e

**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

**(СПбГУТ)**

**Факультет инфоРМАЦИОННЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ (ИТПИ)**

**кафедра программной инженерии и вычислительной техники (пиивт)**

Дисциплина: «Алгоритмы и структуры данных»

Лабораторная работа №2

**Тема: «Методы поиска»**

Отчёт

Выполнили:

Студент группы ИКПИ-33

Коньков М. Д.

Семенихин А. Р.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Принял:

Дагаев А. В.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Санкт-Петербург

2025г.

### Общая формулировка

Целью данной работы является **разработка на языке Python и тщательное исследование нескольких алгоритмов поиска подстроки в тексте**. В ходе исследования планируется **измерить время, затрачиваемое на поиск в текстовых данных разного объема** для каждого алгоритма, с последующим **сравнением их быстродействия**. Полученные **экспериментальные результаты** будут использованы для **оценки практической применимости и анализа теоретической сложности** рассматриваемых методов поиска.

### Общий алгоритм решения

1. **Генерация текстовых данных случайной длины** для проведения испытаний. Создание набора случайных строк различной размерности (например, от 1000 до 15 000 символов с шагом 1000) для имитации различных объемов входных данных.
2. **Разработка на языке Python программных реализаций следующих алгоритмов поиска подстроки:** последовательный алгоритм поиска, алгоритм Бойера-Мура и алгоритм Кнута-Морриса-Пратта. Обеспечение корректной работы каждой реализации для точного сравнения их производительности.
3. **Проведение серии измерений времени выполнения** для каждого алгоритма на текстовых данных различной длины. Для каждой размерности текста выполнить поиск случайной подстроки (ключа) фиксированной длины.
4. **Оценка среднего времени поиска** для каждого алгоритма и каждой размерности текста. Для повышения точности измерений, запуск каждого алгоритма поиска повторяется заданное количество раз (10 000 раз) для каждой комбинации размера текста и алгоритма, с последующим расчетом среднего времени выполнения.
5. **Сохранение результатов измерений в файл формата XLSX (Excel)** для удобства дальнейшего анализа и обработки данных. Организация данных в табличном виде, включающем размер текста, среднее время выполнения для каждого алгоритма и номер итерации.
6. **Сравнительный анализ практической производительности реализованных алгоритмов** на основе экспериментально полученных данных о времени выполнения. Сопоставление среднего времени поиска для каждого алгоритма при различных размерах входных текстовых данных.
7. **Исследование полученных результатов для выявления закономерностей** изменения времени поиска в зависимости от размера текста для каждого алгоритма. Анализ эффективности алгоритмов с ростом объема входных данных и определение, какие алгоритмы демонстрируют лучшую масштабируемость.
8. **Визуализация результатов тестирования** путем построения графиков, отображающих зависимость среднего времени поиска от размера текстовых данных для каждого из трех алгоритмов. Использование средств Excel для создания наглядных графических представлений.
9. **Формулирование итоговых выводов** относительно сравнительной эффективности исследованных алгоритмов поиска подстроки для различных размеров текстовых данных. Рекомендации по применению наиболее подходящего алгоритма в зависимости от ожидаемого объема текстовых данных и требований к скорости поиска.

### Описание алгоритмов поиска

**Алгоритм последовательного поиска:**

Последовательный (линейный) поиск – это один из самых простых алгоритмов для нахождения подстроки в строке. Алгоритм последовательно сравнивает каждый символ основной строки с первым символом подстроки. Если символы совпадают, начинается проверка остальных символов подстроки. Если вся подстрока совпадает с соответствующей частью строки, поиск завершается. В противном случае процесс продолжается до конца строки.

### ****Пример работы:****

#### ****1. Пошаговый поиск подстроки в строке****

Ищем подстроку **ABC** в строке **ABCAFDBCABC**.

1. **Первый шаг**
   * Начинаем с первого символа строки.
   * Сравниваем **ABC** и **ABC**.
   * Все символы совпадают → подстрока найдена на позиции 0.
2. **Второй шаг** (если продолжаем поиск)
   * Продолжаем проверку с четвертого символа строки.
   * Сравниваем **CAF** и **ABC**.
   * Символ C ≠ A → несовпадение, продолжаем поиск.
3. **Третий шаг**
   * Сравниваем **AFD** и **ABC**.
   * Символ A = A, но F ≠ B → несовпадение, продолжаем поиск.
4. **Четвертый шаг**
   * Сравниваем **FDB** и **ABC**.
   * Символ F ≠ A → несовпадение, продолжаем поиск.
5. **Пятый шаг**
   * Сравниваем **DBC** и **ABC**.
   * Символ D ≠ A → несовпадение, продолжаем поиск.
6. **Шестой шаг**
   * Сравниваем **BCA** и **ABC**.
   * Символ B ≠ A → несовпадение, продолжаем поиск.
7. **Седьмой шаг**
   * Сравниваем **CAB** и **ABC**.
   * Символ C ≠ A → несовпадение, продолжаем поиск.
8. **Восьмой шаг**
   * Сравниваем **ABC** и **ABC**.
   * Все символы совпали → подстрока найдена на позиции 8.

Алгоритм завершает работу, так как подстрока найдена дважды в строке.

**Алгоритм Бойера-Мура:**

Алгоритм Бойера-Мура считается одним из самых быстрых методов поиска подстроки в строке. В начале работы строится таблица смещений, после чего осуществляется процесс поиска. Сравнение символов начинается с конца подстроки и идет справа налево. Если последний символ не совпадает с символом основной строки, производится сдвиг на значение из таблицы смещений. В случае несовпадения другого символа подстрока сдвигается на одну позицию вправо. Если все символы совпали – подстрока найдена.

### ****Пример работы:****

#### ****1. Построение таблицы смещений:****

Для построения таблицы смещений определяется удаленность каждого символа от конца подстроки. Используются следующие правила:

* Если символ встречается несколько раз, берется расстояние от последнего вхождения.
* Если символ отсутствует в подстроке, ему присваивается значение, равное длине подстроки.

##### ****Пример: Подстрока ABCABD, строка ABCAFDFABCABD****

Алфавит: A, B, C, D, F

Таблица 1

| Символ | Сдвиг |
| --- | --- |
| A | 2 |
| B | 1 |
| C | 3 |
| D | 0 |
| F | 6 |

#### ****2. Пошаговый поиск подстроки в строке****

Ищем подстроку ABCABD в строке ABCAFDFABCABD.

1. **Первый шаг**
   * Сравниваем ABCABD и ABCAFD.
   * Символ D ≠ F, но F отсутствует в подстроке.
   * Сдвигаем подстроку на всю её длину (6 символов).
2. **Второй шаг**
   * Сравниваем ABCABD и DFABCA.
   * Символ D ≠ A, A есть в подстроке, его сдвиг – 2.
   * Сдвигаем подстроку на 2 символа.
3. **Третий шаг**
   * Сравниваем ABCABD и ABCABD.
   * Все символы совпали.
   * Подстрока найдена.

Алгоритм завершает работу, так как найдено точное совпадение.

**Алгоритм Кнута-Морриса-Пратта:**

Алгоритм Кнута-Морриса-Пратта (КМП) – это эффективный метод поиска подстроки в строке, использующий предварительную обработку подстроки для оптимизации сдвигов. В начале работы строится префикс-функция (таблица длин наибольших совпадающих префиксов и суффиксов подстроки). Затем происходит сравнение символов строки и подстроки. При несовпадении поиск продолжается не с начала подстроки, а с позиции, определяемой префикс-функцией, что ускоряет процесс.

### ****Пример работы:****

### ****Построение префикс-функции:****

**Для построения префикс-функции строки, давайте рассмотрим пример на строках "**ABABABABAC**" и подстроке "**ABABAC**". Префикс-функция помогает находить длину наибольшего префикса, который является одновременно и суффиксом для каждой позиции строки. Рассмотрим, как мы будем строить эту функцию поочередно для каждого символа строки.**

**Начнем с первого символа строки, который находится на позиции 1. Это символ "A". Мы ищем наибольший префикс, который одновременно является суффиксом. Для этого достаточно рассмотреть единственный символ "A", который будет и префиксом, и суффиксом, то есть длина совпадения равна 1. Значение префикс-функции для первой позиции будет равно 1.**

**Далее переходим ко второму символу на позиции 2 — это "B". Рассматриваем префикс "AB", который заканчивается на эту позицию. Однако, суффикс для строки на данный момент может быть только буквой "B", и совпадений с началом строки нет. Следовательно, длина совпадения на этой позиции равна 0, и значение префикс-функции для второй позиции будет равно 0.**

**Теперь смотрим на третий символ на позиции 3 — снова "A". Префикс в этом случае уже будет "ABA", так как строка на данный момент начинается с "AB". Мы видим, что "A" в начале строки совпадает с "A", который стоит на данной позиции. Таким образом, длина совпадения составляет 1, и префикс-функция на этой позиции будет равна 1.**

**На четвёртой позиции стоит символ "B", а префикс к этому моменту уже имеет вид "ABAB". Мы видим, что на позиции 4 и суффикс, и префикс совпадают до двух символов — это "AB". Таким образом, на этой позиции длина совпадения составит 2, и префикс-функция для этой позиции будет равна 2.**

**Далее смотрим на пятую позицию, где стоит символ "A". Префикс на пятой позиции уже будет равен "ABABA", что начинается с тех же символов, что и строка на позиции 1. Здесь у нас совпадение по первым трем символам: "A". Следовательно, длина совпадения будет равна 3, и префикс-функция для пятой позиции будет равна 3.**

**Наконец, переходим к последнему символу подстроки на шестой позиции, где стоит символ "C". На этом этапе префикс на данной позиции будет выглядеть как "ABABAC". Однако, здесь нет символа в строке, который совпадает с буквой "C" в начале. Суффикс будет совсем другим, и нет совпадений. Поэтому длина совпадений на этой позиции равна 0, и префикс-функция для шестой позиции будет равна 0.**

**Таким образом, мы получаем таблицу префикс-функции для строки "ABABAC", в которой указаны значения для каждой позиции (таблица префикс-функций):**

Таблица 2

|  |  |  |
| --- | --- | --- |
| Позиция | Символ | Значение префикс-функции |
| 1 | A | 1 |
| 2 | B | 0 |
| 3 | A | 1 |
| 4 | B | 2 |
| 5 | A | 3 |
| 6 | C | 0 |

### Пошаговый поиск подстроки в строке

Рассмотрим подстроку **ABABAC**. Для неё строится таблица:

Таблица 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Шаг** | **Строка** | **Подстрока** | **Совпадения** | **Сдвиг** |
| 1 | **ABABA**BABAC | ABABAC | ABABA | 3 |
| 2 | AB**ABAB**ABAC | ABABAC | ABAB | 2 |
| 3 | ABAB**ABA**BAC | ABABAC | ABA | 2 |
| 4 | ABABAB**ABAC** | ABABAC | ABAC | 1 |
| 5 | ABAB**ABABAC** | ABABAC | ABABAC | Найдено |

* Если символ совпадает с началом подстроки, значение увеличивается.
* Если несовпадение, используется предыдущее значение для сдвига.

#### ****2. Пошаговый поиск подстроки в строке****

**Ищем ABABAC в строке ABABABABAC.**

****1. Первый шаг****

**◦ Сравниваем ABABAC и ABABAB.**

**◦ Пять первых символов совпали, но C ≠ B.**

**◦ По таблице префиксов сдвиг на 3 позиции.**

****2. Второй шаг****

**◦ Сравниваем ABABAC и ABAB (начиная с 4-й позиции).**

**◦ Четыре первых символа совпали, но A ≠ B.**

**◦ По таблице префиксов сдвиг на 2 позиции.**

****3. Третий шаг****

**◦ Сравниваем ABABAC и ABAC (начиная с 6-й позиции).**

**◦ Все символы совпали.**

**◦ Подстрока найдена на позиции 4.**

**Алгоритм завершает работу, так как найдено полное совпадение.**

### ****Сравнение алгоритмов поиска****

***Таблица 4***

| ****Алгоритм**** | ****Плюсы**** | ****Минусы**** |
| --- | --- | --- |
| **Последовательный поиск** | Прост в реализации, не требует дополнительной памяти | Медленный при больших объемах данных |
| **Бойера-Мура** | Быстро на длинных строках, использует эвристики для оптимизации | Может быть неэффективным на коротких строках или при частых совпадениях |
| **Кнута-Морриса-Пратта** | Использует предварительную обработку для ускорения поиска | Требует дополнительной памяти для таблицы префиксов |

В ходе выполнения сравнения трех алгоритмов поиска подстроки мы измеряли время выполнения для каждого из них при различных размерах строк, повторяя измерения 10 000 раз для минимизации влияния случайных колебаний. Это позволяет более точно оценить производительность каждого алгоритма и исключить случайные отклонения, вызванные внешними факторами.

Проанализировав результаты, можно выделить несколько ключевых наблюдений:

* **Последовательный поиск** показывает худшую производительность среди всех алгоритмов. Время выполнения растет линейно с увеличением размера строки, что делает его малопригодным для работы с большими объемами данных.
* **Алгоритм Бойера-Мура** продемонстрировал наибольшее преимущество при работе с длинными строками, особенно когда текст содержит разнообразный алфавит. Эвристики "плохого символа" и "хорошего суффикса" позволяют значительно сократить количество проверок, что ускоряет поиск в случае больших текстов.
* **Алгоритм Кнута-Морриса-Пратта** в сравнении с другими показал более устойчивую производительность, что связано с особенностями генерации строк. Строка создавалась случайным образом со случайным расстановкой необходимых подстрок, после чего происходило достроение до нового размера .
* При **малых размерах строк** различия между алгоритмами не так заметны, однако с увеличением длины строки преимущества Бойера-Мура становятся более выраженными, а недостатки последовательного алгоритма усиливаются.
* На больших строках **алгоритм Бойера-Мура** показывает значительное превосходство в скорости (в несколько десятков раз быстрее, чем последовательный поиск), а **Кнут-Моррис-Пратт** стабильно работает при наличии повторяющихся фрагментов, но не достигает такой же скорости.

Таким образом, выбор алгоритма поиска зависит от размера данных и структуры текста: для небольших строк разница в производительности не так значительна, но для длинных строк алгоритм Бойера-Мура является наиболее эффективным.

**Описание программы**

Данная программа, написанная на языке Python (версии 3.11.2), предназначена для оценки и сравнения производительности различных алгоритмов поиска подстроки в строке. В частности, программа тестирует три алгоритма: последовательный поиск подстроки, алгоритм Бойера-Мура и алгоритм Кнута-Морриса-Пратта (КМП). В процессе работы программа генерирует случайные тексты различной длины и случайный ключ (подстроку). Затем измеряется время выполнения каждого алгоритма при поиске количества вхождений этого ключа в сгенерированных текстах. Полученные результаты усредняются по множеству итераций и сохраняются в формате Excel (XLSX) для дальнейшего анализа.

Целью программы является проведение сравнительного анализа производительности данных алгоритмов, акцентируя внимание на времени их выполнения в зависимости от длины текста.

**Основные функции программы:**

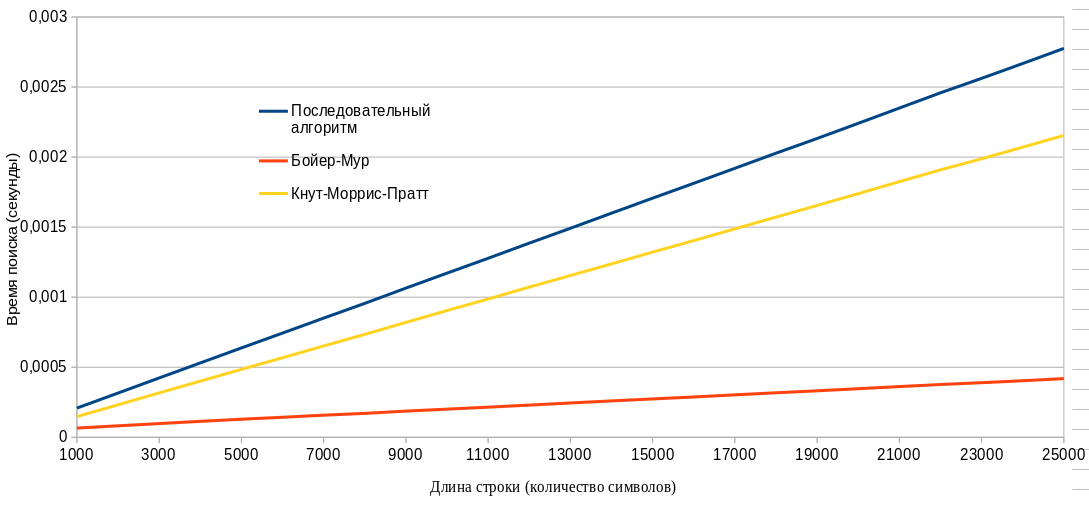
• **generate\_key(length=15)** – создает случайную строку-ключ заданной длины, используя буквы и цифры.  
• **generate\_text(length, key)** – формирует случайный текст указанной длины, встраивая в него несколько вхождений заданного ключа.  
• **sequential\_search(text, key)** – реализует последовательный алгоритм поиска подстроки для подсчета вхождений ключа в тексте.  
• **BoyerMoore.search(text)** – метод класса BoyerMoore, реализует алгоритм Бойера-Мура для поиска вхождений ключа, используя эвристики плохого символа и хорошего суффикса.  
• **KMP.search(text)** – метод класса KMP, реализует алгоритм Кнута-Морриса-Пратта, применяя предварительно вычисленный массив наибольших суффиксов-префиксов (LPS) для эффективного поиска.  
• Основная часть скрипта выполняет итерации по различным длинам текста и множеству повторений, замеряя время выполнения каждого алгоритма поиска и усредняя результаты.  
• Программа сохраняет усредненные значения времени выполнения для каждого алгоритма и длины текста в файл Excel.  
• В заключение, программа выводит в консоль усредненные значения времени выполнения для каждого алгоритма и длины текста.

**Дополнительные компоненты и технологии:**

• **random.choices()** – используется для генерации случайных строк и ключей.  
• **string.ascii\_letters + string.digits + ' '** – задает набор символов (буквы, цифры и пробел), используемый при генерации случайных строк.  
• **time.time()** – применяется для точного измерения времени выполнения алгоритмов.  
• **pandas.DataFrame** – библиотека для создания и манипуляции табличными данными, используемая для хранения и экспорта результатов в Excel.  
• **collections.defaultdict** – используется для хранения накапливаемых значений времени выполнения для каждого алгоритма и длины текста.

Программа позволяет провести наглядное сравнение эффективности различных алгоритмов поиска подстроки в зависимости от размера текста. Путем измерения и фиксации времени выполнения, она дает возможность проанализировать, какой из алгоритмов демонстрирует наилучшую производительность в различных условиях и предоставляет данные для дальнейшего изучения и оптимизации. Выходные данные, сохраненные сохраняются в файл Excel для графиков.

**Результаты работы**



****Рис. 1** Графическое сравнение последовательного поиска, алгоритма Бойера-Мура, алгоритма Кнута-Морриса-Пратта**

**Жёлтым цветов показана работа алгоритма поиска Кнута-Морриса-Пратта, красным — Бойера-Мура, а синим — работа последовательного алгоритма поиска. Было подтверждено, что алгоритм Бойера-Мура работает в десятки раз быстрее последовательного поиска (последний, в свою очередь, сильно теряет в эффективности при увеличении числа элементов). Алгоритм поиска Кнута-Морриса-Пратта показал среднюю эффективность.**

**Вывод:**

**В ходе выполнения данной работы была реализована программа на языке Python (версии 3.11.2) для исследования и сравнительного анализа трех классических алгоритмов поиска подстроки в строке: последовательного алгоритма, алгоритма Бойера-Мура и алгоритма Кнута-Морриса-Пратта. Оценка их производительности проводилась путем измерения среднего времени выполнения на строках различных размеров — от 1000 до 25 000 символов. Для повышения точности результатов каждое измерение повторялось 10 000 раз с последующим усреднением. Использование стандартных средств Python, таких как модуль time для измерения временных затрат и биxблиотека openpyxl для сохранения результатов, позволило наглядно проанализировать различия в эффективности данных алгоритмов. Полученные результаты подтверждают теоретические ожидания и демонстрируют значительное влияние структуры входных данных на скорость выполнения поиска.**

1. **Изучено:**

◦ Алгоритмы поиска подстроки: последовательный алгоритм, алгоритм Бойера-Мура и алгоритм Кнута-Морриса-Пратта в контексте их программной реализации и сравнительного анализа производительности.

1. **Освоено:**

◦ Методика измерения временных затрат алгоритмов поиска в Python с использованием функции time.perf\_counter().

◦ Практическая реализация алгоритмов последовательного поиска, Бойера-Мура и Кнута-Морриса-Пратта на языке Python.

◦ Использование Excel для сохранения и анализа результатов тестирования.

1. **Разработана программа, которая:**

◦ Генерирует случайные строки и подстроки заданной длины.

◦ Реализует три алгоритма поиска и измеряет их среднее время выполнения.

◦ Сохраняет результаты в Excel-файл для дальнейшего анализа.

1. **Анализ результатов показал:**

◦ Последовательный алгоритм продемонстрировал предсказуемую линейную зависимость времени выполнения от длины строки. Несмотря на простоту, его производительность оказалась значительно ниже по сравнению с другими методами.

◦ Алгоритм Бойера-Мура показал наилучшие результаты по скорости поиска благодаря возможности пропуска частей строки при сравнении. Его преимущество особенно заметно при длинных строках.

◦ Алгоритм Кнута-Морриса-Пратта продемонстрировал устойчивую производительность, опережая последовательный метод, но уступая алгоритму Бойера-Мура. Подготовка вспомогательного массива (lps) снижает количество сравнений, но на случайных строках его эффективность не достигает максимума.

****Листинг****

import random

import string

import time

import pandas as pd

from collections import defaultdict

def generate\_key(length=15):

return ''.join(random.choices(string.ascii\_letters + string.digits, k=length))

def generate\_text(length, key):

text = ''.join(random.choices(string.ascii\_letters + string.digits + ' ', k=length))

num\_insertions = random.randint(1, 100)

positions = random.sample(range(length), num\_insertions)

for pos in positions:

text = text[:pos] + key + text[pos:]

return text

def sequential\_search(text, key):

count = 0

n = len(text)

m = len(key)

for i in range(n - m + 1):

match = True

for j in range(m):

if text[i + j] != key[j]:

match = False

break

if match:

count += 1

return count

class BoyerMoore:

def \_\_init\_\_(self, pattern):

self.pattern = pattern

self.bad\_char = self.bad\_character\_table(pattern)

self.good\_suffix = self.good\_suffix\_table(pattern)

def bad\_character\_table(self, pattern):

table = {}

for i in range(len(pattern) - 1):

table[pattern[i]] = len(pattern) - 1 - i

return table

def good\_suffix\_table(self, pattern):

m = len(pattern)

suffix\_table = [0] \* m

last\_prefix\_pos = m

for i in range(m - 1, -1, -1):

if self.is\_prefix(pattern, i + 1):

last\_prefix\_pos = i + 1

suffix\_table[m - 1 - i] = last\_prefix\_pos - i

prefix\_table = [0] \* m

for i in range(m):

prefix\_table[i] = last\_prefix\_pos

for i in range(m - 1):

slen = self.suffix\_length(pattern, i)

prefix\_table[slen] = m - 1 - i

return prefix\_table

def is\_prefix(self, pattern, p):

m = len(pattern)

for i in range(p):

if pattern[i] != pattern[m - p + i]:

return False

return True

def suffix\_length(self, pattern, i):

length = 0

m = len(pattern)

for j in range(i, -1, -1):

if pattern[j] == pattern[m - 1 - i + j]:

length += 1

else:

break

return length

def search(self, text):

m, n = len(self.pattern), len(text)

if m > n:

return 0

bad\_char = self.bad\_char

good\_suffix = self.good\_suffix

count = 0

i = 0

while i <= n - m:

j = m - 1

while j >= 0 and self.pattern[j] == text[i + j]:

j -= 1

if j < 0:

count += 1

i += good\_suffix[0] if good\_suffix[0] != 0 else 1

else:

char = text[i + j]

bad\_char\_shift = bad\_char.get(char, m) - (m - 1 - j) if char in bad\_char else m

good\_suffix\_shift = good\_suffix[j + 1] if j + 1 < m else 1

i += max(bad\_char\_shift, good\_suffix\_shift)

return count

class KMP:

def \_\_init\_\_(self, pattern):

self.pattern = pattern

self.lps = self.compute\_lps(pattern)

def compute\_lps(self, pattern):

m = len(pattern)

lps = [0] \* m

length = 0

i = 1

while i < m:

if pattern[i] == pattern[length]:

length += 1

lps[i] = length

i += 1

else:

if length != 0:

length = lps[length - 1]

else:

lps[i] = 0

i += 1

return lps

def search(self, text):

m, n = len(self.pattern), len(text)

lps = self.lps

i = j = 0

count = 0

while i < n:

if self.pattern[j] == text[i]:

i += 1

j += 1

if j == m:

count += 1

j = lps[j - 1]

elif i < n and self.pattern[j] != text[i]:

if j != 0:

j = lps[j - 1]

else:

i += 1

return count

num\_iterations = 10000

text\_lengths = list(range(1000, 25001, 1000))

results = defaultdict(lambda: {'sequential': 0, 'boyer\_moore': 0, 'kmp': 0})

for \_ in range(num\_iterations):

key = generate\_key()

for length in text\_lengths:

text = generate\_text(length, key)

start = time.time()

sequential\_search(text, key)

time\_sequential = time.time() - start

bm = BoyerMoore(key)

start = time.time()

bm.search(text)

time\_bm = time.time() - start

kmp = KMP(key)

start = time.time()

kmp.search(text)

time\_kmp = time.time() - start

results[length]['sequential'] += time\_sequential

results[length]['boyer\_moore'] += time\_bm

results[length]['kmp'] += time\_kmp

data = {

"Длина строки": [],

"Последовательный алгоритм": [],

"Бойер-Мур": [],

"Кнут-Моррис-Пратт": []

}

for length in text\_lengths:

data["Длина строки"].append(length)

data["Последовательный алгоритм"].append(round(results[length]['sequential'] / num\_iterations, 6))

data["Бойер-Мур"].append(round(results[length]['boyer\_moore'] / num\_iterations, 6))

data["Кнут-Моррис-Пратт"].append(round(results[length]['kmp'] / num\_iterations, 6))

df = pd.DataFrame(data)

df.to\_excel("search\_results.xlsx", index=False, float\_format="%.6f")

for length in text\_lengths:

print(f"Длина строки: {length}")

print(f" Последоватльный алгоритм: {results[length]['sequential'] / num\_iterations:.6f} сек")

print(f" Бойер-Мур: {results[length]['boyer\_moore'] / num\_iterations:.6f} сек")

print(f" Кнут-Моррис-Пратт: {results[length]['kmp'] / num\_iterations:.6f} сек")

print("-")