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-Корасика

Студент гр. 1384 _____ Сочков И.С.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p1, \ldots, pn\}$ $1 \leq |pi| \leq 75$

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

Задание 2.

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P 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≤|T|≤100000)

Шаблон (P,1≤|P|≤40)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Выполнение работы.

Класс Vertex представляет вершину бора, который является древовидной структурой данных для хранения и обработки множества строк.

Атрибуты класса Vertex:

- next_vertexes: словарь, хранящий следующие вершины в боре и связующие символы;
- move: словарь, хранящий функцию перехода, указывающую на следующую вершину, которая соответствует заданному символу;

- flag: флаг, указывающий на то, является ли текущая вершина концом какого-либо шаблона;
- pattern_index: индекс шаблона в списке шаблонов Bor, который заканчивается в этой вершине;
- suffix_link: индекс вершины в боре, которая соответствует самому длинному суффиксу текущей строки;
- good_suffix_link: индекс вершины в боре, которая соответствует суффиксу текущей строки и имеет флаг;
- parent: индекс вершины-родителя;
- symbol: символ, который связывает текущую вершину с ее родительской вершиной.

Был создан класс Bor, предназначенный для поиска заданных шаблонов в строке. Содержит следующие атрибуты:

- bor: список объектов класса Vertex, представляющих узлы бора.
- patterns: список строк, представляющих добавленные в бор образцы (паттерны).
- result: список кортежей, представляющих найденные в тексте сочетания образцов и позиции, в которых они встретились.

Класс Bor имеет следующие методы:

1. __init__(self): конструктор класса, инициализирующий пустой бор, список шаблонов и результат поиска.
2. add_to_bor(self, pattern: str) -> None: добавляет шаблон в бор. Для этого метод последовательно проходит по символам шаблона, создавая новые вершины и связи в боре, если таких еще нет. После прохода по всем символам метод помечает последнюю вершину, соответствующую концу шаблона, как финальную.
3. get_suffix_link(self, vertex: int) -> int: возвращает суффиксную ссылку для заданной вершины. Суффиксная ссылка - это ссылка на самую длинную подстроку шаблона, совпадающую с префиксом данной вершины, за

исключением этого префикса. Для вычисления суффиксной ссылки метод использует рекурсивный алгоритм, переходя от данной вершины к ее родителю и далее по бору, пока не найдет первую вершину, имеющую связь с текущим символом шаблона.

4. get_good_suffix_link(self, vertex: int) -> int: возвращает хорошую суффиксную ссылку для заданной вершины. Хорошая суффиксная ссылка - это ссылка на первую вершину с флагом (последнюю вершину шаблона), находящуюся по пути суффиксных ссылок от данной вершины. Если такой вершины нет, то ссылка устанавливается на вершину, находящуюся на один уровень выше в иерархии бора.

5. get_move(self, vertex: int, symbol: str) -> int: возвращает вершину, соответствующую переходу из заданной вершины по заданному символу. Если такой вершины нет, метод вычисляет суффиксную ссылку для данной вершины и выполняет переход из нее по тому же символу.

6. find(self, text: str) -> None: находит все вхождения шаблонов в заданной строке и сохраняет их позиции в результирующем списке. Для каждого символа входной строки он проходит от текущей вершины бора вниз по дереву, переходя от вершины к вершине по символам строки. Если текущая вершина является финальной для какого-то шаблона, то соответствующее вхождение добавляется в список результатов. Затем он переходит к следующему символу входной строки и продолжает поиск с новой текущей вершины.

7. get_answer(self, text: str) -> List[Tuple[int, int]]: принимает строку text и вызывает метод find, чтобы заполнить список результатов. Затем он сортирует список результатов и возвращает его. Каждый элемент списка представляет собой кортеж из индекса начала вхождения в строку text и индекса шаблона в списке добавленных шаблонов.

Код решения первой задачи представлен в приложении А, в файле task_1.py.

Для решения второй задачи потребовалось модифицировать функцию get_result и создать функцию build_bor, добавляющую подстроки шаблона в бор, разделяя её части по символу-джокеру.

Был модифицирован метод find класса Bor – теперь он сохраняет позиции вхождения шаблонов в тексте, а также позицию этих строк в строке-паттерне.

Итоговый код также представлен в task_2.py (см. в Приложении А).

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 — Результаты тестирования

Входные данные	Выходные данные	Комментарии
NTAG 3 TAGT TAG T	2 2 2 3	Работа программы task_1.py
ACTANCA A$$A$ $	1	Работа программы task_2.cpp

Выводы.

В рамках проведенной лабораторной работы были разработаны программы, основанные на алгоритме Ахо-Корасик, для решения двух задач: точный поиск набора образцов в строке и поиск всех вхождений строки, содержащей символ-джокер, в тексте. Для разработки программ использовался язык программирования Python. Программы были успешно протестированы на платформе Stepik. В результате тестирования было подтверждено, что программы работают корректно и эффективно выполняют свои задачи. Этот опыт помог улучшить понимание алгоритмов поиска в строках и структур данных в целом, а также улучшить навыки программирования на Python.

ПРИЛОЖЕНИЕ А
КОД ДЛЯ РЕШЕНИЯ ЗАДАЧ

task_1.py:

```
from typing import Dict, List, Tuple

'''
Класс Vertex представляет вершину бора, который является древовидной
структурой данных для хранения и обработки множества строк.

Атрибуты класса Vertex:
next_vertexes: словарь, хранящий следующие вершины в боре и связующие символы
move: словарь, хранящий функцию перехода, указывающую на следующую вершину,
которая соответствует заданному символу
flag: флаг, указывающий на то, является ли текущая вершина концом какого-либо шаблона
pattern_index: индекс шаблона в списке шаблонов Bor, который заканчивается в этой вершине
suffix_link: индекс вершины в боре, которая соответствует самому длинному суффиксу текущей строки
good_suffix_link: индекс вершины в боре, которая соответствует суффиксу текущей строки и имеет флаг
parent: индекс вершины-родителя
symbol: символ, который связывает текущую вершину с ее родительской вершиной
'''
class Vertex:
    def __init__(self, parent:int, symbol:str) -> None:
        self.next_vertexes: Dict[str, int] = {}
        self.move: Dict[str, int] = {}
        self.flag = False
        self.pattern_index = -1
        self.suffix_link = -1
        self.good_suffix_link = -1
        self.parent = parent
        self.symbol = symbol

'''
```

```
    Класс Bor предназначен для поиска заданных шаблонов в строке.

    Содержит следующие атрибуты:
    bor: список объектов класса Vertex, представляющих узлы бора.
    patterns: список строк, представляющих добавленные в бор образцы
(паттерны).
    result: список кортежей, представляющих найденные в тексте сочетания
образцов и позиции,
    в которых они встретились.
    '''
    class Bor:
        #конструктор класса, инициализирующий пустой бор, список шаблонов и
результат поиска
        def __init__(self) -> None:
            self.bor: List[Vertex] = [Vertex(0, '$')]
            self.patterns: List[str] = []
            self.result: List[Tuple[int, int]] = []

        #метод добавления шаблона в бор
        def add_to_bor(self, pattern: str) -> None:
            number = 0
            for symbol in pattern:
                if symbol not in self.bor[number].next_vertexes:
                    self.bor.append(Vertex(number, symbol))
                    self.bor[number].next_vertexes[symbol] = len(self.bor)
- 1
                number = self.bor[number].next_vertexes[symbol]

            self.bor[number].flag = True
            self.patterns.append(pattern)
            self.bor[number].pattern_index = len(self.patterns) - 1

        #метод поиска суффиксной ссылки для заданной вершины бора
        def get_suffix_link(self, vertex: int) -> int:
            if self.bor[vertex].suffix_link == -1:
                if vertex == 0 or self.bor[vertex].parent == 0:
                    self.bor[vertex].suffix_link = 0
                else:
```

```
                    self.bor[vertex].suffix_link =
self.get_move(self.get_suffix_link(self.bor[vertex].parent),
self.bor[vertex].symbol)
            return self.bor[vertex].suffix_link

        #метод поиска суффиксной ссылки для заданной вершины бора
        def get_good_suffix_link(self, vertex: int) -> int:
            if self.bor[vertex].good_suffix_link == -1:
                current_suffix_link = self.get_suffix_link(vertex)
                if current_suffix_link == 0:
                    self.bor[vertex].good_suffix_link = 0
                else:
                    self.bor[vertex].good_suffix_link = current_suffix_link
if self.bor[current_suffix_link].flag else
self.get_good_suffix_link(current_suffix_link)
            return self.bor[vertex].good_suffix_link

        #метод получения вершины, соответствующего переходу из заданной
вершины по заданному символу
        def get_move(self, vertex: int, symbol: str) -> int:
            if symbol not in self.bor[vertex].move:
                if symbol in self.bor[vertex].next_vertexes:
                    self.bor[vertex].move[symbol] =
self.bor[vertex].next_vertexes[symbol]
                else:
                    self.bor[vertex].move[symbol] = 0 if vertex == 0 else
self.get_move(self.get_suffix_link(vertex), symbol)
            return self.bor[vertex].move[symbol]

        #метод поиска всех вхождений шаблонов в заданной строке и сохранения
их позиций в результирующем списке
        def find(self, text: str) -> None:
            vertex_number = 0
            for text_index, symbol in enumerate(text):
                vertex_number = self.get_move(vertex_number, symbol)
                vertex = vertex_number
                while vertex != 0:
                    if self.bor[vertex].flag:
```

```
                        self.result.append((text_index + 2 -
len(self.patterns[self.bor[vertex].pattern_index]),
self.bor[vertex].pattern_index + 1))
                    vertex = self.get_good_suffix_link(vertex)

        #метод решения задачи
        def get_answer(self, text: str) -> List[Tuple[int, int]]:
            self.find(text)
            self.result.sort(key=lambda x: (x[0], x[1]))
            return self.result

    #Запуск программы
    if __name__ == "__main__":
        bor = Bor()
        text = input()
        patternQuantity = int(input())
        for _ in range(patternQuantity):
            pattern = input()
            bor.add_to_bor(pattern)
        result = bor.get_answer(text)
        for i in range(len(result)):
            print(*result[i])
```

task_2.py:

```
    from typing import List, Tuple

    '''
    Класс Vertex представляет вершину бора, который является древовидной
    структурой данных для хранения и обработки множества строк.

    Атрибуты класса Vertex:
    next_vertexes: словарь, хранящий следующие вершины в боре и связующие
символы
    move: словарь, хранящий функцию перехода, указывающую на следующую
вершину,
    которая соответствует заданному символу
    flag: флаг, указывающий на то, является ли текущая вершина концом
какого-либо шаблона
    pattern_index: индекс шаблона в списке шаблонов Bor, который
заканчивается в этой вершине
    suffix_link: индекс вершины в боре, которая соответствует самому
длинному суффиксу текущей строки
    good_suffix_link: индекс вершины в боре, которая соответствует
суффиксу текущей строки и имеет флаг
```

```
    parent: индекс вершины-родителя
    symbol: символ, который связывает текущую вершину с ее родительской
вершиной
    '''
    class Vertex:
        def __init__(self, parent: int, symbol: str):
            self.next_vertexes = {}
            self.move = {}
            self.flag = False
            self.pattern_indexes = []
            self.suffix_link = -1
            self.good_suffix_link = -1
            self.parent = parent
            self.symbol = symbol

    '''
    Класс Bor предназначен для поиска заданных шаблонов в строке.

    Содержит следующие атрибуты:
    bor: список объектов класса Vertex, представляющих узлы бора.
    patterns: список строк, представляющих добавленные в бор образцы
(паттерны).
    result: список кортежей, представляющих найденные в тексте сочетания
образцов и позиции,
    в которых они встретились.
    pattern_positions: список позиций каждого образца в строке, где
образец встретился впервые.
    occurrence: список количества вхождений каждого образца в строке.
    found_patterns: список строк, найденных в тексте в процессе поиска
образцов.
    '''
    class Bor:
        #конструктор класса, инициализирующий пустой бор, список шаблонов
и результат поиска
        def __init__(self):
            self.bor = [Vertex(0, '$')]
            self.patterns = []
            self.result = []
            self.pattern_positions = []
            self.occurrence = []
            self.found_patterns = []

        #метод добавления шаблона в бор
        def add_to_bor(self, pattern: str) -> None:
            number = 0
            for symbol in pattern:
                if symbol not in self.bor[number].next_vertexes:
                    self.bor.append(Vertex(number, symbol))
                    self.bor[number].next_vertexes[symbol] = len(self.bor)
- 1
```

```
                number = self.bor[number].next_vertexes[symbol]
            self.bor[number].flag = True
            self.patterns.append(pattern)
            self.bor[number].pattern_indexes.append(len(self.patterns) -
1)

        #метод поиска суффиксной ссылки для заданной вершины бора
        def get_suffix_link(self, vertex: int) -> int:
            if self.bor[vertex].suffix_link == -1:
                if vertex == 0 or self.bor[vertex].parent == 0:
                    self.bor[vertex].suffix_link = 0
                else:
                    self.bor[vertex].suffix_link =
self.get_move(self.get_suffix_link(self.bor[vertex].parent),
self.bor[vertex].symbol)
            return self.bor[vertex].suffix_link

        #метод поиска суффиксной ссылки для заданной вершины бора
        def get_good_suffix_link(self, vertex: int) -> int:
            if self.bor[vertex].good_suffix_link == -1:
                current_suffix_link = self.get_suffix_link(vertex)
                if current_suffix_link == 0:
                    self.bor[vertex].good_suffix_link = 0
                else:
                    self.bor[vertex].good_suffix_link =
current_suffix_link if self.bor[current_suffix_link].flag else
self.get_good_suffix_link(current_suffix_link)
            return self.bor[vertex].good_suffix_link

        #метод получения вершины, соответствующего переходу из заданной
вершины по заданному символу
        def get_move(self, vertex: int, symbol: str) -> int:
            if symbol not in self.bor[vertex].move:
                if symbol in self.bor[vertex].next_vertexes:
                    self.bor[vertex].move[symbol] =
self.bor[vertex].next_vertexes[symbol]
                else:
                    if vertex == 0:
                        self.bor[vertex].move[symbol] = 0
                    else:
                        self.bor[vertex].move[symbol] =
self.get_move(self.get_suffix_link(vertex), symbol)
            return self.bor[vertex].move[symbol]

        #метод поиска всех вхождений шаблонов в заданной строке и
сохранения их позиций в
        #результирующем списке
        def find(self, text: str) -> None:
            vertex_number = 0
            for text_index in range(len(text)):
```

```
                vertex_number = self.get_move(vertex_number,
text[text_index])
                vertex = vertex_number
                while vertex != 0:
                    if self.bor[vertex].flag:
                        for number in self.bor[vertex].pattern_indexes:
                            position_in_text = text_index + 1 -
len(self.patterns[number])
                            self.found_patterns.append((position_in_text,
number))
                    vertex = self.get_good_suffix_link(vertex)

    #Функция построения бора шаблона с символом-джокером
    def build_bor(pattern: str, joker: str, bor: Bor):
        pattern_part = ""
        for pattern_index in range(len(pattern)):
            if pattern[pattern_index] != joker:
                pattern_part += pattern[pattern_index]
                if pattern_index == len(pattern) - 1 or
pattern[pattern_index+1] == joker:
                    bor.add_to_bor(pattern_part)
                    bor.pattern_positions.append(pattern_index -
len(pattern_part) + 2)
                    pattern_part = ""

    #Функция поиска шаблона в тексте с учетом символа-джокера
    def get_result(bor: Bor, text: str, pattern: str):
        bor.find(text)
        bor.occurrence = [0] * len(text)
        for i in range(len(bor.found_patterns)):
            text_position = bor.found_patterns[i][0]
            substring_index =
bor.pattern_positions[bor.found_patterns[i][1]]
            index = text_position - substring_index + 1

            if index + len(pattern) > len(text):
                continue
            if text_position >= substring_index - 1:
                bor.occurrence[index] += 1

        for i in range(len(bor.occurrence)):
            if bor.occurrence[i] == len(bor.patterns):
                print(i + 1)

    #Запуск программы
    if __name__ == "__main__":
        bor = Bor()
        text = input()
        pattern = input()
        joker = input()
```

```
build_bor(pattern, joker, bor)
get_result(bor, text, pattern)
```