Оглавление

[Постановка Задачи 4](#_Toc185943549)

[Глава 1. Теория 5](#_Toc185943550)

[Коды Хаффмана. Основные понятия. Алгоритм построения. 5](#_Toc185943551)

[Построение за O(n) 6](#_Toc185943552)

[Пример построения дерева Хаффмана 7](#_Toc185943553)

[Канонические коды Хаффмана 9](#_Toc185943554)

[Пример построения канонических кодов Хаффмана 10](#_Toc185943555)

[Доказательство корректности алгоритма 10](#_Toc185943556)

[Проблемы 11](#_Toc185943557)

[Асимптотическая оценка временной сложности алгоритма 11](#_Toc185943558)

[Глава 2. Решение задачи 13](#_Toc185943559)

[Тестирование в задаче 13](#_Toc185943560)

[Мини-тесты для проверки работоспособности 15](#_Toc185943561)

[Тесты на реальном тексте 16](#_Toc185943562)

[Заключение 18](#_Toc185943563)

[Список литературы 19](#_Toc185943564)

[Приложения 20](#_Toc185943565)

# Постановка Задачи

Дан алфавит из n символов , каждому из которых приписана частота , где и . Используя входные данные, построить префиксный код с минимальной средней длиной кодирования, т.е. свести к минимуму следующую функцию:

где — длина кодового слова для .

# Глава 1. Теория

Для решения будем использовать алгоритм кодирования Хаффмана.

## Коды Хаффмана. Основные понятия. Алгоритм построения.

Кодирование Хаффмана — это алгоритм сжатия данных без потерь. Идея заключается в назначении входным символам кодов переменной длины. Длины назначенных кодов основаны на частотах соответствующих символов.

Коды переменной длины, назначенные входным символам, являются префиксными кодами, то есть коды назначаются таким образом, что код, назначенный одному символу, не является префиксом кода, назначенного любому другому символу, тем самым обеспечивая однозначность отображения.

Суть алгоритма заключается в построении дерева Хаффмана, которое позволяет декодировать полученное сообщение путём спуска по дереву. Алгоритм строит дерево снизу-вверх. Рассмотрим шаги по построению этого дерева.

1. Для каждого уникального символа из входного алфавита строится узел.
2. На этих узлах строится куча на минимуме.
3. Извлекается 2 узла с минимальной частотой из кучи.
4. Создаётся новый узел с частотой, равной сумме частот двух извлечённых узлов и добавляется в кучу.
5. Повторять шаги 3 и 4 пока в куче не останется 1 узел, который будет являться корнем дерева.

Пусть n — входной алфавит, |n| — размер входного алфавита, f[c] — частота для символа .

function Build\_Huffman\_tree(n):

heap = make\_heap(n, f(n))

**while** (size(heap) > 1) **do** {

left = heap.pop()

right = heap.pop()

top = new Node(0, left->f, right->f)

heap.push(top) }

root = heap.pop()

*Листинг 1.* Псевдокод построения дерева Хаффмана:

Построение кучи занимает О(n) времени, цикл while выполняется n – 1 раз, извлечение и вставка в кучу занимает O(logn). Следовательно, асимптотическая оценка сложности алгоритма составляет O(nlogn).

Попробуем дать верхнюю оценку для алгоритма:

* Построение кучи занимает не более 2n.
* Цикл while выполняется ровно n-1 раз (после каждой итерации размер кучи уменьшается на 1).
* Извлечение 2х элементов потребует 2logn операций.
* Создание узла занимает константное время, 1 операция.
* Добавление узла занимает logn.

Итого имеем:

Оценка является достаточно грубой, но данной оценки вполне достаточно для нашего случая.

## Построение за O(n)

Дерево Хаффмана можно построить за линейное время, при условии, что входные данные отсортированы.

Алгоритм построения:

1. Создаётся 2 пустые очереди.
2. Создаётся узел для каждого символа из входного алфавита. Все узлы помещаются в 1 очередь.
3. Извлечение двух узлов с минимальной частотой, проверив обе очереди.

Алгоритм извлечения:

* 1. Если вторая очередь пуста, извлекается элемент из первой очереди.
  2. Если первая очередь пуста, извлекается элемент из второй очереди.
  3. В противном случае сравниваются начала двух очередей и извлекается минимум.

1. Создаётся узел с частотой, равной сумме частот извлечённых узлов.
2. Новый узел помещается во вторую очередь.
3. Повторять шаги 3-4 пока в очередях суммарно более 1 узла.

function build\_Huffman\_tree\_sorted(n):

Q1, Q2 // Создаются очереди

Q1.push(n, f(n))

**while** (size(Q1) + size(Q2) > 1) **do** {

left = extract\_min(Q1, Q2)

right = extract\_min(Q1,Q2)

top = new Node(0, left->f + right->f)

Q2.push(top) }

root = (size(Q1) == 0 ? q2.pop() : q1.pop())

*Листинг 2.* Псевдокод функции построения дерева Хаффмана на основе двух очередей.

Заполнение кучи Q1 делается за n операций, цикл while работает n-1 раз, каждая итерация занимает O(1). Отсюда получаем асимптотическую оценку O(n).

Пробуем дать верхнюю оценку:

* n операций на заполнение кучи Q1.
* n – 1 итерация цикла while.
* Извлекаем два элемента за константу (1 операция).
* Вставка в кучу Q2 за 1 операцию.

Итого имеем .

**Замечание:** В обоих версиях построения требуется O(n) дополнительной памяти (в первом случае для построения кучи, во втором случае на поддержание очередей).

## Пример построения дерева Хаффмана

Возьмём алфавит и частоты .

Этапы построения дерева:

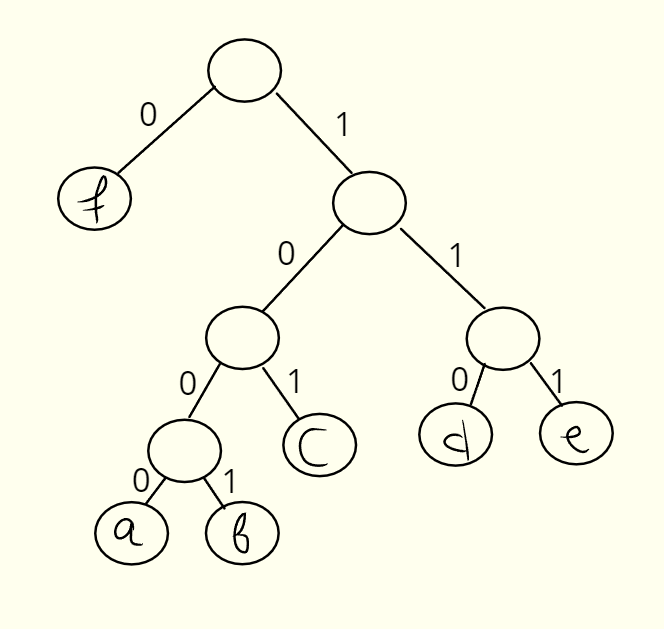
* Узлы a и b объединяются в новый узел с частотой ;
* Узлы и c объединяются в узел с частотой ;
* Узлы d и e объединяются в узел с частотой ;
* Узлы и объединяются в узел с частотой ;
* Узлы и f объединяются в узел (корень дерева) с итоговой частотой .

Итоговое представление дерева представлено на рисунке 1.

По нему можем получить итоговые коды для символов:

.

Пусть нам пришло сообщение , тогда в лучшем случае мы получим код длины — строка из символов f, в худшем — строка из символов a или b.



*Рисунок 1.* Дерево Хаффмана для примера.

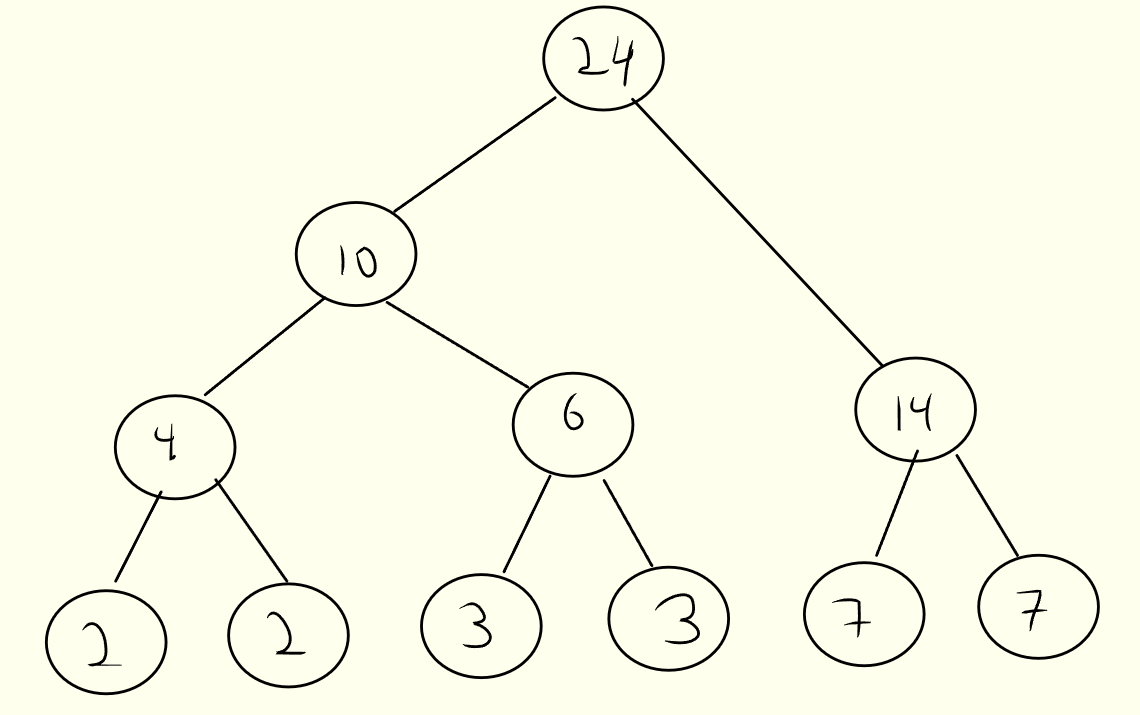
Возникает вопрос: А какая максимальная глубина дерева и как её достичь?

Пусть у нас есть — префикс-суммы для массива частот, т.е. . Тогда худший случай построения дерева Хаффмана выглядит следующим образом:

В данном случае, мы на каждом шаге будем объединять только что добавленный узел с каким-то из символов алфавита (поскольку сумма всех меньших его в точности равна его частоте, а значит эти узлы должны образовывать новый узел на каждой итерации) и итоговая глубина дерева будет .

Чтобы наоборот, достичь минимальной высоты дерева, необходимо подобрать такой набор частот, чтобы при каждом объединении мы как можно дольше оставались на одной глубине дерева. Достичь этого можно в том случае, если в алфавите встречается много пар с одинаковыми частотами. Например:

. Здесь в качестве удобства взяли свои веса, чтобы нагляднее показать разницу.



*Рисунок 2.* Дерево Хаффмана с минимально возможной глубиной.

## Канонические коды Хаффмана

В каноническом кодировании Хаффмана используются длины бит стандартных кодов Хаффмана, сгенерированных для каждого символа. Сначала символы сортируются по длинам бит в неубывающем порядке, а затем для каждой длины бит они сортируются лексикографически. Первый символ получает код, содержащий все нули и имеющий ту же длину, что и исходная длина бит. Для последующих символов, если длина бит символа равна длине бит предыдущего символа, то код предыдущего символа увеличивается на единицу и присваивается текущему символу.

*Чем канонические коды лучше обычных?*

Проблема обычного алгоритма сжатия по Хаффману — недетерминированность. Для похожих последовательностей могут получиться разные деревья, также и одно дерево без правильной сериализации может соответствовать разным последовательностям. Причиной тому является то, что на каждом шаге алгоритма, если существует несколько узлов с одинаковыми частотами, выбор пары узлов для объединения не определён однозначно. Например, символ *А* при одном выборе пар узлов может быть закодирован как , а при другом уже как . Канонические коды решают эту проблему лексикографической сортировкой.

## Пример построения канонических кодов Хаффмана

A – 5, B – 5, C – 2, D – 2, E – 1.

После построения дерева Хаффмана получаем длины кодов:  
A – 2, B – 2, C – 3, D – 3, E – 4.

Теперь располагаем элементы в порядке сортировки (сначала по длине кода, затем лексикографически): A (2), B (2), C (3), D (3), E (4).

Назначаем коды по принципу канонических кодов:

A = 00, B = 01, C = 100, D = 101, E = 1100.

Таким образом мы получаем однозначное отображение для данного набора символов и их частот.

## Доказательство корректности алгоритма

Докажем, что коды Хаффмана сводят функцию к минимуму. Будем доказывать по индукции.

**База индукции:**

Рассмотрим случай, когда . Если алфавит состоит из двух символов и ​ с частотами и ​, то единственный возможный префиксный код — это код, в котором одному символу присваивается кодовое слово длины 1 (например, 0), а другому также длины 1 (например, 1).

Средняя длина кодового слова будет равна: .

Это минимальная возможная длина, так как длины всех кодовых слов неизбежно равны 1.

**Шаг индукции:**

Пусть дан алфавит Σ размера . Пусть и - два символа в Σ с наименьшими частотами и соответственно. Алгоритм Хаффмана на первом шаге объединит эти два символа в один узел с частотой .

Рассмотрим новый алфавит Σ' размера k, полученный из Σ путем удаления и и добавления нового символа .

Пусть T' - оптимальное префиксное дерево для алфавита Σ'. Согласно индукционному предположению, такое дерево существует (поскольку |Σ'| = k).

Теперь построим дерево T для алфавита Σ из дерева T'. В дереве T' найдем лист, соответствующий символу . Заменим этот лист внутренним узлом, у которого левым потомком будет символ , а правым - символ . Коды всех остальных символов в T будут такими же, как и в T'. Коды символов и будут на один бит длиннее, чем код символа в T'.

Получаем, что:

, где — длина кода в дереве для символа .

, что является минимальным приростом.

## Проблемы

Как говорилось ранее, главная проблема классических кодов Хаффмана — это недетерминированность. Решением этой проблемы является либо использование канонических кодов Хаффмана, либо передача полной структуры дерева с возможностью восстановить её со стороны получателя.

Проблемой канонических кодов всё ещё остаётся необходимость в передаче структуры дерева.

Ещё одной значимой проблемой может стать неправильный выбор частоты символов во входных сообщениях. Входные сообщения могут содержать символы, которые встречаются редко (по нашей таблице частот), в довольно большом количестве, что делает кодирование невыгодным и требует перестройки таблицы частот и дерева Хаффмана в целом.

## Асимптотическая оценка временной сложности алгоритма

Подведём итоги работы алгоритма Хаффмана со всеми вариантами.

1. Классические коды Хаффмана:

* на неотсортированных входных данных
* на уже отсортированных данных.

Так же требуется построить структуру дерева, переданную в сообщении (можно сделать за ).

Требует дополнительной памяти.

1. Канонические коды Хаффмана:

Асимптотические оценки, как и у классических кодов Хаффмана.

# Глава 2. Решение задачи

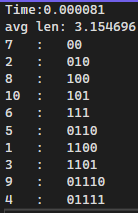
## Тестирование в задаче

На вход подаётся файл, содержащий количество символов и частоты символов (номер частоты = номеру символа в алфавите). На выходе получаем коды для каждого из символов.

1 тестовый пример:

10 37 59 43 27 30 96 96 71 8 76

Результат программы:



*Рисунок 1.* Результат выполнения программы для 1 тестового примера.

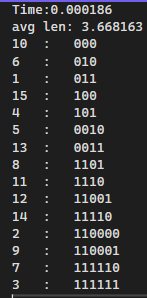
Программа отработала практически моментально, средняя длина L порядка 3.15 (вместо 4 для обычного кодирования). Получаем выигрыш примерно в 21%.

Минимальная длина кодирования — 2 бита, максимальная — 5 бит.

2 тестовый пример:

15 895 121 188 953 378 849 153 579 144 727 589 301 442 327 930

Результат программы:



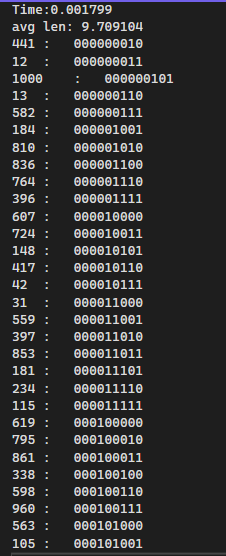
*Рисунок 2.* Результат выполнения программы для 2 тестового примера.

Скорость работы программы пока всё ещё моментальная. Средняя длина L составила 3.67 вместо 4. Прирост примерно на 8%.

Минимальная длина кода — 3 бита, максимальная — 6 бит.

3 тестовый пример:

Содержит 1000 символов, подробнее можно ознакомиться в приложении 1.



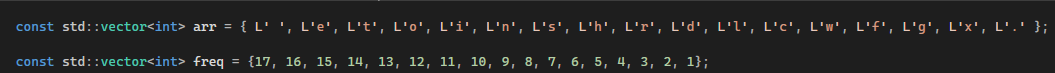
*Рисунок 3.* Результат выполнения программы для 3 тестового примера.

Как видим, скорость выполнения составила всего 0.001 секунду, что примерно составляет . Средняя длина L составляет 9.7 вместо 10 стандартных.

Минимальная длина кодов — 9 бит, максимальная — 19 бит. Полный результат выполнения можно найти с помощью приложения 1.

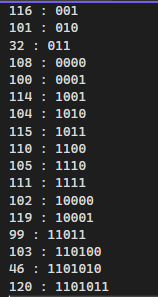
## Мини-тесты для проверки работоспособности

Проверим работоспособность реализованных алгоритмов на простых тестах — входные сообщения должны быть закодированы и декодированы без потерь.



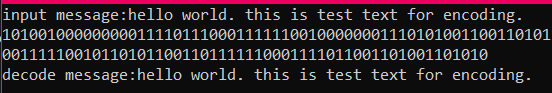
*Рисунок 4.* Входной алфавит и частоты символов для теста.

Классический алгоритм Хаффмана кодирует символы следующим образом:



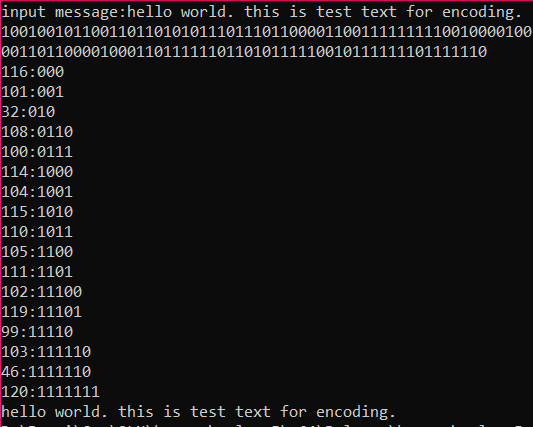
*Рисунок 5.* Результат работы классического алгоритма Хаффмана.

Попробуем зашифровать сообщение и обратно его расшифровать.



*Рисунок 6.* Итог работы кодирования и декодирования классического алгоритма Хаффмана.

Проверим работу канонических Кодов Хаффмана на тех же данных.



*Рисунок 7.* Итог работы канонических кодов Хаффмана.

Итоговая длина сообщения вышла 174 символа, вместо 220(5 бит \* 44 символов входного сообщения). Получаем примерно 21% сжатия, пользуясь таблицей частотностью в текстах русского языка.

## Тесты на реальном тексте

Возьмём в качестве теста произведение Льва Николаевича Толстого “Война и мир” в 4 томах. Общая длина файла составила символов, среди которых 150 уникальных. Поскольку файл очень большой, то расходы на чтение из файла/запись в файл не будут учитываться при подсчёте времени работы программы. Так же рассмотрим исправленный текст, в котором присутствуют только знаки препинания и символы русского языка (размер алфавита 80, размер файла 5 199 469). Таблицу частот возьмём характерную для русского языка.

Ниже приведена итоговая таблица со средним временем работы алгоритмов.

*Таблица 1.* Таблица со средним временем работы алгоритмов на тестах, а также средние длины кодов при кодировании.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Хаффман | Наивный | Хаффман (исправленный) | Наивный (исправленный) |
| Среднее время работы | 0.2 | 0.18 | 0.12 | 0.17 |
| Средняя длина | 4.910499 | 8.0 | 6.015128 | 7.0 |
| Минимальная длина | 3 | 8 | 6 | 7 |
| Максимальная длина | 21 | 8 | 8 | 7 |

Скорость работы сложно оценить в процентном отношении, поскольку размеры алфавита не столь большие в реальных условиях. Однако, глядя на таблицу, можем сделать следующие выводы:

* средняя длина кодов Хаффмана существенно ниже, чем у наивного кодирования;
* средняя длина кодов выходит меньше, если применять его к конкретному сообщению;
* скорость работы с уже имеющейся таблицей частот намного быстрее скорости наивного алгоритма, в виду того, что средняя частота ниже и требуется меньше времени, чтобы закодировать и раскодировать сообщение.

# Заключение

Кодирование Хаффмана является хорошим алгоритмом для сжатия данных без потерь, поскольку не использует большие затраты по дополнительной памяти (), а также имеет хорошую асимптотическую временную оценку сложности — ( на отсортированных данных). Этот алгоритм может быть применён к любому алфавиту, ведь его эффективность зависит только от частотности символов. Суть алгоритма заключается в сведении средней длины кодов к минимуму, что в конечном итоге и помогает нам сжимать данные.

# Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — М.: Издательский дом «Вильямс», 2011. — 1296 с.
2. Каноническое кодирование Хаффмана [Электронный ресурс] / Переведено с помощью Google Translate. URL: <https://www-geeksforgeeks-org.translate.goog/canonical-huffman-coding/?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=wapp> (дата обращения: 19.12.2024).
3. Кодирование Хаффмана [Электронный ресурс] / Переведено с помощью Google Translate. URL:

<https://www-geeksforgeeks-org.translate.goog/huffman-coding-greedy-algo-3/?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=wapp> (дата обращения: 19.12.2024).

1. Кодирование Хаффмана [Электронный ресурс] / Переведено с помощью Google Translate. URL:

<https://en-m-wikipedia-org.translate.goog/wiki/Huffman_coding?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=wapp> (дата обращения: 19.12.2024).

# Приложения

*Приложение 1.* Ссылка на исходные файлы проекта.

<https://drive.google.com/drive/folders/119_ra8QqNzYsFO5jyeAPmbbHSmx7BRy8?usp=sharing>

*Приложение 2*. Описание структуры проекта.

Нижеприведённые тесты находятся в папке tests (далее будут указываться пути, относительные этой папке).

Директории:

* huffman
  + war\_and\_peace\_encode.txt — результат кодирования текста в битовую последовательность классическим методом Хаффмана;
  + war\_and\_peace\_codes.txt — итоговые коды для каждого символа при классическом методе Хаффмана;
  + war\_and\_peace\_decode.txt — результат декодирования битовой последовательности классическим методом Хаффмана;
  + fixed\_war\_and\_peace\_encode.txt — результат кодирования исправленного текста в битовую последовательность классическим методом Хаффмана;
  + fixed\_war\_and\_peace\_codes.txt — итоговые коды для каждого символа для исправленного текста при классическом методе Хаффмана;
  + fixed\_war\_and\_peace\_decode.txt — результат декодирования битовой последовательности исправленного текста классическим методом Хаффмана.
* canonical
  + war\_and\_peace\_encode.txt — результат кодирования текста в битовую последовательность каноническим методом Хаффмана;
  + war\_and\_peace\_codes.txt — итоговые коды для каждого символа при каноническом методе Хаффмана;
  + war\_and\_peace\_decode.txt — результат декодирования битовой последовательности каноническим методом Хаффмана;
* naive
  + war\_and\_peace\_encode.txt — результат кодирования текста в битовую последовательность наивным методом;
  + war\_and\_peace\_codes.txt — итоговые коды для каждого символа при наивном методе;
  + war\_and\_peace\_decode.txt — результат декодирования битовой последовательности наивным методом;
  + fixed\_war\_and\_peace\_encode.txt — результат кодирования исправленного текса в битовую последовательность наивным методом;
  + fixed\_war\_and\_peace\_codes.txt — итоговые коды для каждого символа исправленного текста при наивном методе;
  + fixed\_war\_and\_peace\_decode.txt — результат декодирования битовой последовательности исправленного текста наивным методом.

Исходные файлы с кодом находятся в папке src.