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

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Ознакомиться с алгоритмом Ахо-Корасик поиска набора образцов в строке, применить его в решении поставленных задач.

Задание.

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 $P = \{p_1,\ldots,p_n\}$ $1 \leq |p_i| \leq 75$.

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

Выход:

Все вхождения образцов из P в T.

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

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

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

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.

Основные теоретические положения.

Алгоритм Ахо-Корасик (АК) - классическое решение задачи точного сопоставления множеств.

АК основан на структуре данных "дерево ключевых слов".

Поиск строки S в бору: начинаем в корне, идем по ребрам, отмеченным символами S, пока возможно.

Если с последним символом S мы приходим в вершину с сохраненным идентификатором, то S - слово из словаря.

Если в какой-то момент ребра, отмеченного нужным символом, не находится, то строки S в словаре нет.

Ясно, что это занимает O (|S|) времени. Таким образом, бор - это эффективный способ хранить словарь и искать в нем слова.

Теперь перейдем от бора к автомату, чтобы добиться поиска шаблонов в тексте за линейное время.

Действия автомата определяются тремя функциями, определенными для всех состояний:

Функция goto g(s, a) указывает, в какое состояние переходить из данного состояния s при просмотре символа a.

Функция неудачи f(s) указывает, в какое состояние переходить при просмотре неподходящего символа.

Выходная функция out(s) выдает множество шаблонов, которые обнаруживаются при переходе в состояние s.

Теорема. Поиск шаблонов в тексте T [1…m] с помощью автомата АК занимает O(m + z) времени, где z - количество появлений шаблонов.

Выполнение работы.

Для решения поставленных задач определения вхождения подстроки в строку был реализован алгоритм Ахо-Корасик.

Для реализации алгоритма был создан вспомогательный класс, описывающий узлы дерева ключевых слов – бора.

class NodePatternTree. Как описано раннее, данный класс описывает узлы бора. Представлен полями: *leaves*, *plenty_patterns*, *node_if_fail*. Словарь *leaves* – указывает, в какой узел переходить из узла по ключу при просмотре символа-ключа. Список *plenty_patterns* – множество шаблонов, которые обнаруживаются при переходе по символу. Переменная *node_if_fail* – узел для перехода при просмотре неподходящего символа.

Основные функции:

1) *def create_pattern_tree_aho(patterns)*. Функция, создающая бор – дерево паттернов. На вход подается набор шаблонов, по которым будет строиться дерево. С помощью вспомогательного класса, описанного выше, создается узел для бора. Соответствующие поля класса обновляются для каждого узла: в словарь *leaves* добавляются символы из шаблонов, по которым будут храниться другие узлы для перехода, в список *plenty_patterns* добавляется номер обрабатываемого на данном этапе шаблона. Функция возвращает корень дерева паттернов – бора.

2) *def create_aho_statemachine(patterns)*. Функция, создающая автомат Ахо-Корасик, а именно дерево паттернов и инициализирующая вершины для перехода при просмотре неподходящих символов. Создается очередь для

просмотра каждой вершины и дальнейшей инициализации. Переменная *node_if_fail*, узел для перехода при просмотре неподходящего символа, и список *plenty_patterns* с множеством шаблонов вычисляются для всех вершин в порядке обхода в ширину. Рассмотрим узлы *current_node* и *symbol, leaf in zip(current_node.leaves.keys(), current_node.leaves.values())*, то есть *current_node* – родитель *leaf.* Нужно, чтобы *leaf.node_if_fail* указывало на ту вершину, метка которой является самым длинным суффиксом для *leaf*, являющимся также началом некоторого шаблона из множества *P*. Эта вершина ищется путем просматривания вершин, метки которых являются все более и более короткими суффиксами *current_node*, пока не находится вершина, для которой определен словарь *leaves*. Тогда *leaf.node_if_fail* присваивается найденная вершина. Теперь *leaf.plenty_patterns += leaf.node_if_fail.plenty_patterns.* Это делается потому, что все шаблоны, распознаваемые при переходе в узел *leaf.node_if_fail* являются надлежащими суффиксами *leaf* и должны быть отслежены при переходе в данный узел. Функция возвращает корень дерева паттернов – бора.

3) *def find_all_substrings_aho_first(text, root, patterns).* Функция, находящая все возможные вхождения шаблонов в текст для первого задания. Принимает на вход текст, корень бора и набор шаблонов. Создается список *substrings_in_text*, в который будут помещаться индекс вхождения подстроки в текст и сам номер строки. Осуществляется проход по бору до тех пор, пока символ текста на текущей итерации не окажется среди листьев текущего узла. При нахождении такого узла для него рассматривается список множества шаблонов, которые будут записаны в список *substrings_in_text.* Функция возвращает список, содержащий шаблоны с их индексами вхождения в текст.

4) *def find_all_substrings_aho_second(text, root, patterns).* Функция, находящая все возможные вхождения шаблонов в текст для второго задания. Принимает на вход текст, корень бора и набор шаблонов. Работает так же, как и функция, описанная выше. Различие заключается при добавлении элементов в список *substrings_in_text*, элементы не будут увеличиваться на константные

значения, так как для вывода ответа потребуются дополнительные преобразования.

5) *def create_all_patterns(pattern, joker_symbol)*. Функция, создающая наборы шаблонов для задания с символом джокером. На вход подается единый шаблон и символ джокер. Для того, чтобы алгоритм для второго задания работал так же, как и для первого, единый шаблон необходимо разделить на несколько шаблонов, где разделителем будет выступать введенный символ джокер. Также необходимо запомнить индексы в начальном едином шаблоне для каждого из полученных новых шаблонов. Функция возвращает набор шаблонов и соответствующие им начальные индексы в едином шаблоне.

6) *def calculate_substrings(text, start_indices_in_pattern, substrings_in_text).* Функция, считающая, количество безмасочных подстрок, встретившихся в тексте. Принимает на вход текст, начальные индексы шаблонов в едином шаблоне, список, хранящий шаблон и его индекс вхождения в тексте. Для алгоритма понадобится список *count_substring_without_mask,* где *count_substring_without_mask[i]* – количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции *i*. Появление *i*-ой подстроки в тексте на позиции *j* будет означать возможное появление единого шаблона *pattern* на позиции *j-start_indices_in_pattern[pattern]*. То есть при нахождении *i*-ой подстроки в тексте на позиции *j* увеличиваем на единицу список, объявленный раннее в данном пункте, по индексу, равному позиции возможного появления единого шаблона. Функция возвращает список, хранящий по индексу количество встретившихся подстрок.

7) *def print_first_answer(patterns_in_text).* Функция, выводящая ответ на первое задание. Принимает на вход список, содержащий шаблоны с их индексами вхождения в текст. В начале список сортируется по возрастанию по двум значениям в следующем порядке: номер позиции, затем номер шаблона. Ответ выводится на экран через пробел.

8) *def print_second_answer(count_substrings_by_index, pattern, all_patterns).* Функция, выводящая ответ на второе задание. Принимает на вход список, хранящий по индексу количество встретившихся подстрок. Каждое *i,* для которого *count_substrings_by_index[i] == len(all_patterns)*, является стартовой позицией появления шаблона в тексте. Такое значение, увеличенное на единицу, так как нумерация согласно заданию, начинается с одного, выводится на экран.

9) *def solve_first_task().* Функция, запускающая решение первого задания. Осуществляется считывание текста, количество шаблонов и сами шаблоны. Для решения задачи запускаются следующие функции, которые были описаны раннее: *create_aho_statemachine(), find_all_substrings_aho_first(), print_first_answer().*

10) *def solve_second_task().* Функция, запускающая решение второго задания. Осуществляется считывание текста, шаблона и символа джокера. Для решения задачи запускаются следующие функции, которые были описаны раннее: *create_all_patterns(), create_aho_statemachine(), find_all_substrings_aho_second(), calculate_substrings(), print_second_answer().*

Разработанный программный код смотреть в приложении А.

Выводы.

В ходе лабораторной работы был изучен алгоритм Ахо-Корасик. Разработан программный код, позволяющий решить следующие задачи с помощью данного алгоритма: точный поиск набора образцов в тексте и поиск для одного образца с джокером. На языке программирования *Python* реализованы функции, представляющие собой решение поставленных задач.

Для работы алгоритма Ахо-Корасик изучена структура данных «дерево ключевых слов» или бор. Реализован класс, описывающий узел такого дерева, и функция, строящее такое дерево. Для нахождения шаблонов в тексте реализован автомат Ахо-Корасик, состояниями которого являются узлы бора.

Для решения задачи поиска одного образца с джокером была осуществлена предобработка образца перед началом алгоритма. Образец

делился на подстроки по разделителю, которым являлся символ джокера, а также для получившихся подстрок сохранялись их стартовые позиции в образце. Это позволяет решать данную задачу по такому же принципу, как и первую.

Разработанный программный код для решения поставленных задач успешно прошел тестирование на онлайн платформе *Stepik*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
    # Класс, описывающий узлы для построения дерева ключевых слов(узлы бора)
    # Поля класса: leaves указывает, в какие узлы переходить
    # plenty_patterns множество шаблонов, обнаруживающиеся при переходе по символу
    # node_if_fail узел для перехода при просмотре неподходящего символа
    class NodePatternTree:
        # Метод, инициализирующий поля класса
        def __init__(self):
            self.leaves = {}
            self.plenty_patterns = []
            self.node_if_fail = None

    # Функция, создающая бор - дерево паттернов
    # Принимает на вход набор шаблонов
    # Возвращает корень дерева паттернов - бора
    def create_pattern_tree_aho(patterns):
        root = NodePatternTree()
        for index, string in enumerate(patterns):
            node = root
            for symbol in string:
                node = node.leaves.setdefault(symbol, NodePatternTree())
            node.plenty_patterns += [index]
        return root

    # Функция, реализующая автомат Ахо-Корасик, а именно
    # создается дерево паттернов и инициализируются
    # узлы для перехода при просмотре неподходящих символов
    # Принимает на вход набор шаблонов
    # Возвращает корень дерева паттернов - бора
    def create_aho_statemachine(patterns):
        root = create_pattern_tree_aho(patterns)
```

```
        nodes_queue = []
        for node in root.leaves.values():
            nodes_queue += [node]
            node.node_if_fail = root
        while len(nodes_queue) > 0:
            current_node = nodes_queue.pop(0)
            for symbol, leaf in zip(current_node.leaves.keys(),
current_node.leaves.values()):
                nodes_queue += [leaf]
                fail_node = current_node.node_if_fail
                while fail_node is not None and symbol not in
fail_node.leaves:
                    fail_node = fail_node.node_if_fail
                leaf.node_if_fail = fail_node.leaves[symbol] if
fail_node else root
                leaf.plenty_patterns +=
leaf.node_if_fail.plenty_patterns
        return root

    # Функция, находящая все возможные вхождения шаблонов в текст(1
задание)
    # Принимает на вход текст, корень бора и набор шаблонов
    # Возвращает список, содержащий шаблоны с их индексами вхождения в
текст
    def find_all_substrings_aho_first(text, root, patterns):
        current_node = root
        substrings_in_text = []
        for i in range(len(text)):
            while current_node is not None and text[i] not in
current_node.leaves:
                current_node = current_node.node_if_fail
            if current_node is None:
                current_node = root
                continue
            current_node = current_node.leaves[text[i]]
            for pattern in current_node.plenty_patterns:
                substrings_in_text += [[i - len(patterns[pattern]) + 2,
pattern + 1]]
        return substrings_in_text
```

```
    # Функция, находящая все возможные вхождения шаблонов в текст(2 задание)
    # Принимает на вход текст, корень бора и набор шаблонов
    # Возвращает список, содержащий шаблоны с их индексами вхождения в текст
    def find_all_substrings_aho_second(text, root, patterns):
        current_node = root
        substrings_in_text = []
        for i in range(len(text)):
            while current_node is not None and text[i] not in current_node.leaves:
                current_node = current_node.node_if_fail
            if current_node is None:
                current_node = root
                continue
            current_node = current_node.leaves[text[i]]
            for pattern in current_node.plenty_patterns:
                substrings_in_text += [[i - len(patterns[pattern]) + 1, pattern]]
        return substrings_in_text

    # Функция, создающая наборы шаблонов для задания с символом джокером
    # Принимает на вход единый шаблон и символ джокер
    # Возвращает набор шаблонов и соответствующие им стартовые индексы в едином шаблоне
    def create_all_patterns(pattern, joker_symbol):
        patterns = list(filter(None, pattern.split(joker_symbol)))
        start_indices_in_pattern = []
        start_index = 0
        for string in patterns:
            current_section = pattern[start_index:]
            current_index = current_section.index(string)
            start_indices_in_pattern += [current_index + (len(pattern)-len(current_section))]
            start_index = current_index + len(string) + (len(pattern)-len(current_section))
        return patterns, start_indices_in_pattern
```

```
    # Функция, считающая количество безмасочных подстрок, встретившихся
в тексте
    # Принимает на вход текст, стартовые индексы шаблонов в едином
шаблоне,
    # список, хранящий шаблоны и их индексы вхождения в тексте
    # Возвращает список, хранящий по индексу количество встретившихся
подстрок
    def calculate_substrings(text, start_indices_in_pattern,
substrings_in_text):
        count_substring_without_mask = [0]*len(text)
        for index_in_text, pattern in substrings_in_text:
            current_index = index_in_text -
start_indices_in_pattern[pattern]
            if current_index >= 0 and current_index <
len(count_substring_without_mask):
                count_substring_without_mask[current_index] += 1
        return count_substring_without_mask

    # Функция, выводящая ответ на первое задание
    # Принимает на вход список, содержащий
    # шаблоны с их индексами вхождения в текст
    def print_first_answer(patterns_in_text):
        patterns_in_text.sort(key=lambda value:(value[0], value[1]))
        for i in range(len(patterns_in_text)):
            print(*patterns_in_text[i], sep = ' ')

    # Функция, выводящая ответ на второе задание
    # Принимает на вход список, хранящий по индексу
    # количество встретившихся подстрок
    def print_second_answer(count_substrings_by_index, pattern,
all_patterns):
        for i in range(len(count_substrings_by_index)-len(pattern)+1):
            if count_substrings_by_index[i] == len(all_patterns):
                print(i+1)

    # Функция, запускающая решение первого задания
    def solve_first_task():
        text = input()
```

```
        count_pattern = int(input())
        patterns = []
        for _ in range(count_pattern):
            patterns += [input()]
        root_pattern_tree = create_aho_statemachine(patterns)
        patterns_in_text  =  sorted(find_all_substrings_aho_first(text,
root_pattern_tree, patterns))
        print_first_answer(patterns_in_text)

    # Функция, запускающая решение второго задания
    def solve_second_task():
        text = input()
        pattern = input()
        joker_symbol = input()
        all_patterns,            start_indices_in_pattern            =
create_all_patterns(pattern, joker_symbol)
        root_pattern_tree = create_aho_statemachine(all_patterns)
        substrings_in_text = sorted(find_all_substrings_aho_second(text,
root_pattern_tree, all_patterns))
        count_substrings_without_mask    =    calculate_substrings(text,
start_indices_in_pattern, substrings_in_text)
        print_second_answer(count_substrings_without_mask,     pattern,
all_patterns)

    # Условие для запуска программы
    if __name__ == "__main__":
        solve_first_task()
        solve_second_task()
```