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

ОТЧЕТ

по курсовой работе
по дисциплине «Алгоритмы и структуры данных»
Тема: «Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со статическим методом и методом Фано-Шеннона»

Студент гр. 1381 _____ Таргонский М.А.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург
2022

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент: Таргонский М.А.

Группа: 1381

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

Вариант 1 (На «Хорошо»)

Задание: Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со статическим методом и методом ФаноШеннона.

"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.

Дата выдачи задания: 25.10.2022

Дата сдачи реферата: 23.12.2022

Дата защиты реферата: 24.12.2022

АННОТАЦИЯ

В процессе выполнения курсовой работы создавались программы для сравнения трёх методов кодирования информации на языке Python 3.7.0. Разработка велась на операционной системе Windows 11 x64 в редакторе исходного кода Visual Studio Code.

СОДЕРЖАНИЕ

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 методов по длине закодированного кода, по времени кодирования и по корректности декодирования

2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

2.1. Алгоритмы сжатия текста без потерь

В данной курсовой работе рассматриваются алгоритмы кодирования Хаффмана и Фано-Шеннона, которые предназначаются для сжатия данных без потерь за счёт уменьшения избыточности информации путём уменьшения длинных последовательностей на более короткие. При написании программы необходимо учесть, что все эти алгоритмы являются префиксными, то есть для закодированного кода выполняется условие Фано.

2.2. Реализация алгоритма Шеннона-Фано

Алгоритм Шеннона-Фано состоит из следующих этапов:

- Символы первичного алфавита сортируются по убыванию вероятностей.

- Полученный массив символов делится примерно поровну по сумме вероятностей (левая часть меньше или равна правой по сумме вероятностей).

- Для первой части назначается число “0”, а для другой – “1”.

- Каждая часть продолжает рекурсивно делиться с присвоением чисел каждой ветке, пока размер частей не станет единичным.

- В результате деления символам назначаются префиксные коды разной длины.

Для кодирования алгоритмом Шеннона-Фано была реализована функция ShannonFanoEncode, принимающая входную строку и пустой словарь для записи кодов.

Внутри функции происходит подсчёт вхождений каждого символа в строку (функция countLetters), а затем символы сортируются по количеству вхождений в порядке убывания с помощью макс-кучи (функция heapSort). Сами символы хранятся в классе Letter с полями char (символ), count (кол-во

вхождений), code (поле для кода), а также с переопределённой операцией сравнения для сортировки.

Далее вызывается рекурсивная функция encode, которая принимает упорядоченный список символов и делит его примерно пополам по сумме вероятностей (левая часть меньше или равна правой по сумме вероятностей): массив проходится одновременно с двух сторон, на каждом шаге добавляется один элемент к той части, где сумма вероятностей меньше. Для каждой части эта функция запускается вновь, но с одним отличием: первому "ответвлению" назначается число “0”, а другому – “1”. Рекурсия заканчивается, когда в списке остаётся два элемента, либо один элемент. В первом случае к коду символов добавляется “0” и “1” соответственно, и значения записываются в словарь. Во втором случае значение кода является окончательным и записывается в словарь сразу.

В конце формируется новая строка, где для каждого символа начальной строки записывается его код – эта строка и будет закодированным сообщением, которое возвращается.

Время, за которое выполняется кодирование, в теории должно быть от O(nlog2n) в лучшем случае до O(n^2)в худшем случае. Проверим, будет ли это соответствовать написанной программе (s – кол-во символов в исходном сообщении, n – кол-во уникальных символов):

- countLetters: Сначала входная строка преобразуется в set, а затем обратно в list (обе операции линейны в Python), и создаётся занулённый список размером с кол-во уникальных символов (размер сета). Затем входная строка проходится один раз с подсчётом символов, после чего для каждого символа создаётся объект класса Letter. В итоге, аппроксимация по времени этой функции составляет O(s).
- heapSort: В худшем случае сортировка кучей имеет аппроксимацию O(nlog2n)

- encode: На каждой итерации мы проходим входной список один раз. Рекуррентная формула: T(n) = T(k)+T(n-k)+O(n). В худшем случае, на каждом этапе мы будем отделять один элемент, т.е. T(n) = T(1)+T(n1)+O(n), следовательно T(n) будет O(n^2). В лучшем случае, части будут равны, и тогда T(n) = 2T(n/2)+O(n), следовательно T(n) будет O(nlog2n).

- Общее время = O(nlog2n) в лучшем и O(n^2) в худшем случае.

Таким образом, время реализованного алгоритма совпадает с теоретическими оценками.

2.3. Реализация статического алгоритма Хаффмана

Статический алгоритм Хаффмана состоит из следующих этапов:

- Строится бинарное дерево Хаффмана на основе первичного алфавита.

1) Берутся два узла (символа) с наименьшим весом (вероятностью) и бъединяются в новый узел.

2) Старые узлы удаляются, а новый добавляется

- Коды символов получаются через обход полученного дерева.

Для статического кодирования алгоритмом Хаффмана была реализована функция HuffmanEncode, принимающая входную строку и пустой словарь для записи кодов.

Внутри функции происходит подсчёт вхождений каждой буквы в строку (функция countLetters), а затем строится дерево Хаффмана (функция buildHuffmanTree). Узлы дерева реализованы через класс Letter: класс схож с тем, который описывался в предыдущей главе, только здесь ещё реализованы поля для двух потомков. Узел, который не является символом, содержит пустое поле char.

Построение дерева происходит следующим образом: создаётся минкуча на основе первичного алфавита, затем в цикле из дерева достаются два наименьших элемента (через метод мин-кучи), на их основе создаётся новое,

которое помещается обратно в мин-кучу. Когда остался один элемент в куче, цикл заканчивается, и этот элемент возвращается.

Построение кода проходит в функции buildCodes, в котором дерево рекурсивно обходится с назначением чисел “0” и “1” соответствующим ветвям.

В теории статический алгоритм Хаффмана требует времени O(nlog2n), тем временем в реализованном алгоритме (s – кол-во символов в исходном сообщении, n – кол-во уникальных символов):

- countLetters: Как и прошлом случае, O(s).

- buildHuffmanTree: Создание кучи на основе входного списка требует O(n). Далее идёт цикл с около n итераций, в котором достаётся два раза достаётся верхний элемент и вставляется новый (обе операции O(log2n)). Итого получаем O(nlog2n).

- buildCodes: Простой обход дерева – O(n).

- Общее время = O(nlog2n).

Таким образом, время реализованного алгоритма совпадает с теоретическими оценками.

2.4. Реализация динамического алгоритма Хаффмана

Существует несколько реализаций динамического кода Хаффмана, в данной курсовой работе был выбран FGK алгоритм:

- Инициализируется FGK дерево. У этого дерево несколько особенностей: во-первых, в дереве есть специальный 0-узел, предназначенный для идентификации нового символа; во-вторых, веса идут в порядке убывания; в-третьих, у каждого узла (не листа) всегда есть два потомка.

- 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.

-Для получения кодов нужно просто пройти по дереву.

Этот метод является динамическим, но в целях исследования мы будем использовать входную строку и добавлять каждый её элемент в дерево посимвольно.

Для кодирования реализована функция HuffmanEncode, в котором инициализируется объект класса FGKTree, после чего туда добавляются символы из входящей строки.

Класс FGKTree имеет четыре метода. Первый из них – это конструктор, который создаёт пустой массив под вершины дерева и добавляет туда 0-узел. Элементы дерева реализованы в классе Letter. В отличии от одноимённого класса в предыдущем примере этот класс хранит не только потомков, но и родителя, что позволяет обходить дерево через вершины от корня к листьям и наоборот.

Следующий метод в классе FGKTree – это метод add, который, как не трудно догадаться, добавляет символ в дерево. Сначала происходит поиск этого символа в массиве вершин. Если символ не был найден, то создаётся два новых элемента. Один из них ставиться на место 0-узла, и в качестве потомков ему назначаются второй созданный элемент (в нём хранится символ) и 0-узел. Затем вызывается метод rebuild, которому передаётся вершина с рассматриваемым символом.

Метод rebuild 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.

Чтобы лучше понять ход работы этого метода, рассмотрим пример:

- Пусть уже построено следующее дерево (рис. 2.4.1), и мы хотим увеличить кол-во вхождений ‘b’.

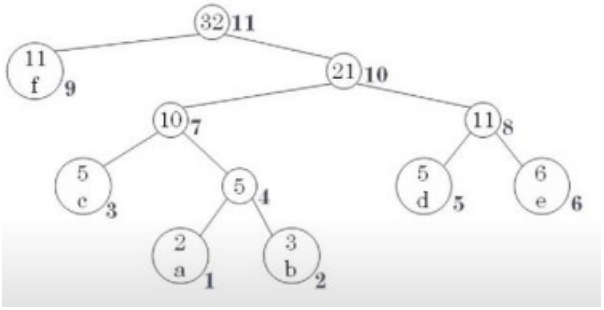


Рис. 2.4.1

- Допустим, мы просто увеличим значения весов ‘b’ и всех родителей по цепочке. Тогда у нас явно нарушается порядок убывания (рис. 2.4.2).

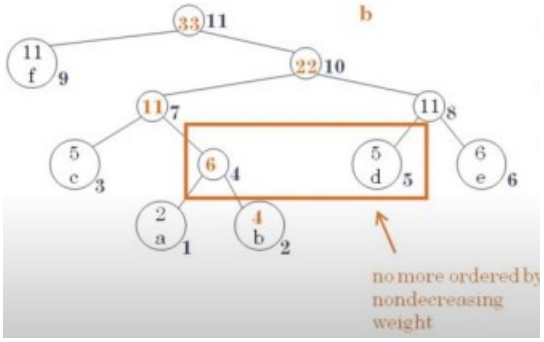
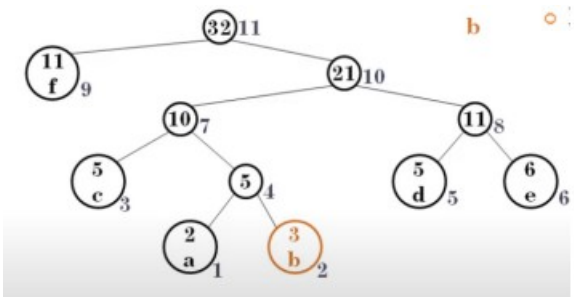
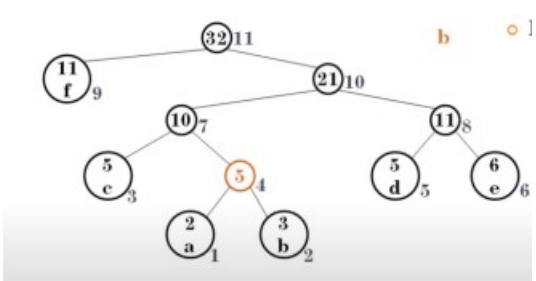


Рис. 2.4.2

- Чтобы восстановить свойство неубывания, перед тем как увеличивать вес очередной вершины, мы пытаемся протолкнуть её вверх (рис. 2.4.3).





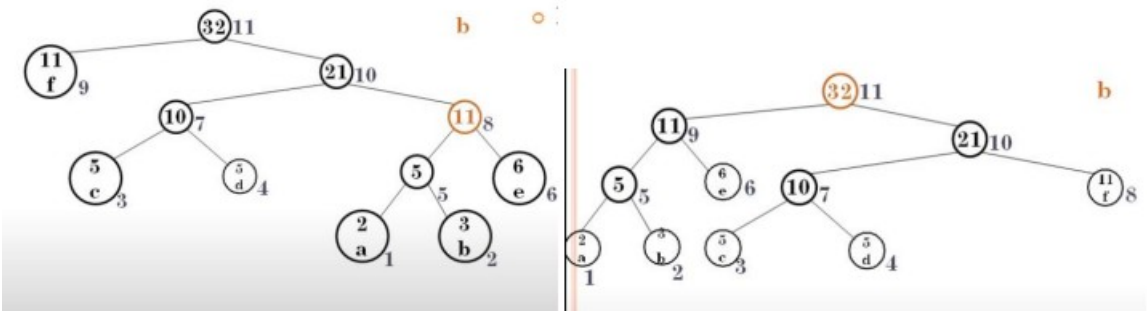


Рис. 2.4.3

- После того, как мы дошли до корня, никаких конфликтов при увеличении весов не будет (рис. 2.4.4).

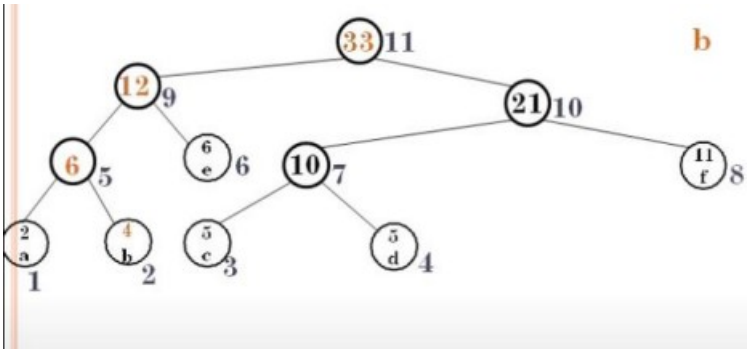


Рис. 2.4.4

Можно заметить, что, по итогу, на каждом этапе FGK дерево по внешнему виду является деревом Хаффмана, поэтому операция построения кодов будет такая же – обход дерева (метод getCodes).

Так как данный динамический алгоритм Хаффмана, в отличие от статического, требует на каждом этапе восстановление свойств FGK дерева, который подразумевает обход дерева на каждом шаге, то по быстродействию он явно будет проигрывать. Рассмотрим реализованный алгоритм:

- add: Поиск символа в дереве – обход – требует O(n). Возможное добавление элемента в дерево требует O(1).

- rebuild: Пусть k – текущее кол-во элементов в дереве. Тогда на каждом вызове этого метода потребуется обработать максимум log2k элементов.

Для каждого элемента мы просматриваем предыдущие и при необходимости меняем. В худшем случае, мы не меняем ветки местами, поэтому для каждой вершины нам необходимо просматривать все вершины до этого.

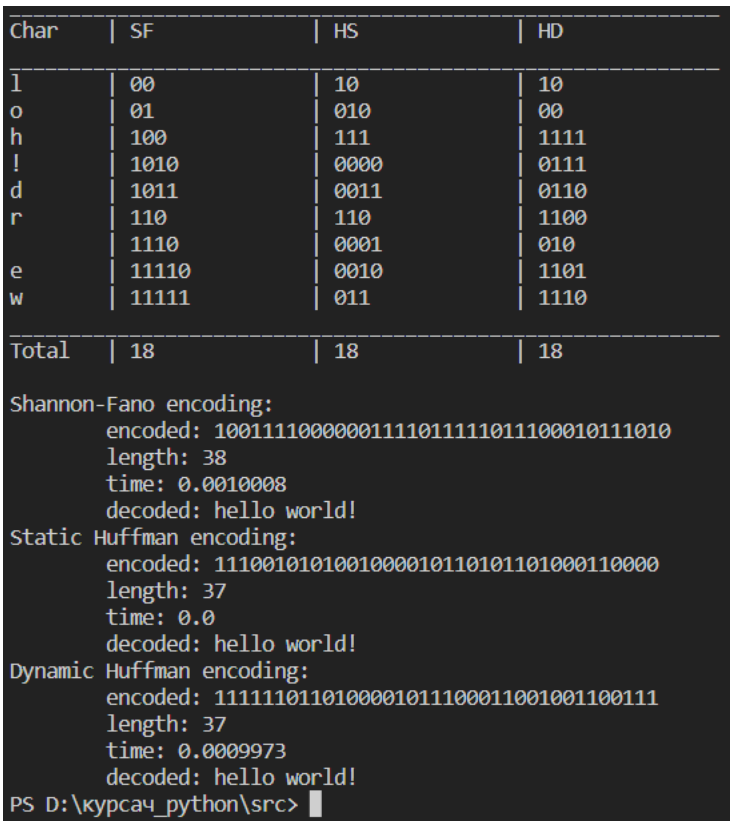
В потенциально лучшем случае, при каждом вызове функции мы будем менять рассматриваемый узел с самым верхним после корня, а это O(s), хотя этого случая на практике скорее всего невозможно добиться, и нам всё равно придётся проходить дерево несколько раз, поэтому более реалистично получить O(slog2s).

- getCodes: обход дерева – O(n)
- Общее время = O(slog2s) → O(s^2)

2.5. Сравнение результатов кодирования

При запуске файла researchEncoding.py, запускается программа для сравнения трёх вышеперечисленных методов. В начале считывается входная строка, которая, затем, кодируется тремя методами.

Затем выводится таблица, содержащая информацию о кодах для каждого символа. Таже в этой таблице подсчитывается длина закодированного алфавита для каждого случая. После этого для каждого метода выводиться следующая информация: название метода, закодированная строка, длина строки, декодированная строка. Для декодирования реализована функция decode, которая принимает закодированную строку и словарь кодов. Эта функция просто считывает комбинацию символов и если эта комбинация является кодом из словаря, то заменяет её исходным символом, поэтому для успешного декодирования необходимо, чтобы соблюдалось условие Фано.



Пример 1 – Сравнение кодирования фразы “hello world!”

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.

2.6. Сравнение времени кодирования

Создание отдельной программы для сравнения времени было обусловлено несколькими причинами. Во-первых, функции time и time_ns библиотеки time в Python имеют маленькую точность в Windows, и при вводе небольших сообщений высок шанс получить нулевое время. Для решения этой проблемы необходимо брать более длинные сообщения. Во-вторых, на основании кодирования сообщений разной длины можно составить графики зависимости времени от кол-ва символов.

В начале программы описана строка string, состоящая из 120 уникальных символов. Далее описана переменная length, хранящая кол-во повторений для каждого символа этой строки. Затем измеряется время кодирования разными методами. Количество измерений для каждого метода – 120 (каждая итерация – это определённое количество символов из string, повторённое length раз).

```
Time measurements
Initial string: ☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→∟↔▲▼ !#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz
Number of repetitions: 500
Step: 2         Symbols: 1000
Step: 3         Symbols: 1500
Step: 4         Symbols: 2000
Step: 5         Symbols: 2500
Step: 6         Symbols: 3000
Step: 7         Symbols: 3500
Step: 8         Symbols: 4000
Step: 9         Symbols: 4500
Step: 10        Symbols: 5000
```

…

```
Step: 110       Symbols: 22000
Step: 111       Symbols: 22200
Step: 112       Symbols: 22400
Step: 113       Symbols: 22600
Step: 114       Symbols: 22800
Step: 115       Symbols: 23000
Step: 116       Symbols: 23200
Step: 117       Symbols: 23400
Step: 118       Symbols: 23600
Step: 119       Symbols: 23800
Step: 120       Symbols: 24000
Finished
```

Рис. 2.6.1 – Подсчёт времени кодирования

В результате работы программы будут получены графики для каждой функции. Также на графиках статического кодирования Хаффмана и Шеннона-Фано будет изображена для сравнения функция t = a*n*log2n, а для динамического кодирования Хаффмана – t = a*n 2 (см. рис. 2.6.2).

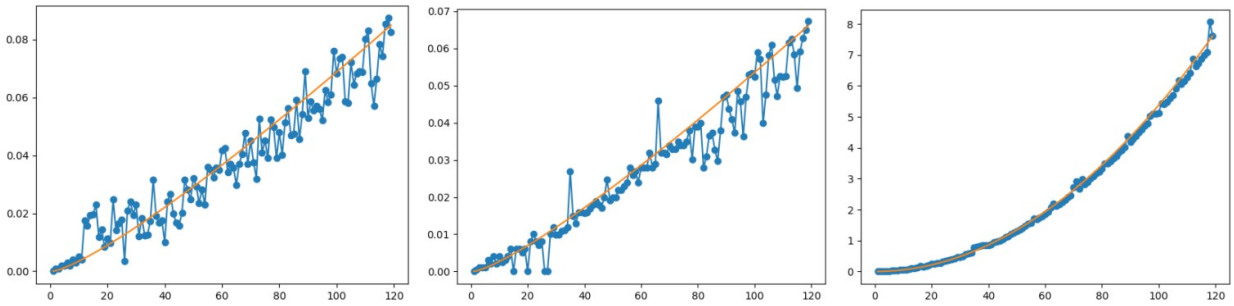


Рис. 2.6.2 – Графики зависимости времени от кол-ва элементов (x500). SF, HS, HD соответственно.

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 (120), то динамический алгоритм во многом зависит от размера входной строки в целом (60 тыс.), иными словами, медлительность обусловлена динамичностью.

Другие примеры работы программы см. в приложении А.

Разработанный программный код см. в приложении В.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы была создана программа на языке Python для сравнительного исследования динамического и статического кода Хаффмана и кода Шеннона-Фано. Программа сравнивает методы по длине закодированного сообщения и по времени кодирования. Также программа может выводить графики зависимости времени от кол-ва символов во входной строке. Можно сделать вывод о соответствии полученного результата поставленной цели.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сайт Huffman coding - Wikipedia - кодирование Хаффмана.

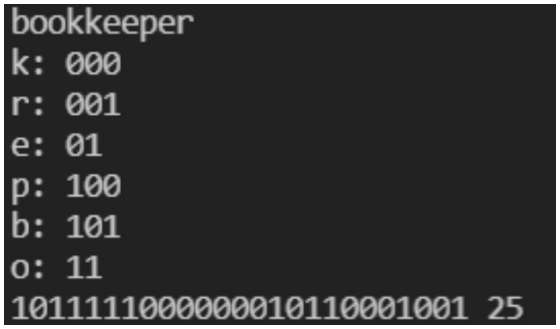
2. Сайт Shannon–Fano coding - Wikipedia - кодирование Шеннона-Фано.

3. Сайт Adaptive Huffman coding - Wikipedia - динамическое кодирование Хаффмана.

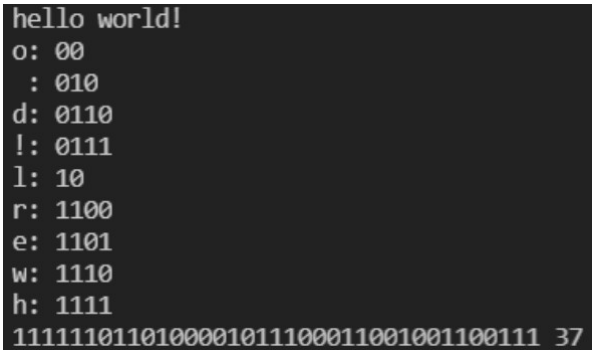
4. Сайт Data Compression - Adaptive Huffman coding - FGK (stringology.org) - FGK алгоритм.

5. Сайт Adaptive Huffman Coding - YouTube – пример FGK алгоритма.

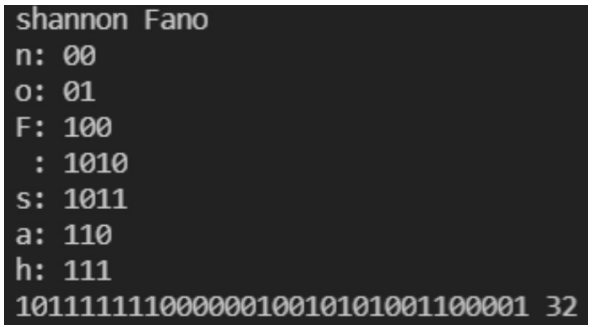
ПРИЛОЖЕНИЕ А
ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ



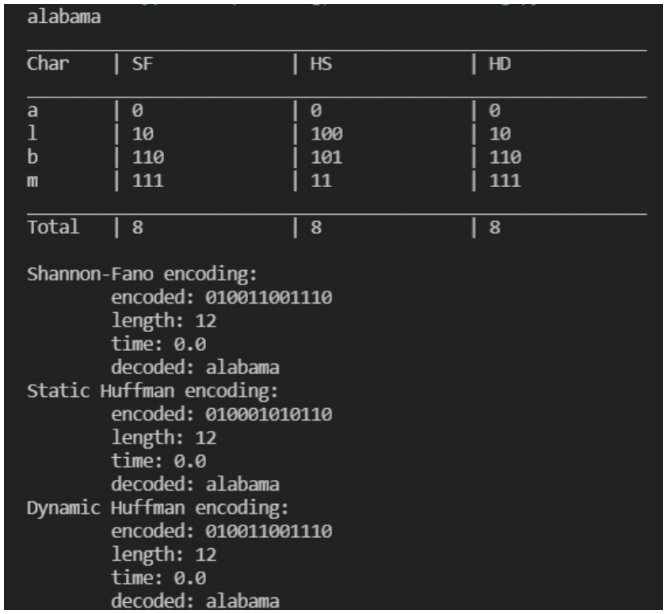
Пример 1 – Запуск файла huffmanStatic.py



Пример 2 – Запуск файла huffmanDynamic.py



Пример 3 – Запуск файла ShannonFano.py



Пример 4 – Запуск файла researchDecoding.py

ПРИЛОЖЕНИЕ В
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл Heap.py:

```
class Heap:
    def __init__(self, arr = [], isMin = True):
        self.heap = []
        self.isMin = isMin
        for el in arr:
            self.insert(el)
    def parent(self, index):
        return (index - 1)//2
    def left(self, index):
        return 2*index + 1
    def right(self, index):
        return 2*index + 2
    def sift_up(self, index):
        if index < 0 or index >= len(self.heap): return
        parent = self.parent(index)
        while index and not (self.heap[parent] < self.heap[index] and self.isMin or self.heap[parent]
> self.heap[index] and not self.isMin):
            self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]
            index, parent = parent, self.parent(index)
    def sift_down(self, index):
        if index < 0 or index >= len(self.heap): return
        mIndex = index
        while True:
            left, right = self.left(index), self.right(index)
            if right < len(self.heap) and (self.heap[right] < self.heap[mIndex] and self.isMin or
self.heap[right] > self.heap[mIndex] and not self.isMin): mIndex = right
            if left < len(self.heap) and (self.heap[left] < self.heap[mIndex] and self.isMin or
self.heap[left] > self.heap[mIndex] and not self.isMin): mIndex = left
            if mIndex == index: return
            else:
```

```
            self.heap[index], self.heap[mIndex] = self.heap[mIndex], self.heap[index]
            index = mIndex
    def pop(self):
        if not self.heap: return
        mElement = self.heap[0]
        self.heap[0] = self.heap[-1]
        del self.heap[-1]
        self.sift_down(0)
        return mElement
    def insert(self, element):
        self.heap.append(element)
        self.sift_up(len(self.heap)-1)
    def size(self):
        return len(self.heap)
```

Файл ShannonFano.py:

```
import Heap
from functools import reduce

class Letter:
    def __init__(self, char, count):
        self.char = char
        self.count = count
        self.code = ""
    def __gt__(self, other):
        if isinstance(other, Letter):
            if self.count != other.count: return self.count > other.count

def countLetters(string):
    alphabet = list(set(string))
    occurances = [0]*len(alphabet)

    for char in string:
```

```
        occurances[alphabet.index(char)] += 1

    letters = []
    for ch in range(len(alphabet)):
        letters.append(Letter(alphabet[ch], occurances[ch]))
    return letters

def heapSort(letters):
    heapSortedArr = []
    heap = Heap.Heap(letters, None)
    while heap.size():
        heapSortedArr.append(heap.pop())
    return heapSortedArr

def listReducer(sum, letter):
    return sum + letter.count

def encode(letters, dictionary, code = ''):
    if len(letters) == 1:
        dictionary[letters[0].char] = code
        return
    if len(letters) == 2:
        dictionary[letters[0].char] = code + '0'
        dictionary[letters[1].char] = code + '1'
        return

    indexLeft, indexRight = 0, len(letters)-1
    leftSum, rightSum = 0, 0
    left, right = [], []
    while indexLeft <= indexRight:
        if(leftSum + letters[indexLeft].count < rightSum or not leftSum):
            leftSum += letters[indexLeft].count
            left.append(letters[indexLeft])
            indexLeft += 1
```

```
        else:
            rightSum += letters[indexRight].count
            right.insert(0, letters[indexRight])
            indexRight -= 1
    encode(left, dictionary, code + '0')
    encode(right, dictionary, code + '1')

def ShannonFanoEncode(string, dictionary):
    letters = heapSort(countLetters(string))
    encode(letters, dictionary)

    code = ''
    for ch in string: code += dictionary[ch]
    return code

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

    codes = dict()
    output = ShannonFanoEncode(string, codes)
    for ch in codes.keys():
        print(f'{ch}: {codes[ch]}')

    print(output, len(output))
```

Файл HuffmanStatic.py:

```
import Heap

# Letter is node in bin Huffman tree
# Letter containing char is leaf
class Letter:
    def __init__(self, char, count, left = None, right = None):
        self.char = char
```

```
        self.count = count
        self.code = ""
        self.left, self.right = left, right
    def __lt__(self, other):
        if isinstance(other, Letter):
            if self.count != other.count: return self.count < other.count
            else:
                if other.char == "": return True
                elif self.char == "": return False
                else: return self.char < other.char

def countLetters(string):
    alphabet = list(set(string))
    occurances = [0]*len(alphabet)

    for char in string:
        occurances[alphabet.index(char)] += 1

    letters = []
    for ch in range(len(alphabet)):
        letters.append(Letter(alphabet[ch], occurances[ch]))
    return letters

def buildHuffmanTree(table):
    heap = Heap.Heap(table)
    while heap.size() > 1:
        left = heap.pop()
        right = heap.pop()
        heap.insert(Letter('', left.count + right.count, left, right))
    return heap.pop()

def buildCodes(top, chars, branch = ''):
    current = top
    right, left = current.right, current.left
```

```
    current.code += branch
    if right:
        right.code = current.code
        buildCodes(right, chars, '0')
    if left:
        left.code = current.code
        buildCodes(left, chars, '1')
    if current.char:
        chars[current.char] = current.code

def HuffmanEncode(string, dictionary):
    top = buildHuffmanTree(countLetters(string))
    buildCodes(top, dictionary)

    code = ''
    for ch in string: code += dictionary[ch]
    return code



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

    codes = dict()
    output = HuffmanEncode(string, codes)
    for ch in codes.keys():
        print(f'{ch}: {codes[ch]}')

    print(output, len(output))
```

Файл HuffmanDynamic.py:

```
from FGKTree import FGKTree
```

```
def HuffmanEncode(string, codes):
    tree = FGKTree()
    for char in string:
        tree.add(char)

    tree.getCodes(codes)

    code = ''
    for ch in string: code += codes[ch]
    return code

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

    codes = dict()
    output = HuffmanEncode(string, codes)

    for ch in codes.keys():
        print(f'{ch}: {codes[ch]}')
    print(output, len(output))
```

Файл FGKTree.py:

```
# Letter is node in Bin FGK tree
# Letter containing char is leaf
class Letter:
    def __init__(self, char="", left = None, right = None, parent = None):
        self.char = char
        self.count = 0
        self.code = ""
        self.left = left
        self.right = right
        self.parent = parent
    def __lt__(self, other):
```

```
        if isinstance(other, Letter):
            return self.count < other.count

# Bin FGK tree for Dynamic Huffman
class FGKTree:
    def __init__(self):
        self.arr = [Letter(),]
    def rebuild(self, element):
        current = element
        while True:
            if not current: break
            parents, childs = [self.arr[0]], []
            end = False
            while not end:
                for letter in parents:
                    if letter.left: childs.append(letter.left)
                    if letter.right: childs.append(letter.right)

                    if letter == current:
                        end = True
                        break
                    if letter.count == current.count:
                        if(current.char and not letter.char): continue
                        if(letter.parent and letter.parent.right == letter): letter.parent.right = current
                        elif letter.parent: letter.parent.left = current
                        if(current.parent.right == current): current.parent.right = letter
                        else: current.parent.left = letter
                        current.parent, letter.parent = letter.parent, current.parent
                        end = True
                        break
                parents = childs
                childs = []
            current.count += 1
            current = current.parent
```

```
def add(self, char):
    charNode = None
    for letter in self.arr:
        if letter.char == char:
            charNode = letter
            break

    if not charNode:
        nullNode = self.arr.pop()
        charNode = Letter(char)
        node = Letter("", charNode, nullNode)

        if nullNode.parent:
            nullNode.parent.left = node
            node.parent = nullNode.parent
        charNode.parent, nullNode.parent = node, node
        node.right, node.left = charNode, nullNode
        self.arr.extend([node, charNode, nullNode])
    self.rebuild(charNode)
def getCodes(self, dict, top = None, branch =''):
    current = top if top else self.arr[0]
    right, left = current.right, current.left
    current.code += branch
    if right:
        right.code = current.code
        self.getCodes(dict, right, '0')
    if left:
        left.code = current.code
        self.getCodes(dict, left, '1')
    if current.char:
        if(current.parent.left and not current.parent.left.count):
            dict[current.char] = current.parent.code
        else: dict[current.char] = current.code
```

Файл researchEncoding.py:

```
from ShannonFano import ShannonFanoEncode
from huffmanStatic import HuffmanEncode as HuffmanEncodeStatic
from huffmanDynamic import HuffmanEncode as HuffmanEncodeDynamic

from time import time_ns as time
from functools import reduce
def countReducer(sum, code):
    return sum + len(code)

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()

    start = time()
    codesSF = dict()
    outputSF = ShannonFanoEncode(string, codesSF)
    decodeSF = decode(outputSF, codesSF)
    end = time()
    timeSF = (end - start)/(10**9)

    start = time()
    codesHS = dict()
    outputHS = HuffmanEncodeStatic(string, codesHS)
```

```
    decodeHS = decode(outputHS, codesHS)
    end = time()
    timeHS = (end - start)/(10**9)

    start = time()
    codesHD = dict()
    outputHD = HuffmanEncodeDynamic(string, codesHD)
    decodeHD = decode(outputHD, codesHD)
    end = time()
    timeHD = (end - start)/(10**9)

    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",
        reduce(countReducer, codesSF.items(), 0),
        reduce(countReducer, codesHS.items(), 0),
        reduce(countReducer, codesHD.items(), 0))
    )
    print('')

    print("Shannon-Fano encoding:\n\tencoded:", outputSF, "\n\tlength:", len(outputSF), "\n\
ttime:", timeSF, "\n\tdecoded:", decodeSF)
    print("Static Huffman encoding:\n\tencoded:", outputHS, "\n\tlength:", len(outputHS), "\n\
ttime:", timeHS, "\n\tdecoded:", decodeHS)
```

```
    print("Dynamic Huffman encoding:\n\tencoded:", outputHD, "\n\tlength:", len(outputHD), "\n\
ttime:", timeHD, "\n\tdecoded:", decodeHD)
```

Файл researchTime.py:

```
from numpy import log2
from ShannonFano import ShannonFanoEncode
from huffmanStatic import HuffmanEncode as HuffmanEncodeStatic
from huffmanDynamic import HuffmanEncode as HuffmanEncodeDynamic
from time import time_ns as time
import matplotlib.pyplot as plt

string = "☺ ☻♥♦♣♠•◘○◙♂♀♪♫☼► ◄↕‼¶§▬↨↑↓→←∟↔▲▼
!#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz"
length = 500

def measureTime(func, string, codes):
    start = time()
    func(string, codes)
    end = time()
    return (end - start)/(10**9)

if __name__ == "__main__":
    print("Time measurements")
    print("Initial string:", string)
    print("Number of repetitions:", length)
    timings = [[],[],[]]
    codes = dict()
    for n in range(1, len(string)):
        print("Step:", n+1, "\tSymbols:", (n+1)*length)
        strToEncode = string[:n]*length
        timings[0].append(measureTime(ShannonFanoEncode, strToEncode, codes))
        timings[1].append(measureTime(HuffmanEncodeStatic, strToEncode, codes))
```

```
    timings[2].append(measureTime(HuffmanEncodeDynamic, strToEncode, codes))

xCoord = range(1, len(string))
yCoord = timings[0]
plt.scatter(xCoord, yCoord)
plt.plot(xCoord, yCoord)
a = (yCoord[-1]+yCoord[-2])/2/xCoord[-1]/log2(xCoord[-1])
yCoord = [a*x*log2(x) for x in xCoord]
plt.plot(xCoord, yCoord)
plt.savefig("SF.png")
plt.clf()

yCoord = timings[1]
plt.scatter(xCoord, yCoord)
plt.plot(xCoord, yCoord)
a = (yCoord[-1]+yCoord[-2])/2/xCoord[-1]/log2(xCoord[-1])
yCoord = [a*x*log2(x) for x in xCoord]
plt.plot(xCoord, yCoord)
plt.savefig("HS.png")
plt.clf()

yCoord = timings[2]
plt.scatter(xCoord, yCoord)
plt.plot(xCoord, yCoord)
a = yCoord[-1]/xCoord[-1]**2
yCoord = [a*x**2 for x in xCoord]
plt.plot(xCoord, yCoord)
plt.savefig("HD.png")
plt.clf()

print("Finished")
```