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: Алгоритм Ахо-Корасик

Студент гр. 1304 Поршнев Р.А.

Преподаватель Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Изучить алгоритм Ахо-Корасик.

Задание.

1. Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n ($1 \leq n \leq 100000$), каждая следующая из n строк содержит шаблон из набора $P = \{p_1, \dots, p_n\}, 1 \leq |p| \leq 75$.

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.

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:

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

Вторая строка содержит 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.

Основные теоретические положения.

1. Данную задачу следует решать классическим алгоритмом Ахо-Корасик.

2. Для решения данной задачи строку-образец следует разбить на подстроки по символу-разделителю *джокеру*. Затем по данным подстрокам нужно построить бор и запустить автомат, который будет последовательно идти по символам строки и узлам бора. После окончания работы автомата следует сопоставить позиции вхождений каждой подстроки с учётом смещения с позициями в строке-образце и в случае сохранения инварианта позиция вхождения сохраняется.

Выполнение работы.

1. В ходе реализации данной задачи реализована структура *NodeInfo*, содержащая в себе информацию о каждом узле бора: суффиксную ссылку на другой узел, является ли данный узел бора образцом, какой узел является родителем данного, быстрая суффиксная ссылка (суффиксная ссылка на ближайший терминальный узел), список потомков данного узла.

Реализован класс *InputData*, предназначенный для считывания начальных данных и их получения. На вход конструктору подаётся символ, который не используется в искомом алфавите и соответствующее поле в классе инициализируется данным значением. Реализованы следующие методы:

- *read_text_and_patterns(self)* – данный метод предназначен для считывания текста, количества образцов и самих образцов непосредственно. К

началу строки каждого образца добавляется специальный символ null_symbol, который нужен для корректного построения бора;

- *get_text(self)* – данный метод предназначен для получения текста;
- *get_patterns(self)* – данный метод предназначен для получения списка образцов.

Для решения задачи реализован класс *AhoCorasick*. Конструктор данного класса принимает на вход текст, образцы и символ, который гарантированно не содержится в образцах. Соответствующие поля класса инициализируются входными значениями. Реализованы следующие методы:

- *run(self)* – данный метод запускает решение задачи: построение бора, создание суффиксных ссылок, создание быстрых суффиксных ссылок, запуск автомата для получения предварительных данных, обработка предварительных данных и получение ответа на задачу, печать ответа на экран;
- *__build_tree(self)* – данный метод строит бор по полученным образцам;
- *__make_suffix_link(self)* – данный метод создает суффиксные ссылки для каждого узла бора. Алгоритм представляет собой "ленивую" динамику, то есть новые суффиксные ссылки создаются на основе уже существующих;
- *__make_fast_suffix_links(self)* – данный метод создает быстрые суффиксные ссылки на основе уже созданных суффиксных ссылок. Алгоритм представляет собой "ленивую" динамику, то есть новые быстрые суффиксные ссылки создаются на основе уже существующих или на основании того, что сейчас указатель автомата находится на терминальном узле;
- *__search_for_occurrences(self)* – данный метод предназначен для получения предварительных данных, которые затем можно интерпретировать как ответ на задачу. Автомат двигается по символам текста и узлам бора. Если есть возможность перейти в новое состояние по потомку узла, то автомат перейдёт к данному потомку, иначе произойдёт переход по суффиксной ссылке и там установится наличие возможности, оговоренной выше. Если текущего символа нет в потомке и для данного узла нет суффиксной ссылки, значит, автомат

пропускает данный символ текста и переходит к следующему. Если автомат оказался в терминальном узле, значит в тексте был найден один из образцов и эти данные нужно зафиксировать для дальнейшей обработки и получения ответа на исходную задачу;

- *__handle_answer(self)* – данный метод предназначен для обработки предварительных данных, которые после обработки и являются ответом на исходную задачу. Предварительные данные содержатся в словаре *self.__answer*, где ключ – это образец, а значения – списки позиций вхождений данного образца в текст. В роли значения ключа выступают списки, а не список позиций, так как входные данные не гарантируют того, что каждый образец входит единожды. Если алгоритм в методе *__search_for_occurrences(self)* находит вхождения некоторого образца, то к ключу данного образца добавляется список позиций вхождений данного образца в текст. Если в будущем данный автомат снова попадёт на данный образец, к значениям данного ключа добавится ещё один список позиций вхождений данного образца в текст. Данный метод раскрывает все эти списки, преобразуя в один. Затем происходит сортировка ключей словаря и значений каждого ключа. Полученный словарь и является ответом на задачу.

Исходный код представлен в Приложении А.

2. В ходе реализации данной задачи реализована структура *NodeInfo*, содержащая в себе информацию о каждом узле бора: суффиксную ссылку на другой узел, является ли данный узел бора образцом, какой узел является родителем данного, быстрая суффиксная ссылка (суффиксная ссылка на ближайший терминальный узел), список потомков данного узла.

Реализован класс *InputData*, предназначенный для считывания начальных данных и их получения. На вход конструктору подаётся символ, который не используется в искомом алфавите и соответствующее поле в классе инициализируется данным значением. Реализованы следующие методы:

- *run(self)* – данный метод запускает работу методов класса, таких как считывание исходных данных, разделения образца на подстроки, вычисление смещения соседних подстрок;

- *__read_text_and_patterns(self)* – данный метод предназначен для считывания текста, образца и символа-джокера;
- *__splitting_pattern(self)* – данный метод разделяет строку-образец на подстроки по символу-джокеру и к каждой подстроке добавляется null_symbol. Построение бора будет происходить поданным подстрокам;
- *__compute_position_differences(self)* – данный метод высчитывает смещение текущей подстроки относительно следующей. Подстроки сформированы в результате разбиения образца по символу-джокеру;
- *get_text(self)* – данный метод предназначен для получения текста;
- *get_patterns(self)* – данный метод предназначен для получения списка образцов;
- *get_position_differences(self)* – данный метод предназначен для получения массива смещений подстрок, который понадобится для оценки предварительных данных и формирования ответа на исходную задачу;
- *get_start_offset(self)* – данный метод возвращает позицию первого не джокер-символа.

Для решения задачи реализован класс *AhoCorasickJoker*. Конструктор данного класса принимает на вход текст, образцы, список смещений подстрок в образце, позицию первого не джокер-символа в образце и символ, который гарантированно не содержится в образцах. Соответствующие поля класса инициализируются входными значениями. Реализованы следующие методы:

- *run(self)* – данный метод запускает решение задачи: построение бора, создание суффиксных ссылок, создание быстрых суффиксных ссылок, запуск автомата для получения предварительных данных, обработка предварительных данных и получение ответа на задачу, печать ответа на экран;
- *__build_trie(self)* – данный метод строит бор по полученным образцам;

- *__make_suffix_links(self)* – данный метод создает суффиксные ссылки для каждого узла бора. Алгоритм представляет собой "ленивую" динамику, то есть новые суффиксные ссылки создаются на основе уже существующих;
- *__make_fast_suffix_links(self)* – данный метод создает быстрые суффиксные ссылки на основе уже созданных суффиксных ссылок. Алгоритм представляет собой "ленивую" динамику, то есть новые быстрые суффиксные ссылки создаются на основе уже существующих или на основании того, что сейчас указатель автомата находится на терминальном узле;
- *__search_for_occurrences(self)* – данный метод преднзначен для получения предварительных данных, которые затем можно интерпретировать как ответ на задачу. Автомат двигается по символам текста и узлам бора. Если есть возможность перейти в новое состояние по потомку узла, то автомат перейдёт к данному потомку, иначе произойдёт переход по суффиксной ссылке и там установится наличие возможности, оговоренной выше. Если текущего символа нет в потомке и для данного узла нет суффиксной ссылки, значит, автомат пропускает данный символ текста и переходит к следующему. Если автомат оказался в терминальном узле, значит в тексте был найден один из образцов и эти данные нужно зафиксировать для дальнейшей обработки и получения ответа на исходную задачу;
- *__handle_positions_of_occurrences(self)* – данный метод предназначен для обработки предварительных данных, которые после обработки и являются ответом на исходную задачу. Предварительные данные содержатся в словаре *self.__positions_of_occurrences*, где ключ – это образец, а значения –список, в котором индекс – позиция начала вхождения подстроки в текст, а значение – является ли данная позиция началом вхождения подстроки в текст. Основываясь на значениях массива позиций начала подстрок *self.__position_differences*, разделённых символа джокера, в исходном образце и данных, записанных в массиве *self.__positions_of_occurrences* происходит формирование ответа на исходную задачу;

- *__print_answer(self)* – данный метод предназначен для печати ответа на экран.

Исходный код представлен в Приложении А.

Выводы.

В ходе выполнения лабораторной работы был изучен алгоритм Ахо-Корасик, из терминологии изучены такие понятия как суффиксные ссылки, сжатые суффиксные ссылки или быстрые суффиксные ссылки, бор и автомат.

Разработана программа, выполняющая считывание с клавиатуры исходных данных в виде текста, количества образцов, непосредственно самих образцов. Затем данная программа строит бор по заданным образцам, формирует суффиксные ссылки и сжатые суффиксные ссылки для всех узлов бора. Далее программа, работая как автомат, находит позиции вхождений исходных образцов в текст, обрабатывает полученные данные и выводит данные позиции и номера образцов на экран. По своему принципу работы данная программа является реализацией алгоритма Ахо-Корасик.

Также разработана программа, выполняющая считывание с клавиатуры исходных данных в виде текста, образца и символа-джокера. Затем программа строит бор по подстрокам, которые были образованы в результате деления образца по символу-джокера, формирует суффиксные ссылки и сжатые суффиксные ссылки для всех узлов бора. Далее программа, работая как автомат, находит позиции вхождений подстрок в исходный текст и фиксирует данные подстроки и позиции. Затем полученные данные обрабатываются и ответ на исходную задачу выводится на экран. По своему принципу работы данная программа является приложением алгоритма Ахо-Корасик.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: task1.py

```
import queue

'''
Данная структура содержит информацию о каждом узле бора: суффиксную ссылку на другой узел,
является ли данный узел бора образцом, какой узел является родителем данного,
быстрая суффиксная ссылка (суффиксная ссылка на ближайший терминальный узел), список потомков данного узла.
'''


class NodeInfo:
    def __init__(self, suffix_link, is_terminal, parent):
        self.suffix_link = suffix_link
        self.is_terminal = is_terminal
        self.parent = parent
        self.fast_suffix_link = None
        self.children_names = []


'''
Класс, содержащий инструменты для решения поставленной задачи.
'''


class AhoCorasick:

    '''
    Конструктор для инициализация начальных данных.
    Входные данные: текст, список образцов, символ, который гарантированно не содержится в образцах.
    '''

    def __init__(self, input_text, input_patterns, null_symbol):
        self.__null_symbol = null_symbol
        self.__text = input_text
        self.__patterns = input_patterns
        self.__trie = {self.__null_symbol: NodeInfo(None, False, self.__null_symbol)}
        self.__answer = {}

    '''
    Данный метод запускает решение задачи: построение бора, создание суффиксных ссылок,
    создание быстрых суффиксных ссылок, запуск автомата для получения предварительных данных,
    обработка предварительных данных -> получение ответа на задачу, печать ответа на экран.
    '''

    def run(self):
```

```
        self.__build_trie()
        self.__make_suffix_links()
        self.__make_fast_suffix_links()
        self.__search_for_occurrences()
        self.__handle_answer()
        self.__print_answer()

    '''
    Данный метод строит бор по полученным образцам.
    '''

    def __build_trie(self):
        for pattern in self.__patterns:
            node_name = ''
            for i in range(len(pattern)):
                node_name += pattern[i]
                if node_name not in self.__trie:
                    node_info   =   NodeInfo(self.__null_symbol,   i   ==
len(pattern) - 1, node_name[:len(node_name) - 1])
                    self.__trie[node_name[:len(node_name)                -
1]].children_names.append(node_name)
                    self.__trie[node_name] = node_info
                elif node_name in self.__trie and i == len(pattern) - 1:
                    self.__trie[node_name].is_terminal = True

    '''
    Данный метод создает суффиксные ссылки для каждого узла бора.
    Алгоритм представляет собой "ленивую" динамику,
    то есть новые суффиксные ссылки создаются на основе уже существующих.
    '''

    def __make_suffix_links(self):
        nodes_names = queue.Queue()
        nodes_names.put(self.__null_symbol)
        while not nodes_names.empty():
            node_name = nodes_names.get()
            transit_node_name                                              =
self.__trie[self.__trie[node_name].parent].suffix_link
            edge_weight = node_name[len(node_name) - 1]
            suffix_link_was_found = False
            while transit_node_name is not None:
                children = self.__trie[transit_node_name].children_names
                for child in children:
                    if child[len(child) - 1] == edge_weight:
                        self.__trie[node_name].suffix_link = child
                        suffix_link_was_found = True
                        break
                if suffix_link_was_found:
                    break
                else:
                    transit_node_name                                      =
self.__trie[transit_node_name].suffix_link
            children_names = self.__trie[node_name].children_names
            for child_name in children_names:
                nodes_names.put(child_name)

    '''
```

```
    Данный метод создает быстрые суффиксные ссылки на основе уже созданных
суффиксных ссылок.
    Алгоритм представляет собой "ленивую" динамику,
    то есть новые быстрые суффиксные ссылки создаются на основе уже
существующих или на основании того,
    что сейчас указатель автомата находится на терминальном узле.
    '''

    def __make_fast_suffix_links(self):
        nodes_names = queue.Queue()
        nodes_names.put(self.__null_symbol)
        while not nodes_names.empty():
            node_name = nodes_names.get()
            transit_node_name = self.__trie[node_name].suffix_link
            while transit_node_name is not None:
                if self.__trie[transit_node_name].is_terminal:
                    self.__trie[node_name].fast_suffix_link             =
transit_node_name
                    break
                elif  self.__trie[transit_node_name].fast_suffix_link  is
not None:
                    self.__trie[node_name].fast_suffix_link             =
self.__trie[transit_node_name].fast_suffix_link
                    break
                transit_node_name                                       =
self.__trie[transit_node_name].suffix_link
            children_names = self.__trie[node_name].children_names
            for child_name in children_names:
                nodes_names.put(child_name)

    '''
    Данный метод предназначен для получения предварительных данных,
    которые затем можно интерпретировать как ответ на задачу.
    Автомат двигается по символам текста и узлам бора. Если есть
возможность перейти в новое состояние по потомку узла,
    то автомат перейдёт к данному потомку, иначе произойдёт переход по
суффиксной ссылке
    и там установится наличие возможности, оговоренной выше. Если текущего
символа нет в потомке и для данного узла нет
    суффиксной ссылки, значит, автомат пропускает данный символ текста и
переходит к следующему.
    Если автомат оказался в терминальном узле, значит в тексте был найден
один из образцов и эти данные нужно
    зафиксировать для дальнейшей обработки и получения ответа на исходную
задачу.
    '''

    def __search_for_occurrences(self):
        current_node_name = self.__null_symbol
        i = 0
        while i < len(self.__text):
            children_names                                              =
self.__trie[current_node_name].children_names
            child_was_found = False
            for child_name in children_names:
                if child_name[-1] == self.__text[i]:
                    current_node_name += child_name[-1]
                    child_was_found = True
```

```
                    i += 1
                    break
            if not child_was_found:
                if self.__trie[current_node_name].suffix_link is not None:
                    current_node_name                                    =
self.__trie[current_node_name].suffix_link
                else:
                    i += 1
                continue
            if   not   self.__trie[current_node_name].is_terminal   and
self.__trie[
                current_node_name].fast_suffix_link is None:
                continue
            pattern_node_name = current_node_name
            while pattern_node_name is not None:
                if self.__trie[pattern_node_name].is_terminal:
                    if i - len(pattern_node_name) + 2 not in self.__answer:
                        self.__answer[i  -  len(pattern_node_name)  +  2]  =
[self.__patterns[pattern_node_name]]
                    else:
                        self.__answer[i    -    len(pattern_node_name)    +
2].append(self.__patterns[pattern_node_name])
                pattern_node_name                                        =
self.__trie[pattern_node_name].fast_suffix_link

    '''
    Данный метод предназначен для обработки предварительных данных,
    которые после обработки и являются ответом на исходную задачу.
    Предварительные данные содержатся в словаре self.__answer, где ключ -
- это образец,
    а значения -- списки позиций вхождений данного образца в текст. В роли
значений выступают списки,
    а не список позиций, так как входные данные не гарантируют того, что
каждый образец входит единожды.
    Если алгоритм в методе __search_for_occurrences(self) находит
вхождения некоторого образца,
    то к ключу данного образца добавляется список позиций вхождений данного
образца в текст.
    Если в будущем данный автомат снова попадёт на данный образец, к
значениям данного ключа добавится ещё один список
    позиций вхождений данного образца в текст. Данный метод раскрывает все
эти списки, преобразуя в один. Затем
    происходит сортировка ключей словаря и значений каждого ключа.
Полученный словарь и является ответом на задачу.
    '''

    def __handle_answer(self):
        for position in self.__answer:
            numbers_of_patterns = []
            for elem in self.__answer[position]:
                for number_of_pattern in elem:
                    numbers_of_patterns.append(number_of_pattern)
            self.__answer[position] = numbers_of_patterns
        self.__answer = dict(sorted(self.__answer.items(), key=lambda x:
x[0]))
        for position in self.__answer:
            self.__answer[position].sort()
```

```
    '''
    Данный метод предназначен для печати ответа на экран.
    '''

    def __print_answer(self):
        for position in self.__answer:
            for number_of_pattern in self.__answer[position]:
                print(position, number_of_pattern)


'''
Данный класс предназначен для считывания начальных данных и их получения.
'''


class InputData:
    '''
    Данный конструктор инициализирует символ null_symbol, который
гарнтирвоанно не будет встречаться ни в образцах,
    ни в тексте.
    '''

    def __init__(self, null_symbol):
        self.__patterns = {}
        self.__text = ''
        self.__null_symbol = null_symbol

    '''
    Данный метод предназначен для считывания текста, количества образцов
и самих образцов непосредственно.
    К началу строки каждого образца добавляется специльный символ
null_symbol,
    который нужен для корректного построения бора.
    '''

    def read_text_and_patterns(self):
        self.__patterns = {}
        self.__text = input()
        number_of_patterns = int(input())
        for i in range(number_of_patterns):
            pattern = self.__null_symbol + input()
            if pattern not in self.__patterns:
                self.__patterns[pattern] = [i + 1]
            else:
                self.__patterns[pattern].append(i + 1)

    '''
    Данный метод предназначен для получения текста.
    '''

    def get_text(self):
        return self.__text

    '''
    Данный метод предназначен для получения списка образцов.
    '''

    def get_patterns(self):
```

```
        return self.__patterns


if __name__ == "__main__":
    input_data_reader = InputData(' ')
    input_data_reader.read_text_and_patterns()
    aho_corasick       =       AhoCorasick(input_data_reader.get_text(),
input_data_reader.get_patterns(), ' ')
    aho_corasick.run()
```

Название файла: task2.py

```
import queue


'''
Данная структура содержит информацию о каждом узле бора: суффиксную ссылку
на другой узел,
является ли данный узел бора образцом, какой узел является родителем
данного,
быстрая суффиксная ссылка (суффиксная ссылка на ближайший терминальный
узел), список потомков данного узла.
'''



class NodeInfo:
    def __init__(self, suffix_link, is_terminal, parent):
        self.suffix_link = suffix_link
        self.is_terminal = is_terminal
        self.parent = parent
        self.fast_suffix_link = None
        self.children_names = []



'''
Класс, содержащий инструменты для решения поставленной задачи.
'''



class AhoCorasickJoker:
    '''
    Конструктор для инициализация начальных данных.
    Входные данные: текст, список подстрок образца, позиции вхождений
каждой подстроки в образец,
```

```
    смещение образца относительно символов-джокеров, стоящих в его начале,
символ,
    который гарантированно не содержится в образцах.
    '''

    def __init__(self, input_text, input_patterns, position_differences,
start_offset, null_symbol):
        self.__position_differences = position_differences
        self.__null_symbol = null_symbol
        self.__text = input_text
        self.__patterns = input_patterns
        self.__trie = {self.__null_symbol: NodeInfo(None, False,
self.__null_symbol)}
        self.__positions_of_occurrences = {}
        self.__answer = []
        self.__start_offset = start_offset

    '''
    Данный метод запускает решение задачи: построение бора, создание
суффиксных ссылок,
    создание быстрых суффиксных ссылок, запуск автомата для получения
предварительных данных,
    обработка предварительных данных -> получение ответа на задачу, печать
ответа на экран.
    '''

    def run(self):
        self.__build_trie()
        self.__make_suffix_links()
        self.__make_fast_suffix_links()
        self.__search_for_occurrences()
        self.__handle_positions_of_occurrences()
        self.__print_answer()

    '''
    Данный метод строит бор по полученным образцам.
    '''

    def __build_trie(self):
```

```
        for pattern in self.__patterns:
            node_name = ''
            for i in range(len(pattern)):
                node_name += pattern[i]
                if node_name not in self.__trie:
                    node_info = NodeInfo(self.__null_symbol, i ==
len(pattern) - 1, node_name[:len(node_name) - 1])
                    self.__trie[node_name[:len(node_name) -
1]].children_names.append(node_name)
                    self.__trie[node_name] = node_info
                elif node_name in self.__trie and i == len(pattern) - 1:
                    self.__trie[node_name].is_terminal = True

    '''
    Данный метод создает суффиксные ссылки для каждого узла бора.
    Алгоритм представляет собой "ленивую" динамику,
    то есть новые суффиксные ссылки создаются на основе уже существующих.
    '''

    def __make_suffix_links(self):
        nodes_names = queue.Queue()
        nodes_names.put(self.__null_symbol)
        while not nodes_names.empty():
            node_name = nodes_names.get()
            transit_node_name =
self.__trie[self.__trie[node_name].parent].suffix_link
            edge_weight = node_name[len(node_name) - 1]
            suffix_link_was_found = False
            while transit_node_name is not None:
                children = self.__trie[transit_node_name].children_names
                for child in children:
                    if child[len(child) - 1] == edge_weight:
                        self.__trie[node_name].suffix_link = child
                        suffix_link_was_found = True
                        break
                if suffix_link_was_found:
                    break
                else:
```

```
                    transit_node_name =
self.__trie[transit_node_name].suffix_link
            children_names = self.__trie[node_name].children_names
            for child_name in children_names:
                nodes_names.put(child_name)

    '''
    Данный метод создает быстрые суффиксные ссылки на основе уже созданных
суффиксных ссылок.
    Алгоритм представляет собой "ленивую" динамику,
    то есть новые быстрые суффиксные ссылки создаются на основе уже
существующих или на основании того,
    что сейчас указатель автомата находится на терминальном узле.
    '''

    def __make_fast_suffix_links(self):
        nodes_names = queue.Queue()
        nodes_names.put(self.__null_symbol)
        while not nodes_names.empty():
            node_name = nodes_names.get()
            transit_node_name = self.__trie[node_name].suffix_link
            while transit_node_name is not None:
                if self.__trie[transit_node_name].is_terminal:
                    self.__trie[node_name].fast_suffix_link =
transit_node_name
                    break
                elif  self.__trie[transit_node_name].fast_suffix_link  is
not None:
                    self.__trie[node_name].fast_suffix_link =
self.__trie[transit_node_name].fast_suffix_link
                    break
                transit_node_name =
self.__trie[transit_node_name].suffix_link
            children_names = self.__trie[node_name].children_names
            for child_name in children_names:
                nodes_names.put(child_name)

    '''
    Данный метод преднзначен для получения предварительных данных,
```

которые затем можно интерпретировать как ответ на задачу.

Автомат двигается по символам текста и узлам бора. Если есть возможность перейти в новое состояние по потомку узла,

то автомат перейдёт к данному потомку, иначе произойдёт переход по суффиксной ссылке

и там установится наличие возможности, оговоренной выше. Если текущего символа нет в потомке и для данного узла нет

суффиксной ссылки, значит, автомат пропускает данный символ текста и переходит к следующему.

Если автомат оказался в терминальном узле, значит в тексте был найден один из образцов и эти данные нужно

зафиксировать для дальнейшей обработки и получения ответа на исходную задачу.

'''

```
    def __search_for_occurrences(self):
        current_node_name = self.__null_symbol
        i = 0
        while i < len(self.__text):
            children_names = self.__trie[current_node_name].children_names
            child_was_found = False
            for child_name in children_names:
                if child_name[-1] == self.__text[i]:
                    current_node_name += child_name[-1]
                    child_was_found = True
                    i += 1
                    break
            if not child_was_found:
                if self.__trie[current_node_name].suffix_link is not None:
                    current_node_name = self.__trie[current_node_name].suffix_link
                else:
                    i += 1
                continue
            if not self.__trie[current_node_name].is_terminal and self.__trie[
                current_node_name].fast_suffix_link is None:
                continue
```

```
            pattern_node_name = current_node_name
            while pattern_node_name is not None:
                if self.__trie[pattern_node_name].is_terminal:
                    if pattern_node_name not in
self.__positions_of_occurrences:
                        self.__positions_of_occurrences[pattern_node_name]
= [False] * (len(self.__text) + 1)

self.__positions_of_occurrences[pattern_node_name][i -
len(pattern_node_name) + 2] = True
                    else:

self.__positions_of_occurrences[pattern_node_name][i -
len(pattern_node_name) + 2] = True
                pattern_node_name =
self.__trie[pattern_node_name].fast_suffix_link

    '''
    Данный метод предназначен для обработки предварительных данных,
    которые после обработки и являются ответом на исходную задачу.
    Предварительные данные содержатся в словаре
self.__positions_of_occurrences, где ключ -- это образец,
    а значения -- список, в котором индекс -- позиция начала вхождения
подстроки в текст, а значение -- является ли
    данная позиция началом вхождения подстроки в текст.
    Основываясь на значениях массива позиций начала подстрок
self.__position_differences, разделённых символа джокера,
    в исходном образце и данных, записанных в массиве
self.__positions_of_occurrences происходит формирование ответа
    на исходную задачу.
    '''

    def __handle_positions_of_occurrences(self):
        start_positions = []
        for i, bit in
enumerate(self.__positions_of_occurrences[self.__patterns[0]]):
            if bit:
                start_positions.append(i)
        for start_position in start_positions:
```

```
            previous_position = start_position
            sequence_is_broken = False
            for i, pattern in enumerate(self.__patterns[1:]):
                if previous_position + self.__position_differences[i] >
len(self.__text) or \
                        not
self.__positions_of_occurrences[pattern][previous_position +

self.__position_differences[i]]:
                    sequence_is_broken = True
                    break
                previous_position = previous_position +
self.__position_differences[i]
            if previous_position + self.__position_differences[-1] - 1 >
len(self.__text):
                sequence_is_broken = True
            if not sequence_is_broken and start_position -
self.__start_offset > 0:
                self.__answer.append(start_position - self.__start_offset)

    '''
    Данный метод предназначен для печати ответа на экран.
    '''

    def __print_answer(self):
        for position in self.__answer:
            print(position)


'''
Данный класс предназначен для считывания начальных данных и их получения.
'''


class InputData:
    '''
    Данный конструктор инициализирует символ null_symbol, который
гарнтирвоанно не будет встречаться ни в образцах,
    ни в тексте.
```

```
    '''

    def __init__(self, null_symbol):
        self.__pattern = ''
        self.__text = ''
        self.__joker_symbol = ''
        self.__patterns = []
        self.__null_symbol = null_symbol
        self.__position_differences = []
        self.__start_offset = 0
    '''
```

Данный метод запускает работу методов класса, таких как считывание исходных данных, разделения образца на подстроки,

вычисление смещения соседних подстрок.

```
    '''
    def run(self):
        self.__read_text_and_patterns()
        self.__splitting_pattern()
        self.__compute_position_differences()

    '''
```

Данный метод предназначен для считывания текста, образца и символа-джокера.

```
    '''

    def __read_text_and_patterns(self):
        self.__text = input()
        self.__pattern = input()
        self.__joker_symbol = input()

    '''
```

Данный метод разделяет строку-образец на подстроки по символу-джокеру и к каждой подстроке добавляется null_symbol.

Бор будет строится по данным подстрокам.

```
    '''

    def __splitting_pattern(self):
        dirty_patterns = self.__pattern.split(self.__joker_symbol)
```

```
        self.__patterns = [self.__null_symbol + pattern for pattern in
dirty_patterns if pattern != '']

    '''
    Данный метод высчитывает смещение текущей подстроки относительно
следующей. Подстроки сформированы в результате
    разбиения образца по символу-джокеру.
    '''

    def __compute_position_differences(self):
        for i in range(len(self.__pattern)):
            if i == 0 and self.__pattern[i] != self.__joker_symbol:
                self.__position_differences.append(i)
                continue
            if self.__pattern[i] != self.__joker_symbol \
                    and self.__pattern[i - 1] == self.__joker_symbol:
                self.__position_differences.append(i)
        self.__start_offset = self.__position_differences[0]
        if self.__pattern[-1] == self.__joker_symbol:
            self.__position_differences.append(len(self.__pattern))
        else:

self.__position_differences.append(self.__position_differences[-1])
        for i in range(len(self.__position_differences) - 1):
            self.__position_differences[i]                                  =
self.__position_differences[i + 1] - self.__position_differences[i]

    '''
    Данный метод предназнчен для получения текста.
    '''

    def get_text(self):
        return self.__text

    '''
    Данный метод предназначен для получения списка подстрок, по которым
будет строится бор.
    '''
```

```
    def get_patterns(self):
        return self.__patterns

    '''
    Данный метод предназначен для получения массива смещений подстрок,
который понадобится для оценки предварительных
    данных и формирования ответа на исходную задачу.
    '''

    def get_position_differences(self):
        return
self.__position_differences[:len(self.__position_differences) - 1]

    '''
    Данный метод возвращает позицию первого не джокер-символа.
    '''
    def get_start_offset(self):
        return self.__start_offset


if __name__ == "__main__":
    input_data = InputData(' ')
    input_data.run()
    aho_corasick_joker     =     AhoCorasickJoker(input_data.get_text(),
input_data.get_patterns(),

input_data.get_position_differences(), input_data.get_start_offset(), ' ')
    aho_corasick_joker.run()
```