**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. 0303 |  | Архипов В. А. |
| Преподаватель |  | Фирсов М. А. |

Санкт-Петербург

2022

## Цель работы.

Изучить алгоритм Ахо-Корасик для нахождения всех вхождений нескольких подстрок в текст и решить с помощью него две задачи: найти все вхождения подстрок в текст; найти все вхождения одной строки с символом джокером, который может быть любым символом.

**Основные теоретические положения.**

Основной идеей в алгоритме Ахо-Корасик является построение бора из символов, составляющих искомые подстроки, а также построение суффиксных ссылок из каждого элемента бора. Суффиксные ссылки нужны для того, чтобы мы имели возможность найти все вхождения за один проход дерева, не имея необходимости совершать «откаты».

## Задание.

1. Разработайте программу, решающую задачу точного поиска набора образцов.  
   **Вход:**  
   Первая строка содержит текст T (). Вторая - число n (), каждая следующая из n строк содержит шаблон из набора  . 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

1. 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 Р в текст Т.  
    Например, образец аb??с? 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, ).  
   Шаблон (P, ).  
   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 2.** Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

**Описание алгоритма.**

1. **Алгоритм Ахо-Корасик состоит из трех этапов:**
2. **Составления бора из паттернов, которые мы будем искать в тексте.**

Бор представляет собой дерево с вершинами в виде символов, строится бор последовательным перебором всех символов каждого паттерна: начиная с корня, если символ уже встречается на следующем уровне дерева среди детей той вершины, в которой мы находимся, то просто переходим к этой вершине, если нет, то добавляем этот символ в детей текущей вершины и переходим к только что добавленной вершине. Основная использованная структура данных – словарь.

Для построения бора нужно один раз пройтись по всем символам всех переданных подстрок, тогда сложность его по времени – , где – длина i-й подстроки.

Также дополнительно в ходе построения бора мы ставим метки на вершины бора, которые завершают искомые подстроки (путь от корня до данной вершины является паттерном, или говорят, что паттерн является меткой данной вершины).

Оценка сложности первой части алгоритма:

- По времени - , где – длина i-й подстроки, поскольку мы единожды итерируемся по каждой из подстрок

- По памяти - , так как нам необходимо хранить сам бор.

1. **Построение конечного автомата.**

Строить конечный автомат для алгоритма нам помогут суффиксные ссылки.

Суффиксная ссылка – это ссылка на другой элемент бора, который является наибольшим суффиксом того префикса, который составляет путь от корня до рассматриваемого элемента бора. Алгоритм построения заключается в последовательном построении суффиксных ссылок по уровням, начиная с корня (обход бора в ширину). Для элемента бора, для которого мы хотим построить суффиксную ссылку, мы смотрим на суффиксную ссылку его родителя (так как мы обходим бор в ширину, то эта ссылка уже есть), если среди детей этой ссылки есть символ аналогичный символу рассматриваемого элемента бора, то строим ссылку от переданного элемента бора до найденного ребенка суффикса, если нет, то переходим по суффиксной ссылке суффикса и снова проверяем. Так продолжаем, пока не встретим подходящий суффикс или пока не дойдем до корня (во втором случае суффиксной ссылкой будет считаться корень). Когда мы нашли подходящий суффикс, то для всех меток найденной вершины вершина, для которой мы искали суффиксную ссылку, так же становится терминальной (это позволяет оптимизировать алгоритм).

Оценка сложности второй части алгоритма:

- По времени - , где – длина i-й подстроки. Стоит заметить, что при поиске подходящих суффиксных ссылок мы можем переходить по ним ограниченное число раз, и это число точно не превысит длины рассматриваемой ветви нашего дерева, поэтому этим числом при оценке сложности можно пренебречь.

- По памяти - , необходимо хранить сам бор.

1. **Алгоритм поиска с помощью конечного автомата (сам алгоритм)*.*** После того, как конечный автомат был построен, можно переходить к самому алгоритму поиска. (Начинаем с корня бора) мы проходимся последовательно по всем символам текста и если символ текста есть среди детей рассматриваемого на данный момент элемента бора, то мы осуществляем переход к этой дочерней вершине. Если же такого узла нет, то мы переходим по суффиксным ссылкам, пока не дойдем до подходящей вершины (вершины, из которой можно осуществить переход по интересующему нас символу). Если такая вершина не найдена, то мы возвращаемся обратно в корневой узел. Если текущий элемент бора является терминальным для какой-то строки \набора строк, то добавляем индекс вхождения и индекс подстроки в массив результатов. Продолжаем до конца текста.

Как видно, этот шаг алгоритма отрабатывает за время O(n), где n – длина текста, в котором мы ищем образцы.

Таким образом общая сложность алгоритма складывается из сложностей трех шагов:

* По времени - , где n – длина текста, – длина i-й подстроки.
* По памяти - необходимо дополнительно хранить бор.

1. **Алгоритм Ахо-Корасик в задаче поиска подстроки с джокером.**

Решение данной задачи заключается в делении строки с джокером на фрагменты по символу джокера и запоминании индексов вхождения получившихся паттернов в изначальный паттерн с джокером. Далее мы запускаем алгоритм Ахо-Корасик для нахождения вхождений этих фрагментов в текст. После этого мы заводим массив, размера n(размер текста, в котором осуществлялся поиск), состоящий из нулей. К индексу массива прибавляется 1, если на этой позиции должен находиться фрагмент (за вычетом смещения этого фрагмента в искомой строке с джокером от позиции, которая была найдена с помощью алгоритма Ахо-Корасика). Если какой-то индекс в массиве становится равным количеству фрагментов, на которые была поделена изначальная искомая строка, то этот индекс является индексом вхождения строки с джокером в текст (стоит так же проверить, что «наложение» на эту часть текста паттерна с джокером не выведет нас за пределы текста).

Сложность по времени O(n + p ), n – длина текста, p – длина паттерна с джокером. К сложности Ахо-Корасик добавляется парсинг строки с джокерами (простой проход за линейное время) и 2 прохода по массиву длины n, поэтому O() сложность не меняется

Сложность по памяти O(p + n), n – длина текста, p – длина паттерна с джокером, помимо хранения бора нам необходимо хранить еще и массив с числами, размера n, поэтому O() сложность алгоритма не меняется.

1. **Подсчет количества вершин в автомате.**

Данная задача решает обычным обходом построенного бора в ширину и с последующей записью всех пройденных элементов во множество. Количество узлов во множестве будет соответствовать количеству вершин в автомате.

Сложность по времени O(, где – длина i-го паттерна.

Сложность по памяти O(, где – длина i-го паттерна.

1. **Вывод списка найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.**

Для начала выполняется поиск паттернов в тексте по алгоритму Ахо-Корасик.

Затем для каждого из паттернов ищутся все его вхождения и выполняется поиск близлежащих паттернов. Выполняется он следующим образом: мы просматриваем все вхождения подстрок в текст, и запоминаем подстроку если:

- ее начало расположено между началом и концом фиксированной подстроки

- между концом и началом данной подстроки расположено начало нашей фиксированной подстроки

Данное действие повторяется для всех появлений подстроки в тексте.

Затем все найденные паттерны пишутся в словарь.

Сложность по времени , где n – длина текста, – i-й паттерн, z – число вхождений паттернов в подстроку. Данная сложность обусловлена тем, что для поиска интересующих нас элементов будь то все вхождения определенного паттерна или все паттерны, которые имеют общую часть с фиксированным, используется итерация по списку решений размера z.

Сложность по памяти O(, где – паттерн, z – число вхождений паттернов в текст, i – число паттернов. Прибавка к бору Ахо-Корасик обусловлена тем, что список вхождений нужно сохранить, также нужно хранить для каждого паттерна число его пересечений с другими.

## Описание функций и структур данных.

- Класс Aho-Node.

Представляет собой узел в боре.

self.goto – словарь вида {key: value},где key - символ, value – узел, на который мы должны перейти при чтении символа key

self.out – список подстрок, для которых данный узел является терминальным.

self.fail – суффиксная ссылка для данной вершины.

- create\_trie(patterns) – функция для построения бора из массива строк pattern.

Аргументы: patterns – список строк, для которых мы хотим построить бор

Возвращаемое значение: root – корневая вершина построенного бора

- create\_machine(patterns) – функция, которая строит бор из массива строк pattern, а затем путем определения суффиксных ссылок для всех вершин бора превращает его в конечный автомат.

Аргументы: patterns – массив строк, для которых мы хотим построить бор, а затем и конечный автомат

Возвращаемое значение: root – начальная вершина построенного автомата.

* aho\_corasick(s, root, patterns) – сам алгоритм Ахо-Корасик.

Аргументы: s – текст, в котором будет осуществляться поиск

root – начальная вершина конечного автомата

patterns – список подстрок, которые мы будем искать в тексте.

Возвращаемое значение: Массив кортежей вида (x, y), где x – индекс вхождения паттерна, а y – порядковый номер этого паттерна в списке patterns.

* count\_vertex(root) – функция для подсчета вершин в конечном автомате.

Аргументы: root – начальная (корневая) вершина конечного автомата

Возвращаемое значение: число вершин конечного автомата

- find\_intersection(search, patterns, find) – ищет пересечения в тексте всех строк-паттернов и одной конкретно заданной.

Аргументы: search - кортеж вида (x, y), где x – индекс вхождения этого паттерна в текст, а y – порядковый номер этого паттерна в списке patterns.

patterns – список паттернов

find - массив кортежей вида (x, y), где x – индекс вхождения паттерна в текст, а y – порядковый номер этого паттерна в списке patterns.

Возвращаемое значение: множество паттернов, которые имеют пересечение в тексте с паттерном из search.

Тестирование см. в приложении А.

Разработанный программный код см. в приложении Б.

## Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм Ахо-Корасик. С помощью данного алгоритма были успешно решены задачи нахождения вхождений подстрок в текст и поиск вхождения строки с джокером.

# **ПРИЛОЖЕНИЕ А ТЕСТИРОВАНИЕ**

Результаты тестирования работы алгоритма для решения двух задач и индивидуализации представлены в табл. 1, табл. 2, табл. 3 и табл. 4.

Таблица 1 – Тестирование работы алгоритма Ахо-Корасик для нахождения всех вхождений подстрок в текст

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | NTAG  3  TAGT  TAG  T | 2 2  2 3 |
|  | ATCGTCACACATCA  4  AT  CG  CAC  TC | 1 1  2 4  3 2  5 4  6 3  8 3  11 1  12 4 |
|  | AAAAAAAAAAA  1  A | 1 1  2 1  3 1  4 1  5 1  6 1  7 1  8 1  9 1  10 1  11 1 |
|  | ACVAVT  0 |  |

Таблица 2 – Тестирование работы алгоритма Ахо-Корасик для нахождения в тексте паттерна с джокером(любым символом).

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | ACTANCA  A$$A$  $ | 1 |
|  | CATGCCATGGGGG  G&  & | 4  9  10  11  12 |
|  | GTCAGTAGTGTGCATCGT  GT$$A  $ | 10 |
|  | ATCGTGATCGT  GAT$  $ | 6 |
|  | ACTGGTTCAGT  WWG  W | 2  3  8 |

Таблица 3 – Тестирование работы программы для нахождения числа вершин в автомате

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | HE  SHE  HER  HERS  HIS | 10 |
|  | AT | 3 |
|  | AT  A | 3 |
|  |  | 1 |
|  | ATCG  ATCA  ATCC  ATCT | 8 |

Таблица 4 – Тестирование работы алгоритма Ахо-Корасик для нахождения всех пересечений подстрок в тексте.

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | SHEERHISHERS  5  HIS  SHE  HER  HERS  HE | HIS  ['SHE']  SHE  ['HE', 'HIS', 'HER', 'HERS', 'HE']  HER  ['SHE', 'HERS', 'HE']  HERS  ['SHE', 'HER', 'HE']  HE  ['SHE', 'SHE', 'HER', 'HERS'] |
|  | CTGCATCCG  3  TGC  CAT  GCA | TGC  ['GCA', 'CAT']  CAT  ['TGC', 'GCA']  GCA  ['TGC', 'CAT'] |
|  | ATGCATANGA  3  ATA  ATG  NGA | ATA  []  ATG  []  NGA  [] |
|  | APESHUTSUNAMI  2  SHUT  TSUNAMI | SHUT  ['TSUNAMI']  TSUNAMI  ['SHUT'] |
|  | CAPABILITYBIL  3  CAPAB  BILI  LITY | CAPAB  ['BILI']  BILI  ['CAPAB', 'LITY']  LITY  ['BILI'] |

**ПРИЛОЖЕНИЕ Б**

**РАЗРАБОТАННЫЙ ПРОГРАММНЫЙ КОД**

Название файла: first.py

import logging

import sys

LEVEL = logging.DEBUG

logging.basicConfig(stream=sys.stdout, level=LEVEL)

class AhoNode:

    # Вспомогательный класс для построения дерева

    def \_\_init\_\_(self, symbol=None):

        # Словарь вида key = символ, value = узел, на который нужно перейти при чтении данного символа

        self.goto = {}

        # Список паттернов, для которых этот узел является терминальным

        self.out = []

        # Суффиксная ссылка для данной вершины

        self.fail = None

        self.symbol = symbol

        self.path = []

    def add\_sym(self, symbol, prev):

        self.path.extend(prev.path)

        self.path.append(symbol)

def create\_trie(patterns):

    # Создать бор - дерево паттернов

    root = AhoNode('root')

    root.path.append('root')

    # Добавление слов в бор

    for path in patterns:

        logging.debug(f'Добавление слова {path} в бор')

        node = root

        # Разбиение строки на символы и добавление каждого в бор последовательно, друг за другом

        for ind, symbol in enumerate(path):

            k = 0

            if symbol in node.goto:

                logging.debug(' ' \* ind + f'Символ \'{symbol}\' уже есть, переходим по нему')

            else:

                logging.debug(' ' \* ind + f'Добавление символа \'{symbol}\' и переход по нему')

                prev = node

                k = 1

            node = node.goto.setdefault(symbol, AhoNode(symbol))

            if k == 1:

                node.add\_sym(symbol, prev)

        # В терминальную вершину записываем слово, которое является меткой для этой терминальной вершины

        node.out.append(path)

        logging.debug(' ' \* ind + f'Вершина \'{symbol}\' является терминальной для слова {path}')

        logging.debug('->'.join(node.path))

    return root

def create\_machine(patterns):

    # Функция, создающая автомат Ахо-Корасика.

    # Фактически создает бор и инициализирует fail-функции (они же суффиксные ссылки)

    # всех узлов, обходя дерево в ширину.

    # Создаем бор, инициализируем непосредственных потомков корневого узла

    root = create\_trie(patterns)

    queue = []

    # Обход в ширину

    logging.debug('\n\nПостроение автомата')

    logging.debug('Инициализация суффиксных ссылок для детей корневой вершины')

    for node in root.goto.values():

        logging.debug(f'Символ \'{node.symbol}\' ссылается на корень')

        queue.append(node)

        # Для детей корневого узла суффиксная ссылка ссылается на корень

        node.fail = root

    # Инициализируем остальные узлы

    while len(queue) > 0:

        rnode = queue.pop(0)

        for key, unode in rnode.goto.items():

            depth = 2

            queue.append(unode)

            # rnode - родитель

            # unode - сын

            # Пусть rnode и unode соединены через ребро с символом 'a'

            # Ищем суффиксную ссылку для вершины unode

            logging.debug(' '\*depth + f'Ищем суффиксную ссылку для вершины ' + '->'.join(unode.path))

            # Переходим по уже имеющейся суффиксной ссылке для rnode

            fnode = rnode.fail

            logging.debug(' '\*depth + f'Переходим по суффиксной ссылке для родителя вершины \'{unode.symbol}\' ' + '| ' + '->'.join(unode.path))

            logging.debug(' '\*depth + 'Пришли в ' + '->'.join(fnode.path))

            depth += 2

            # Если из fnode можно пройти по ребру 'a', то суффиксная ссылка для unode

            # будет указывать на вершину в которую мы перейдем

            # В противном случае запускаем рекурсивный поиск вершины

            while fnode is not None and key not in fnode.goto:

                logging.debug(' '\*depth + f'Из вершины \'{fnode.symbol}\' нельзя пройти по ребру \'{key}\' переходим на следующую суффиксную ссылку')

                fnode = fnode.fail

                if fnode:

                    logging.debug(' ' \* depth + 'Пришли в ' + '->'.join(fnode.path))

                else:

                    logging.debug(' '\*depth + 'Пришли в несуществующий узел')

                depth += 2

            # Если подходящей суффиксной ссылки найти не удалось, то ссылаемся на корень дерева

            unode.fail = fnode.goto[key] if fnode else root

            if fnode:

                logging.debug(' '\*depth + f'Из вершины \'{fnode.symbol}\' можно пройти по ребру \'{key}\'')

                logging.debug(' '\*depth + 'Итого суффиксная ссылка для ' + '->'.join(unode.path) + ' это ' + '->'.join(fnode.goto[key].path))

            else:

                logging.debug(' '\*depth + 'Подходящей суффиксной ссылки найти не удалось, поэтому ссылаемся на корень')

                logging.debug(

                    ' ' \* depth + 'Итого суффиксная ссылка для ' + '->'.join(unode.path) + ' это ' + 'root')

            # Делаем unode терминальной вершиной для тех слов,

            # для которых терминальной вершиной являлась суффиксная ссылка

            if unode.fail.out:

                logging.debug(f'Вершина \'{unode.symbol}\' стала терминальной для следующих слов: ' + ','.join(unode.fail.out))

            else:

                logging.debug(

                    f'Вершина \'{unode.symbol}\' не стала терминальной для новых слов')

            unode.out += unode.fail.out

            logging.debug('\n\n')

    return root

def aho\_corasick(s, root, patterns):

    # Находит все возможные подстроки из набора паттернов в строке.

    node = root

    # Списко для записи решения

    solution = []

    # Итерация по строке

    for i in range(len(s)):

        logging.debug('Мы находимся на вершине ' + '->'.join(node.path))

        logging.debug(f'Рассматриваем символ \'{s[i]}\'')

        # Если мы находимся на вершине и нам некуда перейти, то мы переходим по суффиксной ссылке

        while node is not None and s[i] not in node.goto:

            logging.debug(f'Из вершины ' + '->'.join(node.path) + f' мы не можем перейти по ребру \'{s[i]}\', перейдем по суффиксной ссылке')

            node = node.fail

            logging.debug(f'Теперь мы находимся в вершине  ' + '->'.join(node.path) if node else 'Теперь мы находимся в вершине None' )

        # Если мы попали в несуществующий узел, то возвращаемся на корневой узел

        if node is None:

            logging.debug('Пришли в несуществующий узел, возвращаемся в корневую вершину')

            node = root

            continue

        # Если мы стоим на вершине и у нас есть интересующее нас ребро, выполняем переход

        logging.debug(f'Мы можем выполнить переход по ребру \'{s[i]}\', осуществляем')

        node = node.goto[s[i]]

        logging.debug(f'Пришли в ' + '->'.join(node.path))

        # Если вершина, в которую мы пришли, является терминальной для каких-то слов, то записываем в список кортеж вида

        # (позиция, с которой началось вхождение; номер найденной строки)

        if not node.out:

            logging.debug('Данная вершина не является терминальной ни для одного слова')

        for pattern in node.out:

            logging.debug(f'Вершина ' + '->'.join(node.path) + ' является терминальной для слова: ' + pattern)

            solution.append((i - len(pattern) + 2, patterns.index(pattern) + 1))

            logging.debug(f'Зафиксируем вхождение ' + str(solution[-1]))

        logging.debug('\n\n')

    return solution

s = input()

n = int(input())

patterns = []

for \_ in range(n):

    patterns.append(input())

# Создание автомата для алгоритма

root = create\_machine(patterns)

# Запуск алгоритма

logging.debug('Запуск алгоритма')

res = aho\_corasick(s, root, patterns)

# Вывод решений

res.sort()

for elem in res:

    print(\*elem)

Название файла: second.py

import logging

import sys

LEVEL = logging.DEBUG

logging.basicConfig(stream=sys.stdout, level=LEVEL)

class AhoNode:

    # Вспомогательный класс для построения дерева

    def \_\_init\_\_(self, symbol=None):

        # Словарь вида key = символ, value = узел, на который нужно перейти при чтении данного символа

        self.goto = {}

        # Список паттернов, для которых этот узел является терминальным

        self.out = []

        # Суффиксная ссылка для данной вершины

        self.fail = None

        self.symbol = symbol

        self.path = []

    def add\_sym(self, symbol, prev):

        self.path.extend(prev.path)

        self.path.append(symbol)

def create\_trie(patterns):

    # Создать бор - дерево паттернов

    root = AhoNode('root')

    root.path.append('root')

    # Добавление слов в бор

    for path in patterns:

        logging.debug(f'Добавление слова {path} в бор')

        node = root

        # Разбиение строки на символы и добавление каждого в бор последовательно, друг за другом

        for ind, symbol in enumerate(path):

            k = 0

            if symbol in node.goto:

                logging.debug(' ' \* ind + f'Символ \'{symbol}\' уже есть, переходим по нему')

            else:

                logging.debug(' ' \* ind + f'Добавление символа \'{symbol}\' и переход по нему')

                prev = node

                k = 1

            node = node.goto.setdefault(symbol, AhoNode(symbol))

            if k == 1:

                node.add\_sym(symbol, prev)

        # В терминальную вершину записываем слово, которое является меткой для этой терминальной вершины

        node.out.append(path)

        logging.debug(' ' \* ind + f'Вершина \'{symbol}\' является терминальной для слова {path}')

        logging.debug('->'.join(node.path))

    return root

def create\_machine(patterns):

    # Функция, создающая автомат Ахо-Корасика.

    # Фактически создает бор и инициализирует fail-функции (они же суффиксные ссылки)

    # всех узлов, обходя дерево в ширину.

    # Создаем бор, инициализируем непосредственных потомков корневого узла

    root = create\_trie(patterns)

    queue = []

    # Обход в ширину

    logging.debug('\n\nПостроение автомата')

    logging.debug('Инициализация суффиксных ссылок для детей корневой вершины')

    for node in root.goto.values():

        logging.debug(f'Символ \'{node.symbol}\' ссылается на корень')

        queue.append(node)

        # Для детей корневого узла суффиксная ссылка ссылается на корень

        node.fail = root

    # Инициализируем остальные узлы

    while len(queue) > 0:

        rnode = queue.pop(0)

        for key, unode in rnode.goto.items():

            depth = 2

            queue.append(unode)

            # rnode - родитель

            # unode - сын

            # Пусть rnode и unode соединены через ребро с символом 'a'

            # Ищем суффиксную ссылку для вершины unode

            logging.debug(' '\*depth + f'Ищем суффиксную ссылку для вершины ' + '->'.join(unode.path))

            # Переходим по уже имеющейся суффиксной ссылке для rnode

            fnode = rnode.fail

            logging.debug(' '\*depth + f'Переходим по суффиксной ссылке для родителя вершины \'{unode.symbol}\' ' + '| ' + '->'.join(unode.path))

            logging.debug(' '\*depth + 'Пришли в ' + '->'.join(fnode.path))

            depth += 2

            # Если из fnode можно пройти по ребру 'a', то суффиксная ссылка для unode

            # будет указывать на вершину в которую мы перейдем

            # В противном случае запускаем рекурсивный поиск вершины

            while fnode is not None and key not in fnode.goto:

                logging.debug(' '\*depth + f'Из вершины \'{fnode.symbol}\' нельзя пройти по ребру \'{key}\' переходим на следующую суффиксную ссылку')

                fnode = fnode.fail

                if fnode:

                    logging.debug(' ' \* depth + 'Пришли в ' + '->'.join(fnode.path))

                else:

                    logging.debug(' '\*depth + 'Пришли в несуществующий узел')

                depth += 2

            # Если подходящей суффиксной ссылки найти не удалось, то ссылаемся на корень дерева

            unode.fail = fnode.goto[key] if fnode else root

            if fnode:

                logging.debug(' '\*depth + f'Из вершины \'{fnode.symbol}\' можно пройти по ребру \'{key}\'')

                logging.debug(' '\*depth + 'Итого суффиксная ссылка для ' + '->'.join(unode.path) + ' это ' + '->'.join(fnode.goto[key].path))

            else:

                logging.debug(' '\*depth + 'Подходящей суффиксной ссылки найти не удалось, поэтому ссылаемся на корень')

                logging.debug(

                    ' ' \* depth + 'Итого суффиксная ссылка для ' + '->'.join(unode.path) + ' это ' + 'root')

            # Делаем unode терминальной вершиной для тех слов,

            # для которых терминальной вершиной являлась суффиксная ссылка

            if unode.fail.out:

                logging.debug(f'Вершина \'{unode.symbol}\' стала терминальной для следующих слов: ' + ','.join(unode.fail.out))

            else:

                logging.debug(

                    f'Вершина \'{unode.symbol}\' не стала терминальной для новых слов')

            unode.out += unode.fail.out

            logging.debug('\n\n')

    return root

def aho\_corasick(s, root, patterns):

    # Находит все возможные подстроки из набора паттернов в строке.

    node = root

    # Списко для записи решения

    solution = []

    # Итерация по строке

    for i in range(len(s)):

        logging.debug('Мы находимся на вершине ' + '->'.join(node.path))

        logging.debug(f'Рассматриваем символ \'{s[i]}\'')

        # Если мы находимся на вершине и нам некуда перейти, то мы переходим по суффиксной ссылке

        while node is not None and s[i] not in node.goto:

            logging.debug(f'Из вершины ' + '->'.join(node.path) + f' мы не можем перейти по ребру \'{s[i]}\', перейдем по суффиксной ссылке')

            node = node.fail

            logging.debug(f'Теперь мы находимся в вершине  ' + '->'.join(node.path) if node else 'Теперь мы находимся в вершине None' )

        # Если мы попали в несуществующий узел, то возвращаемся на корневой узел

        if node is None:

            logging.debug('Пришли в несуществующий узел, возвращаемся в корневую вершину')

            node = root

            continue

        # Если мы стоим на вершине и у нас есть интересующее нас ребро, выполняем переход

        logging.debug(f'Мы можем выполнить переход по ребру \'{s[i]}\', осуществляем')

        node = node.goto[s[i]]

        logging.debug(f'Пришли в ' + '->'.join(node.path))

        # Если вершина, в которую мы пришли, является терминальной для каких-то слов, то записываем в список кортеж вида

        # (позиция, с которой началось вхождение; номер найденной строки)

        if not node.out:

            logging.debug('Данная вершина не является терминальной ни для одного слова')

        for pattern in node.out:

            indexes = [k for k in range(len(patterns)) if patterns[k] == pattern]

            for ind in indexes:

                if (i - len(pattern) + 1, ind) not in solution:

                    logging.debug(f'Строка \'{pattern}\' встретилась в тексте на позиции {i}')

                solution.append((i - len(pattern) + 1, ind))

        logging.debug('\n\n')

    solution = list(set(solution))

    return solution

s = input()

pattern = input()

joker = input()

patterns = []

parts\_of\_pattern = []

counter = 0

str = ''

logging.debug('Разделим строку с джокером на подстроки без него')

# Парсинг строки с джокером

for i in range(len(pattern)):

    if pattern[i] != joker:

        str += pattern[i]

        if i + 1 == len(pattern):

            # Записываем строку из не джокеров в список паттернов

            # Записываем строку из не джокеров и индекс ее первого символа в строке паттерна в словарь

            parts\_of\_pattern.append((str, counter))

            logging.debug(f'\tПодстрока без джокера \'{str}\' обнаружена на позиции {counter}')

            patterns.append(str)

    else:

        if str != '':

            parts\_of\_pattern.append((str, counter))

            patterns.append(str)

            logging.debug(f'\tПодстрока без джокера \'{str}\' обнаружена на позиции {counter}')

        counter = i + 1

        str = ''

logging.debug(f'\nСписок подстрок: {patterns}')

# Создание автомата для работы алгоритма

root = create\_machine(patterns)

# Запуск работы алгоритма

res = aho\_corasick(s, root, patterns)

# Сортировка списка решений по возрастанию

res.sort()

# Инициализация массива из нулей с длиной, равной длине строки, в которой происходит поиск

C = [0]\*len(s)

logging.debug(f'Инициализация массива: {C}')

# Если паттерн с джокером может располагаться на этом месте, то увеличиваем значение на этом месте массива на единицу

logging.debug('\n\n')

for elem in res:

    patpat = (parts\_of\_pattern[elem[1]][1]) \* joker + patterns[elem[1]] + (

                len(pattern) - parts\_of\_pattern[elem[1]][1] - len(patterns[elem[1]])) \* joker

    logging.debug(f'Рассматриваем строку \'{patpat}\', содержащую подстроку \'{patterns[elem[1]]}\' на позиции {elem[0]} текста')

    if len(s) > (elem[0] - parts\_of\_pattern[elem[1]][1]) >= 0 and (elem[0] - parts\_of\_pattern[elem[1]][1] + len(pattern) - 1) < len(s):

        logging.debug(s)

        logging.debug((elem[0] - parts\_of\_pattern[elem[1]][1])\*' '+patpat)

        logging.debug(f'Такая строка может содержаться в тексте')

        logging.debug(f'Увеличим в массиве на единицу значение элемента, чей индекс соответствует индексу,'

                      f' с которого может начинаться строка \'{patpat}\'')

        logging.debug(C)

        C[elem[0] - parts\_of\_pattern[elem[1]][1]] += 1

        logging.debug(C)

    else:

        logging.debug(s)

        logging.debug((elem[0] - parts\_of\_pattern[elem[1]][1]) \* ' ' + patpat)

        logging.debug(f'Такая строка не может содержаться в тексте')

    logging.debug('\n\n')

# Итерация по полученному массиву, если значение на данной позиции равно числу паттернов на которые разбилась строка

# с джокером и если мы не выходим за границы строки, то мы нашли индекс вхождения паттерна с джокером в строку

logging.debug(f'Окончательный вид массива {C}, число паттернов {len(patterns)}')

for i in range(len(s)):

    if (i + len(pattern)) <= len(C) and C[i] == len(patterns):

        print(i + 1)

Название файла: var.py

import logging

import sys

LEVEL = logging.DEBUG

logging.basicConfig(stream=sys.stdout, level=LEVEL)

class AhoNode:

    # Вспомогательный класс для построения дерева

    def \_\_init\_\_(self, symbol=None):

        # Словарь вида key = символ, value = узел, на который нужно перейти при чтении данного символа

        self.goto = {}

        # Список паттернов, для которых этот узел является терминальным

        self.out = []

        # Суффиксная ссылка для данной вершины

        self.fail = None

        self.symbol = symbol

        self.path = []

    def add\_sym(self, symbol, prev):

        self.path.extend(prev.path)

        self.path.append(symbol)

def aho\_create\_forest(patterns):

    # Создать бор - дерево паттернов

    root = AhoNode('root')

    root.path.append('root')

    # Добавление слов в бор

    for path in patterns:

        node = root

        # Разбиение строки на символы и добавление каждого в бор последовательно, друг за другом

        for ind, symbol in enumerate(path):

            k = 0

            if symbol in node.goto:

                pass

            else:

                prev = node

                k = 1

            node = node.goto.setdefault(symbol, AhoNode(symbol))

            if k == 1:

                node.add\_sym(symbol, prev)

        # В терминальную вершину записываем слово, которое является меткой для этой терминальной вершины

        node.out.append(path)

    return root

def aho\_create\_statemachine(patterns):

    # Функция, создающая автомат Ахо-Корасика.

    # Фактически создает бор и инициализирует fail-функции (они же суффиксные ссылки)

    # всех узлов, обходя дерево в ширину.

    # Создаем бор, инициализируем непосредственных потомков корневого узла

    root = aho\_create\_forest(patterns)

    queue = []

    for node in root.goto.values():

        queue.append(node)

        # Для детей корневого узла суффиксная ссылка ссылается на корень

        node.fail = root

    # Инициализируем остальные узлы

    while len(queue) > 0:

        rnode = queue.pop(0)

        for key, unode in rnode.goto.items():

            queue.append(unode)

            # rnode - родитель

            # unode - сын

            # Пусть rnode и unode соединены через ребро с символом 'a'

            # Ищем суффиксную ссылку для вершины unode

            # Переходим по уже имеющейся суффиксной ссылке для rnode

            fnode = rnode.fail

            # Если из fnode можно пройти по ребру 'a', то суффиксная ссылка для unode

            # будет указывать на вершину в которую мы перейдем

            # В противном случае запускаем рекурсивный поиск вершины

            while fnode is not None and key not in fnode.goto:

                fnode = fnode.fail

            # Если подходящей суффиксной ссылки найти не удалось, то ссылаемся на корень дерева

            unode.fail = fnode.goto[key] if fnode else root

            # Делаем unode терминальной вершиной для тех слов,

            # для которых терминальной вершиной являлась суффиксная ссылка

            unode.out += unode.fail.out

    return root

###################################################

def count\_vertex(root):

    # Очередь для обхода в ширину

    queue = []

    # Множество для подсчета вершин

    vertex = set()

    vertex.add(root)

    logging.debug(f'\t\tВершина 1 ' + '->'.join(root.path))

    i = 2

    # Запись дочерних вершин корня в очередь для обхода и во множество

    logging.debug(f'Из вершины root можем перейти в {[elem.symbol for elem in root.goto.values()]}')

    for node in root.goto.values():

        queue.append(node)

        vertex.add(node)

        logging.debug(f'\t\tВершина {i} ' + '->'.join(node.path))

        i += 1

    # Обход в ширину с добавлением просмотренного элемента во множество

    while len(queue) > 0:

        rnode = queue.pop(0)

        logging.debug(f'Из вершины ' + '->'.join(rnode.path) + f' можем перейти в {[elem.symbol for elem in rnode.goto.values()]}')

        for key, node in rnode.goto.items():

            queue.append(node)

            vertex.add(node)

            logging.debug(f'\t\tВершина {i} ' + '->'.join(node.path))

            i += 1

    # Число элементов во множестве = число вершин автомата

    return len(vertex)

###################################################

def aho\_find\_all(s, root, patterns):

    # Находит все возможные подстроки из набора паттернов в строке.

    node = root

    # Списко для записи решения

    solution = []

    # Итерация по строке

    for i in range(len(s)):

        # Если мы находимся на вершине и нам некуда перейти, то мы переходим по суффиксной ссылке

        while node is not None and s[i] not in node.goto:

            node = node.fail

        # Если мы попали в несуществующий узел, то возвращаемся на корневой узел

        if node is None:

            node = root

            continue

        # Если мы стоим на вершине и у нас есть интересующее нас ребро, выполняем переход

        node = node.goto[s[i]]

        # Если вершина, в которую мы пришли, является терминальной для каких-то слов, то записываем в список кортеж вида

        # (позиция, с которой началось вхождение; номер найденной строки)

        for pattern in node.out:

            solution.append((i - len(pattern) + 1, patterns.index(pattern)))

    return solution

def find\_intersection(s, search , patterns, find):

    # Строка, для которой мы должны найти пересечения с другими паттернами в тексте

    string = patterns[search[1]]

    # Индекс начала вхождения данной строки в текст

    string\_start = search[0]

    res = list()

    logging.debug(f'\t\t{s}')

    logging.debug('\t\t'+string\_start\*' ' + string + '\n')

    # Итерация по найденным паттернам в тексте

    for elem in find:

        if patterns[elem[1]] != string:

            logging.debug(f'\t\tПоиск пересечения со строкой {patterns[elem[1]]} с началом на позиции {elem[0]}')

        # Если начало какого-либо паттерна расположено между началом и концом фиксированного паттерна,

        # то мы заносим его в список

        if string\_start <= elem[0] <= (string\_start + len(string) - 1):

            if patterns[elem[1]] != string:

                logging.debug(f'\t\tНайдено пересечение с паттерном {patterns[elem[1]]}')

                logging.debug(f'\t\t{s}')

                logging.debug('\t\t' + string\_start \* ' ' + string)

                logging.debug('\t\t'+ ' '\*elem[0] + patterns[elem[1]])

            res.append(patterns[elem[1]])

        # Если наш фиксированный паттерн расположен между началом и концом какого-то паттерна,

        # то мы заносим его в список

        elif elem[0] < string\_start <= (elem[0] + len(patterns[elem[1]]) - 1):

            if patterns[elem[1]] != string:

                logging.debug(f'\t\tНайдено пересечение с паттерном {patterns[elem[1]]}')

                logging.debug(f'\t\t{s}')

                logging.debug('\t\t'+string\_start \* ' ' + string)

                logging.debug('\t\t'+' ' \* elem[0] + patterns[elem[1]])

            res.append(patterns[elem[1]])

        else:

            if patterns[elem[1]] != string:

                logging.debug(f'\t\tНе найдено пересечение с паттерном {patterns[elem[1]]}')

                logging.debug('\t\t'+s)

                logging.debug('\t\t' +string\_start \* ' ' + string)

                logging.debug('\t\t'+' '\*elem[0] + patterns[elem[1]])

    # Избавляемся от дублирования (чтобы в списке не содержался наш фиксированный паттерн,

    # для которого мы ищем пересечения)

        logging.debug('\n')

    res.remove(string)

    return res

s = input()

n = int(input())

patterns = []

for \_ in range(n):

    patterns.append(input())

###################

# s = 'SHEERHISHERS'

# n = 5

# patterns = ['HIS', 'SHE', 'HER', 'HERS', 'HE']

############################################

root = aho\_create\_statemachine(patterns)

res = aho\_find\_all(s, root, patterns)

res.sort()

print('Найденные вхождения')

for elem in res:

    print(\*elem)

print('-----')

# Словарь для записи ответа

inters = dict()

# Поиск всех вхождений конкретного паттерна в текст

for i in range(n):

    logging.debug(f'Рассматриваем строку {patterns[i]}')

    finds = []

    for elem in res:

        if elem[1] == i:

            finds.append(elem)

    logging.debug(f'Индексы всех вхождений строки {patterns[i]}: {[i[0] for i in finds]}')

    intersections = list()

    # Поиск всех пересечений паттерна, расположенного на определенной позиции в тексте, с другими паттернами

    for elem in finds:

        intersections.extend(find\_intersection(s, elem, patterns, res))

    # Запись полученных данных в словарь

    inters[patterns[i]] = intersections

    logging.debug('\n\n')

for elem in inters:

    print(elem)

    print(inters[elem])

print('--'\*20)

#################################

root = aho\_create\_statemachine(patterns)

print(count\_vertex(root))

#################################