**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc198492381)

[ТЕОРЕТИЧЕСКИЙ АНАЛИЗ ЗАДАЧИ И АЛГОРИТМОВ ЕЕ РЕШЕНИЯ 5](#_Toc198492382)

[Формальное описание выбранной задачи 5](#_Toc198492383)

[ИССЛЕДОВАНИЕ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧИ 6](#_Toc198492384)

[Наивный алгоритм 6](#_Toc198492385)

[Алгоритм Кнута-Мориса-Пратта(КМП) 7](#_Toc198492386)

[РЕАЛИЗАЦИЯ ИССЛЕДУЕМОГО АЛГОРИТМА 12](#_Toc198492387)

[Программная реализация 12](#_Toc198492388)

[Unit тестирование написанного кода 18](#_Toc198492389)

[ИНТЕГРАЦИЯ В ВЕБ-ИНТЕРФЕЙС 23](#_Toc198492390)

[Постановка задачи прикладного применения исследуемого алгоритма 23](#_Toc198492391)

[Выбор технологического стека 24](#_Toc198492392)

[Реализация фронт и бэк частей 25](#_Toc198492393)

# ВВЕДЕНИЕ

**Актуальность**: поиск подстроки является одной из ключевых задач в компьютерных науках, находящий применение в обработке текстов, системах антиплагиата и других областях. Эффективность алгоритмов поиска подстроки, зачастую, определяет производительность таких систем.

**Цель работы**: разработать программную реализацию алгоритма Кнута-Морриса-Пратта(КМП) для поиска подстроки и интегрировать его в веб-сервис для практического использования.

**Объект исследования**: алгоритм Кнута-Морриса-Пратт, предназначенный для поиска подстроки в тексте.

Предмет исследования: реализация префикс-функции, которая лежит в основе алгоритма КМП

**Задачи**:

1. Провести теоретический анализ задачи поиска подстроки и методов ее решения.
2. Сравнить точность и скорость работы КМП и других алгоритмов поиска подстроки.
3. Реализовать алгоритм КМП на языках программирования Python и C++ с подробными комментариями
4. Выполнить юнит-тестирование алгоритма
5. Создать веб-интерфейс на JavaScript для взаимодействия с пользователем

**Практическая значимость**: Результаты исследования могут быть использованы в системах антиплагиата для быстрого и точного выявления заимствований в текстах.

Методы исследования:

Структура работы:

# ТЕОРЕТИЧЕСКИЙ АНАЛИЗ ЗАДАЧИ И АЛГОРИТМОВ ЕЕ РЕШЕНИЯ

## Формальное описание выбранной задачи

**Наименование**: задача поиска подстроки  
Описание задачи: Задача поиска подстроки заключается в обнаружении всех вхождений заданного образца (паттерна) в тексте. Формально: для строки T (длины n) и образца P (длины m) требуется найти все индексы i, где подстрока T[i..i+m-1] совпадает с P.

**Математическая модель**

Дано две строки: текст (T) длиной (n) и подстрока (P) длиной (m), при этом 1 **≤** m **≤** n

**Цель**: найти все индексы (i), начиная с которых подстрока (P) содержится в тексте(T), если такого индекса нет – вернуть пустое множество

**Целевая функция**: минимизация времени исполнения алгоритма, выраженная через асимптотическую сложность.

**Ограничения**:

1. Строки T и P состоят из некоторого конечного алфавита.
2. n и m – положительные целые числа, причем n ≥ m
3. Шаблон P может встречаться в тексте T ноль и более раз.

## ИССЛЕДОВАНИЕ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧИ

Рассмотрим дав алгоритма решения задачи поиска подстроки, напишем реализацию(псевдокод), разберем принцип работы.

### Наивный алгоритм

Идея данного алгоритма заключается в проверке каждой возможной позиции i в тексте T и сравнении символов с подстрокой P, если совпадение найдено, то индекс записываем результат.

**Принцип работы**: наивный алгоритм последовательно проверяет каждую возможную позицию i в тексте T. Для каждой позиции символы T[i..i+m-1] сравниваются с P[0..m-1]. При несовпадении проверка прерывается, и алгоритм переходит к следующей позиции i+1.

С псевдокодом данного алгоритма можно ознакомиться на Рисунке 1

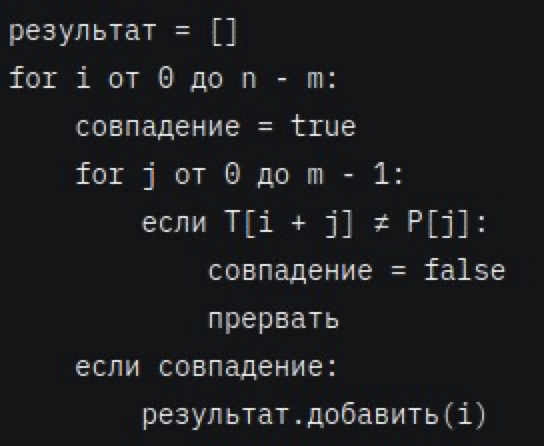


Рисунок 1 – Псевдокод линейного алгоритма

В лучшем случае асимптотическая сложность данного алгоритма составляет O(n), например, если шаблон не найден или сразу найден. Худшая O(n\*m) при множестве частичных совпадений.

Наивный алгоритм прост в реализации, но неэффективен для больших текстов и шаблонов из-за большого числа повторных сравнений.

### Алгоритм Кнута-Мориса-Пратта(КМП)

КМП избегает повторных сравнений символов текста за счет использования **префикс-функции**, которая хранит информацию о совпадающих префиксах и суффиксах образца. Это позволяет выполнять оптимальные сдвиги при несовпадениях.

Алгоритм использует два этапа:

1. Построение префикс-функции (массива pi) для подстроки.
2. Поиск подстрок в тексте с использованием информации из pi.

Для строки P префикс-функция pi (определяется как массив, где pi[i] — длина наибольшего собственного префикса подстроки P[0..i], совпадающего с её суффиксом.

Алгоритм построения pi:

1. Проверка крайних случаев
2. Декларация массива pi и инициализация первого значения pi[0].
3. Использование двух указателей: k (длина текущего совпадающего префикса) и i (индекс для обхода образца).
4. Если не совпадают, уменьшаем pi до pi[pi-1] (возвращаемся к предыдущему совпадению).
5. Если символы P[i] и P[k] совпадают, увеличиваем k и в конце обхода записываем его в pi[i].

Сложность алгоритма нахождения префикс-функции составляет O(m), где m – длина подстроки.

С реализацией данного алгоритма на языке программирования Java можно ознакомиться на Рисунке 2.

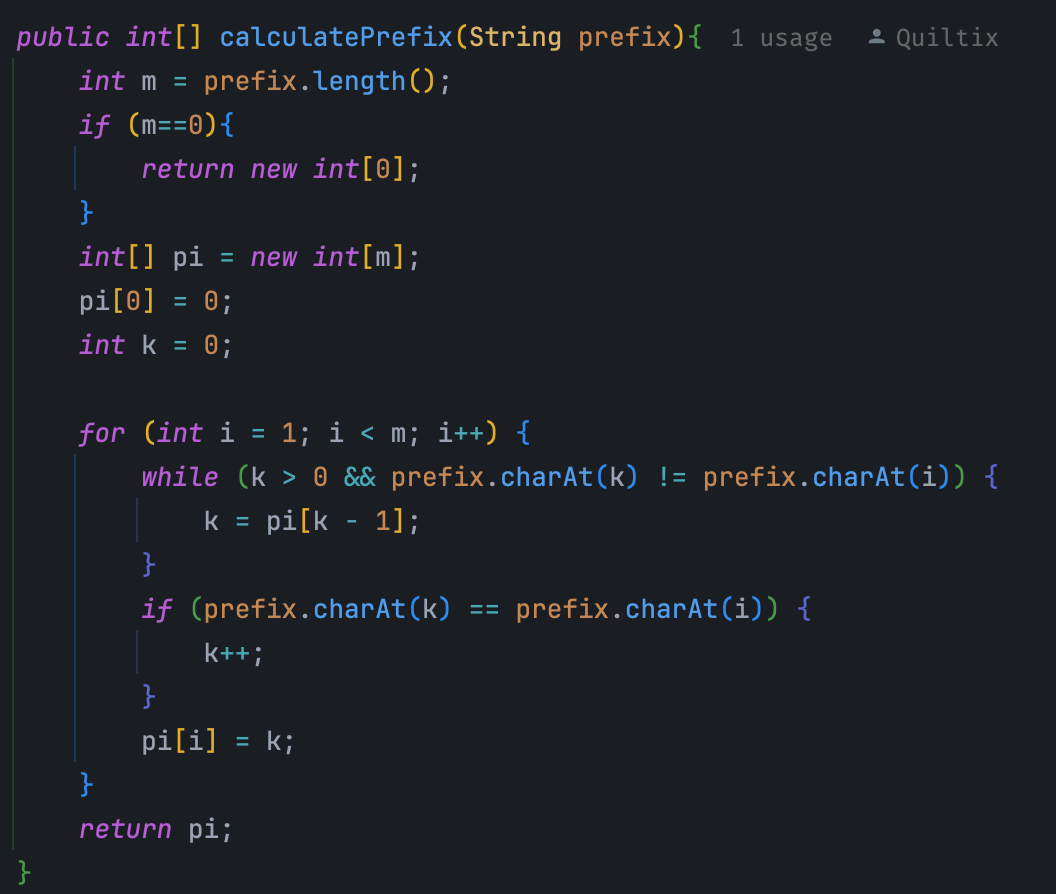


Рисунок 2 – Нахождение префикс функции на языке программирования Java

Следующим этапом является поиск всех вхождений подстроки в строку с использованием префикс-функции

Шаги выполнения алгоритма:

1. Предобработка, которая включает вызов метода для расчета pi
2. Инициализация необходимых переменных и обработка крайних случаев
3. Прохождение по элементам текста и их сравнение с элементами подстроки
   1. В случае совпадения элементов, сдвигаем указатели и снова сравниваем, если таким образом все элементы подстроки совпадают с подряд идущими элементами строки, то записываем индекс начала и сдвигаем указатель подстроки на число, которое записано в предыдущей ячейки префикс-функции (j=p[j-1])
   2. Если элементы не совпадают, то оптимально сдвигаем указатель подстроки на на число, которое записано в предыдущей ячейки префикс-функции (j=p[j-1])
   3. Если совпадений не будет найдено, результатом работы метода станет пустой список

Благодаря такому алгоритму можно избежать избыточных сравнений.

С реализацией данного алгоритма можно ознакомиться на Рисунке 3



Рисунок 3 – Поиск всех вхождений подстроки в строку с использованием префикс-функции на языке программирования Java

Таким образом асимптотическая сложность данного алгоритма - O(n + m). Префикс-функция строится за O(m), поиск выполняется — за O(n). Пространственная сложность – O(m) для хранения массива pi.

КМП гарантирует линейное время выполнения, что делает его идеальным для задач с большими текстами, таких как антиплагиат. Префикс-функция минимизирует повторные сравнения, используя информацию о структуре шаблона.

В результате разбора двух алгоритмов, можно сказать, что Алгоритм КМП превосходит наивный подход за счет использования префикс-функции, что исключает избыточные сравнения. Его асимптотическая сложность O(n + m) делает его предпочтительным для поиска точных совпадений фрагментов текста. Префикс-функция позволяет эффективно пропускать уже проверенные символы, минимизируя количество сравнений. Наивный алгоритм может проще в реализации, но его скорость работы, зачастую, делает его неэффективным для больших текстов и шаблонов из-за большого количества повторений.

# РЕАЛИЗАЦИЯ ИССЛЕДУЕМОГО АЛГОРИТМА

### Программная реализация

Реализуем алгоритм КМП на Java и C++

Реализация алгоритма КМП на Java состоит из двух методов. Метод calculatePrefix вычисляет префикс-функцию, которая определяет, насколько можно сместить шаблон при несоответствии. Метод calculateKMP использует эту функцию для поиска всех вхождений шаблона в тексте, возвращая список индексов. Код эффективен и подходит для обработки больших текстов, например, в системах антиплагиата.

Вход: Строка prefix — шаблон, для которого вычисляется префикс-функция.

Выход: Массив целых чисел pi, где pi[i] — длина самого длинного префикса-суффикса для подстроки prefix[0..i].

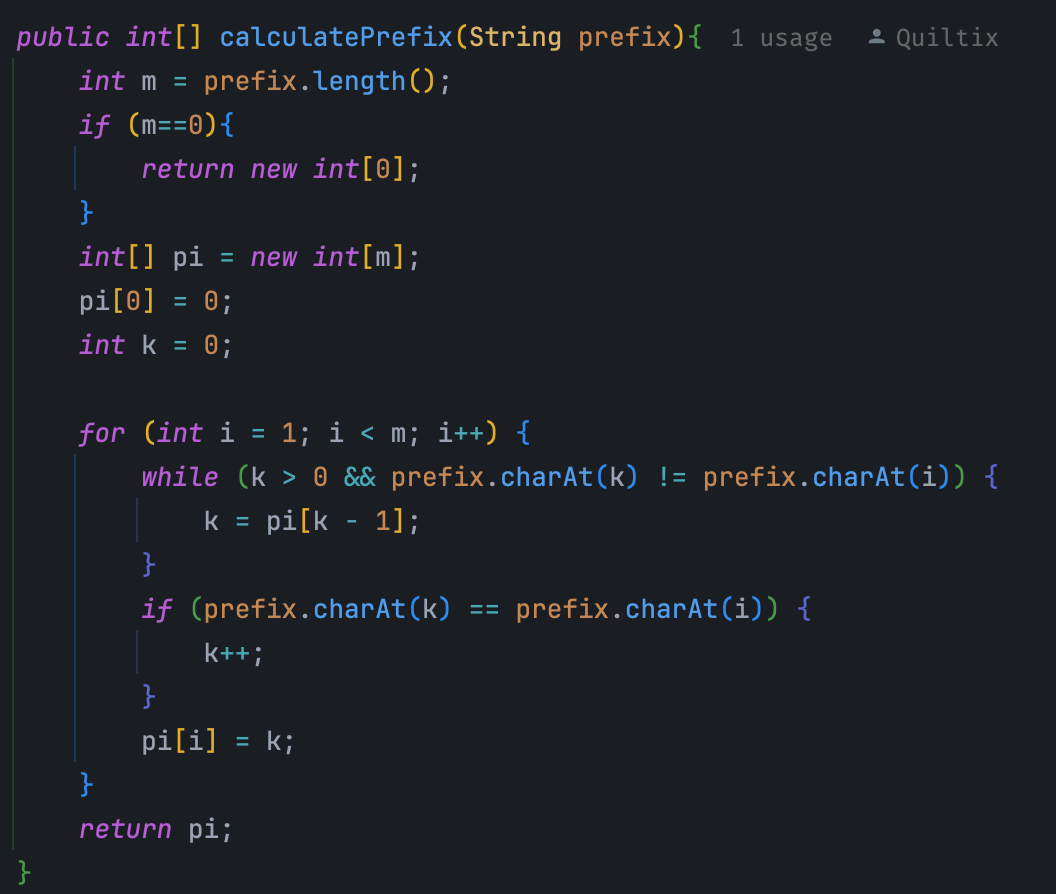


Рисунок 4 –Реализация нахождения префикс-функции на языке программирования Java

Метод calculatePrefix (Рисунок 4) вычисляет префикс-функцию (массив pi)для заданного шаблона. Префикс-функция определяет длину самого длинного собственного префикса подстроки prefix[0..i], который одновременно является суффиксом этой подстроки. Эта информация позволяет алгоритму КМП эффективно смещать шаблон при несоответствии, избегая ненужных сравнений.

Логика работы:

1. **Обработка краевого случая:** если длина шаблона равна нулю (m = 0), метод возвращает пустой массив, так как префикс-функция не имеет смысла для пустой строки.
2. **Инициализация:** создается массив pi размером m, устанавливается pi[0]=0, поскольку для первого символа нет собственных префиксов и вводится переменная k, инициализируемая нулем, которая отслеживает длину текущего префикса-суффикса.
3. **Основной цикл:** Если символы prefix[k] и prefix[i] не совпадают и k > 0, уменьшаем k до значения pi[k-1]. Это позволяет "откатиться" к предыдущему возможному префиксу-суффиксу. Если символы совпадают, увеличиваем k. bЗаписываем pi[i] = k, фиксируя длину префикса-суффикса для текущей позиции.
4. После обработки всех позиций возвращается массив pi.

Метод calculateKMP(Рисунок 5) выполняет поиск всех вхождений шаблона в тексте, используя префикс-функцию для оптимизации процесса. Он возвращает список индексов, с которых начинаются совпадения шаблона в тексте.

Вход: Строки text (текст для поиска) и substring (шаблон).

Выход: Список List<Integer>, содержащий индексы начала каждого вхождения шаблона в тексте.



Рисунок 5 – Реализация метода calculateKMP на языке программирования Java

**Логика работы:**

1. **Предварительная обработка:**
   * Вычисляется префикс-функция для шаблона с помощью метода calculatePrefix, результат сохраняется в массиве p.
   * Инициализируется пустой список result для хранения индексов вхождений.
   * Вводятся указатели:
     + i — текущая позиция в тексте (начинает с 0).
     + j — текущая позиция в шаблоне (начинает с 0).
   * Проверяются краевые случаи:
     + Если длина текста n меньше длины шаблона m, возвращается пустой список.
     + Если текст или шаблон пусты, возвращается пустой список.
2. **Основной цикл поиска:**  
   Пока i < n:
   * Если символы text[i] и substring[j] совпадают:
     + Увеличиваются оба указателя
     + Если j достигает длины шаблона m, найдено вхождение:
       - В список result добавляется индекс начала вхождения i - m Указатель j сбрасывается на основе префикс-функции: j=p[j−1], чтобы продолжить поиск следующего вхождения.
   * Если символы не совпадают:
     + Если j>0, смещаем j на основе префикс-функции: j=p[j−1].
     + Если j=0, увеличиваем i, чтобы перейти к следующему символу текста.
   * Если i достигает, цикл завершается.
3. **Возврат результата:**  
   Возвращается список result, содержащий все индексы вхождений.

Реализация на C++ состоит из двух функций, аналогичных Java-версии: calculatePrefix вычисляет префикс-функцию для шаблона, а calculateKMP выполняет поиск подстроки в тексте.

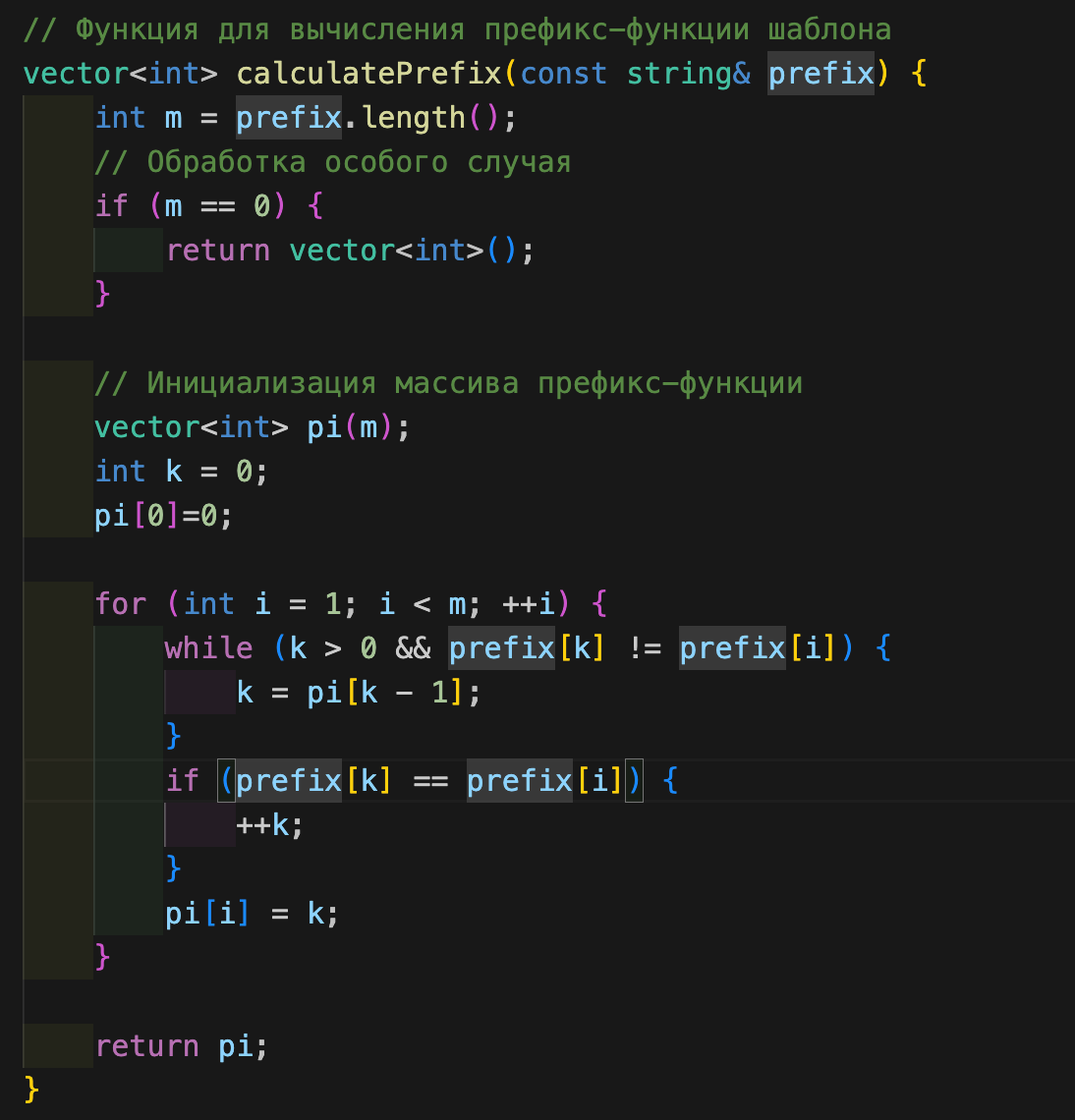


Рисунок 6 –Реализация нахождения префикс-функции на языке программирования C++

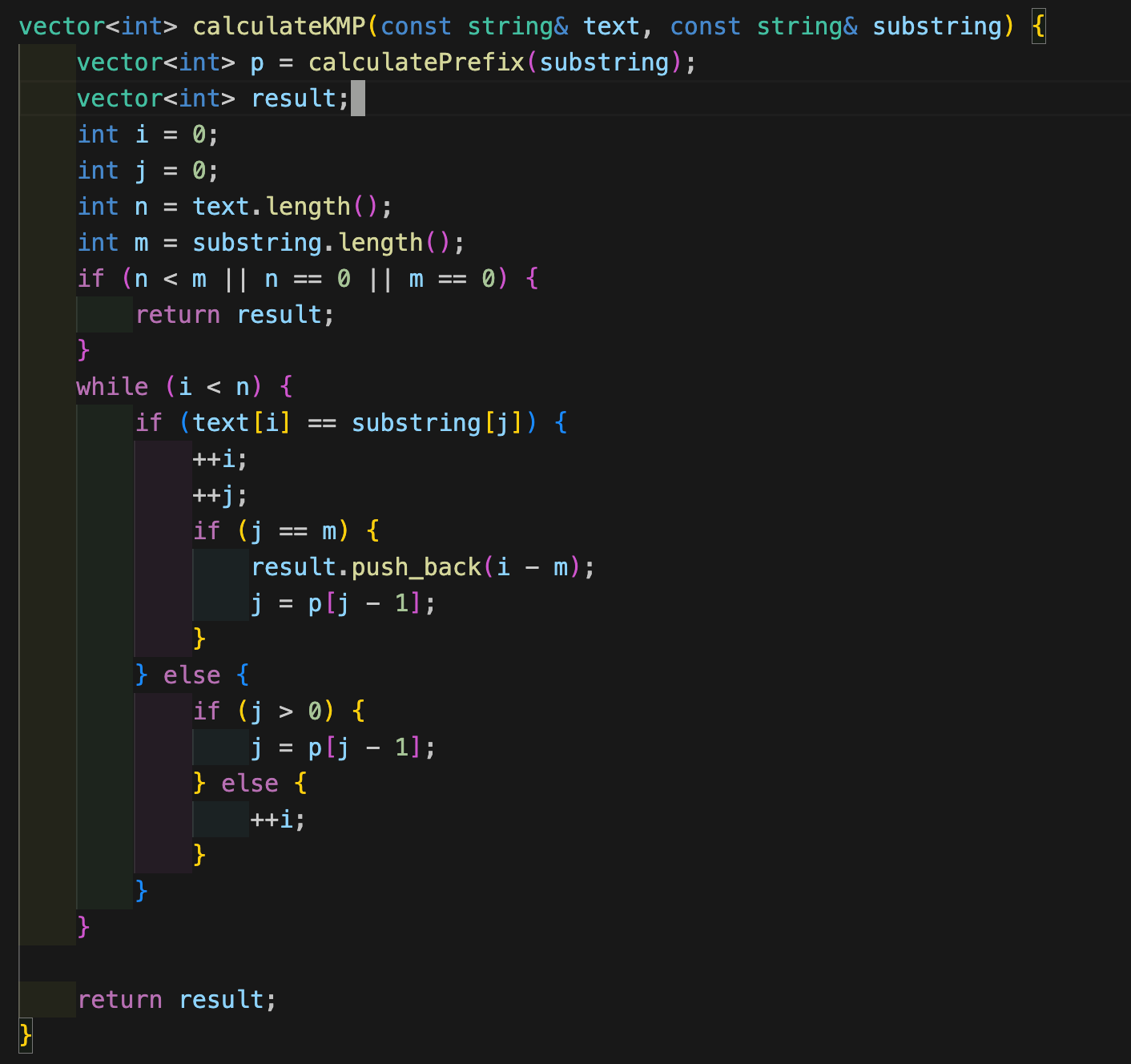


Рисунок 6 – Реализация метода calculateKMP на языке программирования C++

Как можно заметить, реализация алгоритма КМП на C++ аналогична, написанному на Java решению, за исключением использования в C++ такой структуры данных как vector.

### Unit тестирование написанного кода

Юнит-тесты проверяют функциональность методов calculatePrefix и calculateKMP класса AlgorithmService. Тесты разделены на категории: позитивные (корректные входные данные с ожидаемыми результатами), негативные (некорректные или отсутствующие данные) и граничные (краевые случаи, такие как пустые строки или строки минимальной длины). Каждый тест является независимым методом, что упрощает диагностику ошибок и соответствует лучшим практикам тестирования. Тесты используют библиотеку JUnit 5 и основаны на написанном Java коде.

С написанными тестами можно ознакомиться на Рисунках 7-9, результат на Рисунке 10.

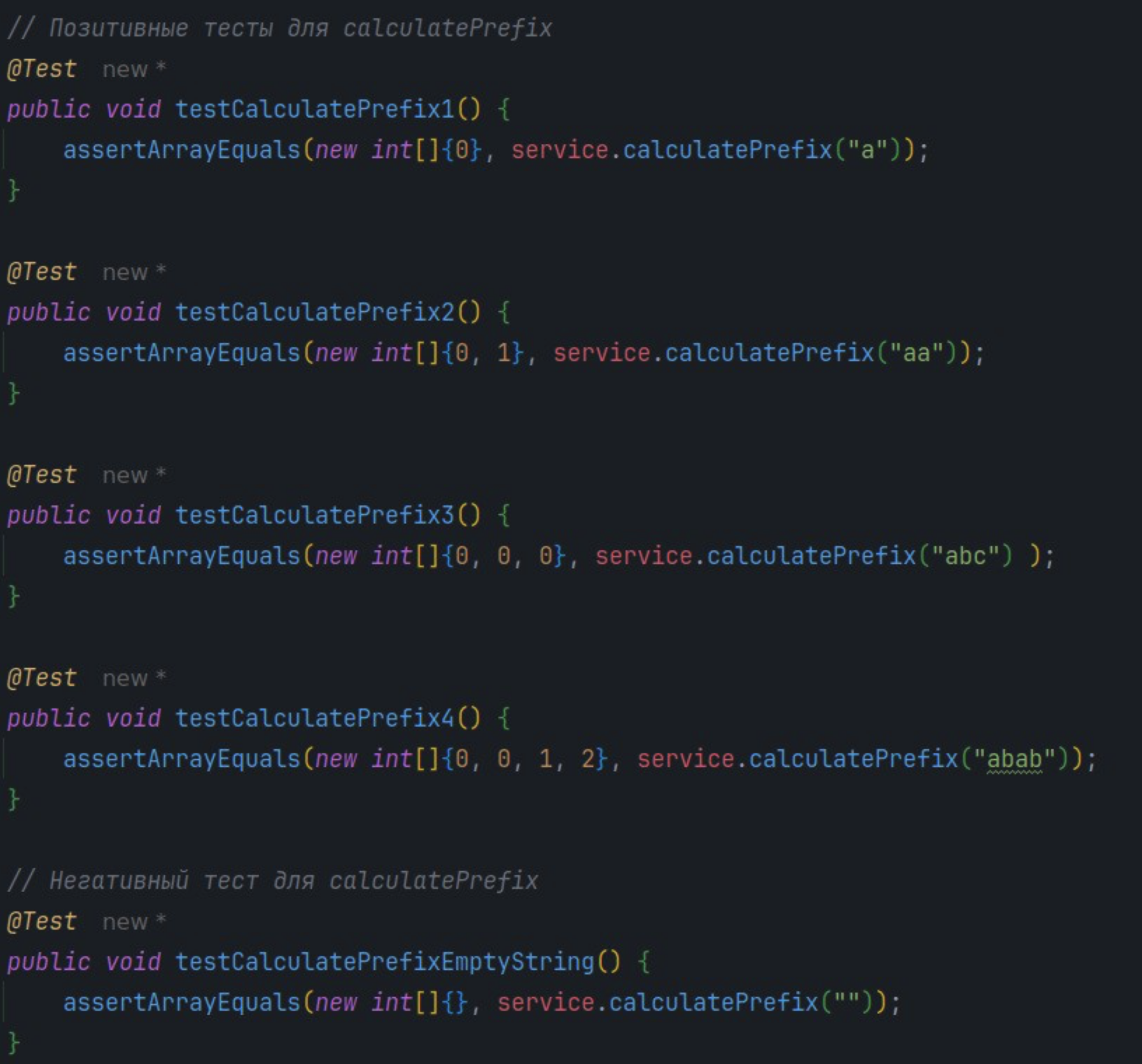


Рисунок 7 – Реализация позитивных и негативных тестов для расчета префикса на Java.



Рисунок 8 – Реализация позитивных тестов для алгоритма КМП на Java



Рисунок 9 – Реализация негативных и граничных тестов для алгоритма КМП на Java.

Юнит-тесты для AlgorithmService обеспечивают полное покрытие функциональности алгоритма КМП, проверяя методы calculatePrefix и calculateKMP на различных сценариях. Позитивные тесты подтверждают корректность работы, негативные — устойчивость к ошибкам, а граничные — обработку краевых случаев. Тесты гарантируют надежность алгоритма для реальных задач, где точность и производительность критичны.



Рисунок 10 – Результат работы тестов

Юнит-тесты для AlgorithmService обеспечивают полное покрытие функциональности алгоритма КМП, проверяя методы calculatePrefix и calculateKMP на различных сценариях. Позитивные тесты подтверждают корректность работы, негативные — устойчивость к ошибкам, а граничные — обработку краевых случаев. Тесты гарантируют надежность алгоритма для реальных задач, где точность и производительность критичны.

# ИНТЕГРАЦИЯ В ВЕБ-ИНТЕРФЕЙС

### Постановка задачи прикладного применения исследуемого алгоритма

Создать веб-сервис для проверки текстов на наличие заимствований (антиплагиат) с использованием алгоритма Кнута-Морриса-Пратта (КМП). Сервис должен позволять пользователям загружать текст и шаблон. Это моделирует реальную задачу антиплагиата, где требуется найти совпадения фрагментов текста с эталонными источниками.

**Описание задачи:**

* **Входные данные:** Пользователь предоставляет текст (например, статью или эссе) и шаблон (фрагмент, который нужно найти).
* **Обработка:** Алгоритм КМП используется для поиска всех вхождений шаблона в тексте. Результаты (индексы совпадений) возвращаются пользователю.
* **Выходные данные:** Список индексов, где шаблон встречается в тексте, с возможностью визуализации совпадений (например, выделение найденных фрагментов).
* **Применение:** Сервис может использоваться в образовательных учреждениях для проверки студенческих работ на плагиат или в системах управления контентом для выявления заимствований.

### Выбор технологического стека

Для реализации веб-сервиса выбран следующий технологический стек, соответствующий предоставленной информации (Spring, Hibernate, Spring Security, HTML/CSS/JS):

* **Бэкенд:**
  + **Spring Boot (Java):** Фреймворк для создания REST API, упрощающий настройку и развертывание. Spring Boot обеспечивает встроенный сервер (Tomcat), управление зависимостями и поддержку REST-контроллеров.
  + **Hibernate (JPA):** ORM-фреймворк для работы с базой данных. Используется для хранения шаблонов и пользовательских данных (логины, роли).
  + **Spring Security:** Обеспечивает авторизацию и аутентификацию пользователей. Используется для ограничения доступа к API только зарегистрированным пользователям.
  + **PostgreSQL:** Мощная база данных для разработки и тестирования обеспечивает расширяемость.
  + **Maven:** Система сборки для управления зависимостями и компиляции проекта.
* **Фронтенд:**
  + **HTML5:** Основа структуры веб-страниц.
  + **CSS3:** Стилизация интерфейса, включая адаптивный дизайн для удобства использования на разных устройствах.
  + **JavaScript (с Fetch API):** Обработка пользовательских действий и асинхронных запросов к бэкенду. Используется для динамического обновления страницы без перезагрузки.
* **Дополнительно:**
  + **REST API:** Для взаимодействия между фронтендом и бэкендом. API принимает текст и шаблон, возвращает результаты поиска.
  + **JSON:** Формат обмена данными между фронтендом и бэкендом.

### Реализация фронт и бэк частей

Бэкенд реализован на Spring Webt с использованием Hibernate, Spring Security и JWT-аутентификации. Он предоставляет REST API для выполнения поиска подстроки, сохранения результатов и получения истории поисков. Основные компоненты:

* **Модель данных:**
  + User: Сущность для хранения данных пользователей (идентификатор, имя пользователя, пароль).
  + Algorithm: Сущность для хранения поисковых запросов (текст, шаблон, результаты, владелец).
* **Сервисы:**
  + AlgorithmService: Содержит логику КМП (calculatePrefix, calculateKMP), выполняет поиск, сохраняет результаты и предоставляет историю поисков.
  + UserService: Управляет регистрацией и аутентификацией пользователей, генерирует JWT-токены.
* **Контроллеры:**
  + AlgorithmController: Обрабатывает запросы на поиск и получение истории.
  + AuthController: Обрабатывает регистрацию и вход пользователей.
* **Репозитории:**
  + UserRepository: Для работы с пользователями.
  + AlgorithmRepository: Для работы с поисковыми запросами.
* **Безопасность:** Spring Security с JWT защищает эндпоинты, требуя токен в заголовке Authorization.

**API-эндпоинты:**

* **POST /api/auth/register**: Регистрация пользователя (принимает RegisterRequestDTO).
* **POST /api/auth/login**: Аутентификация, возвращает JWT-токен (LoginRequestDTO).
* **POST /api/algorithm**: Выполняет поиск КМП, сохраняет результат, возвращает индексы
* **GET /api/algorithm**: Возвращает историю поисков текущего пользователя.
* **GET /api/algorithm/all**: Возвращает все поиски в системе

Фронтенд реализован на HTML, CSS и JavaScript. Он предоставляет интерфейс для регистрации, входа, ввода текста и шаблона, выполнения поиска и просмотра истории.

HTML и JavaScript включают:

* Форму ввода текста и шаблона.
* Кнопки для регистрации и входа (через prompt).
* Секции для отображения истории пользователя и общей истории.
* JavaScript-логику для отправки запросов к бэкенду и обработки ответов.