
11        uint32_t count_suff_array(vector<uint32_t>& array); // returns position
12 };

```

```

1 #include "BWT.hpp"
2
3 using namespace std;
4
5
6
7 BWT::BWT() : Compressor(){}
8
9 BWT::BWT(istream& inpt, ostream& otpt) : Compressor(inpt, otpt){}
10
11 BWT::BWT(istream* inpt, ostream* otpt) : Compressor(inpt, otpt){}
12
13 // O(nlogn) construct from https://e-maxx.ru/algo/suffix_array
14 // it will array of sorted cyclics - GREAT
15
16 uint32_t BWT::count_suff_array(vector<uint32_t>& array){

```

```

17 uint32_t n = in_buffer.size();
18 array.resize(n);
19
20 // equalent classes
21 vector<uint32_t> equality(n);
22 uint32_t classes = 0;
23
24 // help vectors
25 vector<uint32_t> second_array(n), helper(n);
26
27 // second_array - sorted positions of second parts of new substrings
28 // helper - new classes of equality
29
30 // we coding in_buffer fully
31 vector<uint32_t> count(256, 0);
32 // 0 phase - sort chars in string with count sort
33 for(uint32_t i = 0; i < n; ++i){
34     ++count[static_cast<uint8_t>(in_buffer[i])];
35 }
36 for(uint32_t i = 1; i < 256; ++i){
37     count[i] += count[i - 1];
38 }
39 for(uint32_t i = 0; i < n; ++i){
40     array[--count[static_cast<uint8_t>(in_buffer[i])]] = i;
41 }
42 equality[array[0]] = 0;
43
44 for(uint32_t i = 1; i < n; ++i){
45     if(in_buffer[array[i]] != in_buffer[array[i-1]]){
46         ++classes;
47     }
48     equality[array[i]] = classes;
49 }
50 ++classes;
51
52 // all another k phases (k = h + 1)
53
54 for(uint32_t h = 0; (1u << h) < n; ++h){
55     // second_array - includes old sort positions for string with lenght 2^(k-1)
56     // now we should get new sort positions for string with lenght 2^k
57     // for this we can do radix sort for strings S_k[i] = (S_(k-1)[i], S_(k-1)[i
58         // + 2^(k-1)])
59     // but sort of strings S_k[i] by second positions we can get as second_array[i]
60         // = p[i] - 2^(k-1)
61     for(uint32_t i = 0; i < n; ++i){
62         second_array[i] = array[i] < (1u << h) ? array[i] + (n - (1 << h)) : array[
            i] - (1 << h);
63     }
64 }

```

```

63     count.assign(classes, 0);
64
65     // and count sort by first positions
66     for(uint32_t i = 0; i < n; ++i){
67         ++count[equality[second_array[i]]];
68     }
69     for(uint32_t i = 1; i < classes; ++i){
70         count[i] += count[i - 1];
71     }
72     for(uint32_t i = n - 1; i > 0; --i){
73         array[--count[equality[second_array[i]]]] = second_array[i];
74     }
75     array[--count[equality[second_array[0]]]] = second_array[0];
76
77     helper[array[0]] = 0;
78     classes = 0;
79
80     // define classes of equality
81     for(uint32_t i = 1; i < n; ++i){
82         // second - second position of new strings
83         uint32_t second1 = (array[i] + (1u << h)) >= n ?
84             array[i] + (1u << h) - n :
85             array[i] + (1u << h);
86         uint32_t second2 = array[i - 1] + (1u << h) >= n ?
87             array[i - 1] + (1u << h) - n :
88             array[i - 1] + (1u << h);
89
90         if(equality[array[i]] != equality[array[i - 1]] || equality[second1] !=
91             equality[second2]){
92             ++classes;
93         }
94         helper[array[i]] = classes;
95     }
96     ++classes;
97     equality = helper;
98     return equality[0];
99 }
100
101 void BWT::buffer_encode(){
102     vector<uint32_t> suf_array;
103     position = count_suff_array(suf_array);
104     for(uint32_t i = 0; i < in_buffer.size(); ++i){
105         if(suf_array[i] > 0){
106             out_buffer.push_back(in_buffer[suf_array[i] - 1]);
107         }else{
108             out_buffer.push_back(in_buffer[in_buffer.size() - 1]);
109         }
110     }

```

```

111
112     uint32_t pos = position;
113     uint32_t s_mask = 1 << 8; --s_mask; // 00000000000000000000000011111111
114
115     uint8_t byte4 = static_cast<uint8_t>(pos & s_mask);
116     pos >>= 8;
117     uint8_t byte3 = static_cast<uint8_t>(pos & s_mask);
118     pos >>= 8;
119     uint8_t byte2 = static_cast<uint8_t>(pos & s_mask);
120     pos >>= 8;
121     uint8_t byte1 = static_cast<uint8_t>(pos & s_mask);
122
123     out_buffer.push_back(static_cast<char>(byte1));
124     out_buffer.push_back(static_cast<char>(byte2));
125     out_buffer.push_back(static_cast<char>(byte3));
126     out_buffer.push_back(static_cast<char>(byte4));
127 }
128
129 // TO DO:
130 // change algo for stay in_buffer without changes
131
132 void BWT::buffer_decode(){
133     // Reading position number from end of file and pop this number for stay in_buffer
134     // as string for decode
135     position = 0;
136     for(uint8_t i = 0; i < sizeof(uint32_t); ++i){
137         uint8_t byte = static_cast<uint8_t>(in_buffer[in_buffer.size() - 1 - i]);
138         position |= (static_cast<uint32_t>(byte) << (i * 8));
139     }
140
141     // DECODE:
142     // this algo i get from https://neerc.ifmo.ru/wiki/index.php?title=-
143     const uint32_t alphabet = 256;
144     const uint32_t n = in_buffer.size() - sizeof(uint32_t);
145     vector<uint32_t> count(alphabet, 0);
146     vector<uint32_t> t(n);
147
148     // count sort in_buffer and save positions in t
149     for(uint32_t i = 0; i < n; ++i){
150         ++count[static_cast<uint8_t>(in_buffer[i])];
151     }
152
153     uint32_t sum = 0;
154     for(uint32_t i = 0; i < alphabet; ++i){
155         sum += count[i];
156         count[i] = sum - count[i];
157     }
158

```

```

159     for(uint32_t i = 0; i < n; ++i){
160         t[count[static_cast<uint8_t>(in_buffer[i])]]++ = i;
161     }
162
163     // decode in_buffer using t in out_buffer
164     out_buffer.resize(n);
165     uint32_t j = t[position];
166     for(uint32_t i = 0; i < n; ++i){
167         out_buffer[i] = in_buffer[j];
168         j = t[j];
169     }
170 }

```

```

1  #include "Huffman.hpp"
2
3  using namespace std;
4
5
6  Huffman::Huffman() : Compressor({})
7
8  Huffman::Huffman(istream& inpt, ostream& otpt) : Compressor(inpt, otpt){}
9
10 Huffman::Huffman(istream* inpt, ostream* otpt) : Compressor(inpt, otpt){}
11
12
13 void Huffman::buffer_decode(){
14     tree = -1;
15     vertex.clear();
16     uint32_t i = 0;
17     // get tree from input
18     tree_from_input(i);
19
20     uint32_t size = 0;
21
22     // Reading size of decoded buffer
23     for(uint8_t j = 0; j < 4; ++j){
24         // read 1 byte
25         uint8_t byte = static_cast<uint8_t>(in_buffer[i++]);
26         size |= static_cast<uint32_t>(byte) << (8*(3 - j)); // insert bits on needed
           positions
27     }
28
29     // DECODE:
30     // counter of symbols needs for stop decode
31     uint32_t j = 0;
32     int32_t it = tree;
33     for(; i < in_buffer.size(); ++i){
34         // reading byte of information:

```

```

35     uint8_t bits_of_code = in_buffer[i];
36     // create mask for reading bits
37     uint8_t mask = 1 << 7; // 10000000
38     // reading bits
39     for(uint8_t k = 0; k < 8 && j < size; ++k){
40         // if bit - 1 => go right
41         if(bits_of_code & mask){
42             it = vertex[it].right;
43         }else{
44             // if bit - 0 => go left
45             it = vertex[it].left;
46         }
47         // if in list => push decoded simbol in output
48         if(vertex[it].left < 0){
49             out_buffer.push_back(vertex[it].value);
50             // and go back in root
51             it = tree;
52             // apdate counter of simbols
53             ++j;
54         }
55         // update mask for read next bit
56         mask >>= 1;
57     }
58 }
59 }
60
61
62 void Huffman::buffer_encode(){
63     tree = -1;
64     vertex.clear();
65     // At first create tree for encode
66     tree_create();
67     // At second write tree in file
68     tree_write(tree);
69     // it simbol for end of tree
70     // cant do without this but too lazy
71     out_buffer.push_back('2'); // simbol of tree end
72
73     // Create table of simbols codes:
74     map<char, vector<uint8_t>> encode_table;
75     create_encode_table(encode_table);
76
77     // Save size of buffer:
78
79     uint32_t size = in_buffer.size();
80
81     uint32_t s_mask = 1 << 8; --s_mask;
82
83     uint8_t byte4 = static_cast<uint8_t>(size & s_mask);

```

```

84     size >= 8;
85     uint8_t byte3 = static_cast<uint8_t>(size & s_mask);
86     size >= 8;
87     uint8_t byte2 = static_cast<uint8_t>(size & s_mask);
88     size >= 8;
89     uint8_t byte1 = static_cast<uint8_t>(size & s_mask);
90
91
92     out_buffer.push_back(static_cast<char>(byte1));
93
94     out_buffer.push_back(static_cast<char>(byte2));
95
96     out_buffer.push_back(static_cast<char>(byte3));
97
98     out_buffer.push_back(static_cast<char>(byte4));
99
100
101
102     // ENCODING:
103
104     // byte for put in output
105     uint8_t bits_of_code = 0;
106     uint8_t mask = 1 << 7; // 10000000
107     // reading symbols for encode
108     for(uint32_t i = 0; i < in_buffer.size(); ++i){
109         // get code of input symbol
110         vector<uint8_t>& simbol_code = encode_table[in_buffer[i]];
111         // insert code in byte
112         for(uint8_t k = 0; k < simbol_code.size(); ++k){
113             // if bit code - 1 => put 1 on needed position (else nothing to do)
114             if(simbol_code[k]){
115                 bits_of_code |= mask;
116             }
117             // get new position for bit
118             mask >>= 1;
119             // if mask - 00000000 => put encoded byte in output and go to encode new
byte
120             if(!mask){
121                 out_buffer.push_back(static_cast<char>(bits_of_code));
122
123                 mask = 1 << 7;
124                 bits_of_code = 0;
125             }
126         }
127     }
128     // if we encoded last symbols but didnt wrote => writing them
129     if(mask < (1 << 7)){
130         out_buffer.push_back(static_cast<char>(bits_of_code));
131     }

```

```

132 }
133
134
135 // this function needs for walking in tree and construct code table of symbols
136 void Huffman::depth_walker(int32_t obj ,vector<uint8_t>& vec, map<char, vector<uint8_t
    >>& mp){
137     // Wlking in tree and collect way in vector
138     // And after that save way for all alfabet symbols
139     if(vertex[obj].left >= 0){
140         vec.push_back(0);
141         depth_walker(vertex[obj].left, vec, mp);
142         vec.pop_back();
143         vec.push_back(1);
144         depth_walker(vertex[obj].right, vec, mp);
145         vec.pop_back();
146     }else{
147         mp[vertex[obj].value] = vec;
148     }
149 }
150
151
152 void Huffman::create_encode_table(map<char, vector<uint8_t>>& encode_table){
153     // DEPTH WALK FOR CREATE ENCODE TABLE
154     vector<uint8_t> vec;
155     depth_walker(tree, vec, encode_table);
156 }
157
158
159 void Huffman::tree_write(int32_t obj){
160     // Main idea: when we go left put 0
161     // when we come to node put value of node
162     // when go right put 1
163     // P.S. something like this i made in lab2
164
165     // value
166     out_buffer.push_back(vertex[obj].value);
167
168     if(vertex[obj].left >= 0){
169         out_buffer.push_back('0');
170
171         tree_write(vertex[obj].left);
172     }else{
173         // if left - null => right - null in our tree
174         return;
175     }
176     out_buffer.push_back('1');
177     tree_write(vertex[obj].right);
178 }
179

```



```

180 void Huffman::tree_from_input(uint32_t& i){
181     // it very bed code for unerstning but i will try to explain:
182     // We construct tree by path symbols: 0 - go left, 1 - go right, 2 - end of
        building
183     // this symbols - instruction to go in depth walking
184
185     // in structure of tree we have not useful for us attribute: count
186     // we will save in it index of vertex for up move in depth walking
187     // (it not good idea may be, but working)
188
189     // At first create root:
190     vertex.push_back(Huffman::node{-1, -1, 0, in_buffer[i++]});
191     tree = static_cast<int32_t>(vertex.size() - 1);
192     int32_t it = tree;
193
194     // depth walking:
195     while(1){
196         // if go left:
197         // 1) create new node
198         // 2) save in new node index of father in vertex vector
199         // 3) init left and go
200         if(in_buffer[i] == '0'){
201             ++i;
202             vertex.push_back(Huffman::node{-1, -1, static_cast<uint32_t>(it), in_buffer
                [i++]});
203             // create left
204             vertex[it].left = static_cast<int32_t>(vertex.size() - 1);
205             // go left
206             it = vertex[it].left;
207             continue;
208         }
209         // if go right:
210         // 1) move up to vertex for create right child for go right
211         // 2) create new Huffman::node
212         // 3) set here upper travel as upper travel for her father
213         // 4) init right and go
214         if(in_buffer[i] == '1'){
215             ++i;
216             // move up
217             it = vertex[it].count;
218             // here set father travel
219             vertex.push_back(Huffman::node{-1, -1, vertex[it].count, in_buffer[i++]});
220             // create node
221             vertex[it].right = vertex.size() - 1;
222             // go right
223             it = vertex[it].right;
224         }
225         // if read 2 => end of walking
226         if(in_buffer[i] == '2'){

```

```

227         ++i;
228         return;
229     }
230 }
231 }
232
233 void Huffman::tree_create(){
234     // table of counts simbol repeats in buffer
235     map<char, uint32_t> table_of_counts;
236     count_table(table_of_counts);
237
238     // priority queue for Huffman algo(may be not too fast as custom, but create custom
239     // lazy)
240     //priority_queue<Huffman::node, vector<Huffman::node>, comparator> que;
241     priority_queue<pair<int32_t, uint32_t>, vector<pair<int32_t, uint32_t>>, comparator
242     > que;
243
244     // create and insert basic nodes(lists) in queue
245     for(auto it : table_of_counts){
246         vertex.push_back(Huffman::node{-1, -1, it.second, it.first});
247         que.push(pair<int32_t, uint32_t>(vertex.size()-1, vertex.back().count));
248     }
249
250     // situation when only 1 simbol in alfabet: will encode all simbols as 0
251     // it exception from rule:)
252     if(que.size() == 1){
253         vertex.push_back(Huffman::node{-1, -1, vertex[0].count, static_cast<char>(
254             vertex[0].value+1)});
255         vertex.push_back(Huffman::node{0, 1, vertex[0].count, vertex[0].value});
256         tree = 2;
257         return;
258     }
259
260     // build tree from queue: itteration while not 1 elem in queue
261     while(que.size() > 1){
262         // take 2 nodes with less counts
263         int32_t left = que.top().first; que.pop();
264         int32_t right = que.top().first; que.pop();
265         // create new node with left and right childs
266         vertex.push_back(Huffman::node{left, right, vertex[left].count + vertex[right].
267             count, vertex[left].value});
268         // push new node to queue
269         que.push(pair<int32_t, uint32_t>(vertex.size()-1, vertex.back().count));
270     }
271
272     tree = que.top().first;
273     que.pop(); // clear queue(not need)
274 }

```

```

272 |
273 | // count table of repeats in buffer
274 | void Huffman::count_table(map<char, uint32_t>& table_of_counts){
275 |     for(uint32_t i = 0; i < in_buffer.size(); ++i){
276 |         ++table_of_counts[in_buffer[i]];
277 |     }
278 | }

1 | #include "MTF.hpp"
2 |
3 | using namespace std;
4 |
5 | // TO DO
6 |
7 | // change algo for do possible count and insert numbers more than 127
8 |
9 | MTF::MTF() : Compressor({})
10 |
11 | MTF::MTF(istream& inpt, ostream& otpt) : Compressor(inpt, otpt){}
12 |
13 | MTF::MTF(istream* inpt, ostream* otpt) : Compressor(inpt, otpt){}
14 |
15 | void MTF::buffer_encode(){
16 |     alfabet.clear();
17 |     // init alfabet
18 |     for(uint16_t i = 0; i < 256; ++i){
19 |         alfabet.push_back(static_cast<uint8_t>(i));
20 |     }
21 |
22 |     // ENCODE:
23 |     for(char c : in_buffer){
24 |         out_buffer.push_back(
25 |             static_cast<char>(
26 |                 move_to_front_encode(
27 |                     static_cast<uint8_t>(c)
28 |                 )
29 |             )
30 |         );
31 |     }
32 | }
33 |
34 | void MTF::buffer_decode(){
35 |     alfabet.clear();
36 |     // init alfabet:
37 |     for(uint16_t i = 0; i < 256; ++i){
38 |         alfabet.push_back(static_cast<uint8_t>(i));
39 |     }
40 | }

```

```

41 | // DECODE:
42 | for(char c : in_buffer){
43 |     out_buffer.push_back(
44 |         static_cast<char>(
45 |             move_to_front_decode(
46 |                 static_cast<uint8_t>(c)
47 |             )
48 |         )
49 |     );
50 | }
51 | }
52 |
53 | uint8_t MTF::move_to_front_encode(uint8_t c){
54 |     uint8_t i = 0;
55 |     for (auto it = alfabet.begin(); ; ++it, ++i){
56 |         if(*it == c){
57 |             alfabet.erase(it);
58 |             break;
59 |         }
60 |     }
61 |     alfabet.push_front(c);
62 |     return i;
63 | }
64 |
65 |
66 | uint8_t MTF::move_to_front_decode(uint8_t c){
67 |     auto it = alfabet.begin();
68 |     advance(it, c);
69 |     uint8_t i = *it;
70 |     alfabet.erase(it);
71 |     alfabet.push_front(i);
72 |     return i;
73 | }

1 | #include "RLE.hpp"
2 |
3 | using namespace std;
4 |
5 | // TO DO
6 |
7 | // change algo for do possible count and insert numbers more than 127
8 |
9 | RLE::RLE() : Compressor({})
10 |
11 | RLE::RLE(istream& inpt, ostream& otpt) : Compressor(inpt, otpt){}
12 |
13 | RLE::RLE(istream* inpt, ostream* otpt) : Compressor(inpt, otpt){}
14 |

```

```

15 void RLE::buffer_encode(){
16     int8_t count = 0;
17     string buffer;
18     buffer.clear();
19     char last_ch = in_buffer[0]; // set unique
20     // Always save last char from in buffer
21     for(uint32_t i = 1; i < in_buffer.size(); ++i){
22         // if repeat
23         if(in_buffer[i] == last_ch){
24             // if its first repeat, push in output unique char until repeat chars
25             if(count < 0){
26                 out_buffer.push_back(static_cast<char>(count));
27                 out_buffer += buffer;
28                 buffer.clear();
29                 count = 0;
30             }
31             // we have +1 repeating char - last char
32             ++count;
33             // if num of repeats until this char = 127 => push back this info and go
               next
34             if(count == INT8_MAX){
35                 out_buffer.push_back(static_cast<char>(count));
36                 out_buffer.push_back(last_ch);
37                 count = 0;
38             }
39         }else{ // - if not repeat
40             // if its first unique char => push count and char of repeats
41             if(count > 0){
42                 // count + 1 because last char also repeated
43                 ++count;
44                 out_buffer.push_back(static_cast<char>(count));
45                 out_buffer.push_back(last_ch);
46                 count = 0;
47             }else{
48                 // if its not first unique char - push_back it in temp buffer
49                 // and increment count of unique
50                 --count;
51                 buffer.push_back(last_ch);
52                 if (count == INT8_MIN){
53                     out_buffer.push_back(static_cast<char>(count));
54                     out_buffer += buffer;
55                     buffer.clear();
56                     count = 0;
57                 }
58             }
59         }
60     }
61     last_ch = in_buffer[i];
62 }

```

```

63 // we should define group for last char of input and
64 // push nesassary info in output
65 if(count < 0){
66     --count;
67     buffer.push_back(last_ch);
68     out_buffer.push_back(static_cast<char>(count));
69     out_buffer += buffer;
70 }else{
71     ++count;
72     out_buffer.push_back(static_cast<char>(count));
73     out_buffer.push_back(last_ch);
74 }
75
76 }
77
78 void RLE::buffer_decode(){
79     uint32_t i = 0;
80     int8_t count = 0;
81     while(i < in_buffer.size()){
82         count = static_cast<int8_t>(in_buffer[i++]);
83         while(count < 0){
84             out_buffer.push_back(in_buffer[i++]);
85             ++count;
86         }
87         if(count > 0){
88             char tmp = in_buffer[i++];
89             while (count > 0){
90                 out_buffer.push_back(tmp);
91                 --count;
92             }
93         }
94     }
95 }

```

```

1  #include "Arhivator.hpp"
2  #include <cstdio>
3
4  using namespace std;
5
6  Arhivator::Arhivator(){
7      stdinput = true;
8      decode = false;
9      stdoutput = false;
10     hard = false;
11     easy = false;
12     keep = false;
13     recursive = false;
14     check = false;

```

```

15     information = false;
16     path.clear();
17 }
18
19 void Arhivator::set_stdoutput(){
20     stdoutput = true;
21 }
22
23 void Arhivator::set_path(string& str){
24     path = str;
25     if(str.empty()){
26         stdinput = true;
27     }else{
28         stdinput = false;
29     }
30 }
31
32 void Arhivator::set_hard(){
33     hard = true;
34     if(easy){
35         hard = false;
36     }
37 }
38
39 void Arhivator::set_easy(){
40     easy = true;
41     if(hard){
42         hard = false;
43     }
44 }
45
46 void Arhivator::set_recursive(){
47     recursive = true;
48 }
49
50 void Arhivator::set_information(){
51     information = true;
52 }
53
54 void Arhivator::set_check(){
55     check = true;
56 }
57
58 void Arhivator::set_keep(){
59     keep = true;
60 }
61
62 void Arhivator::set_decode(){
63     decode = true;

```

```

64 }
65
66 void Arhivator::start(){
67     if(stdinput){
68         stdoutput = true;
69     }
70     if(information){
71         if(stdinput){
72             check_info(cin);
73         }else{
74             ifstream is(path, ios::in | ios::binary);
75             if(!is){
76                 throw MyException(path + " not a file");
77             }
78             check_info(is);
79             is.close();
80         }
81         return;
82     }
83     if(check){
84         if(stdinput){
85             check_zip(cin);
86         }else{
87             ifstream is(path, ios::in | ios::binary);
88             if(!is){
89                 throw MyException(path + " not a file");
90             }
91             if(check_zip(is)){
92                 cout << "True" << endl;
93             }else{
94                 cout << "False" << endl;
95             }
96             is.close();
97         }
98         return;
99     }
100     if(recursive){
101         throw MyException("Not Ready Yet");
102     }
103     if(hard || easy){
104         throw MyException("Not Ready Yet");
105     }
106     if(decode){
107         decode_file(path);
108         return;
109     }else{
110         encode_file(path);
111         return;
112     }

```



```

113 }
114
115 void Arhivator::encode_file(string& path){
116     ifstream inpt(path, ios::in | ios::binary);
117     if(!inpt){
118         throw MyException(path + " is not a file");
119     }
120     if(!stdoutput){
121         ofstream otpt(path + ".gz", ios::out | ios::binary);
122         encode_stream(inpt, otpt);
123         otpt.close();
124         if(!keep){
125             remove(path.c_str());
126         }
127     }else{
128         encode_stream(inpt, cout);
129         if(!keep){
130             remove(path.c_str());
131         }
132     }
133 }
134
135 void Arhivator::decode_file(string& path){
136     string suf = path.substr(path.size() - 3, 3);
137     if(suf != ".gz"){
138         throw MyException(path + " has unknown suffix");
139     }
140     ifstream inpt(path, ios::in | ios::binary);
141     if(!inpt){
142         throw MyException(path + " is not a file");
143     }
144
145     if(!stdoutput){
146         ofstream otpt(path.substr(0, path.size() - 3), ios::out | ios::binary | ios::
            trunc);
147         decode_stream(inpt, otpt);
148         otpt.close();
149         if(!keep){
150             remove(path.c_str());
151         }
152     }else{
153         decode_stream(inpt, cout);
154         if(!keep){
155             remove(path.c_str());
156         }
157     }
158 }
159
160 void Arhivator::check_info(istream& is){

```

```

161     string pre;
162     pre.resize(prefix.size());
163     uint64_t old_size, new_size;
164     is.read(&pre[0], prefix.size());
165     if(prefix != pre){
166         throw MyException("not in zip format");
167     }
168     is.read(reinterpret_cast<char *>(&old_size), sizeof(uint64_t));
169     is.read(reinterpret_cast<char *>(&new_size), sizeof(uint64_t));
170     if(!is || !check_hash(is)){
171         throw MyException("not in zip format");
172     }
173     cout << "Unompressed size: " << old_size << endl;
174     cout << "Compressed size: " << new_size << endl;
175     cout << "Ratio: "
176     << 100.0 * static_cast<double>(old_size - new_size)
177     / static_cast<double>(old_size) << "%" << endl;
178 }
179
180 // will coding streams:
181 // is - steam of input file(or stdin), os -steam of output file(or stdout)
182 void Arhivator::encode_stream(istream& is, ostream& os){
183     Compressor* cmp;
184     string tmpnm = path;
185     string file1 = tmpnm + ".1";
186     string file2 = tmpnm + ".2";
187     string file3 = tmpnm + ".3";
188     string file4 = tmpnm + ".4";
189     ifstream tmp_in;
190     ofstream tmp_out;
191     uint64_t input_size = 0;
192     uint64_t output_size = 0;
193     uint32_t hash_c = 0;
194
195     // ENCODING:
196
197     // 1 step
198     tmp_out.open(file1, ios::out | ios::binary);
199     cmp = new BWT(is, tmp_out);
200     cmp->encode();
201     input_size = cmp->get_until_size();
202     delete cmp;
203     tmp_out.close(); // TODO exceptions
204
205     // 2 step
206     tmp_in.open(file1, ios::in | ios::binary);
207     tmp_out.open(file2, ios::out | ios::binary);
208     cmp = new MTF(tmp_in, tmp_out);
209     cmp->encode();

```

```

210     delete cmp;
211     tmp_in.close();
212     tmp_out.close();
213     remove(file1.c_str());
214
215     // 3 step
216     tmp_in.open(file2, ios::in | ios::binary);
217     tmp_out.open(file3, ios::out | ios::binary);
218     cmp = new RLE(tmp_in, tmp_out);
219     cmp->encode();
220     delete cmp;
221     tmp_in.close();
222     tmp_out.close();
223     remove(file2.c_str());
224
225     // 4 step
226     tmp_in.open(file3, ios::in | ios::binary);
227     tmp_out.open(file4, ios::out | ios::binary);
228     cmp = new Huffman(tmp_in, tmp_out);
229     cmp->encode();
230     // size of encoded input
231     output_size = cmp->get_after_size();
232     delete cmp;
233     tmp_in.close();
234     tmp_out.close();
235     remove(file3.c_str());
236
237     // gen and add info into output
238     // 1 - add constant prefix for check gzip format
239     os.write(prefix.c_str(), prefix.size());
240     // 2 - add old size of file
241     os.write(reinterpret_cast<const char*>(&input_size), sizeof(uint64_t));
242     // 3 - count new size of file
243     // encoded inpt + preix size + 2 sizes + hash of file
244     output_size += 2 * sizeof(uint64_t) + prefix.size() + sizeof(uint32_t);
245     os.write(reinterpret_cast<const char*>(&output_size), sizeof(uint64_t));
246     // 4 - add hash
247     tmp_in.open(file4, ios::in | ios::binary);
248     hash_c = hash_count(tmp_in);
249     // return to start pos
250     tmp_in.close();
251
252     os.write(reinterpret_cast<const char*>(&hash_c), sizeof(uint32_t));
253
254     // 5 - insert encoed data at least of file
255     tmp_in.open(file4, ios::in | ios::binary);
256     from_stream_to_stream(tmp_in, os);
257     // and remove old file
258     tmp_in.close();

```

```

259     remove(file4.c_str());
260 }
261
262 void Arhivator::decode_stream(istream& is, ostream& os){
263     Compressor* cmp;
264     string tmpnm = path;
265     string file0 = tmpnm + ".0";
266     string file1 = tmpnm + ".1";
267     string file2 = tmpnm + ".2";
268     string file3 = tmpnm + ".3";
269     ifstream tmp_in;
270     ofstream tmp_out;
271
272     // CHECK ZIP FORMAT OF FILE
273
274     // Writing all stream in tmp file
275     tmp_out.open(file0, ios::out | ios::binary);
276     from_stream_to_stream(is, tmp_out);
277     tmp_out.close();
278
279     // Check zip format and take pointer to start of encoded data
280     // after special symbols and numbers
281     tmp_in.open(file0, ios::in | ios::binary);
282
283     if(!check_zip(tmp_in)){
284         throw MyException("not a zip formt");
285     }
286
287     tmp_in.close();
288
289
290     tmp_in.open(file0, ios::in | ios::binary);
291
292     tmp_in.seekg(2*sizeof(uint64_t) + prefix.size() + sizeof(uint32_t), ios::beg);
293
294
295     // DECODING:
296
297     // Decode all characters after specil simbols
298
299     // 1 step
300     tmp_out.open(file1, ios::out | ios::binary);
301     cmp = new Huffman(tmp_in, tmp_out);
302     cmp->decode();
303     delete cmp;
304     tmp_out.close(); // TODO exceptions
305     tmp_in.close();
306     remove(file0.c_str());
307

```

```

308 // 2 step
309 tmp_in.open(file1, ios::in | ios::binary);
310 tmp_out.open(file2, ios::out | ios::binary);
311 cmp = new RLE(tmp_in, tmp_out);
312 cmp->decode();
313 delete cmp;
314 tmp_in.close();
315 tmp_out.close();
316 remove(file1.c_str());
317
318 // 3 step
319 tmp_in.open(file2, ios::in | ios::binary);
320 tmp_out.open(file3, ios::out | ios::binary);
321 cmp = new MTF(tmp_in, tmp_out);
322 cmp->decode();
323 delete cmp;
324 tmp_in.close();
325 tmp_out.close();
326 remove(file2.c_str());
327
328 // 4 step decoding in output
329 tmp_in.open(file3, ios::in | ios::binary);
330 cmp = new BWT(tmp_in, os);
331 cmp->decode();
332 delete cmp;
333 tmp_in.close();
334 remove(file3.c_str());
335 }
336
337 bool Arhivator::check_hash(istream& is){
338     uint32_t hash_f = 0;
339     if(is){
340         is.read(reinterpret_cast<char*>(&hash_f), sizeof(uint32_t));
341     }
342     uint32_t hash_c = hash_count(is);
343     return hash_f == hash_c;
344 }
345
346 bool Arhivator::check_zip(istream& is){
347     string buffer;
348     buffer.resize(prefix.size());
349     if(is){
350         is.read(&buffer[0], prefix.size());
351         if(buffer != prefix){
352             return false;
353         }
354     }
355     uint64_t placebo;
356     if(is){

```

```

357         is.read(reinterpret_cast<char*>(&placebo), sizeof(uint64_t));
358     }
359     if(is){
360         is.read(reinterpret_cast<char*>(&placebo), sizeof(uint64_t));
361     }
362     return check_hash(is);
363 }
364
365 uint32_t Arhivator::hash_count(istream& is){
366     string buffer;
367     uint32_t ans = 0;
368     buffer.resize(max_size_buffer);
369     std::hash<string> func;
370     while(is){
371         is.read(&buffer[0], max_size_buffer);
372         ans += func(buffer);
373     }
374     return ans;
375 }
376
377 void Arhivator::from_stream_to_stream(istream& is, ostream& os){
378     string bufer;
379     bufer.resize(max_size_buffer);
380     while(is){
381         is.read(&bufer[0], max_size_buffer);
382         os.write(bufer.c_str(), is.gcount());
383     }
384 }

```



```

1  class BWT : public Compressor{
2      public:
3          BWT();
4          BWT(istream& inpt, ostream& otpt);
5          BWT(istream* inpt, ostream* otpt);
6      protected:
7          void buffer_encode() override;
8          void buffer_decode() override;
9      private:
10         uint32_t position; // position in sorted cyclic table
11         uint32_t count_suff_array(vector<uint32_t>& array); // returns position
12 };

```



```

1  // Made by Max Bronnikov
2  #ifndef ARHIVATOR_H
3  #define ARHIVATOR_H
4
5  #include <iostream>

```

```

6  #include <fstream>
7  #include <exception> // std::exception
8  #include "Compressors/Compressor.hpp"
9  #include "Compressors/RLE/RLE.hpp"
10 #include "Compressors/MTF/MTF.hpp"
11 #include "Compressors/Huffman/Huffman.hpp"
12 #include "Compressors/BWT/BWT.hpp"
13
14 using namespace std;
15
16
17
18 class MyException: public std::exception
19 {
20 private:
21     string m_error;
22
23 public:
24     MyException(string error) : m_error(error){}
25
26     const char* what() const noexcept { return m_error.c_str(); }
27 };
28
29
30
31 class Arhivator{
32     public:
33         Arhivator();
34         void set_decode();
35         void set_stdoutoutput();
36         void set_path(string& str);
37         void set_hard();
38         void set_easy();
39         void set_keep();
40         void set_recursive();
41         void set_check();
42         void set_information();
43         void start();
44
45     private:
46         void decode_file(string& path);
47         void encode_file(string& path);
48         void check_info(istream& is);
49         // void decode_dir(const string& path);
50         // void encode_dir(const string& path);
51         void encode_stream(istream& is, ostream& os);
52         void decode_stream(istream& is, ostream& os);
53         void from_stream_to_stream(istream& is, ostream& os);
54         uint32_t hash_count(istream& is);

```

```

55     bool check_hash(istream& is);
56     bool check_zip(istream& is);
57
58     bool stdinput; // without filename
59
60     bool stdoutput; // -c
61     bool hard; // -9
62     bool easy; // -1
63     bool keep; // -k
64     bool recursive; // -r
65     bool decode; // -d
66     bool check; // -t
67     bool information; // -l
68
69     uint32_t max_size_buffer = 4194304; // size of bufer for exchage and hash
70     counting
71     string path; // file or directory path
72     const string prefix = "\3\2\1\0"; // for check format
73 };
74
75 #endif

```

```

1 // Made By Max Bronnikov
2 #include <iostream>
3 #include <vector>
4 #include <string>
5 #include "Arhivator/Arhivator.hpp"
6
7 using namespace std;
8
9 // Demo of using Arhivator class
10
11 // This Demo light version of gzip
12 int main(int argc, char const *argv[])
13 {
14     if(argc < 2){
15         cout << "gzip: compressed data not written to a terminal." << endl;
16         cout << "For help, type: gzip -h" << endl;
17         return 0;
18     }
19     vector<string> vector_of_path;
20     Arhivator arhive;
21
22     for(int i = 1; i < argc; ++i){
23         if(argv[i][0] == '-'){
24             switch (argv[i][1]){
25                 case 'd':

```



```

26         archive.set_decode();
27         break;
28     case '1':
29         archive.set_easy();
30         break;
31     case '9':
32         archive.set_hard();
33         break;
34     case 'l':
35         archive.set_information();
36         break;
37     case 't':
38         archive.set_check();
39         break;
40     case 'r':
41         archive.set_recursive();
42         break;
43     case 'k':
44         archive.set_keep();
45         break;
46     case 'c':
47         archive.set_stdoutput();
48         break;
49     case '\\0':
50         vector_of_path.push_back(string());
51         break;
52     default:
53         break;
54     }
55     }else{
56         vector_of_path.push_back(argv[i]);
57     }
58 }
59 if(vector_of_path.empty()){
60     vector_of_path.push_back(string());
61 }
62
63 try{
64     for(unsigned i = 0; i < vector_of_path.size(); ++i){
65         archive.set_path(vector_of_path[i]);
66         archive.start();
67     }
68 }catch(std::exception &expection){
69     cerr << "gzip: " << expection.what() << endl;
70     return 0;
71 }
72
73 return 0;
74 }

```

Полный код на моем *GitHub*: <https://github.com/Bronnikoff/DA/tree/master/KP>

4 Консоль

```
(base) max@max-X550CC:~/DA/KP$ ls
Arhivator  Makefile      SampleSubmission.csv  Train.csv
main.cpp   report.pdf    Test.csv
(base) max@max-X550CC:~/DA/KP$ gzip -k Test.csv Train.csv SampleSubmission.csv
(base) max@max-X550CC:~/DA/KP$ gzip -l Test.csv.gz Train.csv.gz SampleSubmission.csv.gz
compressed      uncompressed  ratio uncompressed_name
1070430          6991963    84.7% Test.csv
1403853          7686426    81.7% Train.csv
251796           1800009    86.0% SampleSubmission.csv
2726079          16478398   83.5% (totals)
(base) max@max-X550CC:~/DA/KP$ make
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/Compressor.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Arhivator.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/RLE/RLE.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/BWT/BWT.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/Huffman/Huffman.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/MTF/MTF.cpp
g++ -std=c++17 -Wall -pedantic -c main.cpp
g++ Compressor.o Arhivator.o RLE.o BWT.o Huffman.o MTF.o main.o -o KP
rm -rf *.o
(base) max@max-X550CC:~/DA/KP$ rm *.gz
(base) max@max-X550CC:~/DA/KP$ ./KP Test.csv
(base) max@max-X550CC:~/DA/KP$ ./KP -l Test.csv.gz
Uncompressed size: 6991963
Compressed size: 1031026
Ratio: 85.2541%
(base) max@max-X550CC:~/DA/KP$ ./KP Train.csv
(base) max@max-X550CC:~/DA/KP$ ./KP -l Train.csv.gz
Uncompressed size: 7686426
Compressed size: 1285648
Ratio: 83.2738%
(base) max@max-X550CC:~/DA/KP$ ./KP SampleSubmission.csv
(base) max@max-X550CC:~/DA/KP$ ./KP -l SampleSubmission.csv.gz
Uncompressed size: 1800009
Compressed size: 55916
Ratio: 96.8936%
(base) max@max-X550CC:~/DA/KP$ ./KP test.djvu
(base) max@max-X550CC:~/DA/KP$ ./KP -l test.djvu.gz
Uncompressed size: 9511403
```

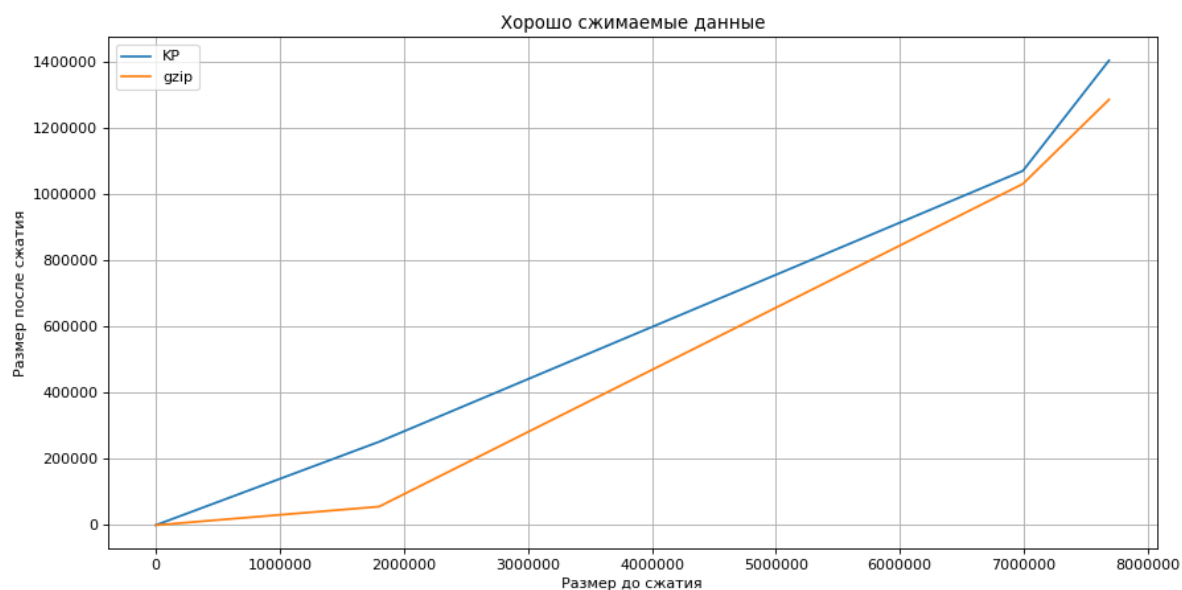
```

Compressed size: 9594466
Ratio: 0.0%
(base) max@max-X550CC:~/DA/KP$ ./KP -k report.pdf
(base) max@max-X550CC:~/DA/KP$ ./KP -l report.pdf.gz
Unompressed size: 325528
Compressed size: 324174
Ratio: 0.41594%
(base) max@max-X550CC:~/DA/KP$ ./KP -d test.djvu.gz
(base) max@max-X550CC:~/DA/KP$ gzip -k test.djvu report.pdf
(base) max@max-X550CC:~/DA/KP$ gzip -l test.djvu.gz report.pdf.gz
compressed      uncompressed  ratio uncompressed_name
9512138          9511403   -0.0% test.djvu
320366           325528    1.6% report.pdf
9832504          9836931   0.0% (totals)
(base) max@max-X550CC:~/DA/KP$ gzip -k test1.pdf
(base) max@max-X550CC:~/DA/KP$ gzip -l test1.pdf.gz
compressed      uncompressed  ratio uncompressed_name
568111          727670    21.9% test1.pdf
(base) max@max-X550CC:~/DA/KP$ ./KP -k test1.pdf
(base) max@max-X550CC:~/DA/KP$ ./KP -l test1.pdf.gz
Unompressed size: 727670
Compressed size: 602360
Ratio: 17.2207%
(base) max@max-X550CC:~/DA/KP$ ./KP -k 191110.en.pdf
(base) max@max-X550CC:~/DA/KP$ ./KP -l 191110.en.pdf.gz
Unompressed size: 284569
Compressed size: 281107
Ratio: 1.21658%
(base) max@max-X550CC:~/DA/KP$ gzip 191110.en.pdf
gzip: 191110.en.pdf.gz already exists; do you wish to overwrite (y or n)? y
(base) max@max-X550CC:~/DA/KP$ gzip -l 191110.en.pdf.gz
compressed      uncompressed  ratio uncompressed_name
275139          284569    3.3% 191110.en.pdf

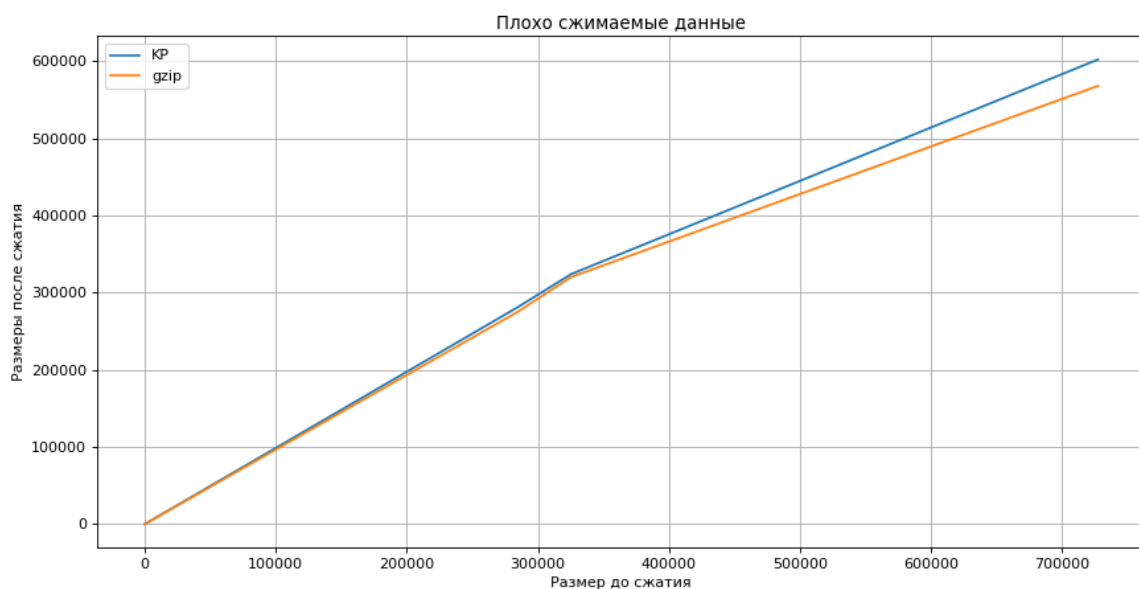
```

Рассмотрим полученные результаты

На хорошо сжимаемых данных размер сжатого файла, а значит и эффективность сжатия у моей программы выше чем у *gzip*:



На плохо сжимаемых данных размер сжатого файла, а значит и эффективность сжатия у моей программы ние чем у *gzip*, однако разница незначительна:



5 Выводы

Выполнив Курсовой проект по курсу «Дискретный анализ», я познакомился с такой тяжелой, но очень интересной темой, как сжатие данных. Данное задание заставило меня вспомнить почти все, что я прошел за 2 семестра изучения дисциплины: деревья, очереди, суффиксные массивы, динамическое программирование, алгоритмы сжатия, сортировки за линейное время - все это мне пришлось использовать в этой работе. Также не обошлось без знаний ООП для реализации наследуемых классов и виртуальных функций.

Из минусов своей реализации я вынужден отметить то, что я применяю последовательно алгоритмы сжатия для временных файлов побуфферно, что тратит огромное количество памяти на жестком диске и занимает время на перезаписывание. Мне стоило считывать кодируемый файл побуфферно, после чего последовательно применять алгоритмы сжатия непосредственно к буферу.

Однако несмотря на всю несовершенство моей реализации я получил огромный опыт при работе над курсовым проектом и, возможно, буду использовать его в дальнейшем в качестве портфолио и демонстрации своих навыков.

Список литературы

- [1] *Суффиксный массив — MAXimal*
URL: https://e-maxx.ru/algo/suffix_array (дата обращения: 05.12.2019).
- [2] *Алгоритм Хаффмана на пальцах — Хабр*
URL: <https://habr.com/ru/post/144200> (дата обращения: 02.12.2019).
- [3] *Преобразование Барроуза-Уилера — ИТМО*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Преобразование_Барроуза-Уилера (дата обращения: 04.12.2019).
- [4] *Преобразование MTF — ИТМО*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Преобразование_MTF (дата обращения: 04.12.2019).
- [5] *Кодирование длин серий — Википедия*
URL: https://ru.wikipedia.org/wiki/Кодирование_длин_серий (дата обращения: 01.12.2019).
- [6] *Алгоритм Карккайнена-Сандерса — ИТМО*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Карккайнена-Сандерса (дата обращения: 04.12.2019).