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

Факультет инфокоммуникационных технологий

Лабораторная работа №7 "Алгоритмы поиска подстрок"

Вариант 7

Выполнили:
Голованов Д.И.,
Шарыпов Е.А.,
Стафеев И.А.
Проверил
Мусаев А.А.

Санкт-Петербург,
2024

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

В данной лабораторной работе необходимо решить следующие задачи:

1. Заполнить массив 500 числами Фибоначчи написанными слитно. Используя алгоритмы поиска подстрок (наивный, Рабина-Карпа, Бойера-Мура, Кнута-Морриса-Пратта), посчитайть количество наиболее часто встречающихся двузначных чисел в образовавшейся строке. Сравнить изученные алгоритмы поиска подстрок.
2. Дан реферат на тему «Логика». Определить количество плагиата (в % от общего количества символов в реферате) в тексте реферата, взяв за основу статью из Википедии «Логика». За плагиат считать любые 3 совпавших слова, идущих подряд. Обосновать выбранный алгоритм поиска.

1 Задача 1

Ниже приведены скриншоты кода для теста алгоритмов 1.1, алгоритма наивного поиска 1.2, алгоритма Рабина-Карпа 1.3, алгоритма Бойера-Мура 1.4 и алгоритма Кнута-Морриса-Пратта 1.5.

```python
from naive import naive
from kmp import kmp
from rk import rk
from bm import bm
import time
nums = [1, 1]

for i in range(500-2):
    nums.append(nums[-1] + nums[-2])

haystack = ''.join(str(num) for num in nums)

def test(f):
    ans = (-1, -1)
    for i in range(10, 100):
        n = len(f(haystack, str(i)))
        if n > ans[1]:
            ans = (i, n)
    return ans

def measure(f):
    correct_ans = (71, 297)
    start = time.time()
    ans = test(f)
    if ans != correct_ans:
        raise Exception
    end = time.time()
    return end - start

algs = {
    "naive": naive,
    "rk": rk,
    "kmp": kmp,
    "bm": bm
}

for name, fn in algs.items():
    print(f'{name}: {int(measure(fn) * 1000)}ms')
```

Рисунок 1.1 — Код для теста алгоритмов

```python
def naive(haystack, needle):
    result = []
    for i in range(len(haystack) - len(needle) + 1):
        if haystack[i:i+len(needle)] == needle:
            result.append(i)
    return result
```

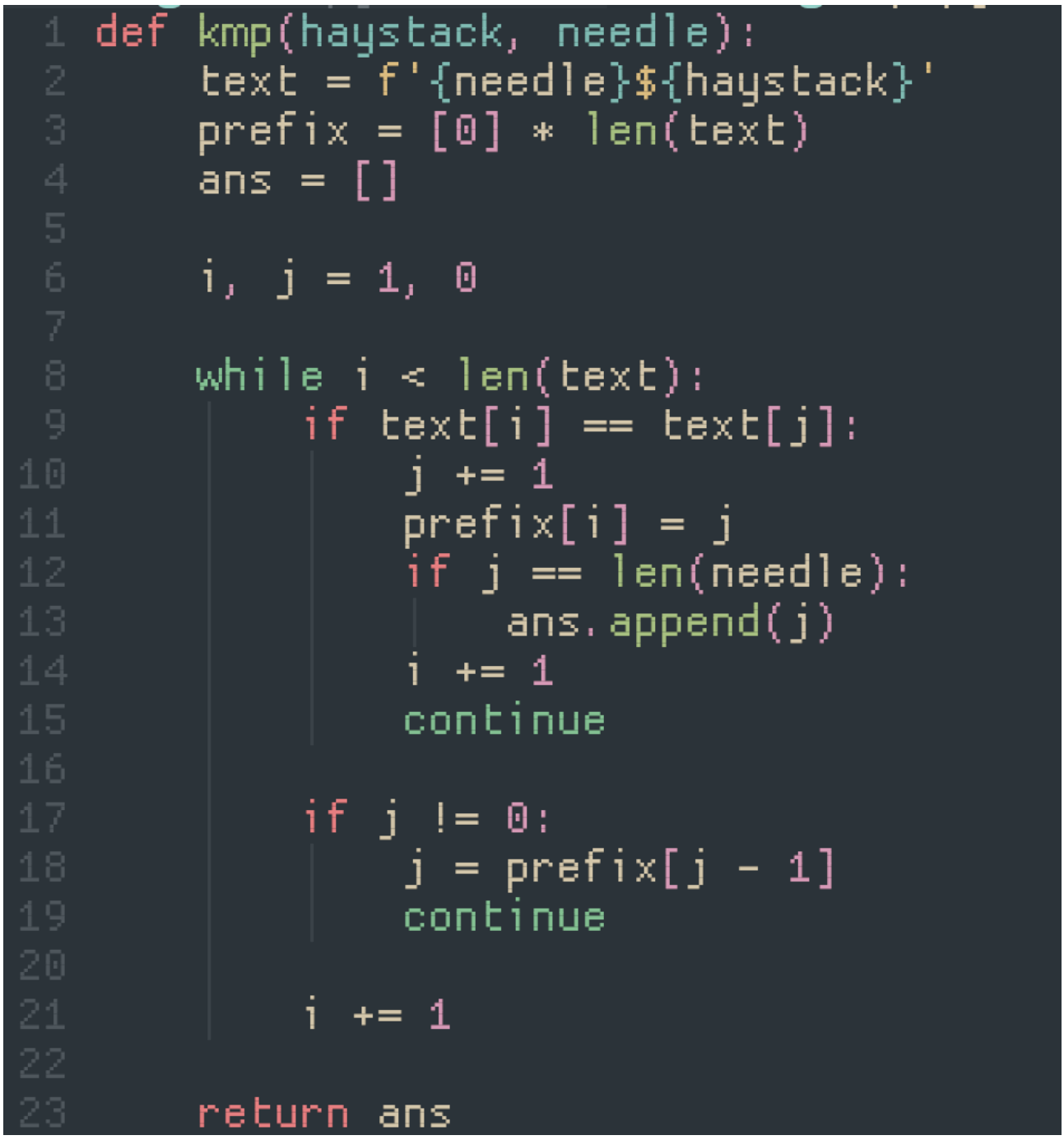
Рисунок 1.2 — Наивный поиск подстроки

```python
q = 101
d = 256

def rk(haystack, needle):
    n = len(needle)

    needle_h = 0
    for sym in needle:
        needle_h = (needle_h * d + ord(sym)) % q

    window_h = 0
    for sym in haystack[:n]:
        window_h = (window_h * d + ord(sym)) % q

    ans = []
    if window_h == needle_h:
        ans.append(0)

    k = d ** (n - 1)
    for l in range(len(haystack) - n):
        old_sym = haystack[l]
        new_sym = haystack[l + n]

        window_h = d * (window_h - k * ord(old_sym)) + ord(new_sym)
        window_h %= q

        if window_h == needle_h and haystack[l + 1:l + n + 1] == needle:
            ans.append(l + 1)

    return ans
```

Рисунок 1.3 — Алгоритм Рабина-Карпа

```python
N = 256

def bad_char_heuristic(string, size):
    bad_chars = [-1] * N

    for i, sym in enumerate(string[:size]):
        bad_chars[ord(sym)] = i

    return bad_chars


def bm(haystack, needle):
    m = len(needle)
    n = len(haystack)
    ans = []

    bad_chars = bad_char_heuristic(needle, m)

    s = 0
    while(s <= n - m):
        j = m - 1

        while j >= 0 and needle[j] == haystack[s+j]:
            j -= 1

        if j < 0:
            ans.append(s)
            s += (m-bad_chars[ord(haystack[s+m])] if s+m < n else 1)
        else:
            s += max(1, j-bad_chars[ord(haystack[s+j])])

    return ans
```

Рисунок 1.4 — Алгоритм Бойера-Мура



```python
def kmp(haystack, needle):
    text = f'{needle}${haystack}'
    prefix = [0] * len(text)
    ans = []

    i, j = 1, 0

    while i < len(text):
        if text[i] == text[j]:
            j += 1
            prefix[i] = j
            if j == len(needle):
                ans.append(j)
            i += 1
            continue

        if j != 0:
            j = prefix[j - 1]
            continue

        i += 1

    return ans
```

Рисунок 1.5 — Алгоритм Кнута-Морриса-Пратта

На рисунке 1.6 приведен скриншот результатов измерения скорости выполнения алгоритмов для исходных данных.

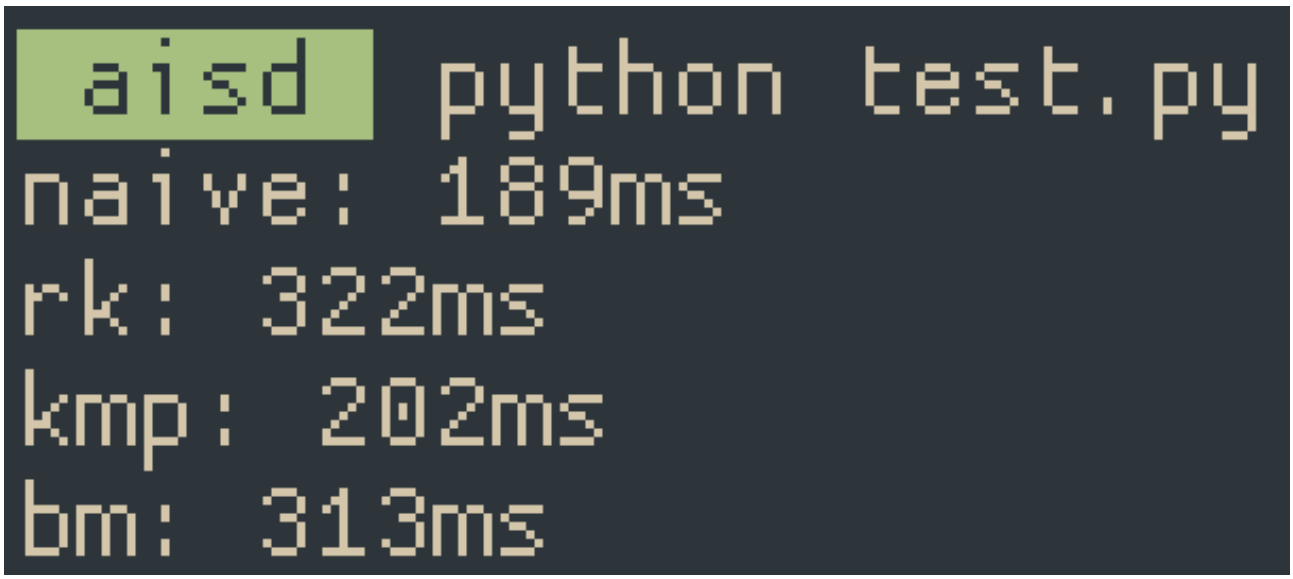


Рисунок 1.6 — Результаты измерения

Наивный алгоритм оказался самым быстрым в данном случае из-за маленькой длины шаблона и дешевого сравнения. Алгоритм Рабина-Карпа оказался довольно медленным, потому что пересчитывание хэша окна в случае шаблона из двух символов занимает столько же времени, сколько подсчёт нового хэша. Если бы мы сохраняли хэши для всех окон при поиске различных чисел, то этот алгоритм был бы значительно быстрее. Алгоритм КМП работает со скоростью наивного из за небольшой длины шаблона. Алгоритм Бойера-Мура довольно медленный из за небольшого алфавита, потому что эвристика плохих символов в этом случае работает плохо.

2 Задача 2

В данной задаче мы применяем алгоритм, похожий на алгоритм Рабина-Карпа. Сначала хэшируются все слова исходного текста, эти слова объеденяются в тройки, для каждой тройки считается хэш на основе хэшей входящих в неё слов. Потом алгоритм итерирует по тройкам слов из статьи, и каждая тройка сопоставляется с увиденными в исходном тексте на основе хэшей сначала тройки, потом слов, потом содержания слов.

На рисунках 2.1, 2.2 и 2.3 приведен код программы, состоящий из хэш-функций для строк и для слов, функция-итераторов по словам и строкам слов и функция для вычисления процента плагиата.

```
import time
d = 256
k = d ** 2
q = 101
def hash_w(word):
    h = 0
    for s in word:
        n = ord(s)
        if n > 1000: # if russian
            n -= 1040

        h = (h * d + n) % q
    return h

def hash_t(t):
    return (t[0][0] * k + t[1][0] * d + t[2][0]) % q

def roll_t(h, old, new):
    return ((h - old[0] * k) * d + new[0]) % q

def len_t(triplet):
    return len(triplet[0][1]) + len(triplet[1][1]) + len(triplet[2][1])

def eq_t(t1, t2):
    eq_h = t1[0][0] == t2[0][0] and t1[1][0] == t2[1][0] and t1[2][0] == t2[2][0]
    if not eq_h:
        return False
    return t1[0][1] == t2[0][1] and t1[1][1] == t2[1][1] and t1[2][1] == t2[2][1]
```

Рисунок 2.1 — Хэш функции для строк и для слов

```python
def words(text):
    word_start = None
    for i, s in enumerate(text):
        if word_start is None and s.isalpha():
            word_start = i
            continue

        if word_start is not None and not s.isalpha():
            content = text[word_start:i]
            word_h = hash_w(content)
            word_start = None
            yield word_h, content

    if word_start != None:
        yield hash_w(text[word_start:]), text[word_start:]

def triplets(text):
    iter_words = words(text)
    w1, w2, w3 = next(iter_words), next(iter_words), next(iter_words)

    t1 = (w1, w2, w3)
    triplet_h = hash_t(t1)
    yield triplet_h, t1

    last_words = [w2, w3]

    for word in iter_words:
        # triplet_h = roll_t(triplet_h, last_words[0], word)
        triplet_h = hash_t((*last_words, word))
        yield triplet_h, (*last_words, word)

        last_words[0] = last_words[1]
        last_words[1] = word
```

Рисунок 2.2 — Функции-итераторы по словам и по тройкам

```
# word: (hash, content)
def hash_triplets(text):
    table = [[] for _ in range(q)] # triplet_hash: (word, word, word)
    for h, triplet in triplets(text):
        # if h not in table:
        #     table[h] = []
        table[h].append(triplet)
    return table

def plagiat(source, text):
    source_triplets = hash_triplets(source)
    ans = 0
    last = -5
    for i, (h, t1) in enumerate(triplets(text)):
        # if h not in source_triplets:
        #     continue
        for t2 in source_triplets[h]:
            if eq_t(t1, t2):
                if i - last == 1:
                    ans += len(t1[2][1])
                    continue
                if i - last == 2:
                    ans += len(t1[2][1]) + len(t1[1][1])
                    continue
                ans += len_t(t1)
                last = i
    return ans / len(text)

text = open('text.txt').read()
wiki = open('wiki.txt').read()

s = time.time()
p = plagiat(wiki, text)
e = time.time()

print(f'{int(p * 100)}%: {int((e - s) * 1000)}ms')
```

Рисунок 2.3 — Подсчет плагиата между двумя текстами

На рисунке 2.4 приведен результат выполнения программы.

```
aisd python plagiat.py
31%: 27ms
```

Рисунок 2.4 — Результат выполнения алгоритма

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Википедия. Статья "Логика"для антиплагиата [Электронный ресурс]: [сайт]. - URL: https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0 (дата обращения: 22.03.2024).

2. Википедия. Алгоритм Бойера-Мура [Электронный ресурс]: [сайт]. - URL: https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm (дата обращения: 22.03.2024).

3. Википедия. Алгоритм Рабина-Карпа [Электронный ресурс]: [сайт]. - URL: https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm (дата обращения: 22.03.2024).

4. Википедия. Алгоритм Кнута-Морриса-Пратта [Электронный ресурс]: [сайт]. - URL: https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm (дата обращения: 22.03.2024).