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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8382 _____ Нечепуренко Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Разработать программу для точного поиска вхождения набора образцов в тесте с использованием алгоритма Ахо-Корасик.

Постановка задачи.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n ($1 \leq n \leq 3\,000$), каждая следующая из n строк содержит шаблон из набора $P = \{p_1, \ldots, p_n\}$, $1 \leq |p_i| \leq 75$. Все строки содержат символы из алфавита {A,C,G,T,N}

Выход:

Все вхождения образцов из P в T.

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

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

(нумерация образцов начинается с 1).

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

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3.

Индивидуальное задание: вариант 2 – подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Описание алгоритма.

Алгоритм Ахо-Корасик позволяет за линейное время найти все позиции вхождений набора образцов в тексте. Из образцов строится детерминированный автомат распознаватель с суффиксными ссылками. Такая структура данных называется Trie (бор). После построения бора происходит итерация по символам текста и, если из текущего состояния автомата можно перейти дальше по текущему символу текста, то происходит переход по ребру, иначе – по суффиксной ссылке. Суффиксная ссылка указывает на узел, путь до которого представляет наибольший суффикс полученного пути. Если на очередной итерации по тексту было достигнуто терминальное состояние автомата – данная позиция означает окончание вхождение данного образца.

Алгоритм Ахо-Корасик может быть использован для поиска по тексту вхождений шаблона со специальным символом (джокером), означающим любой символ алфавита. Для этого необходимо разбить шаблон по джокеру на подстроки, запомнить для каждой подстроки её смещение в исходном шаблоне. Затем найти индексы вхождения всех подстрок в тексте с учётом полученного смещения. Индекс, в котором число найденных подстрок равно их количеству, является индексом вхождения шаблона в тексте. Исходный код программы для поиска шаблона с джокером расположен в Приложении Б.

Реализация алгоритма.

Для реализации узла бора был написан класс TrieNode, хранящий в себе словарь goto, отображение из символа в другой узел, для перехода по бору, out – подстроки, оканчивающиеся на данном узле, fail – суффиксная ссылка на узел и pNumber – номер образца. Полный исходный код расположен в Приложении

A. В функции aho_create_forest единственным аргументом patterns является список образцов. Функция строит дерево для бора.

Функция aho_create_statemachine принимает тот же список образцов, вызывает aho_create_forest, а затем поиском в ширину расставляет суффиксные ссылки. Функция aho_find_all принимает три аргумента: строку с текстом text, корень полученного на предыдущих шагах бора root и словарь соответствий образца и его номера number для формирования ответа.

Функции get_nodes_count и print_trie принимают корень бора и обходом в ширину считают количество узлов или выводят их соответственно.

Для определения пересекающихся подстрок в функции aho_find_all имеется set overlapping, куда добавляется пара индексов при переходе по суффиксной ссылке, не ведущей в корень.

Оценка сложности программы.

Построение бора занимает O(m), где m – сумма длин образцов. Сам алгоритм проходит по тексту один раз, таким образом, общая асимптотика решения O(T + m + n*q), где T – длина текста, n – число вхождений, q – мощность алфавита.

Тестирование.

Проведем тестирование программы для нахождения всех вхождений набора образцов тексте и выводящую согласно индивидуальному варианту число узлов в боре и список пересекающихся образцов. Результат представлен в таблице 1.

Таблица 1 – Тестирование первой программы

Входные данные	Выходные данные
упопабыласобакаонеёлюбил 4 упо опаб ласо	[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0] [Node: goto=dict_keys(['п']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=1]

собак	[Node: goto=dict_keys(['п']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys(['а']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=3] [Node: goto=dict_keys(['о']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=4] [Node: goto=dict_keys(['о']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=1] [Node: goto=dict_keys(['а']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys(['с']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=3] [Node: goto=dict_keys(['б']), suff=[Node: goto=dict_keys(['п']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2], pNumber=4] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['п']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2], pNumber=1] [Node: goto=dict_keys(['б']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys(['о']), suff=[Node: goto=dict_keys(['о']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=4], pNumber=3] [Node: goto=dict_keys(['а']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=4] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['б']), suff=[Node:

	goto=dict_keys(['п']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=2], pNumber=4], pNumber=3] [Node: goto=dict_keys(['к']), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=4] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['у', 'о', 'л', 'с']), suff=None, pNumber=0], pNumber=4] Nodes in trie: 17 Answer: 1 1 3 2 8 3 10 4 Overlapping strings: {('упо', 'опаб'), ('ласо', 'собак')}
ahokorasik 3 hoko koras sik	[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0] [Node: goto=dict_keys(['o']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=1] [Node: goto=dict_keys(['o']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys(['i']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=3] [Node: goto=dict_keys(['k']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=1] [Node: goto=dict_keys(['r']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys(['k']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=3] [Node: goto=dict_keys(['o']), suff=[Node: goto=dict_keys(['o']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=2], pNumber=1] [Node: goto=dict_keys(['a']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None,

	pNumber=0], pNumber=2] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['o']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=2], pNumber=3] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['r']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=2], pNumber=1] [Node: goto=dict_keys(['s']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=2] [Node: goto=dict_keys([]), suff=[Node: goto=dict_keys(['i']), suff=[Node: goto=dict_keys(['h', 'k', 's']), suff=None, pNumber=0], pNumber=3], pNumber=2] Nodes in trie: 13 Answer: 2 1 4 2 8 3 Overlapping strings: {('hoko', 'koras'), ('koras', 'sik')}

Протестируем программу для поиска шаблона с джокером, результат приведён в таблице 2. Для экономии места бор не выводится.

Таблица 2 – Тестирование второй программы

Входные данные	Выходные данные
ACACAACACCCACCACACAC ACAXXCAXXACAXAC X	String biases are: {'ACA': {0, 9}, 'AC': {13}, 'CA': {5}} Got dp array: [1, 2, 1, 1, 0, 4, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0] 6
abacababa a??b ?	String biases are: {'b': {3}, 'a': {0}} Got dp array: [1, 0, 2, 0, 2, 0, 1, 0, 1] 3 5

123452345432 ?4?2 ?	String biases are: {'2': {3}, '4': {1}} Got dp array: [0, 0, 2, 0, 0, 0, 1, 0, 2, 0, 0, 0] 3 9

Обе программы успешно прошли тестирование, больше тестов расположено в директории Tests.

Выводы.

В ходе выполнения лабораторной работы была реализована программа для поиска всех вхождения образцов в текст с использованием алгоритма Ахо-Корасик. Также была реализована программа для нахождения вхождений шаблона с джокером. Была получена асимптотическая оценка времени работы алгоритма.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПЕРВОЙ ПРОГРАММЫ.

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

import operator

class TrieNode:
    ''' Вспомогательный класс для построения дерева
    '''
    def __init__(self):
        self.goto = {}
        self.out = []
        self.fail = None
        self.pNumber = 0

    def __str__(self):
        return f"[Node: goto={self.goto.keys()}, suff={self.fail}, pNumber={self.pNumber}]"


def aho_create_forest(patterns):
    '''Создать бор - дерево паттернов
    '''
    root = TrieNode()

    for idx, path in enumerate(patterns):
        node = root
        for symbol in path:
            node = node.goto.setdefault(symbol, TrieNode())
            node.pNumber = idx + 1
        node.out.append(path)
    return root


def aho_create_statemachine(patterns):
    '''Создать автомат Ахо-Корасика.
    Фактически создает бор и инициализирует fail-функции
    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)
            node.fail = root

        # Инициализируем остальные узлы:
        # 1. Берем очередной узел (важно, что проход в ширину)
        # 2. Находим самую длинную суффиксную ссылку для этой вершины - это и
будет fail-функция
        # 3. Если таковой не нашлось - устанавливаем fail-функцию в корневой
узел
        while len(queue) > 0:
            rnode = queue.pop(0)

            for key, unode in rnode.goto.items():
                queue.append(unode)
                fnode = rnode.fail
                while fnode is not None and key not in fnode.goto:
                    fnode = fnode.fail
                unode.fail = fnode.goto[key] if fnode else root
                unode.out += unode.fail.out

        return root


    def aho_find_all(text, root, number):
        '''Находит все возможные подстроки из набора паттернов в строке.
        '''
        node = root
        answer = []
        overlapping = set()
        for i in range(len(text)):
            while node is not None and text[i] not in node.goto:
                if node is not root and node.fail is not root:
                    overlapping.add((node.pNumber, node.fail.pNumber))
                node = node.fail
            if node is None:
                node = root
                continue
            node = node.goto[text[i]]
            for pattern in node.out:
```

```
                    answer.append((i - len(pattern) + 1 + 1, number[pattern] + 1))
        return sorted(answer, key=operator.itemgetter(0, 1)), overlapping

    def get_nodes_count(root):
        """
        Подсчёт числа узлов поиском в ширину
        """
        queue = [root]
        count = 0
        while len(queue):
            current = queue.pop(0)
            count += 1
            queue.extend(current.goto.values())
        return count

    def print_trie(root):
        """
        Вывод узлов поиском в ширину
        """
        queue = [root]
        while len(queue):
            current = queue.pop(0)
            print(current)
            queue.extend(current.goto.values())

    #Читаем данные
    s = input()
    n = int(input())
    patterns = []
    for _ in range(n):
        patterns.append(input())
    root = aho_create_statemachine(patterns)
    print_trie(root)
    print(f"Nodes in trie: {get_nodes_count(root)}")
    answer, preoverlapping = aho_find_all(s, root, {pattern: idx for idx,
pattern in enumerate(patterns)})
    overlapping = {(patterns[a-1], patterns[b-1]) for a, b in preoverlapping
if a != b} # переведем индексы в образцы
    print("Answer:")
    for row in answer:
        print(*row)
    print(f"Overlapping strings: {overlapping}")
```

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД ВТОРОЙ ПРОГРАММЫ.

```
class TrieNode:
    ''' Вспомогательный класс для построения дерева
    '''
    def __init__(self):
        self.goto = {}
        self.out = []
        self.fail = None
        self.idx = None


def aho_create_forest(patterns):
    '''Создать бор - дерево паттернов
    '''
    root = TrieNode()

    for idx, pattern in enumerate(patterns):
        node = root
        for symbol in pattern:
            node = node.goto.setdefault(symbol, TrieNode())
        node.out.append(pattern)
    return root


def aho_create_statemachine(patterns):
    '''Создать автомат Ахо-Корасика.
    Фактически создает бор и инициализирует fail-функции
    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)
        node.fail = root

    # Инициализируем остальные узлы:
    # 1. Берем очередной узел (важно, что проход в ширину)
    # 2. Находим самую длинную суффиксную ссылку для этой вершины - это и
будет fail-функция
```

```
        # 3. Если таковой не нашлось - устанавливаем fail-функцию в корневой
узел
        while len(queue) > 0:
            rnode = queue.pop(0)

            for key, unode in rnode.goto.items():
                queue.append(unode)
                fnode = rnode.fail
                while fnode is not None and key not in fnode.goto:
                    fnode = fnode.fail
                unode.fail = fnode.goto[key] if fnode else root
                unode.out += unode.fail.out

        return root


    def aho_find_all(s, root, bias):
        '''Находит все возможные подстроки из набора паттернов в строке.
        '''
        node = root
        answer = []
        visited = [set() for x in s]
        dp = [0 for _ in range(len(s))]
        for i in range(len(s)):
            while node is not None and s[i] not in node.goto:
                node = node.fail
            if node is None:
                node = root
                continue
            node = node.goto[s[i]]
            for pattern in set(node.out):
                for bias_ in bias[pattern]:
                    if i >= bias_:
                        dp[i - len(pattern) + 1 - bias_] += 1
        return dp


    text = input()
    pattern = input()
    joker = input()
    patterns = [x for x in pattern.split(joker) if x != '']
    assert len(patterns) > 0
```

```
bias = {}
used = set()
for pattern_ in reversed(sorted(patterns, key=len)):
    for i in range(len(pattern)):
        idx = pattern.find(pattern_, i)
        if idx != -1 and idx not in used:
            bias[pattern_] = bias.get(pattern_, []) + [idx]
            for j in range(len(pattern_)):
                used.add(idx+j)
bias = {key: set(value) for key, value in bias.items()}
print(f"String biases are: {bias}")
root = aho_create_statemachine(patterns)
answer = aho_find_all(text, root, bias)
print(f"Got dp array: {answer}")
for idx, number in enumerate(answer):
    if number == len(patterns) and idx + len(pattern) <= len(text):
        print(idx+1)
```