МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студентка гр. 3343 Гельман П.Е.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Цель данной лабораторной работы состоит в изучении поиска вхождений всех образцов в текст с помощью алгоритма Ахо-Корасик, включая случаи с джокерами.

Задание.

№1

Первая строка содержит текст (T, $1 \leq |T| \leq 100000$).

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p_1, \ldots, p_n\} 1 \leq |p_i| \leq 75$,

Все строки содержат символы из алфавита {A, C, G, T, N}

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p.

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером 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

№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 необходимо найти все вхождения P в текст T.

Например, образец ab??c? с джокером ?? встречается дважды в тексте xabvccbababcax.

Символ джокер не входит в алфавит, символы которого используются в 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}

Вход:

Текст (T,$1 \leq |T| \leq 100000$)

Шаблон (P,$1 \leq |P| \leq 40$)

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

Задание варианта:

Вариант 2. Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Примечания для варианта:

1) Для обоих заданий на программирование должны быть версии кода с выводом промежуточных данных. В них, в частности, должны выводиться построение бора и автомата, построенный автомат (в виде, например, описания каждой вершины автомата), процесс его использования.

2) В автомате должны быть и использоваться не только суффиксные ссылки, но и конечные ссылки.

Выполнение работы.

Задача алгоритма:

Дан набор строк в алфавите размера k суммарной длины m. Необходимо найти для каждой строки все ее вхождения в текст.

Бор (англ. trie, луч, нагруженное дерево) — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах.

Реализованные части программы:

1. def num(c) -> int – функция, предназначенная для перевода символа строки в номер буквы алфавита по заданному соотношению
2. class Vertex – класс вершины бора:
    a. Поле id – идентификатор вершины
    b. Поле next – ссылки на следующую вершину, массив переходов по символам
    c. Поле is_terminal – флаг, отвечающий за терминальность вершины
    d. Поле pattern_indices – массив, который хранит номера шаблонов, заканчивающихся в текущей вершине
    e. Поле parent – ссылка на родителя
    f. Поле pchar – символ, по которому перешли в вершину
    g. Поле sufflink – суффиксная ссылка на другую вершину
    h. Поле go – массив переходов по суффиксным ссылкам
    i. Поле uplink – сжатая суффиксная ссылка

Для этого класса созданы сеттеры и геттеры.

3. Class Trie – класс самого бора.
    a. Поле alpha – размер алфавита, поле vertices – массив вершин бора, поле root – корень бора
    b. def size(self) -> int – возвращает количество вершин в боре

c. def last(self) -> Vertex – возвращает последнюю добавленную вершину
d. def add(self, s, index) -> None – метод добавления вершины в бор. В методе мы проходим по каждому символу шаблона, получаем номер буквы алфавита из символа, если перехода с этой буквой и вершины с этим номером не существует, то создается новая вершина, добавляется в vertices, для текущей вершины устанавливает созданная в качестве next на позицию idx, далее переходим к следующей вершине по символу, на котором стоим (если переход существовал, то просто переходим по нему). Последняя вершина, к которой пришли, отмечается терминальной (шаблон в ней закончился), а также добавляется номер законченного шаблона.
e. def get_link(self, v) -> Vertex – метод получения суффиксной ссылки для вершины. Суффиксная ссылка — это ссылка на узел, соответствующий самому длинному суффиксу, который не заводит бор в тупик. Если суффиксная ссылка еще не вычислена, то сначала проверяем ее на root. В случае, если вершина – корень, или ее родитель – корень, то суффиксная ссылка ведет в root аналогично, иначе мы получаем суффиксную ссылку родителя, переходим по символу перехода текущей вершины из вершины, на которую ссылается суффиксная ссылка родителя, устанавливаем найденную ссылку. Если ссылка была вычислена, просто ее возвращаем.
f. def get_uplink(self, v) -> Vertex – нахождение сжатой суффиксной ссылки (терминальной, конечной). Сжатая суффиксная ссылка – ссылка на ближайшую терминальную вершину в цепочке суффиксных ссылок. При поиске мы

можем сразу перейти к терминальной вершине, не проходя всю цепочку суффиксных ссылок. Это ускоряет поиск, особенно если шаблонов много. Если сжатая ссылка не найдена: получаем обычную ссылку, проверяем, не ссылается ли она на корень, если ссылается, то сжатой не существует, если вершина, на которую она ссылается терминальная, то сжатая = обычная, иначе устанавливаем рекурсивно найденную сжатую ссылку по принципу других условий.

g. def go(self, v, c) -> Vertex – метод для перехода по символу c из вершины v. Сначала преобразовываем символ в индекс, если переход еще не найден, проверяем, что его возможно совершить. Если возможно, то переходим к вершине next по символу, иначе если нельзя перейти по символу, и вершина корень, то переход сам в себя, иначе рекурсивно ищем переход по символу через суффиксные ссылки.
h. def search(self, text, patterns) – метод поиска паттернов в тексте. Осуществляется проход по каждому символу текста, если вершина с помощью перехода по текущему символу существует, то для каждого шаблона, заканчивающегося в этой вершине, вычисляем начало совпадения шаблона с текущим имеющимся набором символов и переходим по сжатой суффиксной ссылке к следующей вершине.
i. def search_wildcard(self, text, pattern_infos, pat_len) – поиск вхождений с джокером. Логика схожа с обычным поиском, но здесь учитывается смещение подстроки внутри шаблона и длина подстроки при вычислении старта вхождения. Если мы видим, что шаблон поместился, то увеличиваем счетчик для позиции старта.
j. def print_auto(self) – метод, который печатает автомат.

4. def search_with_wildcard(text, pattern, wildcard) – функция, которая предварительно разбивает шаблон на подстроки, чтобы найти вхождения с джокерами. В случае, если количество совпадений подстрок, начиная с i, равно количеству подстрок, на которые был разбит шаблон, то в тексте найдено совпадение.
5. def find_overlapping_patterns(matches, patterns) – поиск перекрытий шаблонов в тексте. С помощью уже вычисленных вхождений паттернов в текст и их индексов вычисляется старт и конец того или иного шаблона. Далее сравниваются все возможные комбинации шаблонов между собой, если начало одного меньше, чем конец другого, и конец первого больше старта второго, то имеем перекрытие.

Сложность алгоритма.

Так как мы храним таблицу переходов автомата как индексный массив, то расход памяти O(nσ), вычислительная сложность O(nσ+H+k), где H – длина текста, в котором производится поиск, n – общая длина всех слов в словаре, σ – размер алфавита, k – общая длина всех совпадений. O(H+k) получается из-за прохода по всему тексту. O(nσ) получается из-за создания автомата.

В случае с джокерами: расход по памяти такой же, вычислительная сложность O(nσ+H+k*A), A – количество подстрок в паттерне (без джокеров), n – суммарная длина всех подстрок в паттерне, H – длина текста, k – количество всех совпадений

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ATGCGA	Результаты:	Верно

	2 AT GC	позиция в тексте - 1, номер паттерна - 1 позиция в тексте - 3, номер паттерна - 2	
2.	AAAAA 3 AA AAA A	Результаты: позиция в тексте - 1, номер паттерна - 1 позиция в тексте - 1, номер паттерна - 2 позиция в тексте - 1, номер паттерна - 3 позиция в тексте - 2, номер паттерна - 1 позиция в тексте - 2, номер паттерна - 2 позиция в тексте - 2, номер паттерна - 3 позиция в тексте - 3, номер паттерна - 1 позиция в тексте - 3, номер паттерна - 2 позиция в тексте - 3, номер паттерна - 3 позиция в тексте - 4, номер паттерна - 1 позиция в тексте - 4, номер паттерна - 3 позиция в тексте - 5, номер паттерна - 3	Верно
3.	ACGTNACGT A*T *	-	Верно
4.	ACTANCA A$$A$ $	Результаты: Позиция совпадения шаблона в тексте : 1	Верно
5.	ACTGTAC XT X	Результаты: Позиция совпадения шаблона в тексте : 2 Позиция совпадения шаблона в тексте : 4	Верно
6.	CATCATG _ _	Результаты: Позиция совпадения шаблона в тексте : 1 Позиция совпадения шаблона в тексте : 2 Позиция совпадения шаблона в тексте : 3 Позиция совпадения шаблона в тексте : 4 Позиция совпадения шаблона в тексте : 5	Верно

		Позиция совпадения шаблона в тексте : 6 Позиция совпадения шаблона в тексте : 7 Позиция совпадения шаблона в тексте : 8	

Выводы.

В ходе лабораторной работы был реализован алгоритм Ахо-Корасик, проанализирована его временная сложность и сложность по памяти.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
def num(c) -> int:
    # перевод символа строки в номер буквы алфавита
    return {'A': 0, 'C': 1, 'G': 2, 'T': 3, 'N': 4}[c]


class Vertex:
    def __init__(self, id, alpha, parent, pchar) -> None:
        self._id = id  # идентификатор вершины
        self._next = [None] * alpha  # массив переходов по символам
        self._is_terminal = False  # флаг, является ли вершина терминальной
        self._pattern_indices = []  # номера шаблонов, заканчивающихся в
этой вершине
        self._parent = parent  # родитель текущей вершины
        self._pchar = pchar  # символ, по которому был осуществлен переход
от родителя к этой вершине
        self._sufflink = None  # суффиксная ссылка на другую вершину
(ссылка на наибольший собственный суффикс)
        self._go = [None] * alpha  # массив переходов по суффиксным ссылкам
        self._uplink = None  # сжатая суфф ссылка

    def get_next(self) -> list:  # получение массива переходов по символам
        return self._next

    def set_next(self, index, vertex) -> None:  # заполнение массива
переходов по символам
        self._next[index] = vertex

    def is_terminal(self) -> bool:  # проверка на терминальность
        return self._is_terminal

    def set_terminal(self, value: bool) -> None:  # установка
терминальности вершины
        self._is_terminal = value

    def get_pattern_indices(self) -> list:  # получение массива номеров
шаблонов, заканчивающихся в этой вершине
        return self._pattern_indices

    def add_pattern_index(self, index) -> None:  # добавление номера
шаблона, который заканчивается в этой вершине
        self._pattern_indices.append(index)

    def get_parent(self):  # получение родительской вершины
        return self._parent

    def set_parent(self, parent) -> None:  # установка родительской вершины
        self._parent = parent

    def get_pchar(self) -> str:  # получение символа, по которому перешли в
вершину
        return self._pchar

    def get_sufflink(self):  # получение суффиксной ссылки
        return self._sufflink

    def set_sufflink(self, link) -> None:  # установка суффиксной ссылки
        self._sufflink = link
```

```
    def get_go(self) -> list:  # получение массива переходов по ссылкам
        return self._go

    def set_go(self, index, vertex) -> None:  # установка очередного
перехода по ссылке
        self._go[index] = vertex

    def get_uplink(self):  # получение сжатой суффиксной ссылки
        return self._uplink

    def set_uplink(self, link) -> None:  # установка сжатой суффиксной
ссылки
        self._uplink = link


class Trie:
    def __init__(self, alpha=5) -> None:
        self.alpha = alpha  # размер алфавита, по умолчанию 5 (исходя из
условий задания)
        self.vertices = [Vertex(0, alpha, None, None)]  # вершины бора
        self.root = self.vertices[0]  # корень бора

    def size(self) -> int:  # размер бора (количество вершин)
        return len(self.vertices)

    def last(self) -> Vertex:  # последняя вершина
        return self.vertices[-1]

    def add(self, s, index) -> None:  # добавление вершины в бор
        print(f"\nДобавление шаблона '{s}' с индексом {index}")
        v = self.root
        for c in s:
            idx = num(c)
            print(f"Шаг {i + 1}: символ '{c}' (индекс {idx})")
            if v.get_next()[idx] is None:
                new_vertex = Vertex(self.size(), self.alpha, v, c)  #
создаем объект Vertex
                self.vertices.append(new_vertex)
                print(f"Создана новая вершина (id={new_vertex._id}) от
вершины {v._id} по символу '{c}'")
                v.set_next(idx,
                           self.last())  # устанавливаем для текущей
вершины новый объект в качестве следующей вершины
            else:
                print(f"Переход из вершины {v._id} в вершину
{v.get_next()[idx]._id} по символу '{c}'")
            v = v.get_next()[idx]  # переходим к следующей вершине далее
        v.set_terminal(True)  # последняя вершина, до которой дошли -
терминальная
        v.add_pattern_index(index)  # добавляем номер шаблона, который
закончился сейчас
        print(f"Вершина {v._id} помечена как терминальная для шаблона '{s}'
(индекс {index})")

    def get_link(self, v) -> Vertex:  # получение суффиксной ссылки для
вершины
        print(f"\n[ОБЫЧНАЯ] Вычисление суффиксной ссылки для вершины
{v._id}")
        if v.get_sufflink() is None:  # если суффиксная ссылка еще не
вычислена
            if v == self.root or v.get_parent() == self.root:
                v.set_sufflink(self.root)  # ссылаемся на рут, если вершина
или ее родитель - рут
```

```
                print(f"[ОБЫЧНАЯ] Суффиксная ссылка вершины {v._id}
установлена на корень (id=0)")
            else:
                print(f"[ОБЫЧНАЯ] Рекурсивный вызов для родителя вершины
{v._id} (вершина {v.get_parent()._id})")
                parent_link = self.get_link(v.get_parent())  # ссылка
родителя
                print(f"[ОБЫЧНАЯ] Переход из вершины {parent_link._id} по
символу '{v.get_pchar()}'")
                linked = self.go(parent_link,
                                 v.get_pchar())  # переходим по символу
перехода текущей вершины из суффиксной ссылки родителя
                v.set_sufflink(linked)  # устанавливаем найденную ссылку
                print(f"[ОБЫЧНАЯ] Суффиксная ссылка вершины {v._id}
установлена на вершину {linked._id}")
        else:
            print(f"[ОБЫЧНАЯ] Суффиксная ссылка вершины {v._id} уже
вычислена: ведет к вершине {v.get_sufflink()._id}")
        return v.get_sufflink()  # возвращаем вычисленную суффиксную ссылку

    def get_uplink(self, v) -> Vertex:  # получение сжатой суффиксной
ссылки
        print(f"\n[СЖАТАЯ] Вычисление сжатой суффиксной ссылки для вершины
{v._id}")
        if v.get_uplink() is None:  # если она еще не найдена
            slink = self.get_link(v)  # получаем обычную ссылку
            if slink == self.root:  # если она рут, то сжатой не существует
                v.set_uplink(None)
                print(f"[СЖАТАЯ] Сжатая суффиксная ссылка вершины {v._id}
не существует (ведет на корень)")
            elif slink.is_terminal():  #если она терминальная, то сжатая -
обычная суффиксная ссылка
                v.set_uplink(slink)
                print(
                    f"[СЖАТАЯ] Сжатая суффиксная ссылка вершины {v._id}
установлена на терминальную вершину {slink._id}")
            else:
                uplink = self.get_uplink(slink)  # иначе устанавливаем
рекурсивно найденную сжатую ссылку
                v.set_uplink(uplink)
                if uplink:
                    print(f"[СЖАТАЯ] Сжатая суффиксная ссылка вершины
{v._id} установлена на вершину {uplink._id}")
                else:
                    print(f"[СЖАТАЯ] Сжатая суффиксная ссылка вершины
{v._id} не существует")
        else:
            if v.get_uplink():
                print(
                    f"[СЖАТАЯ] Сжатая суффиксная ссылка вершины {v._id} уже
вычислена: ведет к вершине {v.get_uplink()._id}")
            else:
                print(f"[СЖАТАЯ] Сжатая суффиксная ссылка вершины {v._id}
не существует")
        return v.get_uplink()

    def go(self, v, c) -> Vertex:  # функция для перехода по символу c из
вершины v
        idx = num(c)  # преобразуем символ c в индекс, используя функцию
num
        print(f"\nПопытка перехода из вершины {v._id} по символу '{c}'
(индекс {idx})")
        if v.get_go()[idx] is None:
```

```
            if v.get_next()[idx] is not None:  #проверяем, что такой
переход возможен
                v.set_go(idx, v.get_next()[idx])  # переходим к следующей
вершине
                print(f"Прямой переход из вершины {v._id} в вершину
{v.get_next()[idx]._id} по символу '{c}'")
            elif v == self.root:  # если вершина корень
                v.set_go(idx, self.root)  # то сам в себя
                print(f"Переход из корня по символу '{c}' ведет обратно в
корень")
            else:
                print(f"Рекурсивный вызов для суффиксной ссылки вершины
{v._id}")
                # иначе рекурсивно ищем переход через суффиксную ссылку
                linked = self.go(self.get_link(v), c)
                v.set_go(idx, linked)
                print(
                    f"Переход из вершины {v._id} по символу '{c}' ведет в
вершину {linked._id} (через суффиксную ссылку)")
        else:
            print(f"Переход из вершины {v._id} по символу '{c}' уже
вычислен: ведет в вершину {v.get_go()[idx]._id}")
        return v.get_go()[idx]

    def search(self, text, patterns):
        print(f"\nНачало поиска в тексте '{text}'")
        v = self.root
        result = []
        for j in range(len(text)):  # проходим по символам текста
            c = text[j]
            print(f"\nПозиция {j + 1}: символ '{c}'")
            v = self.go(v, c)  # текущая вершина
            print(f"Текущая вершина: {v._id}")
            check = v
            while check is not None:  # если вершина существует
                for pattern_idx in check.get_pattern_indices():  # для
каждого номера шаблона
                    start_pos = j - len(patterns[pattern_idx - 1]) + 2  #
вычисляем начало совпадения шаблона
                    print(f"Найден шаблон '{patterns[pattern_idx - 1]}'
(индекс {pattern_idx}) на позиции {start_pos}")
                    result.append((start_pos, pattern_idx))
                check = self.get_uplink(check)  # переход по сжатой ссылке
далее
                if check:
                    print(f"Переход по сжатой суффиксной ссылке в вершину
{check._id}")
        return result

    def search_wildcard(self, text, pattern_infos, pat_len):
        v = self.root
        print(f"\nНачало поиска с джокерами в тексте '{text}'")
        counts = [0] * (len(text) + 1)  # counts[i] хранит, сколько
подстрок из шаблона совпало, если полное
        # совпадение начнётся с позиции i
        for j in range(len(text)):  # идем по тексту
            v = self.go(v, text[j])  # текущая вершина
            print(f"\nПозиция {j + 1}: символ '{text[j]}'")
            print(f"Текущая вершина: {v._id}")
            check = v
            while check is not None:
                for pattern_idx in check.get_pattern_indices():
                    offset, length = pattern_infos[pattern_idx - 1]  #
offset - смещение подстроки внутри шаблона,
```

```
                    # length - длина подстроки
                    start_pos = j - length + 1 - offset  # j - конец
найденной подстроки в тексте
                    print(f"Найдена подстрока шаблона (индекс
{pattern_idx}), offset={offset}, length={length}")
                    if 0 <= start_pos <= len(text) - pat_len:  # если
шаблон поместился, увеличиваем счетчик
                        print("Шаблон полностью поместился")
                        counts[start_pos] += 1
                        print(f"Увеличиваем счетчик для позиции
{start_pos}")
                    else:
                        print("Шаблон не поместился, счетчик не
увеличивается")
                check = self.get_uplink(check)  # прыжок на ближайшую
терминальную вершину

        return counts

    def print_auto(self):
        print("\nСтруктура автомата:")
        print(
            "Вершина (id) -> символ: следующая вершина, суффиксная ссылка,
сжатая ссылка, терминальность, конец шаблона")
        for vertex in self.vertices:
            transitions = []
            for idx in range(self.alpha):
                if vertex.get_next()[idx] is not None:
                    char = ['A', 'C', 'G', 'T', 'N'][idx]

transitions.append(f"{char}:{vertex.get_next()[idx]._id}")

            sufflink = vertex.get_sufflink()._id if vertex.get_sufflink()
is not None else "-"
            uplink = vertex.get_uplink()._id if vertex.get_uplink() is not
None else "-"

            terminal = "T" if vertex.is_terminal() else "F"
            patterns = vertex.get_pattern_indices() if
vertex.get_pattern_indices() else "[]"

            if vertex._id == 0:
                print(f"(0 [root]) -> {' '.join(transitions)},
sl:{sufflink}, ul:{uplink}, {terminal}, {patterns}")
            else:
                print(f"({vertex._id}) -> {' '.join(transitions)},
sl:{sufflink}, ul:{uplink}, {terminal}, {patterns}")


def search_with_wildcard(text, pattern, wildcard):
    print(f"\nПоиск шаблона '{pattern}' с джокером '{wildcard}' в тексте
'{text}'")
    chunks = []  # список подстрок между джокерами.
    pattern_infos = []  # позиции этих подстрок в шаблоне
    i = 0
    offset = 0  # позиция в полном шаблоне
    index = 1
    print("\nРазделение шаблона на подстроки без джокеров:")
    while i < len(pattern):
        if pattern[i] == wildcard:  # пропускаем джокеры
            print(f"текущий символ - джокер {pattern[i]}, пропускаем его")
            i += 1
            offset += 1
            continue
```

```
        j = i  # находим кусок без джокеров, сохраняем его и его смещение в
chunks и pattern_infos
        while j < len(pattern) and pattern[j] != wildcard:
            j += 1
        chunk = pattern[i:j]
        chunks.append((chunk, offset))
        pattern_infos.append((offset, len(chunk)))
        print(f"Найдена подстрока без джокеров '{chunk}' на позиции
{offset} длиной {len(chunk)}")
        offset += j - i
        print(f"Сместились на длину подстроки {j - i} в исх. тексте")
        i = j  # ищем далее подстроки
        print(f"i равен концу подстроки {chunk} (j = {j})")

    print("Создаем бор")
    trie = Trie()
    for idx, (chunk, _) in enumerate(chunks):
        trie.add(chunk, idx + 1)  # добавляем каждый кусок в бор

    trie.print_auto()

    pat_len = len(pattern)  # длина полного шаблона
    counts = trie.search_wildcard(text, pattern_infos, pat_len)  #
адаптированный поиск под джокеров
    print(f"\nОбщая длина шаблона: {pat_len}")
    print(f"Количество подстрок: {len(chunks)}")
    trie.print_auto()
    result = []
    for i in range(len(counts)):
        if counts[i] == len(chunks):  # если подсчитанное число совпадений
подстрок, начиная с i, равно длине
            # массива подстрок, на которые была разделена входная, то
совпадение в тексте найдено
            result.append(i + 1)
            print(f"Найдено полное совпадение на позиции {i + 1}")

    return result


def find_overlapping_patterns(matches, patterns):
    matches.sort()
    print("Поиск пересекающихся в тексте шаблонов, благодаря сохранению
позиции вхождения паттерна в текст")
    # словарь для хранения позиций каждого шаблона
    pattern_positions = {}
    for pos, pattern_idx in matches:
        pattern = patterns[pattern_idx - 1]
        start = pos - 1
        end = start + len(pattern) - 1
        if pattern_idx not in pattern_positions:
            pattern_positions[pattern_idx] = []
        pattern_positions[pattern_idx].append((start, end))

    # проверяем пересечения
    overlapping = set()
    all_patterns = list(pattern_positions.keys())

    for i in range(len(all_patterns)):  # первый шаблон
        for j in range(i + 1, len(all_patterns)):  # сраниваем со вторым
            pat1 = all_patterns[i]
            pat2 = all_patterns[j]
            print(f"======== Сраниваем {patterns[pat1 - 1]} и
{patterns[pat2 - 1]} ========")
```

```
            for (s1, e1) in pattern_positions.get(pat1,
                                                  []):  # кортежи,
содержащие начало и конец вхождения паттерна в текст
                for (s2, e2) in pattern_positions.get(pat2, []):
                    print(f"Текущие значения (start_i, end_i): ({s1},
{e1}), ({s2}, {e2})")
                    if not (e1 < s2 or e2 < s1):  # проверка на пересечение
                        overlapping.add(pat1)
                        overlapping.add(pat2)
                        print(
                            f"Пересекаются {patterns[pat1 - 1]} и
{patterns[pat2 - 1]}, (start1 - {s1}, end1 - {e1}), "
                            f"(start2 - {s2}, end2 - {e2}) \n")

    overlapping_patterns = sorted([patterns[idx - 1] for idx in
overlapping])
    return overlapping_patterns


if __name__ == '__main__':
    print("Без джокера:")
    print("Введите текст:")
    T = input().strip()
    print("Введите количество шаблонов: ")
    n = int(input())
    P = []
    print("Введите шаблоны:")
    for _ in range(n):
        P.append(input().strip())

    t = Trie()
    for i in range(n):
        t.add(P[i], i + 1)

    t.print_auto()
    matches = t.search(T, P)
    t.print_auto()
    matches.sort()

    print("Результаты:")
    for pos, pattern_index in matches:
        print(f"позиция в тексте - {pos}, номер паттерна -
{pattern_index}")
    print('=' * 150)
    print(f"Количество вершин в автомате: {t.size()}")

    res = find_overlapping_patterns(matches, P)
    print(f"Все шаблоны, которые с каким-либо точно пересекаются - {',
'.join(res)}")

    # exit(1)
    print('=' * 150)
    print("С джокером:")
    print('Введите текст:')
    T = input().strip()
    print('Введите подстроку:')
    P = input().strip()
    print('Введите символ джокера:')
    wildcard = input().strip()

    matches = search_with_wildcard(T, P, wildcard)

    print('Результаты:')
    for pos in matches:
```

```
print(f"Позиция совпадения шаблона в тексте : {pos}")
```