Санкт-Петербургский Национальный Исследовательский Университет

Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

**Домашнее задание №1**

**Вариант №1**

Выполнил:

Бацанова Е. А.

Проверил

Мусаев А.А.

Санкт-Петербург,

2024

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 3](#_fmjc2tazxxvi)

[Задание 1 4](#_ffln2tea40r9)

[Задание 2 10](#_94lvfsnaijyx)

[ЗАКЛЮЧЕНИЕ 13](#_72dscdhz5eso)

[СПИСОК ЛИТЕРАТУРЫ 14](#_lyztr5onyyqo)

# **ВВЕДЕНИЕ**

Целью данной работы является ознакомление с алгоритмами поиска подстрок, их сравнение, а также применение данных алгоритмов для решения практических задач. Задачами данной лабораторной работы являются:

1. реализация алгоритмов поиска подстрок, а именно наивного алгоритма, алгоритма Рабина-Карпа, Бойера-Мура и Кнута-Морриса-Пратта;
2. сравнение изученных алгоритмов поиска подстрок, выделение их достоинств и недостатков;
3. решение задачи подсчета количества наиболее часто встречающихся двузначных чисел в строке, состоящей из 500 чисел Фибоначчи с использованием каждого из алгоритмов поиска подстрок;
4. решение задачи определения количества плагиата в тексте реферата с помощью одного (оптимального) из реализованных ранее алгоритмов.

# **Задание 1**

**Задание:** 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:**

Взят вариант 1 – заполняем массив числами Фибоначчи. Создадим строку из чисел Фибоначчи с помощью функции *fibonacci()*  (рис. 1).

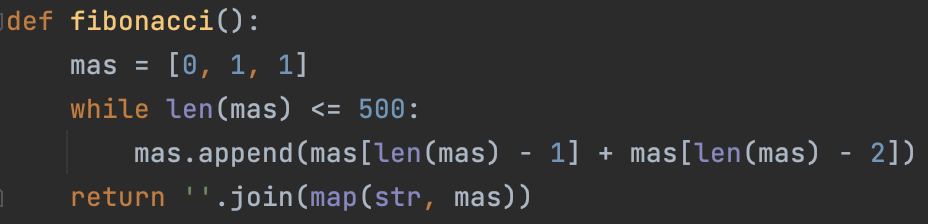


Рисунок 1 – Реализация функции, формирующей строку из 500 чисел Фибоначчи

Реализуем алгоритмы поиска.

*Примечание.* Все функции находят все вхождения подстроки *substring* в строку *line* и возвращает количество таких вхождений – *cnt*. Переменные *m* и *n* в контексте данного задания всегда будут обозначать одно и то же – длину искомой подстроки и исходной строки соответственно.

* Реализуем наивный алгоритм поиска подстрок – за это будет отвечать функция *naive(line, substring)* (рис. 2)

Сам алгоритм наивного поиска подстроки заключается в том, что программа перебирает все возможные позиции, в которых может начинаться подстрока в строке и проверяет, совпадает ли подстрока с отрезком строки, начинающимся с текущей позиции. Для этого программа сначала определяет длину подстроки *substring* и строки *line*, а затем проходит циклом по позициям начала подстроки в строке (от *0* до  *n - m*). Для каждой позиции программа проверяет соответствие символов подстроки и строки, начиная с текущей позиции. Если хотя бы один символ не совпадает, флаг *flag* устанавливается в 1 и производится выход из внутреннего цикла. Если после прохода по всем символам подстроки флаг *flag* остается равен 0, это означает, что найдено вхождение подстроки в строку, и счетчик *cnt* увеличивается на 1. По завершении всех проверок, программа возвращает общее количество найденных вхождений подстроки в строку.

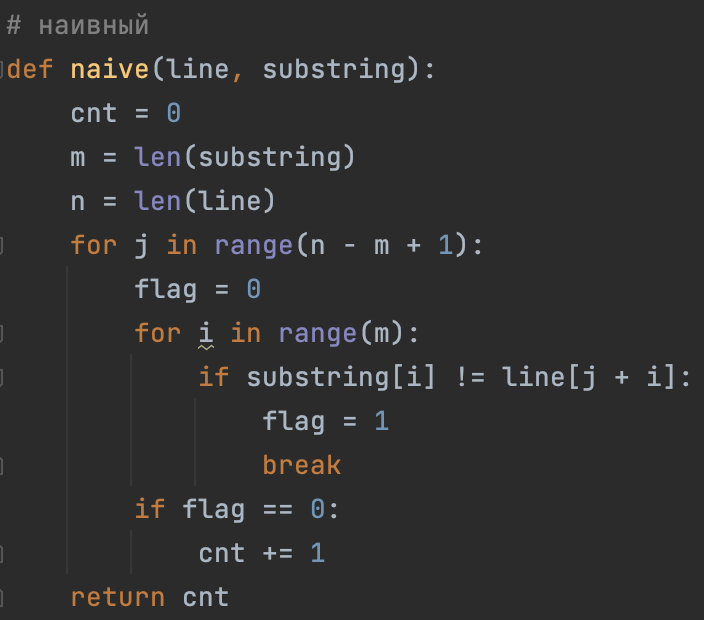


Рисунок 2 – Реализация наивного алгоритма поиска подстроки в строке

* Реализуем алгоритм Рабина-Карпа – за это будут отвечать функции *r\_k\_own\_hash(line, substring)* и *r\_k(line, substring)* (рис. 3 и рис. 4)

Алгоритм Рабина-Карпа работает на основе хэширования – каждому символу в исходной строке присваивается числовое значение, которое затем используется для расчета хэша строк.

Попробуем рассчитать хэш самостоятельно. Сначала создадим словарь *numbers\_for\_symbols*, где ключами будут являться символы из исходной строки *line*, а значениями – соответствующие им числовые значения. Затем программа вычисляет хэш искомой подстроки *h\_substring* с использованием числовых значений символов из словаря *numbers\_for\_symbols*. Далее программа вычисляет хэш *h\_line\_part* для каждой подстроки длиной *m* в строке *line*. Если хэши двух подстрок (искомой и текущей в строке) совпадают, выполняется посимвольное сравнение строк аналогично наивному алгоритму. Если символы в подстроках совпадают, увеличивается счетчик *cnt*, который в конце вычислений вернется в качестве результата данной функции - количества раз, которое подстрока встречается в исходной строке. Реализация программы представлена на рисунке 3.

При этом хэш можно вычислить с помощью встроенного метода *hash()*, тогда программа примет вид, изображенный на рисунке 4. Далее мы убедимся, что использование встроенной функции значительно ускоряет алгоритм.

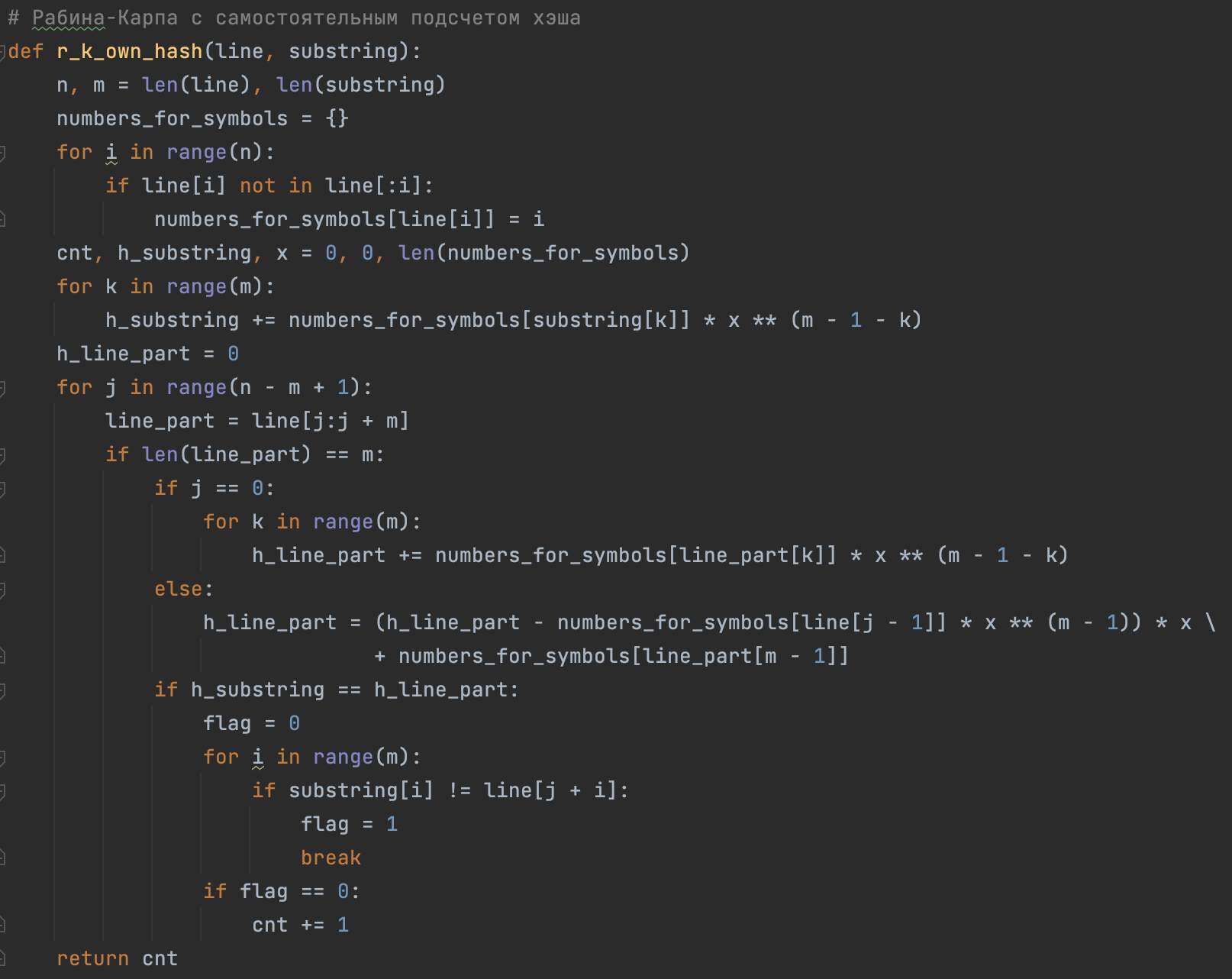


Рисунок 3 – Реализация алгоритма Рабина-Карпа поиска подстроки в строке (собственная реализация расчета хэша)

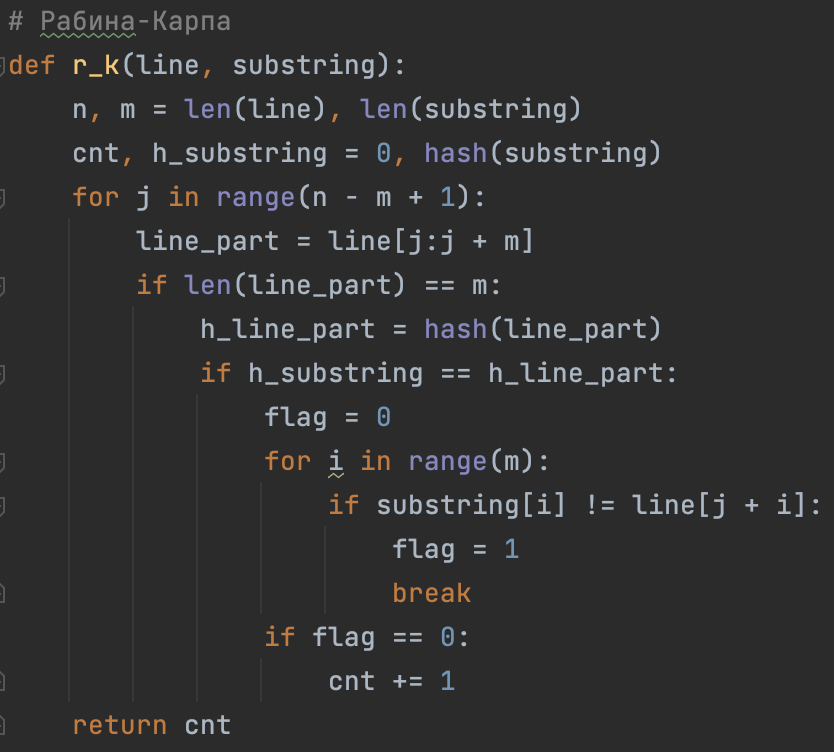


Рисунок 4 – Реализация алгоритма Рабина-Карпа поиска подстроки в строке (расчета хэш с помощью *hash()*)

* Реализуем алгоритм Бойера-Мура – за это будет отвечать функция *b\_m(line, substring)*  (рис. 5)

Для реализации алгоритма необходимо сформировать внутри функции таблицу смещений *d*, чтобы определить, на какую позицию нужно сместить поиск в случае несовпадения символов. Данная таблица обеспечивает более эффективный алгоритм поиска. Затем производится поиск подстроки в строке с помощью двух вложенных циклов. В первом цикле осуществляется проход по строке *line*, начиная с позиции *m-1*. Во втором цикле осуществляется сравнение символов подстроки с символами строки. В случае несовпадения символов подстроки и строки, вычисляется смещение *off* и совершается переход на новую позицию для сравнения символов. Если все символы подстроки совпадают с соответствующими символами строки, счетчик *cnt* увеличивается на 1, и продолжается поиск с новой позиции.

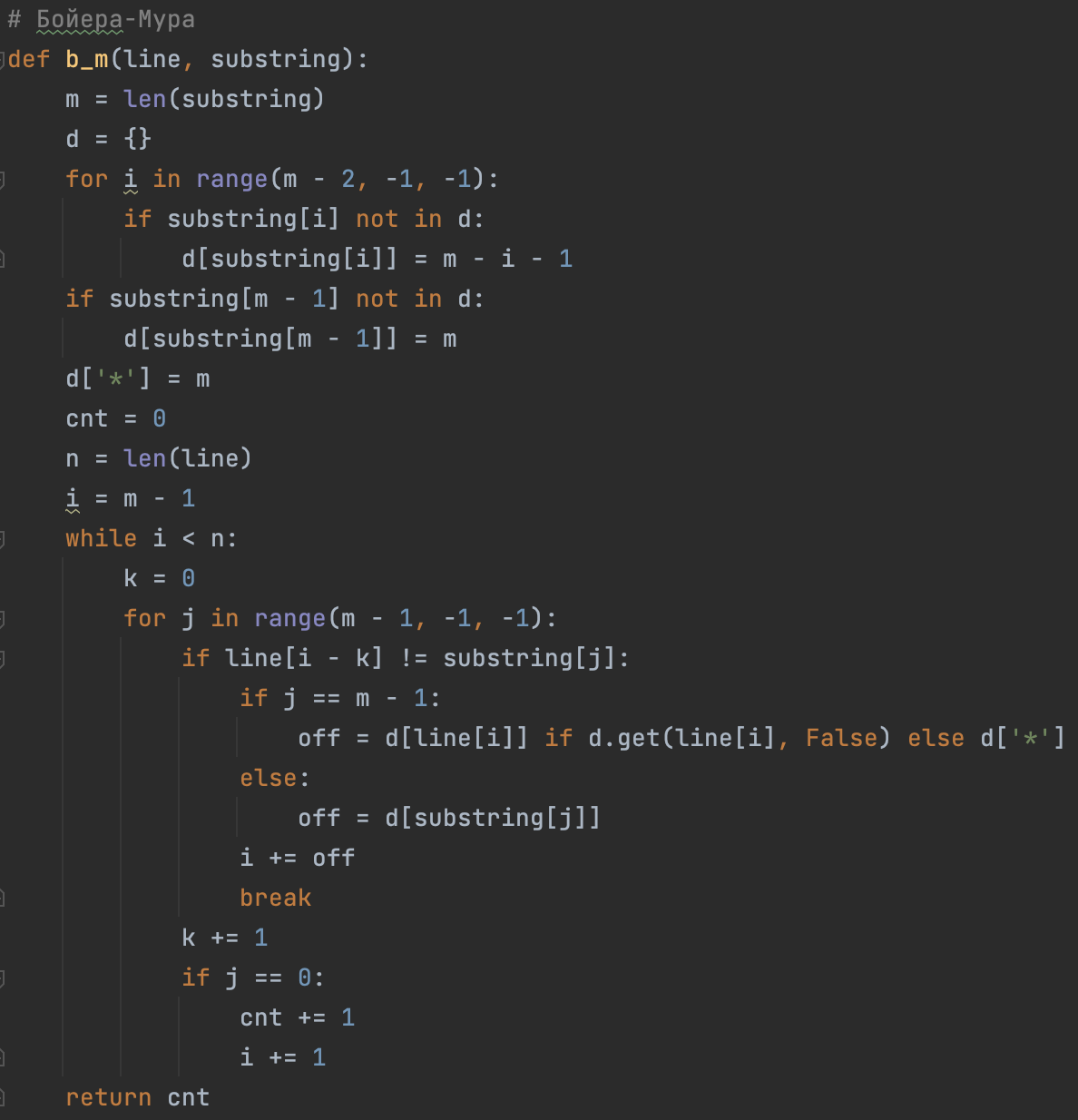


Рисунок 5 – Реализация алгоритма Бойера-Мура поиска подстроки в строке

* Реализуем алгоритм Кнута-Морриса-Пратта – за это будет отвечать функция *k\_m\_p(line, substring)*  (рис. 6)

Сначала в функции создается список *pi*, хранящий значения префикс-функций строк (то есть длину максимального суффикса подстроки, который является одновременно и её префиксом). Это необходимо для оптимизации сравнения символов при поиске подстроки. Затем программа начинает перебирать символы строки *line* и подстроки *substring*, сравнивая их. Если символы совпадают, перебор продолжается дальше. Если символы не совпадают, программа использует таблицу *pi* для определения сдвига, который можно сделать без потери информации.

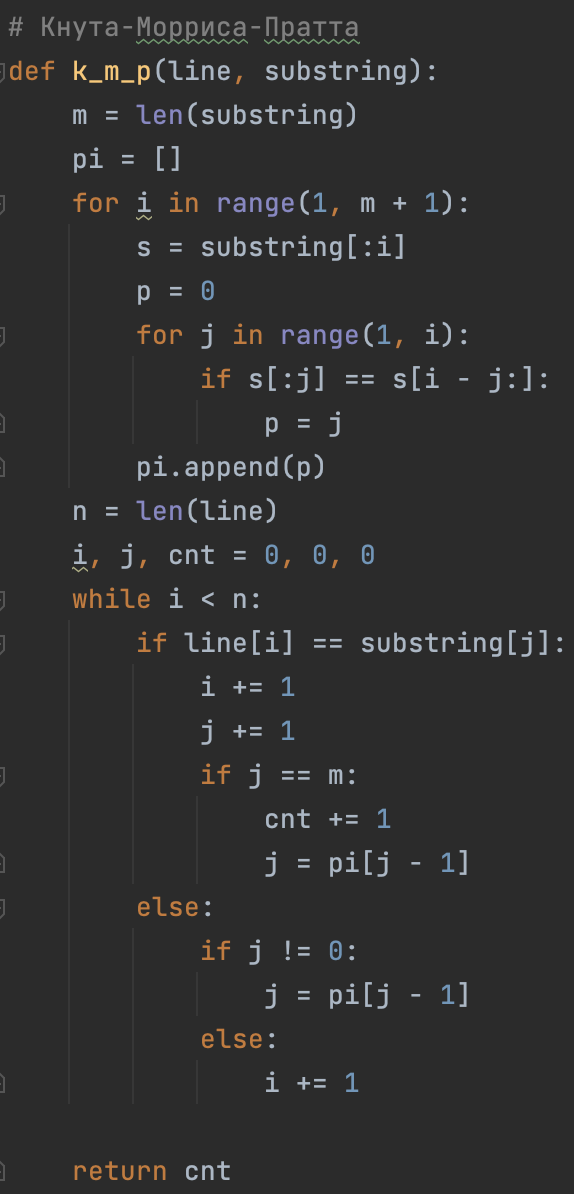


Рисунок 6 – Реализация алгоритма Кнута-Морриса-Пратта поиска подстроки в строке

Используя каждый из этих алгоритмов, посчитаем количество наиболее часто встречающихся двузначных чисел в строке, состоящей из 500 чисел Фибоначчи. Параллельно измерим время, за которое происходит поиск.

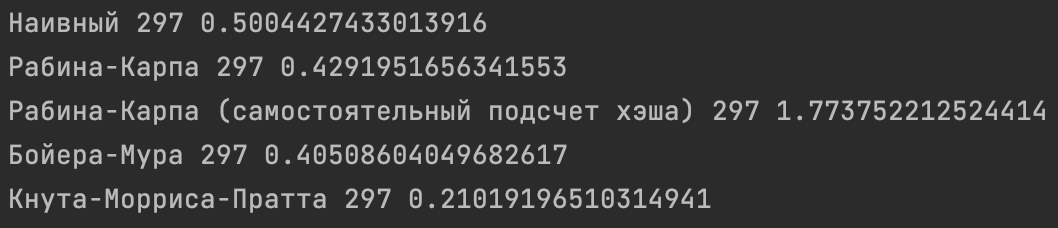


Рисунок 7 – Количество наиболее часто встречающихся двузначных чисел в строке, подсчитанное различными алгоритмами, а также время выполнения алгоритмов

Видим, что для эффективной работы алгоритма Рабина-Карпа необходимо использовать встроенную функцию подсчета хэша.

Таблица 1. Сравнение алгоритмов поиска подстрок.

| Название алгоритма | Достоинства | Недостатки |
| --- | --- | --- |
| Наивный | - простота и понятность реализации. | - неэффективен;  - медленно работает на больших текстах. |
| Рабина-Карпа | - эффективен на текстах с большим размером;  - можно искать сразу несколько подстрок, заранее вычислив их хэш. | - возможность коллизий при вычислении хеш-функции, что может приводить к ложным совпадениям. |
| Бойера-Мура | - крайне эффективен на больших текстах за счет использования предварительной обработки подстроки. | - требует дополнительной памяти для хранения таблицы смещений. |
| Кнута-Морриса-Пратта | - эффективен на больших текстах;  - эффективно обходит повторяющиеся символы | - требует предварительной обработки подстроки, что может быть неэффективно на коротких подстроках. |

# 

# **Задание 2**

**Задание:** 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:**

Взят вариант 1 – файл «Логика.rtf».

Сначала переведем информацию из файла и страницы википедии в текстовые строки. Для перевода содержимого .rtf файла в текст пользуемся модулем *rtf\_to\_text* библиотеки *striprtf*. Для чтения содержимого страницы Википедии пользуемся модулем *wikipedia*. Далее удаляем из обеих строк ненужные символы, оставляя только буквы (так как будем смотреть только на совпавшие слова), приводим все буквы к одному регистру командой *lower()* и удаляем множественные пробелы строкой *text = ' '.join(text.split()).* Полный код программы представлен на рисунке 8.

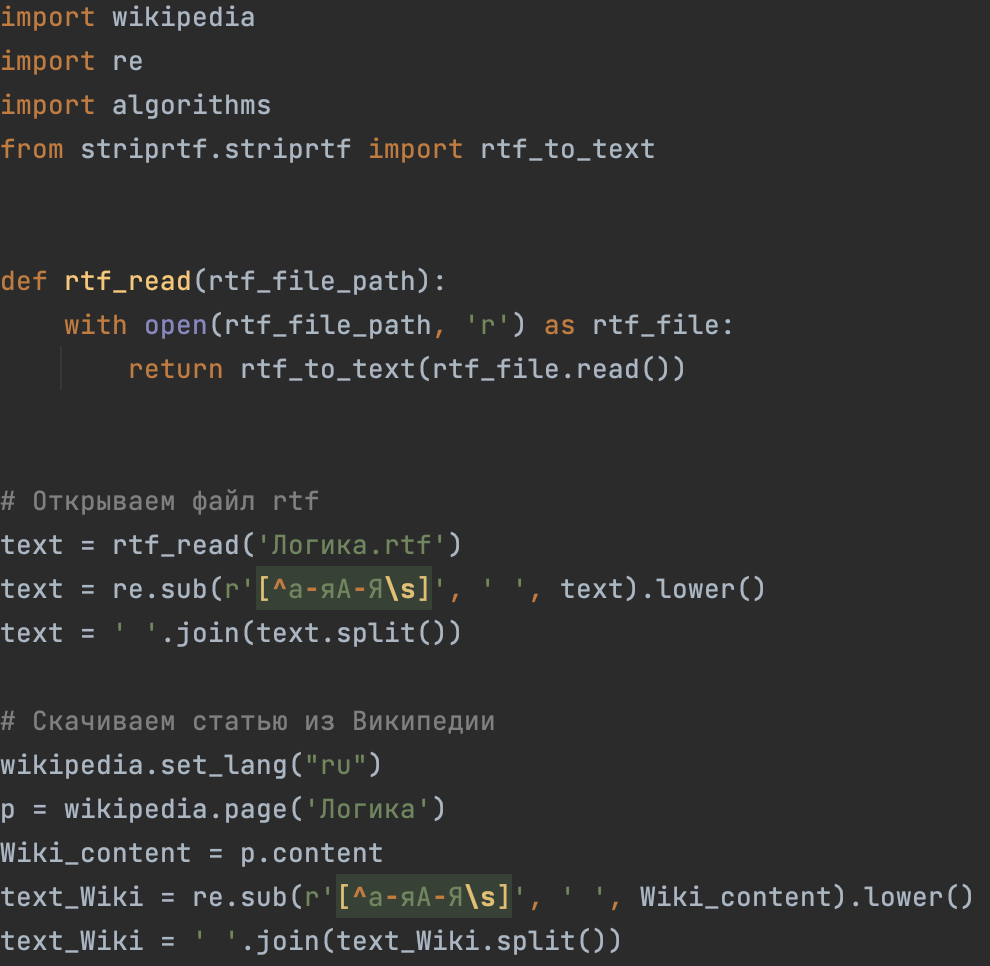


Рисунок 8 – Формирование строк из .rtf файла и страницы Википедии

Далее посчитаем количество слов, которые можно считать плагиатом. Сохраним данное значение в переменную *plagiat*. Для поиска подстрок из Википедии в строке, содержащей текст реферата, воспользуемся методом поиска Бойера-Мура, так как он обладает следующими преимуществами:

* Скорость работы: алгоритм Бойера-Мура является одним из самых быстрых алгоритмов поиска подстроки. Он работает эффективно даже при больших объемах текста;
* Эффективность поиска: данный алгоритм построен на таблице смещений, которая позволяет ему пропускать большие участки текста, если он уверен, что в них нет совпадения. В нашем случае, когда подстрока относительно большая, ведь она почти всегда содержит 3 объединенных слова, это сильно снижает количество сравнений и улучшает производительность.

Алгоритм Бойера-Мура импортируем из прошлого задания. Для определения количества плагиата в первую очередь, необходимо разделить исходный текст и текст из Википедии на слова. После этого проделывается поиск каждой тройки подряд расположенных слов из текста из Википедии внутри строки из набора слов исходного текста. В случае, если находится совпадение, оно записывается как плагиат, который прибавляется к переменной *plagiat*. Если мы нашли совпадение трех слов, то пробуем находить совпадения далее, формируя искомую строку уже из 4 и более слов. Во избежание ситуации повторного нахождения одной и той же подстроки инициируем переменную *k* – с помощью неё мы будем пропускать ненужные итерации *i*, для которых подстроки уже были найдены. В конце выводится количество плагиата в процентах относительно общего количества символов в реферате (то есть количество слов, идентифицированных как плагиат, деленное на общее количество слов).

# 

Рисунок 9 – Реализация программы для подсчета количества плагиата

# 

Рисунок 10 – Вывод программы для файла «Логика.rtf»

# 

# **ЗАКЛЮЧЕНИЕ**

В ходе данной лабораторной работы были изучены и реализованы основные алгоритмы поиска подстрок – наивный, алгоритм Рабина-Карпа, Бойера-Мура и Кнута-Морриса-Пратта.

Была решена задача подсчета количества наиболее часто встречающихся двузначных чисел в строке, состоящей из 500 чисел Фибоначчи с использованием каждого из алгоритмов поиска подстрок. Самым медленным из примененных алгоритмов оказался наивный поиск, самым быстрым – алгоритм Кнута-Морриса-Пратта (время, затраченное на поиск подстроки наивным методом превысило время поиска подстроки методом Кнута-Морриса-Пратта примерно в 2.5 раза). Наивный алгоритм поиска подстрок, хоть и прост в реализации, имеет квадратичную сложность и неэффективен на больших объемах данных. Быстродействие алгоритма Рабина-Карпа сильно зависит от метода расчета хэша – при использовании встроенной функции *hash()* алгоритм сравним по времени с алгоритмом Бойера-Мура. Тогда время работы обоих алгоритмов является средним между наивным поиском и алгоритмом Кнута-Морриса-Пратта – они демонстрируют высокую эффективность при поиске подстроки в больших объемах данных. Можно сказать, что алгоритм Бойера-Мура имеет преимущество в том, что он позволяет совершать просмотры текста справа налево, что может сократить количество сравнений. Однако при самостоятельном вычислении хэша работа алгоритма Рабина-Карпа сильно замедляется.

При анализе плагиата в тексте реферата был выбран алгоритм Бойера-Мура в силу его высокой скорости работы, возможности эффективного обнаружения совпадений и умения пропускать большие участки текста, если он уверен, что в них нет совпадения. Он позволяет искать совпадения сразу в нескольких местах одновременно и сокращает время поиска.

Таким образом, в зависимости от ситуации можно выбрать подходящий алгоритм поиска подстрок для оптимальной работы с данными.

# 

# **СПИСОК ЛИТЕРАТУРЫ**

1. Wikipedia. [Алгоритм Рабина-Карпа](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A0%D0%B0%D0%B1%D0%B8%D0%BD%D0%B0_%E2%80%94_%D0%9A%D0%B0%D1%80%D0%BF%D0%B0). [Электронный ресурс] – (Дата последнего обращения 19.05.2024);
2. Wikipedia. [Алгоритм Бойера-Мур](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%BE%D0%B9%D0%B5%D1%80%D0%B0_%E2%80%94_%D0%9C%D1%83%D1%80%D0%B0)а. [Электронный ресурс] – (Дата последнего обращения 19.05.2024);
3. Wikipedia. [Алгоритм Кнута-Морриса-Пратта](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9A%D0%BD%D1%83%D1%82%D0%B0_%E2%80%94_%D0%9C%D0%BE%D1%80%D1%80%D0%B8%D1%81%D0%B0_%E2%80%94_%D0%9F%D1%80%D0%B0%D1%82%D1%82%D0%B0) [Электронный ресурс] – (Дата последнего обращения 19.05.2024).

# 

# **ПРИЛОЖЕНИЕ**

Для удобства все файлы выгружены на GitHub: <https://github.com/kathykkKk/Algorithms-and-Data-Structures-ICT.git>