Санкт-Петербургский Национальный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №1

Выполнили:

Оншин Д.Н.

Садовая А.Р.

Петрова Н.Г.

Проверил:

Мусаев А.А.

Санкт-Петербург,

2023

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Данная работа представляет собой отчет о выполненных заданиях:

1. Используя каждый изученный алгоритм поиска подстрок (наивный, Рабина-Карпа, Бойера-Мура, Кнута-Морриса-Пратта), посчитать количество наиболее часто встречающихся двузначных чисел в строке, состоящей из 500 чисел фибоначчи, написанных слитно. Сравние изученных алгоритмов поиска подстрок. Вывод о их достоинствах и недостатках.
2. Выбрав любой алгоритм поиска, определить количество плагиата (в процентах от общего количества символов в реферате) в тексте реферата, взяв за основу соответствующие статьи из Википедии. Обоснование выбранного алгоритма поиска.

1 Задание 1

```python
def nav_alg(s):
    start = time.time()
    mas = [0] * 100 #создаем масссив для подсчёта чисел
    for j in range(10, 100): #Поиск двузначных чисел
        st = str(j)
        for i in range(len(s) - 1):
            if st[0] == s[i] and st[1] == s[i + 1]:
                mas[j] += 1
    for j in range(len(mas)):
        if mas[j] == max(mas):
            print('Самое часто встречающееся число:', j)
            print('Количество совпадений:', max(mas))
            break
    end = time.time() - start
    print('Время выполнения алгоритма: ', end)
    print('')
```

Рисунок 1.1 — Код функции для наивного алгоритма

```
Самое часто встречающееся число: 71
Количество совпадений: 296
Время выполнения алгоритма:  0.23027729988098145
```

Рисунок 1.2 — Результат работы наивного алгоритма

```python
def Rabin_Karp(s):
    start = time.time()
    counter = [0] * 100
    alf = 10
    m = 1
    hashmas = [0] * 100
    for i in range(10, 100):
        shablon = i
        hash = (shablon // 10) * alf ** m + (shablon % 10) * alf ** (m - 1)
        hashmas[i] = hash

    for i in range(len(s) - 1):
        ch = int(s[i] + s[i + 1])
        hash = (ch // 10) * alf ** m + (ch % 10) * alf ** (m - 1)
        for j in range(10, len(hashmas)):
            if hashmas[j] == hash:
                if j == ch:
                    counter[ch] += 1

    for j in range(len(counter)):
        if counter[j] == max(counter):
            print('Самое частое число: ', j)
            print('Количество совпадений: ', max(counter))
            break
    end = time.time() - start
    print('Время выполнения алгоритма: ', end)
    print('')
```

Рисунок 1.3 — Код функции для алгоритма Рабина-Карпа

```
Самое частое число:  71
Количество совпадений:  296
Время выполнения алгоритма:  0.1590414047241211
```

Рисунок 1.4 — Результат работы алгоритма Рабина-Карпа

```python
def Boyer_Moore(s):
    start = time.time()
    num_list = [0] * 100
    for i in range(10, 100):
        num = str(i)
        j = 1
        while j < len(s):
            if s[j] == num[1]:
                if s[j - 1] == num[0]:
                    num_list[i] += 1
                    j += 2
                else:
                    j += 2
            else:
                if s[j] == num[0]:
                    j += 1
                else:
                    j += 2
    for j in range(10, len(num_list)):
        if num_list[j] == max(num_list):
            print('Самое частое число: ', j)
            print('Количество совпадений: ', max(num_list))
            break
    end = time.time() - start
    print('Время выполнения алгоритма: ', end)
    print('')
```

Рисунок 1.5 — Код функции для алгоритма Бойера-Мура

```
Самое частое число:  71
Количество совпадений:  296
Время выполнения алгоритма:  0.2945120334625244
```

Рисунок 1.6 — Результат работы алгоритма Бойера-Мура

```python
def Knuth_Morris_Pratt(s):
    start = time.time()
    num_list = [0] * 100
    for i in range(10, 100):
        num = str(i)
        if num[0] == num[1]:
            pref_func = [0, 0]
        else:
            pref_func = [0, 1]
        j = 0
        while j < len(s) - 1:
            if num[0] == s[j]:
                if num[1] == s[j + 1]:
                    num_list[i] += 1
                    j += 2
                else:
                    j += 1 - pref_func[1] + 1
            else:
                j += 0 - pref_func[0] + 1
    for j in range(10, len(num_list)):
        if num_list[j] == max(num_list):
            print('Самое частое число: ', j)
            print('Количество совпадений: ', max(num_list))
            break
    end = time.time() - start
    print('Время выполнения алгоритма: ', end)
    print('')
```

Рисунок 1.7 — Код функции для алгоритма Кнута-Морриса-Пратта

```
Самое частое число:  71
Количество совпадений:  296
Время выполнения алгоритма:  0.6225271224975586
```

Рисунок 1.8 — Результат работы алгоритма Кнута-Морриса-Пратта

Рассмотрим процесс работы программы, в которой пользователь выбирает разные методы поиска.

```
 1 - наивный алгоритм
 2 - алгоритм Рабина-Карпа
 3 - алгоритм Бойера-Мура
 4 - алгоритм Кнута-Морриса-Пратта
 0 - выход
Введите номер алгоритма: 4

Самое частое число:  71
Количество совпадений:  296
Время выполнения алгоритма:  0.6501479148864746
```

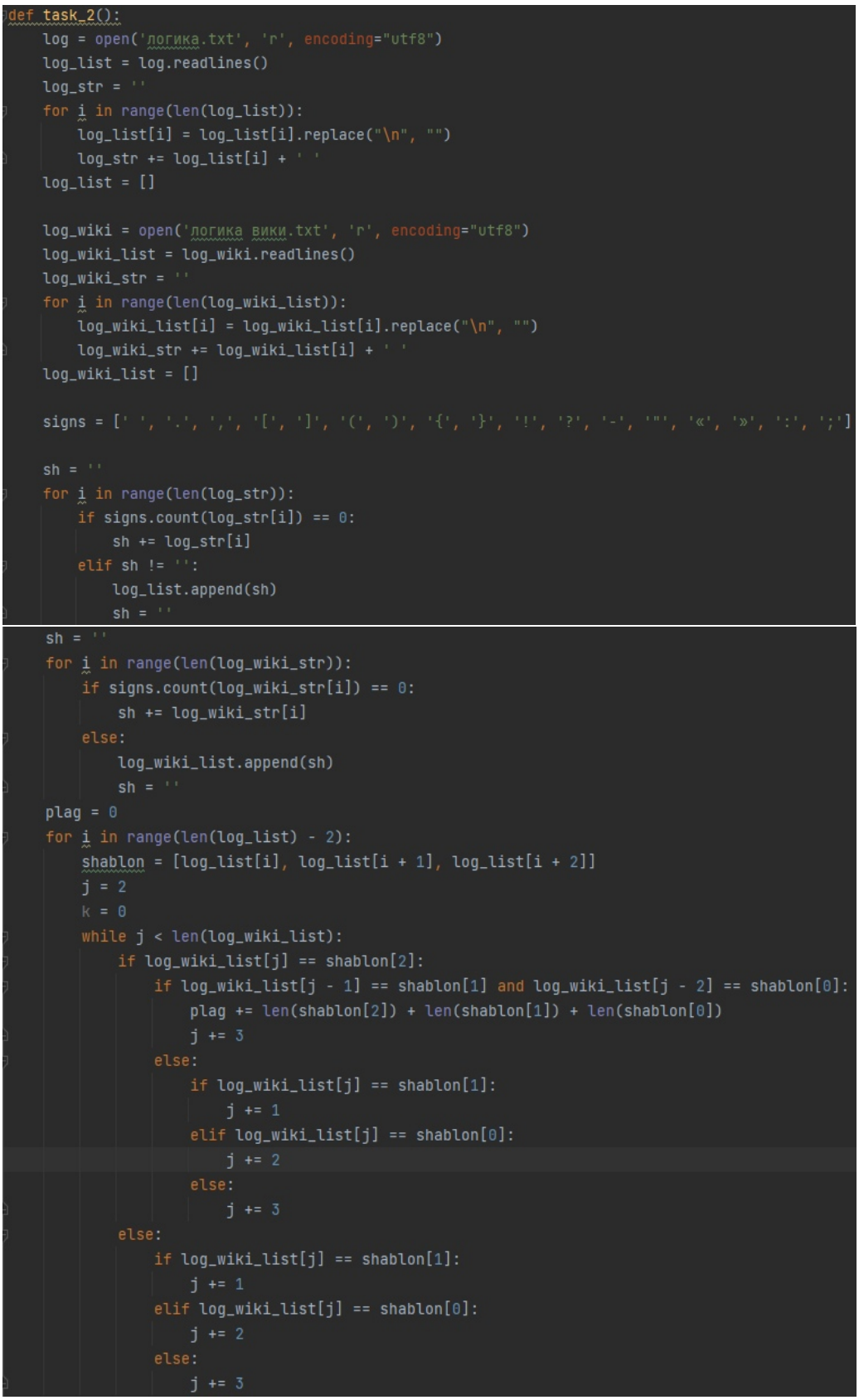
Рисунок 1.9 — Пример работы алгоритма Кнута-Морриса-Пратта

Вывод: Наивный алгоритм является малозатратным и не нуждается в предварительной обработке и в дополнительном пространстве. Большинство сравнений алгоритма прямого поиска являются лишними. Поэтому в худшем случае алгоритм будет малоэффективен, так как его сложность будет пропорциональна O((n-m+1)*m), где n и m – длины строки и подстроки соответственно. Алгоритм Д. Кнута, Д. Морриса и В. Пратта: для данного алгоритма требуется порядка O(m+n) сравнений символов (где n и m – длины строки и подстроки соответственно), что значительно лучше, чем при прямом поиске, при этом, в данном случае применение этого алгоритма неэффективно. Алгоритм Бойера и Мура на хороших данных очень быстр, а вероятность появления плохих данных крайне мала. Таким образом, данный алгоритм является наиболее эффективным в обычных ситуациях, а его быстродействие повышается при увеличении подстроки или алфавита. Алгоритм Рабина-Карпа в наихудшем случае имеет сложность O(m+n). Несмотря на наличие более производительных алгоритмов поиска одиночных строк, алгоритм Рабина-Карпа может оказаться довольно эффективным при поиске множественных шаблонов, которые и нужно было найти в задании, а следовательно, по времени, данный алгоритм показал наилучший результат.

2 Задание 2

Для выполнения задания были взяты 2 файла формата txt. Один с рефератом, второй с исходной статьей из википедии. И с помощью поиска алгоритмом Бойера Мура, учитывая, что плагиатом считается 3 подряд сопадающих слова был определен процент плагиата.

Обоснование выбора алгоритма:
Преимущество этого алгоритма в том, что ценой некоторого количества предварительных вычислений над шаблоном (но не над строкой, в которой ведётся поиск) шаблон сравнивается с исходным текстом не во всех вариантах — часть проверок пропускаются как заведомо не дающие результата. Кроме того, данный алгоритм более понятен для восприятия и соответственно есть возможность более точно адаптировать его под конкретную задачу.



```
def task_2():
    log = open('логика.txt', 'r', encoding="utf8")
    log_list = log.readlines()
    log_str = ''
    for i in range(len(log_list)):
        log_list[i] = log_list[i].replace("\n", "")
        log_str += log_list[i] + ' '
    log_list = []

    log_wiki = open('логика вики.txt', 'r', encoding="utf8")
    log_wiki_list = log_wiki.readlines()
    log_wiki_str = ''
    for i in range(len(log_wiki_list)):
        log_wiki_list[i] = log_wiki_list[i].replace("\n", "")
        log_wiki_str += log_wiki_list[i] + ' '
    log_wiki_list = []

    signs = [' ', '.', ',', '[', ']', '(', ')', '{', '}', '!', '?', '-', '"', '«', '»', ':', ';']

    sh = ''
    for i in range(len(log_str)):
        if signs.count(log_str[i]) == 0:
            sh += log_str[i]
        elif sh != '':
            log_list.append(sh)
            sh = ''
    sh = ''
    for i in range(len(log_wiki_str)):
        if signs.count(log_wiki_str[i]) == 0:
            sh += log_wiki_str[i]
        else:
            log_wiki_list.append(sh)
            sh = ''
    plag = 0
    for i in range(len(log_list) - 2):
        shablon = [log_list[i], log_list[i + 1], log_list[i + 2]]
        j = 2
        k = 0
        while j < len(log_wiki_list):
            if log_wiki_list[j] == shablon[2]:
                if log_wiki_list[j - 1] == shablon[1] and log_wiki_list[j - 2] == shablon[0]:
                    plag += len(shablon[2]) + len(shablon[1]) + len(shablon[0])
                    j += 3
                else:
                    if log_wiki_list[j] == shablon[1]:
                        j += 1
                    elif log_wiki_list[j] == shablon[0]:
                        j += 2
                    else:
                        j += 3
            else:
                if log_wiki_list[j] == shablon[1]:
                    j += 1
                elif log_wiki_list[j] == shablon[0]:
                    j += 2
                else:
                    j += 3
```

Рисунок 2.1 — Код функции для определения процента плагиата

```
    plag_sum = 0
    for i in range(len(log_wiki_list)):
        plag_sum += len(log_wiki_list[i])
    plag /= plag_sum
    print('Процент плагиата равен ', plag * 100)
    log.close()
    log_wiki.close()
```

Рисунок 2.2 — Код функции для определения процента плагиата(продолжение)

```
Введите номер задания: 2
Процент плагиата равен  41.107847598560376
```

Рисунок 2.3 — Вывод функции на определение процента плагиата для статьи "Логика"

ЗАКЛЮЧЕНИЕ

Таким образом, для каждой задачи были написаны программы на языке программирования Python. Были изучены 4 алгоритма поиска и рассмотрены их достоинства и недостатки. Для каждого алгоритма приведены программы.

Все программы можно найти на репозитории в GitHub [1].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GitHub [Электронный ресурс]: `https://github.com/NatalyaPetrova/Algoritms` (дата обращения 20.02.2023).