**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: **Ахо-Корасик**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Гребнев Е.Д, |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

**Цель работы.**

Изучить принцип работы алгоритма Кнута-Морриса\_Пратта. Написать функцию, вычисляющую для каждого элемента строки максимальное значение длины префикса и с помощью данной функции решить поставленные задачи. А именно написать программу, осуществляющую поиск вхождений подстроки в строку, а также программу, определяющую, являются ли строки циклическим сдвигом друг друга, найти индекс начала вхождения второй строки в первую.

**Задание №1.**

Разработайте программу,  решающую задачу точного поиска набора образцов.  
  
**Вход:**  
Первая строка содержит текст (T,1≤∣T∣≤100000*T*,1≤∣*T*∣≤100000 ).  
Вторая - число n*n* (1≤n≤30001≤*n*≤3000), каждая следующая из n*n* строк содержит шаблон из набора P={p1,…,pn}1≤∣pi∣≤75*P*={*p*1​,…,*pn*​}1≤∣*pi*​∣≤75  
Все строки содержат символы из алфавита {A,C,G,T,N}{*A*,*C*,*G*,*T*,*N*}  
**Выход:**  
Все вхождения образцов из P*P* в T*T*.  
Каждое вхождение образца в текст представить в виде двух чисел - i *i*  p*p*  
Где i*i* - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p*p*  
(нумерация образцов начинается с 1).  
Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

**Sample Input:**

NTAG

3

TAGT

TAG

T

**Sample Output:**

2 2

2 3

**Задание №2.**

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*.  
  
В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P*P* необходимо найти все вхождения РР в текст ТТ.  
  
Например, образец аb??с?а*b*??с? с джокером ?? встречается дважды в тексте *xabvccbababcax*.  
  
Символ джокер не входит в алфавит, символы которого используются в T*T*. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида ??? недопустимы.  
Все строки содержат символы из алфавита {A,C,G,T,N}{*A*,*C*,*G*,*T*,*N*}  
  
**Вход:**  
Текст (T,1≤∣T∣≤100000*T*,1≤∣*T*∣≤100000 )  
Шаблон (P,1≤∣P∣≤40*P*,1≤∣*P*∣≤40)  
Символ джокера  
**Выход:**  
Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).  
Номера должны выводиться в порядке возрастания.

**Sample Input:**

ACTANCA

A$$A$

$

**Sample Output:**

1

**Вариант 1**. На месте джокера может быть любой символ, за исключением заданного.

**Описание алгоритмов.**

**Описание алгоритма Ахо-Корасик.**

Алгоритм создает префиксное дерево из букв искомых подстрок. Затем в полученном дереве ищутся суффиксные ссылки. Суффиксная ссылка вершины u – это вершина v, такая что строка v является максимальным суффиксом строки u. Для корня и вершин, исходящих из корня, суффиксной ссылкой является корень. Для остальных вершин осуществляется переход по суффиксной ссылке родителя и, если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет. Далее создаются терминальные ссылки – такие суффиксные ссылки, которые ведут в вершину, которая является терминальной.

Текст, в котором нужно найти подстроки побуквенно передается в автомат. Начиная из корня, автомат переходит по ребру, соответствующему переданному символу. Если нужного ребра нет, переходит по ссылке. Если встреченная вершина является терминальной, значит была встречена подстрока. Если найдено совпадение нужно пройти по терминальным ссылкам, если они не None, чтобы вывести все шаблоны заканчивающиеся на этом месте. Номер подстроки (подстрок) хранится в поле *terminate* вершины. В ответ сохранятся индекс, на котором началась эта подстрока в тексте и сам номер подстроки.

алгоритма для нахождения шаблонов с маской.

**Описание модифицированного алгоритма.**

Алгоритм тот же, но в качестве подстрок берутся кусочки шаблона, разделенные джокером, запоминаются позиции полученных подстрок в исходном шаблоне. Создается массив С длины s, где s – длина текста, где ищется шаблон. При нахождении подстроки, в массиве С увеличивается на единицу число по индексу, соответствующему возможному началу шаблона. Индекс высчитывается по формуле: текущий индекс - (длина найденной подстроки - 1) - (позиция подстроки в шаблоне -1). Затем проходим по полученному массиву, каждый i для которого С[i] = количеству подстрок, является вероятным началом шаблона. В соответствии с индивидуализацией, для каждого найденного шаблона проверяются буквы, стоящие на месте джокера. Если не было встречено запрещенного символа, найденный шаблон добавляется в ответ.

#### 1. ****Класс****Node

Представляет узел в префиксном дереве (trie). Содержит:

* parent: ссылка на родительский узел.
* children: словарь дочерних узлов (ключ — символ, значение — узел).
* suffix\_link: суффиксная ссылка для оптимизации переходов при несовпадении символа.
* terminal\_link: ссылка на ближайший терминальный узел.
* terminate: идентификатор шаблона, если узел терминальный.
* name: символьное имя узла (для отладки).
* Логирование состояния через verbose.

#### 2. ****Класс****AhoCorasickAutomaton

Реализует автомат для поиска множества шаблонов в тексте. Основные этапы:

* Инициализация дерева (\_build\_automaton).
* Построение trie (build\_trie).
* Создание суффиксных (build\_suffix\_links) и терминальных ссылок (build\_terminal\_links).
* Метод search\_patterns для поиска вхождений в текст.

#### 3. ****Метод**** build\_trie

Строит trie из переданных шаблонов. Для каждого символа шаблона:

* Создает дочерние узлы, если их нет.
* Помечает терминальные узлы идентификаторами шаблонов.
* Логирует процесс при verbose=True.

#### 4. ****Метод****\_build\_suffix\_links

Создает суффиксные ссылки через обход в ширину (BFS):

* Для корневых детей ссылки ведут в корень.
* Для остальных узлов: переход по суффиксным ссылкам родителя до нахождения совпадения символа.

#### 5. ****Метод****build\_terminal\_links

Создает терминальные ссылки через BFS:

* Для каждого узла проверяет суффиксные ссылки до первого терминального узла.
* Устанавливает terminal\_link на найденный терминальный узел.

#### 6. ****Метод****search\_patterns

Алгоритм поиска:

* Посимвольный обход текста с переходами по суффиксным ссылкам при несовпадении.
* Проверка терминальных узлов через terminal\_link для обнаружения вхождений.
* Возвращает позиции найденных шаблонов в формате [start\_pos, pattern\_id].

#### 7. ****Функция****read\_input

Читает шаблоны из ввода, присваивая уникальные идентификаторы.  
Формат ввода:

* Первая строка — число шаблонов n.
* Следующие n строк — шаблоны.

#### 8. ****Функция****main

Координирует процесс:

* Чтение текста и шаблонов.
* Построение автомата.
* Запуск поиска и вывод результатов.

### Описание расширенной версии с джокерами (task2.py)

#### 1. ****Класс****TrieNode

Расширенная версия узла:

* depth: глубина узла в дереве (используется для определения позиции в тексте).
* terminate: список смещений для шаблонов, заканчивающихся в этом узле.
* Остальные поля аналогичны Node из task1.

#### 2. ****Класс****Trie

Инкапсулирует логику построения trie:

* add\_pattern: добавляет шаблон в дерево с указанием смещений.
* Логирует создание узлов при verbose=True.

#### 3. ****Класс****AhoCorasick

Расширенный автомат для работы с джокерами:

* Принимает шаблоны с указанием смещений (например, части шаблона между джокерами).
* Методы \_build\_suffix\_links и \_build\_terminal\_links аналогичны task1, но с поддержкой depth.

#### 4. ****Метод****search

Поиск с учетом джокеров:

* Подсчитывает количество совпадений подшаблонов для каждой позиции текста.
* Проверяет, чтобы совпали все части исходного шаблона (игнорируя джокеры).
* Возвращает позиции, где исходный шаблон полностью совпадает с текстом.

#### 5. ****Функция****get\_pattern\_parts

Разделяет шаблон с джокерами на подшаблоны:

* Например, шаблон ab\*c?d разделяется на ab, c с соответствующими смещениями.
* Возвращает словарь вида {подшаблон: [список\_смещений]}.

#### 6. ****Функция****main

Логика для задачи с джокерами:

* Ввод текста, шаблона с джокерами и символа-джокера.
* Разделение шаблона на части.
* Построение автомата и поиск.
* Вывод позиций, где шаблон полностью совпадает с текстом.

**task1.py (Базовый алгоритм Ахо-Корасик)**

**Временная сложность:**

1. **Построение trie:**
   * Зависит от суммы длин всех шаблонов: **O(Σ|Pᵢ|)**, где |Pᵢ| — длина i-го шаблона.
2. **Суффиксные ссылки:**
   * Обработка всех узлов через BFS: **O(Σ|Pᵢ|)**.
3. **Терминальные ссылки:**
   * Обход узлов и переходы по суффиксным ссылкам: **O(Σ|Pᵢ|)**.
4. **Поиск в тексте:**
   * Линейный проход по тексту: **O(|T|)**, где |T| — длина текста.
   * Сбор результатов: **O(Z)**, где Z — количество найденных вхождений.

**Итог по времени:**  
**O(Σ|Pᵢ| + |T| + Z)**.

**Пространственная сложность:**

* Хранение trie: **O(Σ|Pᵢ|)**.
* Дополнительные структуры для ссылок: **O(Σ|Pᵢ|)**.

**Итог по памяти:**  
**O(Σ|Pᵢ|)**.

**task2.py (Расширение с джокерами)**

**Временная сложность:**

1. **Разделение шаблона на подшаблоны (**get\_pattern\_parts**):**
   * Линейный проход по исходному шаблону: **O(L)**, где L — длина шаблона с джокерами.
2. **Построение trie:**
   * Сумма длин всех подшаблонов: **O(Σ|Sⱼ|)**, где |Sⱼ| — длина j-го подшаблона.
3. **Суффиксные и терминальные ссылки:**
   * Аналогично task1.py: **O(Σ|Sⱼ|)**.
4. **Поиск в тексте (**search**):**
   * Линейный проход текста: **O(|T|)**.
   * Проверка совпадений всех подшаблонов: **O(|T| · L)** (из-за цикла по длине исходного шаблона).
   * Сбор результатов: **O(Z')**, где Z' — количество совпадений.

**Итог по времени:**  
**O(Σ|Sⱼ| + |T| · L + Z')**.

**Пространственная сложность:**

* Хранение trie: **O(Σ|Sⱼ|)**.
* Массив для подсчёта совпадений: **O(|T|)**.

**Итог по памяти:**  
**O(Σ|Sⱼ| + |T|)**.

**Тестирование.**

|  |  |  |
| --- | --- | --- |
| Входные данные | Ответ | Комментарий |
| NTAG  3  TAGT  TAG  T | 2 2  2 3 | Верно |
| ACCGTACA  2  AC  GT | 1 1  4 2  6 1 | Верно |
| ACGT  3  ACGT  CG  GT | 1 1  2 2  3 3 | Верно |

Таблица 1 – Тестирование алгоритма Ахо-Корасик

|  |  |  |
| --- | --- | --- |
| Входные данные | Ответ | Комментарий |
| ACTANCA  A$$A$  $  G | 1 | Верно |
| ACACAA  ACXA  X | 3 | Верно |
| ACGANGAAAT  A$G  $ | 1  4 | Верно |

Таблица 1 – Тестирование алгоритма поиска с джокером

Результат работы программы с отладочным выводом для первого задания (см. рис 1, 2, 3).

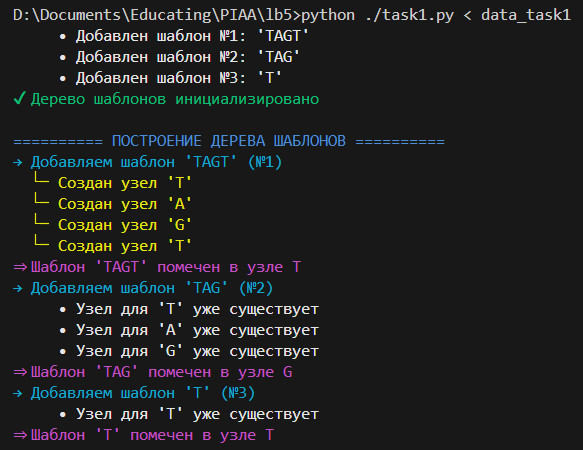


Рисунок 1

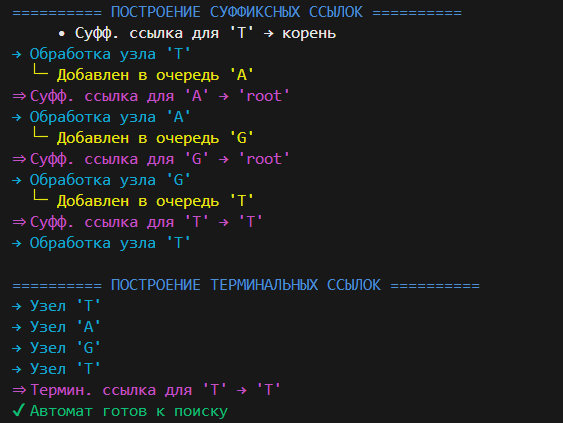


Рисунок 2

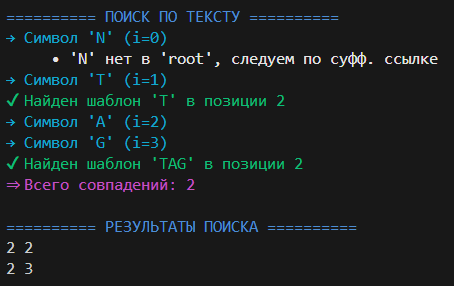


Рисунок 3

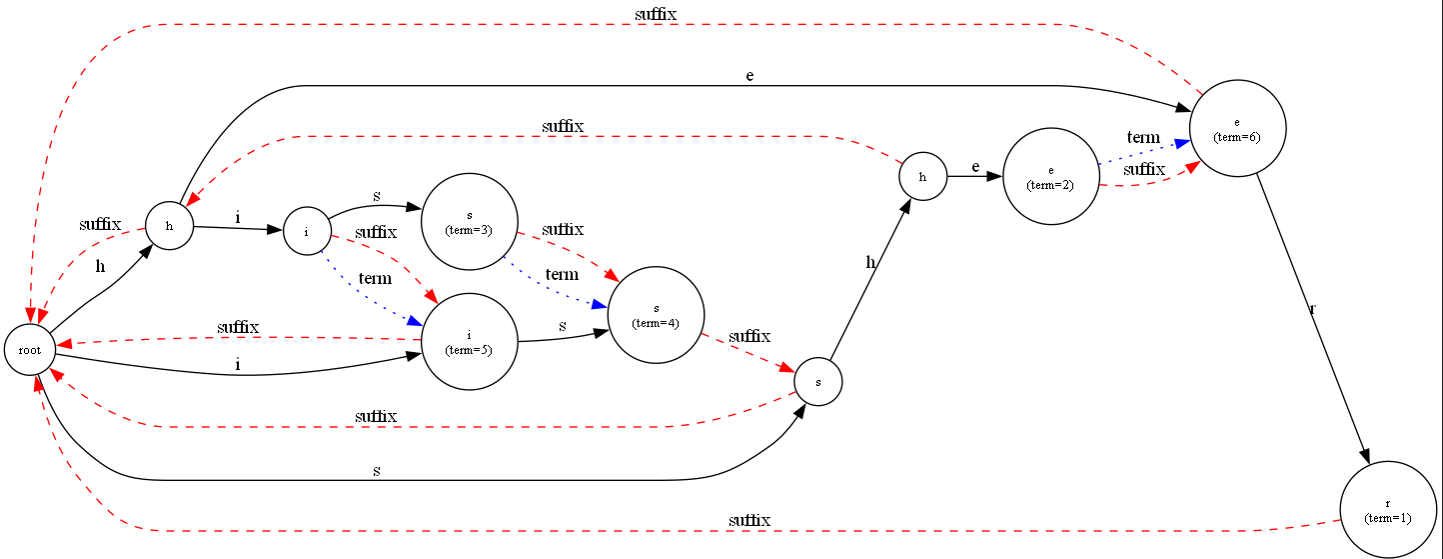
****

Рис. 4 **–** Визуализация для {'her':1, 'she':2, 'his':3, 'is':4, 'i':5, 'he':6}

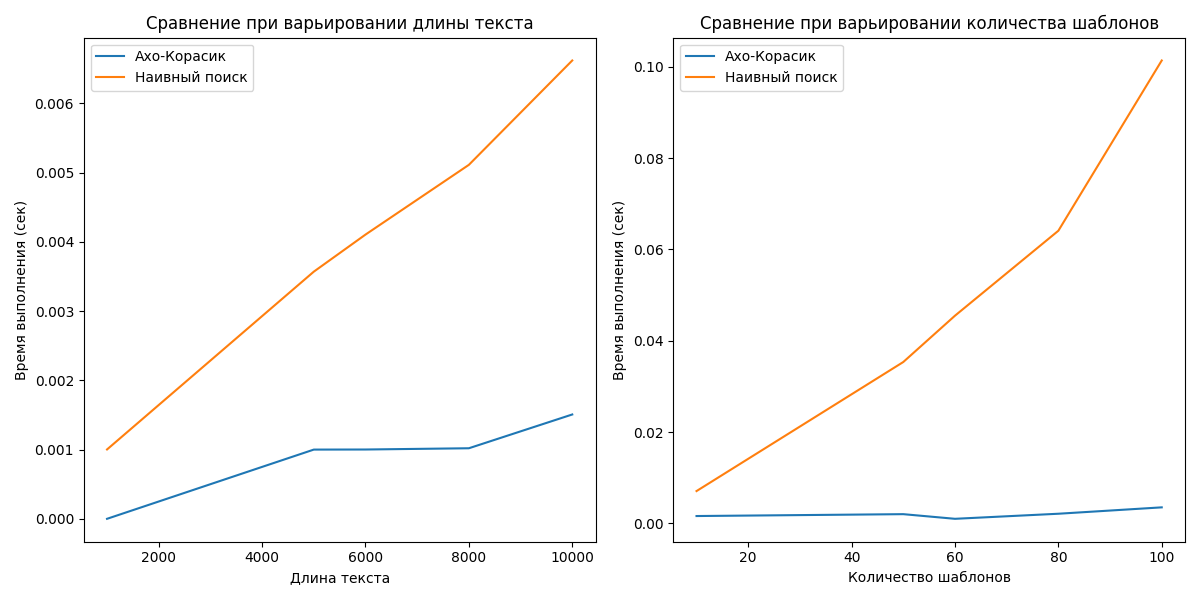
**Исследование.**

Рисунок 8 – Сравнение алгоритма Ахо-Корасик и наивного поиска

Можно сделать вывод, что Ахо-Корасик выполняется значительно быстрее, чем наивный алгоритм.

**Выводы.**

Изучен принцип работы алгоритма Ахо-Корасик. Написаны программы, корректно решающие задачу поиска набора подстрок в строке, в также программа поиска подстроки с джокером.

# **ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

**task1.py**

from operator import itemgetter

from collections import deque

from utils import \*

class Node:

    def \_\_init\_\_(self, link=None, name="root", verbose=False):

        self.parent = None

        self.children = {}

        self.suffix\_link = link

        self.terminal\_link = None

        self.terminate = 0  # Номер шаблона, если узел терминальный

        self.name = name    # Имя узла (символ или "root")

        self.verbose = verbose

    def \_\_str\_\_(self):

        parent\_name = self.parent.name if self.parent else None

        suffix\_name = self.suffix\_link.name if self.suffix\_link else None

        children\_keys = list(self.children.keys()) if self.children else None

        return (

            f"Node(name={self.name}, "

            f"parent={parent\_name}, "

            f"children={children\_keys}, "

            f"suffix\_link={suffix\_name}, "

            f"terminate={self.terminate})"

        )

class AhoCorasickAutomaton:

    def \_\_init\_\_(self, patterns: dict, verbose=False):

        self.verbose = verbose

        if self.verbose:

            log\_success("Дерево шаблонов инициализировано")

        self.root = Node(verbose=verbose)

        self.patterns = patterns

        self.patterns\_inverse = {v: k for k, v in patterns.items()}

        self.\_build\_automaton()

        if self.verbose:

            log\_success("Автомат готов к поиску")

    def \_build\_automaton(self):

        """Построение автомата Ахо-Корасик в три этапа"""

        self.\_build\_trie()

        self.\_build\_suffix\_links()

        self.\_build\_terminal\_links()

    def \_build\_trie(self):

        """Построение исходного бора"""

        if self.verbose:

            log\_section("Построение дерева шаблонов")

        for pattern, pattern\_id in self.patterns.items():

            if self.verbose:

                log\_substep(f"Добавляем шаблон '{pattern}' (№{pattern\_id})")

            current\_node = self.root

            for symbol in pattern:

                if symbol not in current\_node.children:

                    new\_node = Node(name=symbol, link=self.root, verbose=self.verbose)

                    new\_node.parent = current\_node

                    current\_node.children[symbol] = new\_node

                    if self.verbose:

                        log\_action(f"Создан узел '{symbol}'")

                    current\_node = new\_node

                else:

                    current\_node = current\_node.children[symbol]

                    if self.verbose:

                        log\_info(f"Узел для '{symbol}' уже существует")

            current\_node.terminate = pattern\_id

            if self.verbose:

                log\_result(f"Шаблон '{pattern}' помечен в узле {current\_node.name}")

    def \_build\_suffix\_links(self):

        """Построение суффиксных ссылок в ширину (BFS)"""

        if self.verbose:

            log\_section("Построение суффиксных ссылок")

        queue = deque(self.root.children.values())

        # Для детей корня суффиксная ссылка ведет в корень

        for child in queue:

            child.suffix\_link = self.root

            if self.verbose:

                log\_info(f"Суфф. ссылка для '{child.name}' → корень")

        while queue:

            current\_node = queue.popleft()

            if self.verbose:

                log\_substep(f"Обработка узла '{current\_node.name}'")

            for child in current\_node.children.values():

                queue.append(child)

                if self.verbose:

                    log\_action(f"Добавлен в очередь '{child.name}'")

                self.\_set\_suffix\_link\_for\_child(child)

                if self.verbose:

                    log\_result(f"Суфф. ссылка для '{child.name}' → '{child.suffix\_link.name}'")

    def \_set\_suffix\_link\_for\_child(self, node):

        """Установка суффиксной ссылки для конкретного узла"""

        link = node.parent.suffix\_link

        while link and (node.name not in link.children):

            link = link.suffix\_link

        if link:

            node.suffix\_link = link.children.get(node.name, self.root)

        else:

            node.suffix\_link = self.root

    def \_build\_terminal\_links(self):

        """Построение терминальных ссылок в ширину (BFS)"""

        if self.verbose:

            log\_section("Построение терминальных ссылок")

        queue = deque(self.root.children.values())

        while queue:

            current\_node = queue.popleft()

            if self.verbose:

                log\_substep(f"Узел '{current\_node.name}'")

            # Добавляем детей текущего узла в очередь

            queue.extend(current\_node.children.values())

            # Ищем ближайшую терминальную вершину по суффиксным ссылкам

            temp = current\_node.suffix\_link

            while temp != self.root:

                if temp.terminate:

                    current\_node.terminal\_link = temp

                    if self.verbose:

                        log\_result(f"Термин. ссылка для '{current\_node.name}' → '{temp.name}'")

                    break

                temp = temp.suffix\_link

    def search\_patterns(self, text: str) -> list[str]:

        """Поиск всех вхождений шаблонов в тексте"""

        if self.verbose:

            log\_section("Поиск по тексту")

        results = []

        current\_node = self.root

        for position, symbol in enumerate(text):

            if self.verbose:

                log\_substep(f"Символ '{symbol}' (i={position})")

            # Переходим по суффиксным ссылкам, пока не найдем подходящего ребенка

            while current\_node and symbol not in current\_node.children:

                if self.verbose:

                    log\_info(f"'{symbol}' нет в '{current\_node.name}', следуем по суфф. ссылке")

                current\_node = current\_node.suffix\_link

            if current\_node:

                current\_node = current\_node.children.get(symbol, self.root)

                temp\_node = current\_node

                # Проверяем терминальные узлы по терминальным ссылкам

                while temp\_node:

                    if temp\_node.terminate:

                        pattern = self.patterns\_inverse[temp\_node.terminate]

                        start\_pos = position - len(pattern) + 2  # +2 для 1-based индексации

                        results.append([start\_pos, temp\_node.terminate])

                        if self.verbose:

                            log\_success(f"Найден шаблон '{pattern}' в позиции {start\_pos}")

                    temp\_node = temp\_node.terminal\_link

            else:

                current\_node = self.root

        # Сортируем результаты и форматируем вывод

        results.sort(key=itemgetter(0, 1))

        formatted = [' '.join(map(str, item)) for item in results]

        if self.verbose:

            log\_result(f"Всего совпадений: {len(formatted)}")

        return formatted

def read\_input(verbose=False):

    """Чтение входных данных"""

    n = int(input())

    patterns = {}

    for pattern\_id in range(1, n+1):

        pattern = input().strip()

        patterns[pattern] = pattern\_id

        if verbose:

            log\_info(f"Добавлен шаблон №{pattern\_id}: '{pattern}'")

    return patterns

def main():

    verbose = True

    text = input().strip()

    patterns = read\_input(verbose)

    automaton = AhoCorasickAutomaton(patterns, verbose)

    results = automaton.search\_patterns(text)

    if results:

        if verbose:

            log\_section("Результаты поиска")

        print('\n'.join(results))

    else:

        log\_warning("Совпадения не найдены")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**task2.py**

from collections import deque

from utils import \*

# === Структуры ===

class TrieNode:

    def \_\_init\_\_(self, parent=None, name="root", verbose=False):

        self.parent = parent

        self.children = {}

        self.suffix\_link = None

        self.terminal\_link = None

        self.terminate = []

        self.name = name

        self.depth = parent.depth + 1 if parent else 0

        self.verbose = verbose

    def \_\_repr\_\_(self):

        return f"TrieNode(name='{self.name}', depth={self.depth}, terminate={self.terminate})"

class Trie:

    def \_\_init\_\_(self, verbose=False):

        self.root = TrieNode(verbose=verbose)

        self.verbose = verbose

        if self.verbose:

            log\_success("Дерево шаблонов инициализировано")

    def add\_pattern(self, pattern, offsets):

        if self.verbose:

            log\_substep(f"Добавление шаблона: '{pattern}' с оффсетами: {offsets}")

        node = self.root

        for symbol in pattern:

            if symbol not in node.children:

                if self.verbose:

                    log\_action(f"Создание нового узла для символа: '{symbol}'")

                node.children[symbol] = TrieNode(parent=node, name=symbol, verbose=self.verbose)

            else:

                if self.verbose:

                    log\_action(f"Используется существующий узел для символа: '{symbol}'")

            node = node.children[symbol]

        node.terminate = offsets

        if self.verbose:

            log\_result(f"Шаблон '{pattern}' успешно добавлен. Терминальный узел: {node}")

class AhoCorasick:

    def \_\_init\_\_(self, patterns, verbose=False):

        self.verbose = verbose

        if self.verbose:

            log\_section("Инициализация автомата Ахо-Корасика")

        self.trie = Trie(verbose=verbose)

        self.patterns = patterns

        self.\_build\_trie()

        self.\_build\_suffix\_links()

        self.\_build\_terminal\_links()

        if self.verbose:

            log\_success("Построение автомата завершено")

    def \_build\_trie(self):

        if self.verbose:

            log\_section("Построение дерева шаблонов")

        for pattern, offsets in self.patterns.items():

            self.trie.add\_pattern(pattern, offsets)

    def \_build\_suffix\_links(self):

        if self.verbose:

            log\_section("Построение суффиксных ссылок")

        root = self.trie.root

        queue = deque(root.children.values())

        for child in queue:

            child.suffix\_link = root

            if self.verbose:

                log\_info(f"Суффиксная ссылка для '{child.name}' установлена на корень")

        while queue:

            current = queue.popleft()

            if self.verbose:

                log\_substep(f"Обработка узла: '{current.name}' (глубина: {current.depth})")

            for symbol, child in current.children.items():

                queue.append(child)

                if self.verbose:

                    log\_action(f"Потомок '{child.name}' добавлен в очередь")

                link = current.suffix\_link

                while link is not None and symbol not in link.children:

                    if self.verbose:

                        log\_info(f"Символ '{symbol}' не найден в '{link.name}', переход по ссылке")

                    link = link.suffix\_link

                if link and symbol in link.children:

                    child.suffix\_link = link.children[symbol]

                    if self.verbose:

                        log\_result(f"Суффиксная ссылка для '{child.name}' установлена на '{child.suffix\_link.name}'")

                else:

                    child.suffix\_link = root

                    if self.verbose:

                        log\_result(f"Суффиксная ссылка для '{child.name}' установлена на корень")

    def \_build\_terminal\_links(self):

        if self.verbose:

            log\_section("Построение терминальных ссылок")

        root = self.trie.root

        queue = deque(root.children.values())

        while queue:

            current = queue.popleft()

            if self.verbose:

                log\_substep(f"Обработка узла: '{current.name}'")

            queue.extend(current.children.values())

            temp = current.suffix\_link

            while temp and temp != root:

                if temp.terminate:

                    current.terminal\_link = temp

                    if self.verbose:

                        log\_result(f"Терминальная ссылка для '{current.name}' установлена на '{temp.name}'")

                    break

                temp = temp.suffix\_link

    def search(self, text, pattern\_input, joker):

        if self.verbose:

            log\_section("Поиск по тексту")

        result = [0] \* len(text)

        node = self.trie.root

        for index, symbol in enumerate(text):

            if self.verbose:

                log\_substep(f"Символ '{symbol}' на позиции {index + 1}")

            while node and symbol not in node.children:

                node = node.suffix\_link

            if node:

                node = node.children.get(symbol, self.trie.root)

            else:

                node = self.trie.root

            temp = node

            while temp:

                if temp.terminate:

                    matched = text[index - temp.depth + 1: index + 1]

                    for offset in self.patterns[matched]:

                        pos = index - temp.depth - offset + 1

                        if pos >= 0:

                            result[pos] += 1

                temp = temp.terminal\_link

        output = []

        total\_patterns = sum(len(v) for v in self.patterns.values())

        for i in range(len(result) - len(pattern\_input) + 1):

            if result[i] == total\_patterns:

                valid = True

                for j in range(len(pattern\_input)):

                    if pattern\_input[j] == joker and text[i + j] == joker:

                        valid = False

                        break

                if valid:

                    output.append(str(i + 1))

        return output

def get\_pattern\_parts(pattern, joker, verbose=False):

    if verbose:

        log\_section("Разделение шаблона по джокеру")

    parts = {}

    last\_j = -1

    for i, char in enumerate(pattern):

        if char == joker:

            if last\_j < i - 1:

                sub = pattern[last\_j + 1: i]

                parts.setdefault(sub, []).append(last\_j + 1)

                if verbose:

                    log\_action(f"Найден подшаблон: '{sub}' (оффсет: {last\_j + 1})")

            last\_j = i

    if last\_j != len(pattern) - 1:

        sub = pattern[last\_j + 1:]

        parts.setdefault(sub, []).append(last\_j + 1)

        if verbose:

            log\_action(f"Найден подшаблон: '{sub}' (оффсет: {last\_j + 1})")

    if verbose:

        log\_success(f"Итоговые подшаблоны: {parts}")

    return parts

def main():

    verbose = False

    text = input().strip()

    pattern\_input = input().strip()

    joker = input().strip()

    patterns = get\_pattern\_parts(pattern\_input, joker, verbose)

    if not patterns:

        log\_warning("Не найдено ни одного подшаблона")

        return

    automaton = AhoCorasick(patterns, verbose)

    result = automaton.search(text, pattern\_input, joker)

    if result:

        if verbose:

            log\_section("Результаты")

        print('\n'.join(result))

    else:

        log\_warning("Шаблон не найден в тексте")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**visualization.py**

from graphviz import Digraph

class Visualizer:

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

self.automaton = automaton

self.graph = Digraph(format='png')

self.graph.attr('node', shape='circle', fontsize='10')

self.graph.attr(rankdir='LR')

def \_add\_nodes(self, node, visited):

# Label includes name and terminate flag

label = f"{node.name}"

if node.terminate:

label += f"\\n(term={node.terminate})"

self.graph.node(str(id(node)), label)

visited.add(node)

for child in node.children.values():

if child not in visited:

self.\_add\_nodes(child, visited)

def \_add\_trie\_edges(self, node, visited):

visited.add(node)

for symbol, child in node.children.items():

self.graph.edge(str(id(node)), str(id(child)), label=symbol)

if child not in visited:

self.\_add\_trie\_edges(child, visited)

def \_add\_suffix\_edges(self, node, visited):

visited.add(node)

if node is not self.automaton.root:

target = node.suffix\_link

self.graph.edge(

str(id(node)), str(id(target)),

label='suffix', style='dashed', color='red'

)

for child in node.children.values():

if child not in visited:

self.\_add\_suffix\_edges(child, visited)

def \_add\_terminal\_edges(self, node, visited):

visited.add(node)

if node.terminal\_link:

target = node.terminal\_link

self.graph.edge(

str(id(node)), str(id(target)),

label='term', style='dotted', color='blue'

)

for child in node.children.values():

if child not in visited:

self.\_add\_terminal\_edges(child, visited)

def render(self, filename='aho\_automaton'):

# Build graph

self.\_add\_nodes(self.automaton.root, set())

self.\_add\_trie\_edges(self.automaton.root, set())

self.\_add\_suffix\_edges(self.automaton.root, set())

self.\_add\_terminal\_edges(self.automaton.root, set())

# Render to file

self.graph.render(filename, view=True)

from task1 import AhoCorasickAutomaton

patterns = {'her':1, 'she':2, 'his':3, 'is':4, 'i':5, 'he':6}

automaton = AhoCorasickAutomaton(patterns, verbose=False)

vis = Visualizer(automaton)

vis.render('aho\_visualization')