Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Факультет Инфокоммуникационных технологий (ИКТ)

Образовательная программа Мобильные и сетевые технологии

О Т Ч Е Т

по Лабораторной работе 4

Дисциплина: Алгоритмы и структуры данных.

Тема: Алгоритмы поиска подстрок.

Специальность: 09.03.03 Прикладная информатика.

Проверил:
Мусаев А. А. _____
Дата: «__»_____2023 г.
Оценка: _____

Выполнил:
Балцат К. И.,
студент группы К33401

Санкт-Петербург
2023

ВЫПОЛНЕНИЕ

1 Задача 1

Заполните массив 500 числами (четный вариант – простые числа, нечетный вариант – числа Фибоначчи) написанными слитно. Используя каждый изученный алгоритм поиска подстрок (наивный, Рабина-Карпа, Бойера-Мура, Кнута-Морриса-Пратта), посчитайте количество наиболее часто встречающихся двузначных чисел в образовавшейся строке. Сравните изученные алгоритмы поиска подстрок. Сделайте вывод о их достоинствах и недостатках.

Решение:

Создаем необходимый массив из слитно написанных чисел Фибоначчи. Затем определяем функцию для подсчета количества наиболее часто встречающихся двузначных чисел в строке. Функция использует регулярное выражение для поиска всех двузначных чисел в строке и словарь для подсчета количества вхождений каждого числа. Функция возвращает список кортежей, где каждый кортеж содержит двузначное число и количество его вхождений в строку. Кортежи отсортированы по убыванию количества вхождений.

```python
import numpy as np
import regex as re

# Создаем массив из первых 500 чисел Фибоначчи:
fibonacci = [0, 1]
for i in range(2, 500):
    fibonacci.append(fibonacci[i-1] + fibonacci[i-2])

# Объединяем числа в строку и заменяем пробелы на пустые символы:
fibonacci_str = ''.join(map(str, fibonacci))
fibonacci_str = fibonacci_str.replace(' ', '')

def count_numbers(string):
    numbers = re.findall(r'\d{2}', string, overlapped=True)
    print(len(string), len(numbers))
    counts = {}
    for num in numbers:
        if num in counts:
            counts[num] += 1
        else:
            counts[num] = 1
    sorted_counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)
    return sorted_counts
re_counts = count_numbers(fibonacci_str)
print(re_counts[:10])
```

Наивный алгоритм поиска подстрок

Определяем функцию для наивного алгоритма поиска подстрок:

```
# Наивный алгоритм поиска подстрок
def naive_search(text, pattern_len=2):
    n = len(text)
    count_dict = {}
    for i in range(n - pattern_len + 1):
        pattern = text[i:i+pattern_len]
        if pattern.isdigit() and len(pattern) == pattern_len:
            if pattern in count_dict:
                count_dict[pattern] += 1
            else:
                count_dict[pattern] = 1
    sorted_counts = sorted(count_dict.items(), key=lambda x: x[1], reverse=T
    return sorted_counts

naive_counts = naive_search(fibonacci_str, pattern_len=2)
print(naive_counts[:10])
```

Алгоритм Рабина-Карпа

Определяем функцию для алгоритма Рабина-Карпа и внутри полиномиальную кольцевую Хеш-функцию:

```
# Алгоритм Рабина-Карпа
def rabin_karp_search(string, pattern_len=2):
    # Задаем полиномиальную кольцевую!! hash-функцию.
    def H(substring: str, x=10, m=1):
        '''
        same as int
        '''
        h = 0
        for i in range(len(substring)):
            h += int(substring[i])*x**(m-i)
        return h
    counts = {f'{i:02}': 0 for i in range(1, 100)}
    hash_table = []
    for i in range(len(string) - pattern_len + 1):
        hash_table.append(H(string[i:i+pattern_len]))
    for pattern in counts.keys():
        pattern_hash = H(pattern)
        for hash in hash_table:
            if pattern_hash == hash:
                # Посимвольная проверка не требуется. В нашем случае hash-фу
                counts[pattern] += 1
    sorted_counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)
    return sorted_counts
```

Алгоритм Бойера-Мура

Определяем функцию для алгоритма Бойера-Мура:

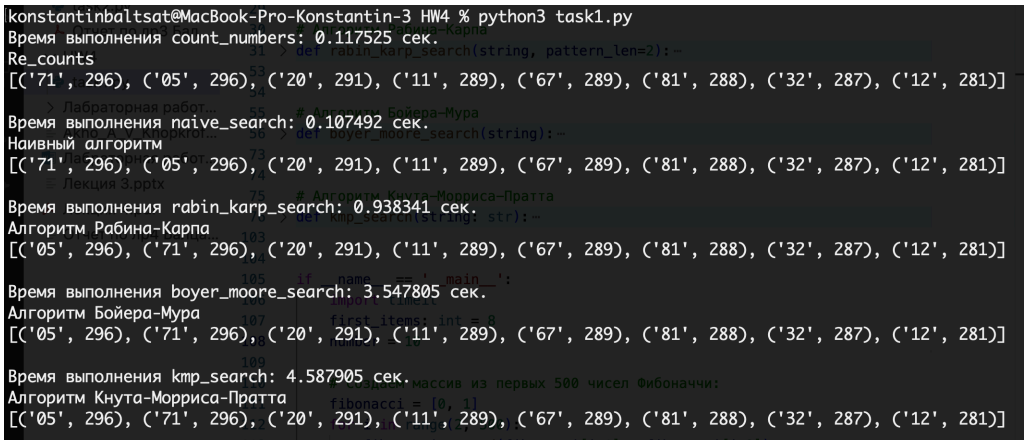
```python
# Алгоритм Бойера-Мура
def boyer_moore_search(string):
    counts = {f'{i:02}': 0 for i in range(1, 100)}
    for pattern in counts.keys():
        pos = 1  # len(pattern)==2
        while pos < len(string):
            if (string[pos] == pattern[1]) is False:
                if string[pos] in pattern:
                    pos += 1  # len(pattern)==2
                else:
                    pos += 2
            elif (string[pos-1] == pattern[0]) is False:
                pos+=1
            else:
                counts[pattern] += 1
                pos += 1
    sorted_counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)
    return sorted_counts
```

Алгоритм Кнута-Морриса-Пратта

Определяем функцию для алгоритма Кнута-Морриса-Пратта и внутри префикс-функцию:

```python
# Алгоритм Кнута-Морриса-Пратта
def kmp_search(string: str):
    def prefix(s) -> list:
        n = len(s)
        pi = [0] * n  # создаем список для хранения значений префикс-функции
        for i in range(1, n):
            j = pi[i-1]
            while j > 0 and s[j] != s[i]:
                j = pi[j-1]
            if s[j] == s[i]:
                j += 1
            pi[i] = j
        return pi

    counts = {f'{i:02}': 0 for i in range(1, 100)}
    for pattern in counts.keys():
        P = prefix(pattern)
        k = 0
        for i in range(len(string)):
            while k>0 and pattern[k] != string[i]:
                k = P[k-1]
            if pattern[k] == string[i]:
                k += 1
            if k == len(pattern):
                counts[pattern]+=1
                k = P[k-1]
    sorted_counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)
    return sorted_counts
```

Результат работы программы



Краткое сравнение алгоритмов:

Наивный алгоритм:

Достоинства:

Простота реализации.

Работает для любых строк.

Недостатки:

Очень медленный для больших текстов и/или шаблонов.

В худшем случае требует O(nm) операций.

Алгоритм Рабина-Карпа:

Достоинства:

Очень быстрый в среднем случае.

Возможность поиска нескольких вхождений.

Недостатки:

Некоторые коллизии могут приводить к ложным срабатываниям.

Требуется хранение хеш-значений всех подстрок текста, что занимает много памяти.

Может работать медленно в худшем случае.

Алгоритм Бойера-Мура:

Достоинства:

Очень быстрый на практике.

Не требует дополнительной памяти за исключением массива прыжков.

Хорошо справляется с худшим случаем.

Недостатки:

Сложен для понимания и реализации.

Не работает для некоторых типов шаблонов.

Алгоритм Кнута-Морриса-Пратта:

Достоинства:

Быстрый и эффективный в среднем и худшем случаях.

Не требует дополнительной памяти.

Недостатки:

Сложен для понимания и реализации.

Не так быстр как алгоритм Бойера-Мура на практике.

Итак, выбор алгоритма зависит от конкретной задачи. Если важна производительность на больших данных, то следует использовать алгоритм Бойера-Мура или Кнута-Морриса-Пратта. Если необходимо искать несколько вхождений, то Рабина-Карп может оказаться хорошим выбором. Если же важна простота реализации и работа с короткими строками, то можно использовать наивный алгоритм.

2 Задача 2

Дан набор рефератов. Выберите любой алгоритм поиска и определите количество плагиата (в % от общего количества символов в реферате) в тексте реферата, взяв за основу соответствующие статьи из Википедии (название файла = название статьи). За плагиат считать любые 3 совпавших слова, идущих подряд. Обоснуйте выбранный алгоритм поиска.

Решение:

Определим функцию обработки текста.

```
import os
import re
import docx2txt
import string
import requests
from bs4 import BeautifulSoup

# Set the directory where the files are located
dir_path = "./texts/"

def clean_text(text):
    # Remove unwanted characters from the text
    text = re.sub(r'\[[0-9]+\]', '', text)
    text = re.sub(r'\n+', '\n', text)
    text = re.sub(r'\t+', ' ', text)

    # Remove punctuation except periods in lists
    for match in re.findall(r'(?m)^\d+[.)]\s+', text):
        text = text.replace(match, '')
    text = re.sub(r'(?m)^[^\S\n]*[-•—][^\S\n]+', '', text)
    text = text.replace('.', '')
    text = text.translate(str.maketrans(
        '', '', string.punctuation.replace('.', '')+'↑'))
    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))
    # Convert to lowercase
    text = text.lower()
    # Remove extra whitespace
    text = ' '.join(text.split())
    return text

```

Определим функцию чтения статьи с Википедии.

```
# Define the function to get the Wikipedia article
def get_wiki_article(article_name):
    # Replace spaces with underscores in article name
    article_name = article_name.replace(" ", "_")
    # Build the URL for the article
    url = f"https://ru.wikipedia.org/wiki/{article_name}"
    # Make a request to the URL and get the HTML content
    response = requests.get(url)
    html_content = response.content
    # Parse the HTML content with BeautifulSoup
    soup = BeautifulSoup(html_content, 'html.parser')
    # Find the content div and extract the text
    content_div = soup.find('div', {'class': 'mw-parser-output'})
    # Remove unwanted elements from the content
    [s.extract() for s in content_div('sup')]
    [s.extract() for s in content_div('img')]
    [s.extract() for s in content_div('audio')]
    [s.extract() for s in content_div('video')]
    [s.extract() for s in content_div('map')]
    [s.extract() for s in content_div('timeline')]
    [s.extract() for s in content_div('table')]
    # Get the cleaned text
    text = content_div.get_text()
    text = clean_text(text)
    # Remove the [править | править код] artifact
    text = text.replace('[править | править код]', '')
    # Return the cleaned text
    return re.sub(r'\[[0-9]+\]', '', text)
```

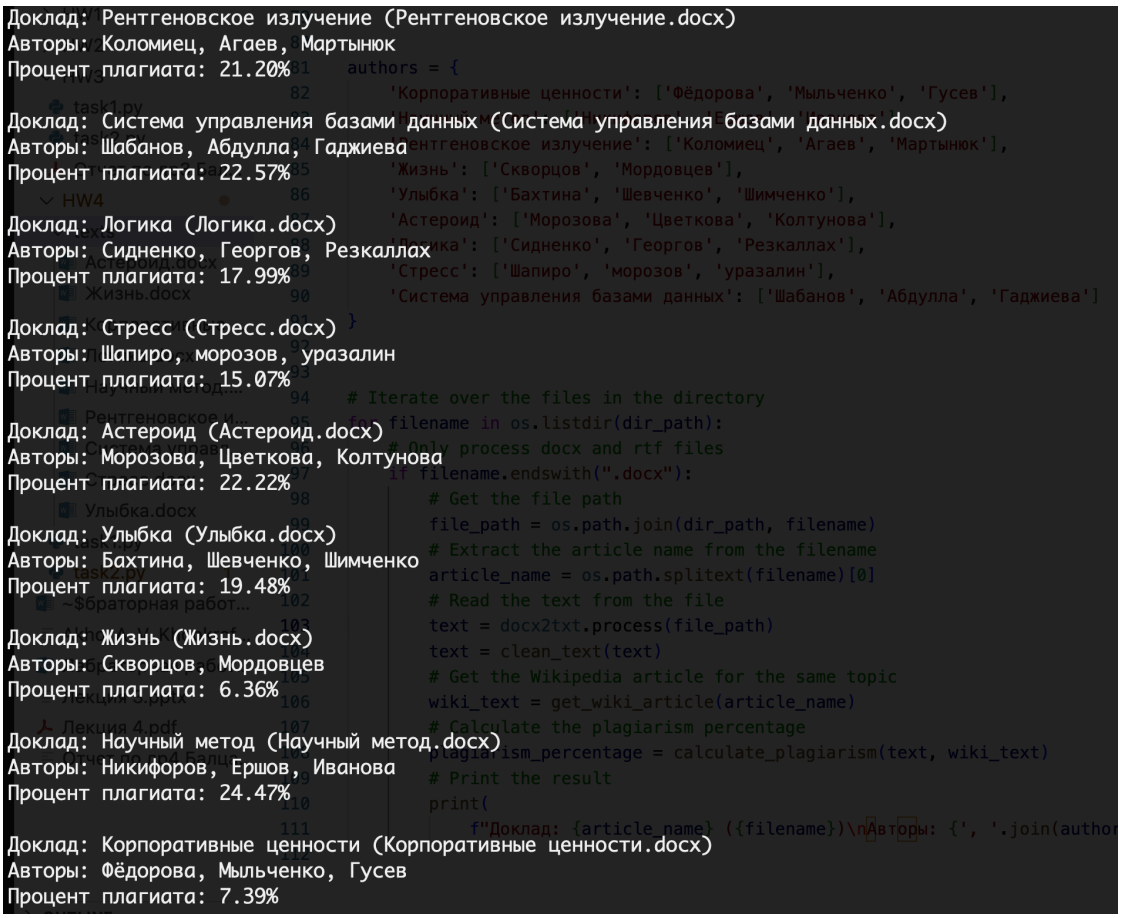
Определим функцию подсчета плагиата.

```python
# Define the function to calculate plagiarism
def calculate_plagiarism(text, wiki_text):
    # Split the text and wiki_text into words
    text_words = text.split()
    # Iterate over the text_words and compare to wiki_words
    match_count = 0
    for i in range(len(text_words)-2):
        text_tri = text_words[i] + " " + \
            text_words[i+1] + " " + text_words[i+2]
        if text_tri in wiki_text:
            match_count += 1
    # Calculate and return the plagiarism percentage
    plagiarism_percentage = (match_count / len(text_words)) * 100
    return plagiarism_percentage

```

Прочтем и обработаем файлы.

```python
authors = {
    'Корпоративные ценности': ['Фёдорова', 'Мыльченко', 'Гусев'],
    'Научный метод': ['Никифоров', 'Ершов', 'Иванова'],
    'Рентгеновское излучение': ['Коломиец', 'Агаев', 'Мартынюк'],
    'Жизнь': ['Скворцов', 'Мордовцев'],
    'Улыбка': ['Бахтина', 'Шевченко', 'Шимченко'],
    'Астероид': ['Морозова', 'Цветкова', 'Колтунова'],
    'Логика': ['Сидненко', 'Георгов', 'Резкаллах'],
    'Стресс': ['Шапиро', 'морозов', 'уразалин'],
    'Система управления базами данных': ['Шабанов', 'Абдулла', 'Гаджиева']
}


# Iterate over the files in the directory
for filename in os.listdir(dir_path):
    # Only process docx and rtf files
    if filename.endswith(".docx"):
        # Get the file path
        file_path = os.path.join(dir_path, filename)
        # Extract the article name from the filename
        article_name = os.path.splitext(filename)[0]
        # Read the text from the file
        text = docx2txt.process(file_path)
        text = clean_text(text)
        # Get the Wikipedia article for the same topic
        wiki_text = get_wiki_article(article_name)
        # Calculate the plagiarism percentage
        plagiarism_percentage = calculate_plagiarism(text, wiki_text)
        # Print the result
        print(
            f"Доклад: {article_name} ({filename})\nАвторы: {', '.join(authors[article_name])}\nПро
```

В результате работы программы получим:



Выбранный алгоритм для вычисления плагиата - это метод сравнения на основе триграмм. Метод берет входной текст и разбивает его на триграммы (группы из трех соседних слов), а затем сравнивает эти триграммы с триграммами, извлеченными из статьи Википедии.

Этот алгоритм эффективен, поскольку он учитывает контекст, в котором слова встречаются в тексте, а не просто сравнивает отдельные слова. Сравнивая группы из трех соседних слов, алгоритм может лучше уловить смысл и структуру текста и сравнить его с исходным текстом. Кроме того, метод сравнения на основе триграмм более устойчив к незначительным изменениям в тексте, таким как изменение порядка слов или небольшие изменения в формулировках. Это объясняется тем, что триграммы способны уловить сходство между текстами даже при изменении отдельных слов.

В целом, метод сравнения на основе триграмм является хорошо зарекомендовавшим себя подходом к обнаружению плагиата и доказал свою эффективность на практике.

ВЫВОД

Я изучил алгоритмы поиска подстрок и на практике понял достоинства и недостатки каждого. Теперь я способен эффективно применять каждый изученный алгоритм в зависимости от решаемой задачи.