МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе № 5
по дисциплине «Построение и Анализ Алгоритмов»
Тема: «Ахо-Корасик»
Вариант 5

Студент гр. 3343 Жучков О.Д.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы

Написать программы на основе алгоритма Ахо-Корасик для нахождения вхождения всех образцов в строке, а также найти индексы вхождения образцов с джокерами.

Задание

№1

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.

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3

№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,$1 \leq |T| \leq 100000$)

Шаблон (P,$1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер). Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Вариант 5:

Вычислить максимальное количество дуг, исходящих из одной вершины в боре; вырезать из строки поиска все найденные образцы и вывести остаток строки поиска.

Выполнение работы

Алгоритм создает префиксное дерево из букв искомых подстрок. Вершины, в которых искомая подстрока заканчивается называет терминальной и выделяется специальным цветом в графическом представлении.

Суффиксная ссылка вершины u – это вершина v, такая что строка v является максимальным суффиксом строки u. Для корня и вершин, исходящих из корня, суффиксной ссылкой является корень. Для остальных вершин осуществляется переход по суффиксной ссылке родителя и, если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет. Суффиксные ссылки находятся не автоматически для каждой вершины, а вычисляются во время работы программы при обращении к специальному методу.

Текст, в котором нужно найти подстроки побуквенно передается в автомат. Начиная из корня, автомат переходит по ребру, соответствующему переданному символу. Если нужного ребра нет, переходит по ссылке. Если встреченная вершина является терминальной, значит была встречена подстрока. Если найдено совпадение нужно пройти по терминальным ссылкам, чтобы вывести все шаблоны заканчивающиеся на этом месте.

Класс *Node* – вершина бора, имеет поля описывающие суффиксную и терминальную ссылки, дуги входящие и исходящие, каждая вершина нумеруется от 0 до n. Имеет метод *count_children,* которая считает количество исходящих дуг.

Класс *Trie* – бор, хранит состояние автомата, корневую вершину дерева.

Метод *get_all_nodes* совершает проход в ширину для сбора подробной информации о боре и его вершинах для вывода.

Методы *get_link, get_suffix_link, get_terminal_link* отвечают за нахождение суффиксной и терминальной ссылки вершины.

Метод *add_string* добавляет строку-шаблон в бор.

Метод *process_text* ищет все вхождения шаблонов в данный текст с помощью построенного дерева.

Программа выводит промежуточные результаты при построении бора и обработке текста, включая подробное описание всех вершин.

```
[0] текст - root;   потомки: A->1, C->5, G->7;  суффиксная ссылка - 0;  терминальная ссылка - 0
[1] текст - A;  потомки: C->2;  суффиксная ссылка - 0;  терминальная ссылка - 0
[2] текст - AC; потомки: G->3;  суффиксная ссылка - 5;  терминальная ссылка - 0
[3] текст - ACG;    потомки: T->4;  суффиксная ссылка - 6;  терминальная ссылка - 6
[4] текст - ACGT;   потомки: ;  суффиксная ссылка - 8;  терминальная ссылка - 8
[5] текст - C;  потомки: G->6;  суффиксная ссылка - 0;  терминальная ссылка - 0
[6] текст - CG; потомки: ;  суффиксная ссылка - 7;  терминальная ссылка - 0
[7] текст - G;  потомки: T->8;  суффиксная ссылка - 0;  терминальная ссылка - 0
[8] текст - GT; потомки: ;  суффиксная ссылка - 0;  терминальная ссылка - 0
Макс. количество дуг, исходящих из вершины: 3
```

Рисунок 1 – Пример построенного бора

Помимо этого в методе *process_text_with_mask* был реализован алгоритм поиска шаблонов с масками. Для этого он разбивается на непрерывные подстроки без масок, которые добавляются в автомат. Далее автомат используется для поиска всех их вхождений в текст. Когда подстрока Qi находится в тексте на позиции j, это означает, что шаблон может начинаться в позиции j - li + 1 (где li – позиция начала Qi в шаблоне). Для этого заводится массив счётчиков C, где C[i] - количество подстрок, начавшихся в тексте в такой позиции, что шаблон может начинаться с i. Если в какой-либо позиции i счётчик равен числу подстрок (k), значит, в позиции i найдено полное вхождение шаблона с масками.

Оценка сложности алгоритма.

M – суммарный размер шаблонов, α – размер алфавита, тогда необходимо создать не более M вершин, каждая из которых может иметь до a исходящих дуг. O(M* α) – временная и пространственная сложность создания бора.

Поиск займет O(N+Z) времени, где N – длина текста, Z – количество найденных вхождений.

Для алгоритма Ахо-Корасик с джокером сложность по времени O(M · α + N + Z·K), где K – количество сегментов в паттерне (без джокеров), M – суммарная длина всех сегментов паттерна (без джокеров). Сложность по памяти не изменяется.

Тестирование

Таблица 1 – Тестирование алгоритма

№ п/п	Входные данные	Выходные данные	Комментарии
1	NCTAGTCN 6 TAGT TCN TA TG GT NC	1 6 3 1 3 3 5 5 6 2	Результат соответствует ожиданиям.
2	AANCTGATAAACANA AA$C$$A $	1 9	Проверка алгоритма с джокерами
3	AAAAA $$$ $	Шаблон состоит только из масок	Проверка особого случая

Выводы

Изучен принцип работы алгоритма Ахо-Корасик. Написаны про-граммы, корректно решающие задачу поиска набора подстрок в строке, в также программа поиска подстроки с джокером.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class Node:
    def __init__(self, alphabet, number):
        self.children = dict(zip(alphabet, [None] * len(alphabet)))
        self.go = dict(zip(alphabet, [None] * len(alphabet)))
        self.parent = None
        self.suffix_link = None
        self.terminal_link = None
        self.char = ''
        self.is_leaf = False
        self.leaf_pattern_number = []
        self.text = None
        self.number = number

    def count_children(self):
        count = 0
        for i in self.children:
            if self.children[i]:
                count += 1
        return count


class Trie:
    def __init__(self, alphabet=('A', 'C', 'G', 'T', 'N')):
        self.alphabet = alphabet
        self.alphabet_size = len(self.alphabet)
        self.root = Node(self.alphabet, 0)
        self.root.parent = self.root
        self.root.suffix_link = self.root
        self.root.terminal_link = self.root
        self.patterns = []
        self.node_count = 1

    def get_all_nodes(self):
        found = set()
        found.add(0)
        nodes = [self.root]
        node_queue = [self.root]
        while node_queue:
            cur = node_queue[0]
            for c in cur.children:
                v = cur.children[c]
                if v:
                    if v.number not in found:
                        nodes.append(v)
                        found.add(v)
                        node_queue.append(v)
            node_queue.pop(0)
        info_text = []
        for v in sorted(nodes, key=lambda x: x.number):
            info_text.append(f"[{v.number}] текст -
{self.get_text(v)};\tпотомки: {', '.join(f"{v.children[i].char}->{v.chil-
dren[i].number}" for i in v.children if v.children[i])};\t"
```

```
                    f"суффиксная ссылка - {self.get_suffix_link(v,
False).number if self.get_suffix_link(v, False) else "None"};\t"
                    f"терминальная ссылка - {self.get_terminal_link(v,
False).number if self.get_terminal_link(v, False) else "None"}")
            info_text.append(f"Макс. количество дуг, исходящих из вер-
шины: {max([v.count_children() for v in nodes])}")
            return info_text

        def get_node_char(self, v):
            return v.char if v != self.root else 'root'

        def get_text(self, v):
            if v == self.root:
                return 'root'
            if v.text:
                return v.text
            result = ''
            while v != self.root:
                result = v.char + result
                v = v.parent
            v.text = result
            return result

        def get_link(self, v, c, verbose=True):
            if v.go[c] is None:
                if v.children[c] is not None:
                    v.go[c] = v.children[c]
                    if verbose:
                        print(f"\tПерейдем по бору {v.number}
{self.get_text(v)} -> {v.go[c].number} {self.get_text(v.go[c])}")
                elif v == self.root:
                    v.go[c] = self.root
                    if verbose:
                        print(f"\tПерейдем в root")
                else:
                    v.go[c] = self.get_link(self.get_suffix_link(v,
verbose), c, verbose)
                    if verbose:
                        print(f"\tПерейдем по суффиксной ссылке {v.num-
ber} {self.get_text(v)} -> {v.go[c].number} {self.get_text(v.go[c])}")
            else:
                if verbose:
                    print(f"\tПереход по автомату {v.number}
{self.get_text(v)} -> {v.go[c].number} {self.get_text(v.go[c])}")
            return v.go[c]

        def get_suffix_link(self, v, verbose=True):
            if v.suffix_link is None:
                if v == self.root or v.parent == self.root:
                    v.suffix_link = self.root
                    if verbose:
                        print(f"\tСтроим суффиксную ссылку на root")
                else:
                    if verbose:
                        print(f"\tИщем суффикс в боре")
                    v.suffix_link = self.get_link(self.get_suf-
fix_link(v.parent, verbose), v.char, verbose)
                    if v.suffix_link != self.root:
```

```
                        if verbose:
                            print("\tСуффикс найден")
                    else:
                        if verbose:
                            print("\tМаксимальный суффикс пустой")
                    if verbose:
                        print(f"\tСтроим суффиксную ссылку {v.number}
{self.get_text(v)} -> {v.suffix_link.number} {self.get_text(v.suf-
fix_link)}")
            else:
                if verbose:
                    print(f"\tПереходим по суффиксной ссылке {v.number}
{self.get_text(v)} -> {v.suffix_link.number} {self.get_text(v.suf-
fix_link)}")
            return v.suffix_link

        def get_terminal_link(self, v, verbose=True):
            if v.terminal_link is None:
                suffix_link = self.get_suffix_link(v, verbose)
                if suffix_link.is_leaf:
                    v.terminal_link = suffix_link
                elif suffix_link == self.root:
                    v.terminal_link = self.root
                else:
                    v.terminal_link = self.get_terminal_link(suf-
fix_link, verbose)
                if verbose:
                    print(f"\tСтроим терминальную ссылку {v.number}
{self.get_text(v)} -> {v.terminal_link.number} {self.get_text(v.termi-
nal_link)}")
            else:
                if verbose:
                    print(f"\tПереходим по терминальной ссылке {v.num-
ber} {self.get_text(v)} -> {v.terminal_link.number} {self.get_text(v.ter-
minal_link)}")
            return v.terminal_link

        def add_string(self, s, pattern_number):
            print(f"Добавим строку {s} в бор")
            cur = self.root
            for c in s:
                print(f"Находимся в {cur.number} {self.get_text(cur)}")
                if cur.children[c] is None:
                    print(f"\tДобавляем {c}")
                    new = Node(self.alphabet, self.node_count)
                    new.char = c
                    new.parent = cur
                    cur.children[c] = new
                    self.node_count += 1
                else:
                    print(f"\t{c} уже существует")
                cur = cur.children[c]
            print(f"Находимся в {cur.number} {self.get_text(cur)}")
            print(f"\t{c} - терминальный символ")
            cur.is_leaf = True
            cur.leaf_pattern_number.append(pattern_number)
            self.patterns.append(s)
```

```
        def process_text(self, text):
            result = []
            current = self.root

            for i in range(len(text)):
                c = text[i]
                print(f"Рассмотрим вершину {c} на позиции {i + 1} в
тексте {text}")
                current = self.get_link(current, c)
                if current == self.root:
                    print("\tПодстрока не встречается в тексте")
                else:
                    print(f"\tПерешли в состояние {self.get_text(cur-
rent)}")

                temp_node = current
                while temp_node != self.root:
                    if temp_node.is_leaf:
                        for num in temp_node.leaf_pattern_number:
                            pattern_length = len(self.patterns[num])
                            start_pos = i - pattern_length + 1
                            result.append((start_pos, num))
                            print(f"\tВершина {temp_node.number} терми-
нальная, обнаружено вхождение подстроки {patterns[num]}")
                    term = self.get_terminal_link(temp_node)
                    temp_node = term
                    print(f"\tПереходим по терминальной ссылке
{temp_node.number} {self.get_text(temp_node)} -> {term.number}
{self.get_text(term)}")

            print(f"Количество вершин в автомате = {self.node_count}")
            return result

        def process_text_with_mask(self, pattern, text, wildcard):
            if all(c == wildcard for c in pattern):
                print("Шаблон состоит только из масок")
                return []

            print("Разобьем строку на подстроки без маскок")
            substrings = list()
            substring_positions = list()
            i = 0
            while i < len(pattern):
                if pattern[i] == wildcard:
                    i += 1
                    continue
                start = i
                while i < len(pattern) and pattern[i] != wildcard:
                    i += 1
                substrings.append(pattern[start:i])
                substring_positions.append(start)
            print(f"Подстроки без масок: {", ".join(substrings)} на по-
зициях: {", ".join(map(str, substring_positions))}")

            print("Добавим подстроки в бор")
            for i in range(len(substrings)):
                self.add_string(substrings[i], i)
```

```
            counter = [0] * len(text)
            current = self.root
            print("Подсчитаем вхождения подстрок")
            for i in range(len(text)):
                c = text[i]
                print(f"Рассмотрим вершину {c} на позиции {i + 1} в
тексте {text}")
                current = self.get_link(current, c)
                if current == self.root:
                    print("\tПодстрока не встречается в тексте")
                else:
                    print(f"\tПерешли в состояние {self.get_text(cur-
rent)}")
                temp_node = current
                while temp_node != self.root:
                    term = self.get_terminal_link(temp_node)
                    if temp_node.is_leaf:
                        for num in temp_node.leaf_pattern_number:
                            substring_position = substring_posi-
tions[num]
                            substring_length = len(substrings[num])
                            start_pos = i - substring_length - sub-
string_position + 1
                            if start_pos < 0:
                                continue
                            if start_pos + len(pattern) <= len(text):
                                counter[start_pos] += 1
                        print(f"\tВершина {temp_node.char} терминаль-
ная, обнаружено вхождение подстроки {self.patterns[num]}")
                    temp_node = term

            print(f"Найдем вхождения шаблона")
            print(f"Получившийся счетчик совпадений: {counter}")
            result = []
            for i, count in enumerate(counter):
                if count == len(substrings):
                    result.append(i + 1)
                    print(f"\tКоличество вхождений совпало для позиции
{i + 1} с числом {count}")
            print(f"Количество вершин в автомате = {self.node_count}")
            return result


    var = int(input("Выберите вариант\n\t1: Поиск набора образцов\n\t2:
Поиск образца с джокером\n"))
    if var == 1:
        print("1. Текст; 2. Число N шаблонов; 3. N строк с шаблонами")
        text = input().strip()
        n = int(input())
        patterns = [input() for _ in range(n)]

        ac = Trie()
        print("Создание бора и добавление строк")
        for i, pattern in enumerate(patterns):
            ac.add_string(pattern, i)

        print("Преобразуем бор")
        matches = ac.process_text(text)
```

```
        print("Вывод вхождений в текст")
        matches.sort()
        mask = set()
        for pos, pattern_num in matches:
            print(pos+1, pattern_num+1)
        for pos, pattern_num in matches:
            print(f"Шаблон {patterns[pattern_num]} встречается в тексте
{text} на позиции {pos}")
            for i in range(pos, pos+len(patterns[pattern_num])):
                mask.add(i)
        cut = ""
        for i in range(len(text)):
            if i not in mask:
                cut += text[i]

        print("\n".join(ac.get_all_nodes()))
        print("Текст с вырезанными фрагментами:")
        print(cut)


    elif var == 2:
        text = input()
        wildcard_pattern = input()
        wildcard = input()

        ac = Trie()
        matches = ac.process_text_with_mask(wildcard_pattern, text,
wildcard)

        print(f"Вывод найденных вхождений шаблона")
        print(f"Шаблон {wildcard_pattern} встречается в тексте {text}
на позициях {", ".join(map(str, matches))}.")
        print("\n".join(ac.get_all_nodes()))
        print("Текст с вырезанными фрагментами:")
        cut = ""
        mask = set()
        for pos in matches:
            for i in range(pos-1, pos-1+len(wildcard_pattern)):
                mask.add(i)
        for i in range(len(text)):
            if i not in mask:
                cut += text[i]
        print(cut)
```