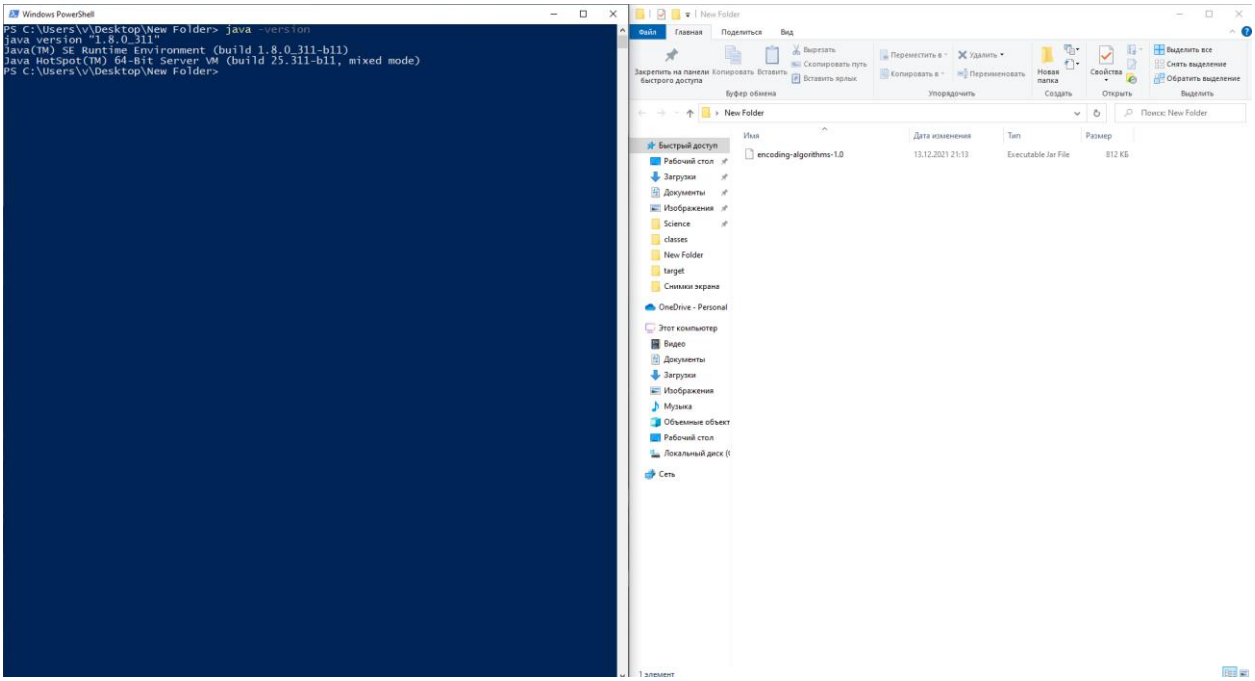
Свидиров Кирилл, 11-902

Отчёт по алгоритмам

Запуск программы

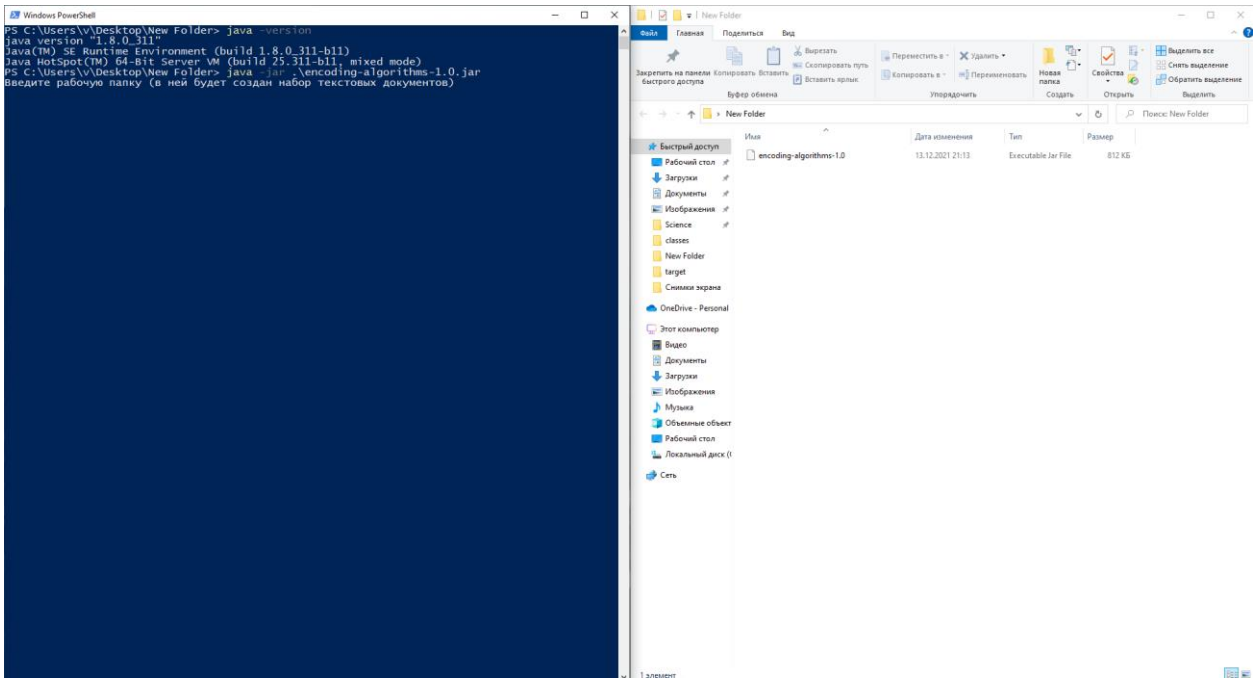
1) Убедитесь, что у Вас установлена Java. Для этого воспользуйтесь командой:
   java -version



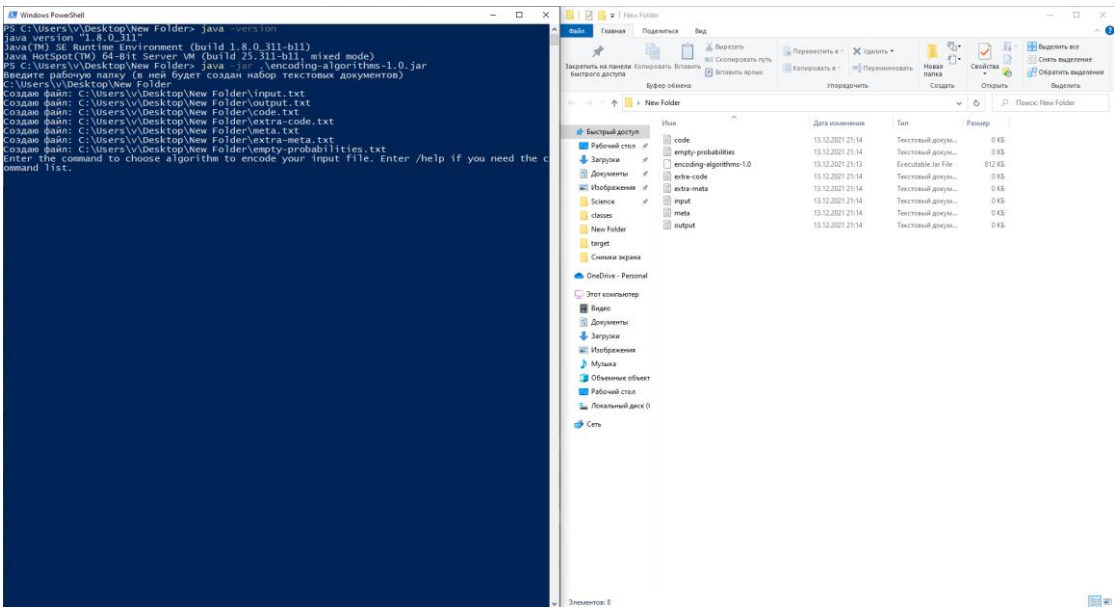
2) Создайте отдельную папку для промежуточных файлов программы (в идеале на рабочем столе – файлы будут создаваться, поэтому не нужны проблемы с правами). В эту же папку можно закинуть jar-ник (на скрине выше в правой части). В ней же запустите консоль.

   ВАЖНО!!! Путь к папке, как и пути к файлам с тестовыми данными, не должны содержать русские символы.
   Причина – под рукой нет готового модуля, который мог бы определить кодировку консоли и корректно считать не ASCII символы.

3) Вводим команду, чтобы запустить jar-файл (путь может быть другим, если jar находится в другом месте)
java -jar .\encoding-algorithms-1.0.jar



4) Программа запрашивает полный путь до папки, в которой будут лежать стартовые текстовые файлы (путь должен быть абсолютным). Вводим его и получаем доступ к консольному приложению (все файлы успешно создаются.



Работа с программой.

Я постарался сделать приложение как можно понятнее. Используя команду /help, можно получить информацию о всех возможностях программы.

Дублирую сюда эти возможности:

/set [param] [param-value] – функция установки параметров, которые влияют на работу приложения

Список параметров ([param]):

- text-logging-flag (true / false, Default = false) – флаг вывода в консоль считанного, закодированного и раскодированного текстов. Использовался в основном для дебага, удобен для небольших текстов. Но слишком избыточен для больших, поэтому выключен.
- distionaries-logging-flag (true / false, Default = true) – флаг вывода в консоль промежуточных структур (таблица вероятностей, таблица кодов и т.д). Не слишком избыточен и нагляден, поэтому включен.
- size-limit (целое положительное число, Default = 65536) – количество байт, ограничивающее основные структуры (в основном размер буфера reader / writer, а также размер закодированной / раскодированной строки, хранящейся в памяти). Позволяет выводить большие тексты по частям и не забивать память
- encoding-alphabet-size (число в отрезке [2, 10], Default = 2) – размер кодирующего алфавита. Для арифметического кодирование и кодов Хемминга данная функция не работает (система счисление – 2 всегда).
- block-size (целое положительное число, Default = 4) – размер блока, кодируемого в алгоритме Хемминга. Работают произвольные значения, но default соответствует заданию.
- input-path – абсолютный путь из входного файла
- output-path – абсолютный путь до файла вывода
- code-path – абсолютный путь до кодового файла
- meta-path – абсолютный путь до файла с мета-информацией

Команды выполнения алгоритмов:

/e [algorithm] – кодирование входного файла

/d [algorithm] – декодирование файла с кодом

/e+d [algorithm] – кодирование входного файла и сразу же декодирование кодового

Виды алгоритмов ([algorithm]):

- huf – алгоритм Хаффмана
- ari – арифметическое кодирование
- b+m – BWT + MTF
- ham – алгоритм Хемминга

/exit – завершение работы приложения

Принцип работы алгоритмов

Алгоритмы берут входные данные из файла input-path, после чего кодировщик пишет мета-информацию в файл meta-path и сам код в файл code-path. После чего декодер берёт код из файла code-path и мета-информацию из файла meta-path и выводит раскодированное сообщение в файл output-path.

Исключением является третий алгоритм (BWT + MTF), в котором схема выглядит следующим образом:

Input -> meta, code -> extra-meta, extra-code -> code -> output

Файлы meta и код используются модулем BWT, а extra-meta и extra-code – модулем MTF.

Основные сценарии тестирования.

P.s. Файлы можно задавать двумя способами:

1) Можно не менять пути в программе методом /set и копировать необходимое сообщение для кодирования в файл input.txt

2) Можно каждый раз менять абсолютные пути

Примеры будут показываться с использованием второго способа – не зря ведь он поддерживается!

Задание 1. Алгоритм Хаффмана.

1) Задаём путь до файла для кодирования

/set input-path PATH

2) Запускаем кодировщик и декодер

/e+d huf

(Если нужно кодировать и декодировать отдельными операциями):

/e huf

/d huf

3) Проверяем результат:

- Файл code.txt – результат после кодирования
- Файл meta.txt – мета-информация для декодера (строка символов и их кодов)
- Файл output.txt – результат после декодирования
- Консоль – выведены посчитанные вероятности и полученные коды для символов.

Задание 2. Арифметическое кодирование.

1) Задаём путь до файла для кодирования

/set input-path PATH

2) Запускаем кодировщик и декодер

/e+d ari

(Если нужно кодировать и декодировать отдельными операциями):

/e ari

/d ari

3) Проверяем результат:

- Файл code.txt – результат после кодирования
- Файл meta.txt – мета-информация для декодера (список вероятностей)
- Файл output.txt – результат после декодирования
- Консоль – выведены посчитанные вероятности для символов.

Задание 3. BWT + MTF.

1) Задаём путь до файла для кодирования

/set input-path PATH

2) Запускаем кодировщик и декодер

/e+d b+m

(Если нужно кодировать и декодировать отдельными операциями):

/e b+m

/d b+m

3) Проверяем результат:

- Файл code.txt – результат после преобразования BWT
- Файл meta.txt – мета-информация для BWT – декодера (номер исходной строки в сортированной матрице)
- Файл extra-code.txt – результат после преобразования MTF
- Файл extra-meta.txt – мета-информация для BWT – декодера (начальная последовательность символов в стопке книг)
- Файл output.txt – результат после декодирования
- Консоль – выведены посчитанные вероятности для символов, а также принятая MTF-декодером строка

Задание 4. Алгоритм Хемминга.

1) Задаём путь до файла для кодирования

/set input-path PATH

2) Запускаем кодировщик

/e ham

В файле code.txt можно поменять биты в блоках, чтобы проверить, как будут исправляться ошибки

Затем запускаем декодеровщик

/d ham

(перед каждый декодированием не обязательно по новой кодировать файл – он не очищается. Т.е. можно закодировать один раз, попробовать раскодировать без ошибок и после попробовать раскодировать, добавив ошибки)

3) Проверяем результат:

- Файл code.txt – результат после кодирования
- Файл meta.txt – мета-информация для декодера (размер блока)
- Файл output.txt – результат после декодирования
- Консоль – если были ошибки, то приложение сообщит, в каких блоках они были совершены и на какие символы были произведены замены

Пример указания нового файла для кодирования:

/set input-path C:\Users\v\Desktop\New Folder\new-input.txt

Особенности алгоритмов.

Алгоритм Хаффмана.

- Есть возможность задать вероятности (properties.txt). Если вероятностей нет, алгоритм сам считает их (для удобной задачи вероятностей символы \n и \r обрабатываются особым образом, чтобы можно было их ввести в таком представлении)

- Для считывания и вывода файлов используются BufferedReader и BufferedWriter с размером буфера – sizeLimit (чтобы не держать в памяти сразу весь текстовый файл)

- Для построения дерева Хаффмана используется самописное дерево:

  HuffmanEntry – это класс – вершина дерева. Она может хранить в себе либо набор подвершин (если вершина не является листом), либо символ (если вершина – лист).

  Алгоритм построения дерева – как на практике. Строим дерево снизу вверх, на каждом шаге объединяем s последних вершин в одну, их

удаляем, новую вставляем в список и продолжаем. Алгоритм построения продолжается, пока не останется одна вершина – корень дерева (getRootHuffmanEntry)

После чего происходит рекурсивный спуск по дереву до листьев и получение кодового слова для символа (getHuffmanCodes) (рекурсия – чтобы при спуске в вершину мы точно провели все листья, которые лежат ниже её – чтобы не пришлось потом проверять её снова. Чуть эффективнее простого прохода до каждого листа)

- В качестве мета данных для декодера выводится строка ‘символ’код’символ’… без разделителей (было желание передать строку в кратчайшем виде для данного алгоритма). Чтобы понимать, когда цифра является символом, а когда – частью кодового слова, было добавлено экранирование (SPECIAL_SYMBOL). Чтобы передать сам специальный символ, нужно экранировать и его.
- Генерация кода происходит посимвольно – чтобы не забивать память входной строкой. Более того, ограничение sizeLimit работает и для закодированной строки – каждый раз, когда буфер кода переполняется, он сразу выводится в файл.
- Для декодирование строится ещё одно дерево – фактически восстанавливается дерево, которое строилось при кодировании, только вершины теперь не содержат вероятностей (buildDictionaryTree) Специально для этого дерева создан класс DictionaryTreeWorker, чтобы была возможность ходить по дереву. Начальное положение – корень

  Декодер посимвольно считывает код. Считав символ, декодер говорит DictionaryTreeWorker перейти на следующую вершину по нему. Переход осуществляется и возвращается значение char в этой вершине. Если оно есть – то вершина была листом, и декодер обновляет текущую вершину DictionaryTreeWorker на корневую (начинаем читать новый код). Если его не было – то вершина не являлась листом, и нам нужно продолжать читать символы дальше.

- Аналогично кодированию декодер имеет буфер и периодически выводит раскодированную строку.

Арифметическое кодирование.

- Есть возможность задать вероятности (properties.txt). Если вероятностей нет, алгоритм сам считает их (для удобной задачи вероятностей символы \n и \r обрабатываются особым образом, чтобы можно было их ввести в таком представлении)

- В метаданные выводятся таблица вероятностей и количество символов в исходном сообщении (оно считается отдельным проходом по файлу – метод подсчёта вероятностей вынесен в отдельный класс и используется в других алгоритмах, менять его не хочется

- Символы упорядочиваются в порядке убывания их вероятности, и для них строится HashMap <Character, Interval> - словарь, ставящий в соответствие каждому символу интервал, соответствующий ему на исходном интервале длины [0, 1)

- Начиная со стартовых значений начала и конца интервала 0, 1, кодер начинает считывать символы и менять их. Новые координаты вычисляются относительно текущих линейными формулами.

- Каждый раз, когда длина чисел начала-конца достигает определённой длины (зависящей от sizeLimit), алгоритм проверяет, может ли он запомнить отдельно общую часть. Для этого он переводит числа в строки и, начиная с начала, запоминает их общие части. После чего длины чисел уменьшаются и облегчают дальнейшие вычисления.

- После получения значения интервала он собирается из общей части (если она есть) и последнего значения интервала. После чего проверяется, равна ли первая граница нулю или последняя граница единице (два крайних случая, когда мы подаём последовательность самых частых или самых редких символов). Для них ставится в соответствие код 0 и 1 соответственно

  Иначе мы последовательно к текущей сумме (начальная сумма = 0) пытаемся добавить дроби 1/2^n (начальное n = 1) и понять, попала ли сумма в интервал или нет.

- Алгоритм декодирования происходит аналогично – считываем таблицу вероятностей и длину сообщения из мета-файла, составляем список интервалов для символов, считываем код (полностью), переводим его в десятичную систему, движемся по интервалам и запоминаем символы на каждом шаге.

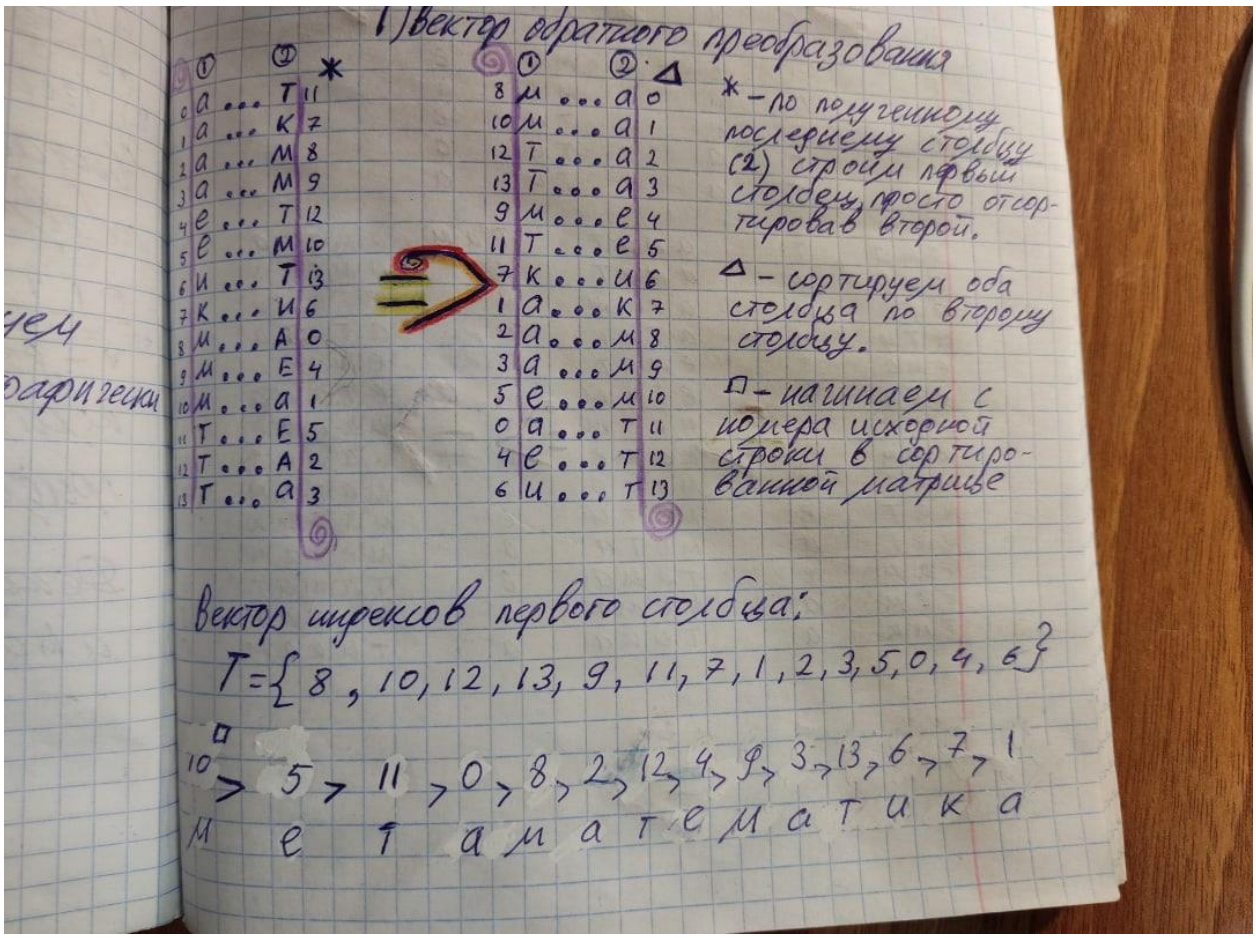
  К особенности можно отнести то, что на каждом шаге поиск соответствующего интервала происходит бинарным поиском – чтобы уменьшить время. Именно поэтому декодер содержит структуру-список, а не словарь – чтобы были индексы. Каждая запись (ArithmeticEntry) содержит символ и соответствующий ему интервал на интервале [0, 1).

BWT.

- Кодирование происходит алгоритмом, аналогичным практическому – считывается строка (полностью), строится матрица сдвигов, сортируется, и из последнего столбца составляется строка преобразования. В мета-информацию попадает номер исходной строки в отсортированной матрице.

- Декодирование осуществляется векторной версией

  Для начала нам нужно получить первую таблицу (на рисунке ниже). Для этого создаётся список из BwtEntry, запись == строчке, то есть хранит первую цифру, первый символ, вторую цифру и второй символ. При создании списка заполняем колонку вторых символов (это входное слово-преобразование) и первый столбец цифр (он просто равен возрастающей нумерации, так как соответствует индексам отсортированных символов)



Теперь для получения остальных данных из этих же записей создаётся второй список, который сортируется по второй колонке символов (то есть сортируем нашу входную строку).

После циклом проходимся параллельно по двум спискам – обычному и отсортированному.

```
for (int i = 0; i < entriesToSort.size(); i++){
    entriesToSort.get(i).setSecondIndex(i);
    entries.get(i).setFirstSymbol(entriesToSort.get(i).getSecondSymbol());
}
```

На второй строчке мы каждому элементу из отсортированного списка на место второго индекса ставим текущее значение счётчика цикла. Логически – обращаясь к записи из отсортированного списка, мы обращаемся одновременно и к записи из исходного списка. То есть, например, на первом шаге мы берём первую запись и в качестве второго индекса ставим единицу. Но, получается, что в исходном списке эта запись тоже получит значение. И её второй индекс будет равняться её месту в отсортированном списке – то что нужно!

После на третьей строчке мы элементу из несортированного списка ставим в качестве первого символа соответствующий символ из элемента сортированного списка. Логически (на примере первого шага) – в запись к первому символу второй колонки мы ставим первым символом первый символ из отсортированной строки.

После чего полученная неотсортированная таблица вновь сортируется по второй колонке, чтобы получить вторую таблицу, по которой можно восстанавливать слово.

Таким образом, мы можем быстро сделать обратное преобразование, не восстанавливая матрицу, при этом алгоритм построения нужного вектора очень краток!

- Восстановление исходной строки происходит практически аналогично алгоритму с практики. Разница лишь в том, что там мы делали переходы по вектору следующим образом:
  Выбираем стартовый элемент, берём соответствующее значение символа, переходим на элемент, индекс которого равен текущему элементу и так далее…

  В своём алгоритме я реализовал то же самое хождение по цепи (полученный вектор – цепь из символов исходного слова в соответствующем порядке), только скрыто через индексы массивов, а явно – через строчки матрицы

MTF.

- Исходное расположение книг на полке строится в зависимости от вероятностей – символ с большей вероятностью изначально будет выше. На то есть 2 причины:
  1) Хотелось добавить возможность влиять на исходное положение книг на полке
  2) В любом случае нужно знать набор символов, чтобы составить стартовую полку. То есть проходиться придётся. Тогда можно и вероятности параллельно посчитать.

- В качестве структуры книжкой полки используется связный список:

Если мы будем брать любую структуру на уровне массива, то сложность будет равна n (найти символ в массиве, проходясь по элементам последовательно) + n (удалить элемент из списка – достать книгу – и при этом сдвинуть все нижележащие) + n (вставить новый элемент в начало и снова сдвинуть весь список) = 3n

В случае связного списка аналогичная операция будет весить n (поиск ноды, в которой лежит нужный символ) + 1 (для удаления достаточно кинуть ссылку предыдущего элемента на следующий – а их можно запомнить) + 1 (для добавления в начало достаточно запомнить элемент в качестве корневого, предварительно указав ссылку в нём на текущий корень) = n

Мелочи, но зато точно сделано самостоятельно с:

- Код, соответствующий символу, равен его текущему индексу в связном списке – книжной полке. Коды хранятся отдельно в специальном листе (codeList), предварительно просчитанные (также они равномерные, индексы были переведены в s-ичную систему и дополнены спереди нулями до равномерных строк). Доступ к получения кода равен O(1) – ведь мы знаем текущий индекс символа на полке, и достаточно достать из списка кодов строку по этому индексу.

- В качестве мета-данных просто передаётся строка символов, соответствующая исходному положению книжной полки

- Считывание происходит посимвольно, вывод – при переполнении буфера. Декодирование считывает код поблочно, а вывод – также при переполнении буфера.

Алгоритм Хемминга.

- В качестве мета-данных передаётся размер блока, который преобразовывается и передаётся

- Алгоритм был сделан по примеру его применения к последовательности бит, поэтому могут быть неоптимальные шаги напрямую
  https://habr.com/ru/post/140611/

- Считывание из файлов происходит посимвольно, а вывод – при переполнении буфера, который ограничен переменной sizeLimit

- Для полноты алгоритма переданные и обратно преобразованные биты сначала переводятся в символы, а потом уже выводятся. В связи с предыдущим пунктом может возникнуть проблема, что необходимо вывести буфер, но какой-то символ пришёл частично (при размере блока <32). Для решения этой проблемы функция checkLastBytes проверяет последние 32 биты буфера и по особенностям строения кода UTF-32 понимает, есть ли неполный символ в буфере и сколько байт пока что не нужно переводить.