

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

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и Структуры Данных»
Тема: Динамическое кодирование и декодирование по Хаффману –
сравнительное исследование со статическим методом и методом
Фано-Шеннона.

Студент гр. 1384

Усачева Д.В.

Преподаватели

Иванов Д.В.

Санкт-Петербург

2022

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент: Усачева Д.В.

Группа: 1384

Вариант 1

Тема работы: Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со статическим методом и методом Фано-Шеннона.

Задание: Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со статическим методом и методом Фано-Шеннона.

"Исследование" – реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Предполагаемый объем пояснительной записки: Не менее 15 страниц.

Дата выдачи задания: 25.10.2022

Дата сдачи реферата:

Дата защиты реферата:

Студент гр. 1384

Усачева Д.В.

Преподаватели

Иванов Д.В.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1. Описание алгоритмов	5
1.1. Алгоритм Шеннона-Фано	5
1.2. Статический алгоритм Хаффмана	6
1.3. Динамический алгоритм Хаффмана	8
2. Сравнение алгоритмов и результатов кодирования	11
2.1. Сравнение работы алгоритмов	11
2.2. Сравнение результатов кодирования	12
2.3. Сравнение времени кодирования	13
ЗАКЛЮЧЕНИЕ	16
ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ	17

ВВЕДЕНИЕ

Цель работы – создать программу для сравнения динамического и статического методов Хаффмана и метода Фано-Шеннона.

Для выполнения работы необходимо выполнить следующие задачи:

- Реализовать структуры данных и алгоритмы для каждого метода.
- Реализовать сравнение результатов различных алгоритмов сжатия

текста.

1. ОПИСАНИЕ АЛГОРИТМОВ

Тема, рассматриваемая в курсовой, работе-это алгоритмы сжатия текста без потерь. Алгоритмы Хаффмана и Шеннона-Фано, сжимают текст путем уменьшения избыточности информации, заменой длинных последовательностей на более короткие.

1.1. Алгоритм Шеннона-Фано

Этапы алгоритма Шеннона-Фано:

- Символы алфавита выписывают по убыванию вероятностей.
- Символы полученного алфавита делят на две части, суммарные вероятности символов которых приблизительно равны друг другу.
- В префиксном коде для первой части алфавита присваивается двоичная цифра «0», второй части — «1».
- Полученные части рекурсивно делятся, и их частям назначаются соответствующие двоичные цифры в префиксном коде.

Для реализации данного алгоритма были написаны функции `count_Symbols`, `encode_ShF`, `encode_ShF_str` и класс `Symbol_SF`.

В классе `Symbol_SF` описаны два поля – символ и его встречаемость в полученной строке. Также было переопределено сравнение элементов для сортировки.

Функция `count_Symbols` получает на вход строку, из которой мы формируем наш алфавит, состоящий из элементов класса `Symbol_SF`. В алфавите записаны все уникальные элементы и количество их вхождений. Алфавит был отсортирован при помощи метода `sorted` (алгоритмом сортировки в данном методе является Timsort, скорость работы которого в лучшем случае $O(n)$, а в худшем и среднем случае $O(n \log n)$). В этой функции совершается первое прохождение исходной строки ($O(n)$, где n -количество символов в строке).

Для кодирования алгоритмом Шеннона-Фано была реализована

функция `encode_ShF_str`, принимающая входную строку и пустой словарь для записи кодов. Внутри нее при помощи функции `encode_ShF` формируется словарь, ключами которого являются символы алфавита, а значениями их двоичные коды различной длины. Далее мы уже второй раз проходимся по исходной строке ($O(n)$) и формируем новую строку, где для каждого символа начальной строки записывается его код. Таким образом, мы и получаем закодированное сообщение.

Функция `encode_ShF` получает на вход упорядоченный алфавит, длину строки, пустой словарь и текущий код для кодирования символов. Функция работает рекурсивно, она каждый раз делит алфавит примерно поровну (по количеству вхождений символов левая часть меньше или равна правой по сумме вероятностей), и продолжает пока длина поступающего алфавита не станет равна 1 или 2, в этом случае значения кода символа записывается в словарь. При каждом делении в левой части массива к коду добавляется «0», а правой «1». Рекуррентная формула работы алгоритма $T(n) = T(k) + T(n-k) + O(n)$, где $k < n$ или же в лучшем случае $T(n) = 2T(n/2) + O(n)$, тогда скорость работы будет $O(n^2)$ и $O(n \log n)$ соответственно.

Теоретическое общее время работы алгоритма $O(n \log n)$ в лучшем и $O(n^2)$ в худшем случае, что совпадает с практически полученными значениями.

1.2. Статический алгоритм Хаффмана

Этапы статического алгоритма Хаффмана:

- Строится бинарное дерево Хаффмана на основе первичного алфавита.
- Берутся два узла (символа) с наименьшим весом и объединяются в новый узел.
- Старые узлы удаляются, а новый добавляется
- Коды символов получаются через обход полученного дерева.

Для реализации данного алгоритма были написаны функции `count_Symbols`, `st_Huffman_Tree`, `encode_stH`, `encode_stH_str` и реализованы классы `Symbol_stH` и `Hear`.

В классе `Symbol_stH` описаны четыре поля – символ, его встречаемость в полученной строке и левый и правый потомки. Также было переопределено сравнение элементов для сортировки.

Класс `Hear` был реализован как мин куча. Определены методы получения левого и правого ребёнка, родителя и размера кучи. Также были определены методы вставки, извлечения минимума, просеивания вниз и вверх.

Функция `count_Symbols` была реализована аналогично одноименной функции в предыдущем разделе, за исключением сортировки полученного алфавита. В данном алгоритме сортировка происходит в другой функции. ($O(n)$)

Функция построения двоичного дерева `st_Huffman_Tree`. Сначала мы определяем полученный алфавит как кучу, поочередно (для получения отсортированной кучи) добавляя каждый элемент. Далее в цикле мы каждый раз извлекаем два элемента с минимальной встречаемостью и объединяем их в один, пока размер кучи не станет 1. Возвращаемое значение данной функции — это корень полученного дерева. (создание отсортированной кучи $O(n \log n)$, построение дерева $O(\log n)$). Итого: $O(n \log n)$)

Код каждого символа мы получаем в функции `encode_stH`, где мы рекурсивно проходимся по дереву от корня к листьям, добавляя к текущему коду «1», если мы идем к правому потомку и «0», если мы идем к левому. Условием для записи кода в словарь является, отсутствие потомков и непустое значение символа в узле(листе). ($O(n)$ -простой обход дерева)

Для кодирования статическим алгоритмом Хаффмана функция `encode_stH_str`, принимает входную строку и пустой словарь для записи кодов. Получив коды каждого символа, при помощи функции `encode_stH`, мы обходим входную строку посимвольно, на каждом шаге записывая в результирующую строку код текущего символа. Таким образом, мы и

получаем закодированное сообщение. ($O(n)$)

Теоретическое общее время работы алгоритма $O(n \log n)$, что совпадает с практически полученными значениями.

1.3. Динамический алгоритм Хаффмана

FGK алгоритм:

Он позволяет динамически регулировать дерево Хаффмана, не имея начальных частот. В ФГК дереве Хаффмана есть особый внешний узел, называемый 0-узел, используемый для идентификации входящих символов. То есть, всякий раз, когда встречается новый символ — его путь в дереве начинается с нулевого узла. Самое важное — то, что нужно усекать и балансировать ФГК дерево Хаффмана при необходимости, и обновлять частоту связанных узлов. Как только частота символа увеличивается, частота всех его родителей должна быть тоже увеличена. Это достигается путём последовательной перестановки узлов, поддеревьев или и тех и других. В дереве веса символов идут в порядке убывания.

Важной особенностью ФГК дерева является принцип братства (или соперничества): каждый узел имеет два потомка (узлы без потомков называются листьями) и веса идут в порядке убывания. Благодаря этому свойству дерево можно хранить в обычном массиве, что увеличивает производительность.

Этапы динамического алгоритма Хаффмана (FGK алгоритм):

- Инициализируется FGK дерево.
- Если символ встречается в первый раз, то он добавляется в качестве брата для 0-узла, и происходит обновление дерева. Если символ уже есть в дереве, то его частота увеличивается, и дерево обновляется.
- Коды символов получаются через обход полученного дерева.

Для реализации данного алгоритма была написана функция

encode_dinH_str и реализованы классы Symbol_dinH и Din_tree.

В классе Symbol_dinH описаны пять полей – символ, его встречаемость в полученной строке, родитель и левый и правый потомки. Также было переопределено сравнение элементов для сортировки.

Класс Din_tree имеет одно поле и 4 метода для работы с деревом.

Первый метод – это вставка, он используется если добавляется новый элемент. Сначала создается два новых элемента, один из которых ставится на место нулевого узла, и в качестве потомков ему назначаются нулевой узел и второй, созданный нами узел, содержащий наш символ. Затем мы вызываем метод rebuild, чтобы сбалансировать наше дерево. ($O(1)$ без балансирования)

Второй метод — это добавление встречаемости элемента, который уже есть в нашем дереве. В нем мы ищем место элемента в нашем дереве, и переопределяем входящий элемент на элемент из дерева (чтобы сохранить данные о родителе, потомках, встречаемости). Затем мы вызываем метод rebuild, чтобы сбалансировать наше дерево. ($O(n)$ без балансирования)

Третий метод — это балансировка и усечение нашего дерева. Он занимается увеличением кол-ва вхождения для рассматриваемого символа (как и всей ветки), а также восстановлением свойств FGK дерева. Происходит это следующим образом: если в дереве встречается вершина с таким же весом, как и у рассматриваемой вершины, но на более верхнем уровне либо на том же уровне, но левее, то эти узлы меняются местами, включая всех потомков. Далее вес этой вершины увеличивается, и мы рассматриваем родителя. Алгоритм повторяется, пока не дойдём до корня. Таким образом мы увеличиваем вес для всех вершин ветки, при этом не нарушая свойств FGK дерева. (k – текущее количество элементов в нашем дереве, тогда нам необходимо обработать до $\log k$ элементов (высота дерева), лучший случай работы алгоритма будет в случае, если нам необходимо поменять последний элемент с самым верхним после корня.)

Четвертый метод — это получение кода для каждого символа путем рекурсивного обхода дерева, метод аналогичен encode_stH из предыдущего

раздела, за исключением одного аспекта — учёт нулевого узла. ($O(n)$ обход дерева).

Функция `encode_dinH_str` обходит полученную строку посимвольно, вставляя новые элементы и увеличивая значения уже вошедших. При помощи метода `encode_dinH` мы получаем код каждого символа. Далее мы обходим входную строку посимвольно, на каждом шаге записывая в результирующую строку код текущего символа. Таким образом, мы и получаем закодированное сообщение.

Теоретическое общее время работы алгоритма $O(n \log n)$ в лучшем и $O(n^2)$ в худшем случае, что совпадает с практически полученными значениями.

2. СРАВНЕНИЕ АЛГОРИТМОВ И РЕЗУЛЬТАТОВ КОДИРОВАНИЯ

2.1. Сравнение работы алгоритмов

Различие между статическими алгоритмами и динамическим.

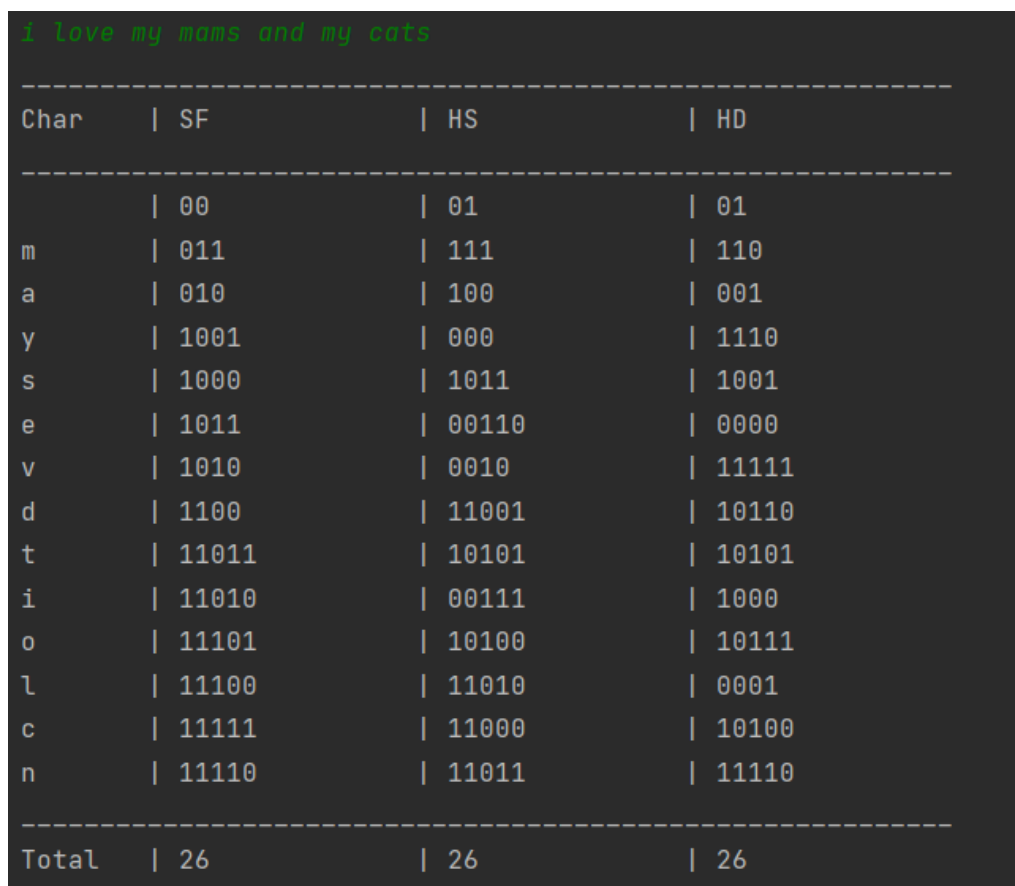
Так как статические алгоритмы сжимают данные за счёт вероятностей появления символов в источнике, следовательно, для того чтоб успешно что-то сжать и разжать нам потребуется знать эти самые вероятности для каждого символа. Статические алгоритмы перед тем как начать сжатие файла программа быстренько пробегается по самому файлу и подсчитывает какой символ сколько раз встречается. Затем, в соответствии с вероятностями появлений символов, извлекаются соответствующие каждому символу коды разной длины. И на третьем этапе снова осуществляется проход по исходному файлу, когда каждый символ заменяется на свой древесный код. Таким образом статическому алгоритму требуется два прохода по файлу источнику, чтоб закодировать данные.

Динамический алгоритм позволяет реализовать однократную модель сжатия. Не зная реальных вероятностей появлений символов в исходном файле - программа постепенно изменяет двоичное дерево с каждым встречаемым символом увеличивая частоту его появления в дереве и перестраивая в связи с этим само дерево. Однако становится очевидным, что, выиграв в количестве проходов по исходному файлу - мы начинаем терять в качестве сжатия, так как в статическом алгоритме частоты встречаемости символов были известны с самого начала и длины кодов этих символов были более близки к оптимальным, в то время как динамическая модель изучая источник постепенно доходит до его реальных частотных характеристик и узнаёт их лишь полностью пройдя исходный файл.

2.2. Сравнение результатов кодирования

Для всех алгоритмов была написана общая функция декодирования, которая принимает закодированную строку и словарь со значениями кода для каждого символа, возвращаемое значение функции — это исходная строка. Для сравнения результатов кодирования запускается программа, в которой одна и та же строка кодируется и декодируется каждым алгоритмом.

Затем выводится таблица (см. Рисунок 1), содержащая информацию о кодах каждого символа и общая длина алфавита. После этого выводиться информация о каждом из методов. (см. Рисунок 2)



```
i love my mams and my cats
```

Char	SF	HS	HD
	00	01	01
m	011	111	110
a	010	100	001
y	1001	000	1110
s	1000	1011	1001
e	1011	00110	0000
v	1010	0010	11111
d	1100	11001	10110
t	11011	10101	10101
i	11010	00111	1000
o	11101	10100	10111
l	11100	11010	0001
c	11111	11000	10100
n	11110	11011	11110
Total	26	26	26

Рисунок 1 - Пример таблицы сравнения методов

```
Shannon-Fano encoding:
  encoded: 1101000111001110110101011000111001000110100111000000101111011000001110010011111010110111000
  length: 91
  decoded: i love my mams and my cats
Static Huffman encoding:
  encoded: 0011101110101010000100011001111000011111001111011011001101111001011110000111000100101011011
  length: 91
  decoded: i love my mams and my cats
Dynamic Huffman encoding:
  encoded: 100001000110111111100000111011100111000111010010100111110101100111011100110100001101011001
  length: 91
  decoded: i love my mams and my cats
```

Рисунок 2 - Пример информации о каждом из методов

По результатам примера можно увидеть, что все три метода кодируют сообщения одинаково коротко, несмотря на то, что коды некоторых символов отличаются. Также можно заметить, что все закодированные строки корректно декодируются.

2.3. Сравнение времени кодирования

Для сравнения времени кодирования алгоритмов была создана отдельная программа. Для наглядности необходимо брать более длинные сообщения. Каждый шаг строка увеличивается на 500 символов.

В начале программы описана строка, состоящая из 120 уникальных символов. Далее описана переменная, хранящая кол-во повторений для каждого символа этой строки. Затем измеряется время кодирования разными методами. Количество шагов для каждого метода равно 120.

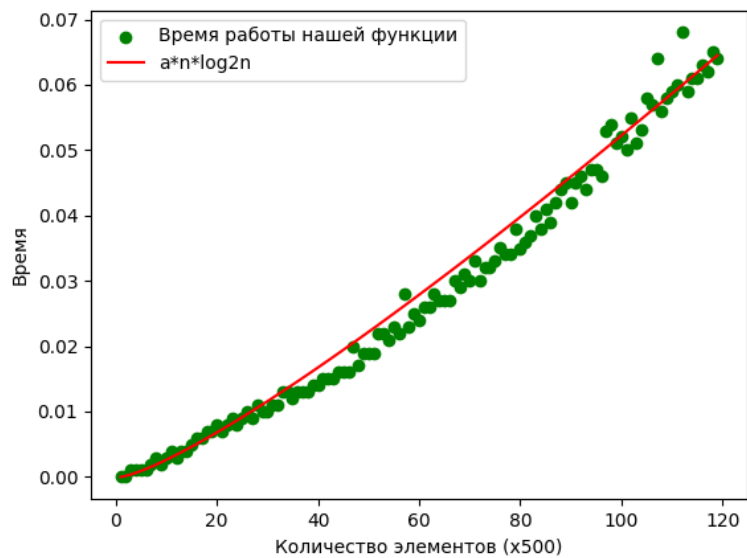


Рисунок 3 – График зависимости времени работы алгоритма Шеннона-Фано от количества элементов

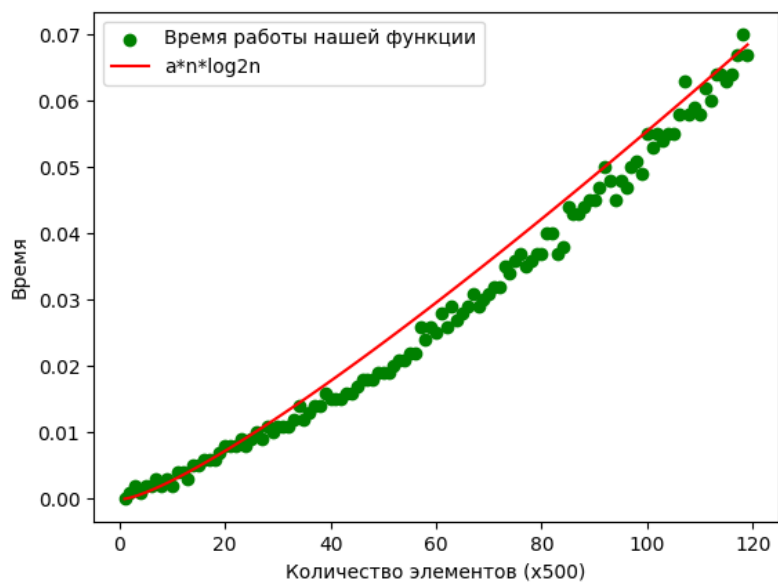


Рисунок 4 – График зависимости времени работы статического алгоритма Хаффмана от количества элементов

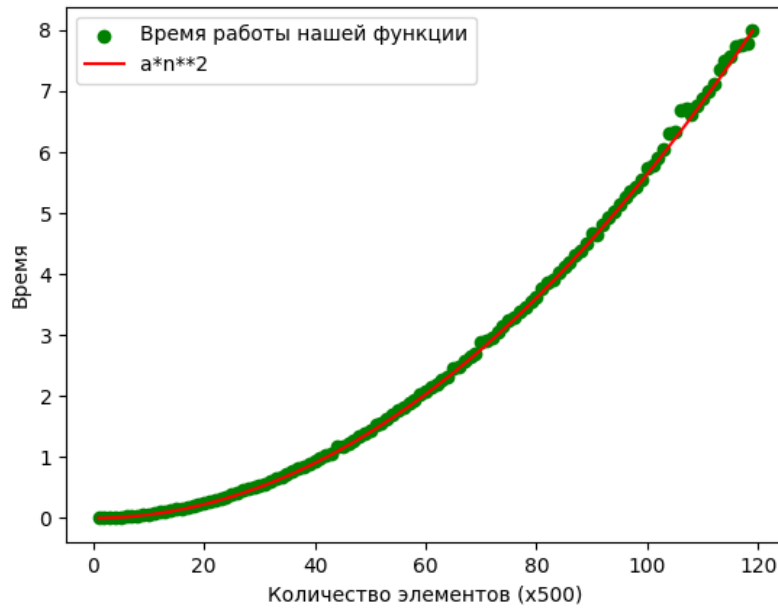


Рисунок 5 – График зависимости времени работы динамического алгоритма Хаффмана от количества элементов

По результатам выполнения программы можно заметить, что кодирование исходной строки по статическому методу Хаффмана занимает примерно столько же времени, сколько и по методу Шеннона-Фано.

Кодирование динамическим алгоритмом Хаффмана оказалось в разы дольше других, что не удивительно, ведь если первые два метода, в основном, зависят от размера первичного алфавита, то динамический алгоритм во многом зависит от размера входной строки в целом, иными словами, медлительность обусловлена динамичностью.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы была создана программа на языке Python для динамического кодирования и декодирования по Хаффману, проведено сравнительное исследование со статическим методом и методом Фано-Шеннона.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл alg_ShF.py

```
class Symbol_SF:
    def __init__(self, char, count):
        self.char = char
        self.count = count

    def __lt__(self, other):
        if isinstance(other, Symbol_SF):
            if self.count != other.count:
                return self.count > other.count

def count_Symbols(string):
    symbols_set = list(set(string))
    occ_in_str = [0] * len(symbols_set)
    for char in string:
        occ_in_str[symbols_set.index(char)] += 1
    symbols_s_c = []
    for char in range(len(symbols_set)):
        symbols_s_c.append(Symbol_SF(symbols_set[char],
occ_in_str[char]))
    symbols_s_c = sorted(symbols_s_c)
    return symbols_s_c

def encode_ShF(symbols_s_c, sum, dictionary, code=''):
    if len(symbols_s_c) == 1:
        dictionary[symbols_s_c[0].char] = code
        return 0
    if len(symbols_s_c) == 2:
        dictionary[symbols_s_c[0].char] = code + '1'
        dictionary[symbols_s_c[1].char] = code + '0'
        return 0
    left_id, right_id = 0, len(symbols_s_c)
    left_Sum, right_Sum = 0, 0
    left, right = [], []
    while left_Sum < sum // 2:
        left.append(symbols_s_c[left_id])
        left_Sum += symbols_s_c[left_id].count
        left_id += 1
    for i in range(left_id, right_id):
        right.append(symbols_s_c[i])
        right_Sum += symbols_s_c[i].count
    encode_ShF(left, left_Sum, dictionary, code + '0')
    encode_ShF(right, right_Sum, dictionary, code + '1')

def encode_ShF_str(string, dictionary):
    symbols_s_c = count_Symbols(string)
    encode_ShF(symbols_s_c, len(string), dictionary)
    code_str = ''
    for char in string:
```

```

        code_str += dictionary[char]
    return code_str

if (__name__ == "__main__"):
    string = input()
    codes = dict()
    if len(count_Symbols(string)) <= 1:
        print(string[0] + ': ', '0')
        print('0' * len(string), len(string))
    else:
        output = encode_ShF_str(string, codes)
        for char in codes.keys():
            print(f'{char}: {codes[char]}')
        print(output, len(output))

```

Файл Heap.py

```

class Heap:

    def __init__(self, heap):
        self.heap = []
        for elem in heap:
            self.insert(elem)

    @staticmethod
    def get_parent(index):
        return (index - 1) // 2

    @staticmethod
    def get_left_child(index):
        return 2 * index + 1

    @staticmethod
    def get_right_child(index):
        return 2 * index + 2

    def size(self):
        return len(self.heap)

    def insert(self, element):
        self.heap.append(element)
        self.sift_up(len(self.heap)-1)

    def extract_min(self):
        min_element = self.heap[0]
        self.heap[0] = self.heap[-1]
        del self.heap[-1]
        self.sift_down(0)
        return min_element

    def sift_up(self, index):
        if index < 0 or index >= len(self.heap):
            return
        parent = self.get_parent(index)
        while index > 0 and self.heap[parent] > self.heap[index]:
            self.heap[parent], self.heap[index] = self.heap[index],
self.heap[parent]

```

```

        index = parent
        parent = self.get_parent(index)

    def sift_down(self, index):
        if index < 0 or index >= len(self.heap):
            return
        left = self.get_left_child(index)
        right = self.get_right_child(index)
        if left >= self.size() and right >= self.size():
            return
        if right >= self.size():
            min_index = left if self.heap[left] < self.heap[index] else
index
        else:
            min_index = left if self.heap[left] < self.heap[right] else
right
            min_index = min_index if self.heap[min_index] <
self.heap[index] else index
        if min_index != index:
            self.heap[min_index], self.heap[index] = self.heap[index],
self.heap[min_index]
            self.sift_down(min_index)

```

Файл alg_stH.py

```

import Heap

class Symbol_stH:
    def __init__(self, char, count, left=None, right=None):
        self.char = char
        self.count = count
        self.left, self.right = left, right

    def __lt__(self, other):
        if self.count != other.count:
            return self.count < other.count
        else:
            return self.char < other.char

    def __gt__(self, other):
        if self.count != other.count:
            return self.count > other.count
        else:
            return self.char > other.char

def count_Symbols(string):
    symbols_set = list(set(string))
    occ_in_str = [0] * len(symbols_set)
    for char in string:
        occ_in_str[symbols_set.index(char)] += 1
    symbols_s_c = []
    for char in range(len(symbols_set)):
        symbols_s_c.append(Symbol_stH(symbols_set[char],
occ_in_str[char]))
    return symbols_s_c

```

```

def st_Huffman_Tree(symbols_s_c):
    heap = Heap.Heap(symbols_s_c)
    while heap.size() > 1:
        left = heap.extract_min()
        right = heap.extract_min()
        sum_count_lr = left.count + right.count
        heap.insert(Symbol_stH('', sum_count_lr, left, right))
    return heap.extract_min()

def encode_stH(hf_tree, dictionary, code=''):
    cur = hf_tree
    right, left = cur.right, cur.left
    if right != None:
        encode_stH(right, dictionary, code + '1')
    if left != None:
        encode_stH(left, dictionary, code + '0')
    if cur.char != '':
        dictionary[cur.char] = code

def encode_stH_str(string, dictionary):
    hf_tree = st_Huffman_Tree(count_Symbols(string))
    encode_stH(hf_tree, dictionary)
    code_str = ''
    for char in string:
        code_str += dictionary[char]
    return code_str

if (__name__ == "__main__"):
    string = input()
    if len(count_Symbols(string)) <= 1:
        print(string[0] + ':', '0')
        print('0' * len(string), len(string))
    else:
        codes = dict()
        output = encode_stH_str(string, codes)
        for ch in codes.keys():
            print(f'{ch}: {codes[ch]}')
        print(output, len(output))

```

Файл alg_dinH.py

```

from din_tree import Din_tree

def encode_dinH_str(string, codes):
    tree = Din_tree()
    tree.insert(string[0])
    for i in range(1, len(string)):
        if string[i] in string[:i]:
            tree.add_count(string[i])
        else:
            tree.insert(string[i])
    tree.encode_dinH(codes)
    code = ''
    for char in string:

```

```

        code += codes[char]
    return code

```

```

if (__name__ == "__main__"):
    string = input()
    codes = dict()
    output = encode_dinH_str(string, codes)
    for ch in codes.keys():
        print(f'{ch}: {codes[ch]}')
    print(output, len(output))

```

Файл din_tree.py

```

class Symbol_dinH:
    def __init__(self, char="", left=None, right=None, parent=None):
        self.char = char
        self.count = 0
        self.left = left
        self.right = right
        self.parent = parent

    def __lt__(self, other):
        if self.count != other.count:
            return self.count < other.count
        else:
            return self.char < other.char

    def __gt__(self, other):
        if self.count != other.count:
            return self.count > other.count
        else:
            return self.char > other.char

class Din_tree:
    def __init__(self):
        self.arr = [Symbol_dinH()]

    def insert(self, element):
        element = Symbol_dinH(element)
        null_el = self.arr.pop()
        node = Symbol_dinH("", element, null_el)
        if null_el.parent:
            null_el.parent.left = node
            node.parent = null_el.parent
        element.parent, null_el.parent = node, node
        node.right, node.left = element, null_el
        self.arr.extend([node, element, null_el])
        self.rebuild(element)

    def add_count(self, element):
        for i in self.arr:
            if i.char == element:
                element = i
                break
        self.rebuild(element)

```

```

def rebuild(self, element):
    cur = element
    while True:
        if not cur:
            break
        parents, childs = [self.arr[0]], []
        end = False
        while not end:
            for symbol_dh in parents:
                if symbol_dh.left:
                    childs.append(symbol_dh.left)
                if symbol_dh.right:
                    childs.append(symbol_dh.right)
                if symbol_dh == cur:
                    end = True
                    break
            if symbol_dh.count == cur.count:
                if cur.char and not symbol_dh.char:
                    continue
                if symbol_dh.parent and symbol_dh.parent.right
== symbol_dh:
                    symbol_dh.parent.right = cur
                elif symbol_dh.parent:
                    symbol_dh.parent.left = cur
                if cur.parent.right == cur:
                    cur.parent.right = symbol_dh
                else:
                    cur.parent.left = symbol_dh
                    cur.parent, symbol_dh.parent =
symbol_dh.parent, cur.parent
            end = True
            break
        parents = childs
        childs = []
        cur.count += 1
        cur = cur.parent

def encode_dinH(self, dictionary, top=None, code=''):
    if top:
        cur = top
    else:
        cur = self.arr[0]
    right, left = cur.right, cur.left
    if right and right.count != 0:
        self.encode_dinH(dictionary, right, code + '0')
    if left and left.count != 0:
        self.encode_dinH(dictionary, left, code + '1')
    if cur.char != '':
        if cur.parent.left and not cur.parent.left.count:
            dictionary[cur.char] = code[:-1]
        else:
            dictionary[cur.char] = code

```

Файл decode.py

```

from alg_ShF import encode_ShF_str
from alg_stH import encode_stH_str

```

```

from alg_dinH import encode_dinH_str

def decode(string, codes):
    outputString = ''
    codedChar = ''
    for char in string:
        codedChar += char
        if codedChar in codes.values():
            outputString +=
list(codes.keys())[list(codes.values()).index(codedChar)]
            codedChar = ''
    return outputString

if (__name__ == "__main__"):
    string = input()

    codesSF = dict()
    outputSF = encode_ShF_str(string, codesSF)
    decodeSF = decode(outputSF, codesSF)
    len_SF = 0
    for i in codesSF.values():
        len_SF += len(i)

    codesHS = dict()
    outputHS = encode_stH_str(string, codesHS)
    decodeHS = decode(outputHS, codesHS)
    len_HS = 0
    for i in codesHS.values():
        len_HS += len(i)

    codesHD = dict()
    outputHD = encode_dinH_str(string, codesHD)
    decodeHD = decode(outputHD, codesHD)
    len_HD = 0
    for i in codesHD.values():
        len_HD += len(i)

    tableLayout = "{:<8}| {:<15}| {:<15}| {:<15}"
    headers = tableLayout.format("Char", "SF", "HS", "HD")
    print("_" * len(headers))
    print(headers)
    print("_" * len(headers))
    for ch in codesSF.keys():
        print(tableLayout.format(ch, codesSF[ch], codesHS[ch],
codesHD[ch]))
    print("_" * len(headers))
    print(tableLayout.format("Total", len_SF, len_HS, len_HD))
    print('')

    print("Shannon-Fano encoding:\n\tencoded:", outputSF,
"\n\tlength:", len(outputSF),
"\n\tdecoded:", decodeSF)
    print("Static Huffman encoding:\n\tencoded:", outputHS,
"\n\tlength:", len(outputHS),
"\n\tdecoded:", decodeHS)
    print("Dynamic Huffman encoding:\n\tencoded:", outputHD,

```



```

# plt.show()
plt.savefig("alg_sH.png")
plt.clf()

yCoord = timings[2]
plt.scatter(xCoord, yCoord, color="green")
# plt.plot(xCoord, yCoord)
a = yCoord[-1] / xCoord[-1] ** 2
yCoord = [a * x ** 2 for x in xCoord]
plt.plot(xCoord, yCoord, color="red")
plt.xlabel("Количество элементов (x500)")
plt.ylabel("Время")
plt.legend(["Время работы нашей функции", "a*n**2"])
# plt.show()
plt.savefig("alg_dinH.png")
plt.clf()

print("Finished")

```