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: Поиск набора подстрок в строке (Ахо-Корасик)
Вариант: 2

Студентка гр. 3388 _____ Титкова С.Д.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы:

Изучить принцип работы алгоритма Ахо-Корасик для нахождения подстрок в строке. Решить с его помощью задачи.

Задание 1:

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,1≤|T|≤100000T).

Вторая - число n (1≤n≤3000), каждая следующая из n строк содержит шаблон из набора $P=\{p\_1,\ldots,p\_n\}$ $1\le|p\_i|\le75$.

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?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

Реализация

Описание алгоритма Ахо-Корасика для точного поиска образцов:

Алгоритм Ахо-Корасика предназначен для эффективного поиска всех вхождений множества заданных шаблонов P={p_1,p_2,…,p_n} в текст T. Он использует структуру данных, называемую бором (trie), дополненную суффиксными и конечными ссылками, что позволяет выполнять поиск за один проход по тексту независимо от числа шаблонов. В данной реализации алгоритм также подсчитывает количество вершин в автомате и определяет шаблоны, имеющие пересечения с другими вхождениями в тексте.

Шаги алгоритма

Создаётся корневое состояние (0) с fail = 0 (суффиксная ссылка) и outputlink = −1 (конечная ссылка). Определяется максимальное число состояний как сумма длин всех шаблонов плюс корень (∑|pi|+1).

Для каждого шаблона p_i из P. Начиная с корня, для каждого символа c в p_i добавляется новое состояние, если перехода по с ещё нет. Счётчик вершин state_count увеличивается на 1 для каждого нового состояния. Конечное состояние шаблона отмечается его индексом i в массиве output.

Далее строится автомат. Используется очередь для обхода состояний в порядке ширины (BFS). Для каждого состояния из корня: fail = 0, output_link устанавливается как само состояние, если оно содержит шаблон, иначе -1.

Для остальных состояний. Суффиксная ссылка fail[next_state] определяется поиском самого длинного суффикса текущей строки, доступного через fail[current_state]. Конечная ссылка output_link[next_state] указывает на ближайший узел с шаблоном по цепочке fail, либо на −1.

Затем производится поиск в тексте. Проход по символам текста T с использованием текущего состояния current_state. Если перехода по символу нет, алгоритм следует по fail до корня или подходящего состояния. При достижении состояния с непустым output или через output_link фиксируются все вхождения шаблонов с вычислением их позиций.

Также по заданию из варианта требовалось осуществить: подсчёт вершин: возвращается state_count. Определение пересечений: для каждого вхождения вычисляются начало и конец, проверяются пересечения между парами вхождений.

.

Описание функций и структур:

Был создан класс *AhoCorasick* в нём определены следующие поля:

- *Patterns* - используется для построения бора и вычисления позиций вхождений.
- *Transitions* - представляет бор, обеспечивая быстрый доступ к следующему состоянию
- *Output* - хранит информацию о завершении шаблонов в вершинах бора
- *Fail* - суффиксные ссылки для быстрого перехода при несовпадении символов
- *Output_link* - конечные ссылки для эффективного извлечения всех совпадени
- *State_count* - используется для подсчёта вершин и ограничения массивов.

Также были реализованы методы класса:

- *__init__(self, patterns)* – метод, который инициализирует автомат Ахо-Корасика для заданного набора шаблонов.
- *Build_automaton(self)* – метод, который строит бор и автомат Ахо-Корасика с суффиксными и конечными ссылками.
- *Search(self, text)* - метод, который ищет все вхождения шаблонов в тексте
- *Get_vertex_count(self)* – метод, который возвращает количество вершин в автомате
- *Find_overlapping_patterns(self, text, results)* – метод, который определяет шаблоны, вхождения которых пересекаются с другими.

Также была написана функция *process_search(T, patterns)*, которая обрабатывает поиск шаблонов в тексте и возвращает результаты.

Оценка сложности алгоритма:

Временная сложность

Построение бора:

- Для каждого символа всех шаблонов создаётся или используется состояние: O(∑|pi|), где| ∑|pi| — суммарная длина шаблонов

Построение автомата:

- Обход BFS по всем состояниям (до ∑|pi|+1) с проверкой переходов для каждого символа алфавита.
- O(∑|pi|) для небольшого алфавита, так как |Σ| считается константой.

Поиск:

- Проход по тексту: O(|T|) переходов, каждый с O(1) по fail и output_link благодаря конечным ссылкам.
- Извлечение совпадений: O(z), где z — число вхождений.

Итог: O(|T|+z)

Пространственная сложность

Бор:

- transitions: O(∑|pi|) состояний, каждый с O(|Σ|) переходов (в худшем случае).
- output: O(∑|pi|) для хранения индексов шаблонов.

Ссылки:

- fail, output_link: O(∑|pi|) элементов.

Результаты:

- results: O(z) пар.
- overlap_patterns: O(n) в худшем случае, где n — число шаблонов.

Итог: O(∑|pi|) для автомата плюс O(z) для результатов, итого O(∑|pi|+z).

Тестирование

Таблица 1. Тестирование.

Входные данные	Выходные данные
ACGT 2 ACGTACGT CGTA	
AAAAA 2 A AA	1 1 1 2 2 1 2 2 3 1 3 2 4 1 4 2 5 1
ACGTACGT 3 AC CG GT	1 1 2 2 3 3 5 1 6 2 7 3
ACGTACGT 3 A AC ACG	1 1 1 2 1 3 5 1 5 2 5 3

Описание Алгоритма Ахо-Корасика с джокерами:

Алгоритм Ахо-Корасика в данной реализации адаптирован для поиска всех вхождений шаблона P с джокерами (специальный символ, обозначающий совпадение с любым символом) в тексте T. Он использует бор (trie) с суффиксными и конечными ссылками для поиска всех бездженкерных подстрок шаблона P, а затем проверяет полное соответствие P в найденных позициях с учётом джокеров. Алгоритм также подсчитывает количество вершин в автомате и определяет шаблоны с пересекающимися вхождениями.

Шаги алгоритма

Создаётся корневое состояние (0) с fail = 0 (суффиксная ссылка) и outputlink = −1 (конечная ссылка). Определяется максимальное число состояний как сумма длин всех шаблонов плюс корень (∑|pi|+1).

Для каждой подстроки Q_i из P. Начиная с корня, создаются состояния для каждого символа c, если перехода нет. Счётчик state_count увеличивается для новых состояний. Конечное состояние подстроки помечается её индексом в output.

Далее производится построение автомата. Для этого будет использоваться обход в ширину (BFS). Для детей корня: fail = 0, output_link— само состояние, если оно конец шаблона, иначе −1. Для остальных: fail определяется поиском через fail родителя, output_link— ближайший узел с шаблоном.

Далее проходим по T с использованием transitions и fail для переходов. Совпадения извлекаются через output и output_link.

Для каждого совпадения Q_i на позиции j: C[j - l_i] += 1, где l_i — стартовая позиция Q_i в P. Позиции i, где C[i] = k и T[i:i+|P|] соответствует P с джокерами, добавляются в результат.

Также по заданию из варианта требовалось осуществить: подсчёт вершин: возвращается state_count. Определение пересечений: для каждого вхождения вычисляются начало и конец, проверяются пересечения между парами вхождений

Описание функций и структур:

Был создан класс *AhoCorasick* в нём определены следующие поля:

- *Patterns* - используется для построения бора и вычисления позиций вхождений.
- *Transitions* - представляет бор для поиска подстрок.
- *Output* - хранит совпадения подстрок
- *Fail* - суффиксные ссылки для быстрого перехода при несовпадении символов
- *Output_link* - конечные ссылки для эффективного извлечения всех совпадени
- *State_count* - используется для подсчёта вершин и ограничения массивов.

Также были реализованы методы класса:

- *__init__(self, patterns)* – метод, который инициализирует автомат Ахо-Корасика для заданного набора шаблонов.
- *Build_automaton(self)* – метод, который строит бор и автомат Ахо-Корасика с суффиксными и конечными ссылками.
- *Search(self, text)* - метод, который ищет все вхождения шаблонов в тексте
- *Get_vertex_count(self)* – метод, который возвращает количество вершин в автомате
- *Find_overlapping_patterns(self, text, results)* – метод, который определяет шаблоны с пересечениями

Также была написана функция *find_pattern_with_wildcards(T, P, wildcard)*, которая находит вхождения P с джокерами в T.

Оценка сложности алгоритма:

Временная сложность:

Разбиение P:

- O(|P|) для разбиения по wildcard и вычисления start_positions.

Построение бора:

- O(∑|Qi|) для добавления подстрок, где ∑|Qi|≤|P|.

Построение автомата:

- O(∑|Qi|) для BFS с фиксированным алфавитом (|Σ| — константа).

Поиск:

- O(|T|) для прохода по тексту, O(z) для извлечения совпадений (z — число вхождений).

Проверка джокеров:

- O(|T|·|P|) для анализа C и проверки символов P в каждой позиции.

Пересечения:

- O(z^2) для сравнения всех пар вхождений.

Итого: O(∑|Qi|+|T|·|P|+z2)

Пространственная сложность

Бор:

- transitions: O(∑|Qi|) состояний.
- output, fail, ooutput_link: O(∑|Qi|).

Поиск:

- C: O(|T|).
- results: O(z) O(z) O(z).
- overlap_patterns: O(k), где k≤|P|.

Общая: O(∑|Qi|+|T|+z), z≤|T|.

Тестирование

Таблица 2. Тестирование.

Входные данные	Выходные данные
AAAAA A*A *	1 2 3
ACGTACGT *CG* *	1 5
ACTANCA A$$A$ $	1
NTAG T*G *	2

Вывод

В ходе лабораторной работы были написаны программы с использованием алгоритма Ахо-Корасика. Также дополнительно было сделано: подсчёт вершин и определение пересечений.

Исходный код программы см. в ПРИЛОЖЕНИИ А.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

aho_1.py

```
from collections import deque

DEBUG = False

class AhoCorasick:
    def __init__(self, patterns):
        self.patterns = [p for p in patterns if p]
        self.num_patterns = len(self.patterns)
        self.max_states = sum(len(p) for p in self.patterns) + 1
        self.transitions = [{} for _ in range(self.max_states)]
        self.output = [[] for _ in range(self.max_states)]
        self.fail = [0] * self.max_states
        self.output_link = [-1] * self.max_states
        self.state_count = 0
        self.build_automaton()

    def build_automaton(self):
        root = 0
        self.fail[root] = root
        self.output_link[root] = -1
        self.state_count = 1

        if DEBUG:
            print("Построение бора:")
        for i in range(self.num_patterns):
            current_state = root
            for char in self.patterns[i]:
                if char not in self.transitions[current_state]:
                    self.transitions[current_state][char] =
self.state_count
                    if DEBUG:
                        print(f"Добавлено ребро из состояния
{current_state} в состояние {self.state_count} по символу '{char}'")
                    self.state_count += 1
                current_state = self.transitions[current_state][char]
            self.output[current_state].append(i)
            if DEBUG:
                print(f"Добавлен шаблон {i} в состояние
{current_state}")

        if DEBUG:
            print("\nПостроение автомата:")
        queue = deque()
        for char in self.transitions[root]:
            state = self.transitions[root][char]
            self.fail[state] = root
            self.output_link[state] = state if self.output[state] else
-1
            queue.append(state)
            if DEBUG:
                print(f"Установлен fail-переход для состояния {state}
в состояние {root}")
```

```
        while queue:
            current_state = queue.popleft()
            for char in self.transitions[current_state]:
                next_state = self.transitions[current_state][char]
                queue.append(next_state)

                fail_state = self.fail[current_state]
                while fail_state != root and char not in
self.transitions[fail_state]:
                    fail_state = self.fail[fail_state]
                self.fail[next_state] =
self.transitions[fail_state].get(char, root)
                if DEBUG:
                    print(f"Установлен fail-переход для состояния
{next_state} в состояние {self.fail[next_state]}")

                fail = self.fail[next_state]
                self.output_link[next_state] = fail if
self.output[fail] else self.output_link[fail]
                if DEBUG:
                    print(f"Установлен output-переход для состояния
{next_state} в состояние {self.output_link[next_state]}")

        if DEBUG:
            print("\nПостроенный автомат:")
            for state in range(self.state_count):
                print(f"Состояние {state}:")
                print(f"  Переходы: {self.transitions[state]}")
                print(f"  Выходы: {self.output[state]}")
                print(f"  Fail-переход: {self.fail[state]}")
                print(f"  Output-переход: {self.output_link[state]}")

    def search(self, text):
        current_state = 0
        results = []
        if DEBUG:
            print("\nПроцесс поиска:")
        for i in range(len(text)):
            char = text[i]
            while current_state != 0 and char not in
self.transitions[current_state]:
                current_state = self.fail[current_state]
                if DEBUG:
                    print(f"Переход по fail-ссылке в состояние
{current_state}")

            if char in self.transitions[current_state]:
                current_state = self.transitions[current_state][char]
                if DEBUG:
                    print(f"Переход в состояние {current_state} по
символу '{char}'")
            else:
                current_state = 0
                if DEBUG:
                    print(f"Символ '{char}' не найден, переход в
корневое состояние")
```

```
            temp_state = current_state
            visited = set()
            while temp_state != -1 and temp_state not in visited:
                visited.add(temp_state)
                if self.output[temp_state]:
                    for pattern_index in self.output[temp_state]:
                        pos = i - len(self.patterns[pattern_index]) +
2
                        results.append((pos, pattern_index + 1))
                        if DEBUG:
                            print(f"Найден шаблон {pattern_index + 1}
на позиции {pos}")
                temp_state = self.output_link[temp_state]
        return sorted(results)

    def get_vertex_count(self):
        return self.state_count

    def find_overlapping_patterns(self, text, results):
        if not results:
            return set()

        occurrences = [(pos, pos + len(self.patterns[pattern_num - 1])
- 1, pattern_num)
                       for pos, pattern_num in results]
        overlap_patterns = set()

        for i in range(len(occurrences)):
            start1, end1, pattern1 = occurrences[i]
            for j in range(len(occurrences)):
                if i != j:
                    start2, end2, pattern2 = occurrences[j]
                    if start1 <= end2 and start2 <= end1:
                        overlap_patterns.add(pattern1)
                        overlap_patterns.add(pattern2)
        return overlap_patterns

def process_search(T, patterns):
    automaton = AhoCorasick(patterns)
    results = automaton.search(T)
    vertex_count = automaton.get_vertex_count()
    overlapping = automaton.find_overlapping_patterns(T, results)
    return results, vertex_count, overlapping

if __name__ == "__main__":
    T = input().strip()
    n = int(input())
    patterns = [input().strip() for _ in range(n)]

    results, vertex_count, overlapping = process_search(T, patterns)

    print(f"\nКоличество вершин в автомате: {vertex_count}")
    for pos, pattern_num in results:
        print(pos, pattern_num)
    if overlapping:
        print("Шаблоны с пересечениями:", ", ".join(str(p) for p in
sorted(overlapping)))
```

```
    else:
        print("Шаблоны с пересечениями отсутствуют")
```

aho_mask.py

```
from collections import deque

DEBUG = False

class AhoCorasick:
    def __init__(self, patterns):
        self.patterns = [p for p in patterns if p]
        self.num_patterns = len(self.patterns)
        self.max_states = sum(len(p) for p in self.patterns) + 1
        self.transitions = [{} for _ in range(self.max_states)]
        self.output = [[] for _ in range(self.max_states)]
        self.fail = [0] * self.max_states
        self.output_link = [-1] * self.max_states
        self.state_count = 0
        self.build_automaton()

    def build_automaton(self):
        root = 0
        self.fail[root] = root
        self.output_link[root] = -1
        self.state_count = 1

        if DEBUG:
            print("Построение бора:")
            print(f"Создано состояние {root} (корень)")

        for i in range(self.num_patterns):
            if DEBUG:
                print(f"\nДобавление шаблона {i + 1}:
'{self.patterns[i]}'")
            current_state = root
            for char in self.patterns[i]:
                if char not in self.transitions[current_state]:
                    new_state = self.state_count
                    self.transitions[current_state][char] = new_state
                    if DEBUG:
                        print(f"Создано состояние {new_state} с
переходом из {current_state} по '{char}'")
                    self.state_count += 1
                else:
                    new_state = self.transitions[current_state][char]
                    if DEBUG:
                        print(f"Переход из {current_state} по '{char}'
в существующее состояние {new_state}")
                current_state = new_state
            self.output[current_state].append(i)
            if DEBUG:
                print(f"Состояние {current_state} отмечено как конец
шаблона {i + 1}")

        if DEBUG:
            print("\nПостроение суффиксных и конечных ссылок:")
        queue = deque()
        for char in self.transitions[root]:
```

```
            state = self.transitions[root][char]
            self.fail[state] = root
            self.output_link[state] = state if self.output[state] else
-1
            if DEBUG:
                print(f"Состояние {state}: fail = {root}, output_link
= {self.output_link[state]}")
            queue.append(state)

        while queue:
            current_state = queue.popleft()
            for char in self.transitions[current_state]:
                next_state = self.transitions[current_state][char]
                queue.append(next_state)

                if DEBUG:
                    print(f"\nОбрабатываем переход из {current_state}
по '{char}' в {next_state}")
                fail_state = self.fail[current_state]
                while fail_state != root and char not in
self.transitions[fail_state]:
                    if DEBUG:
                        print(f"Состояние {fail_state} не имеет
перехода по '{char}', переходим к fail = {self.fail[fail_state]}")
                    fail_state = self.fail[fail_state]
                self.fail[next_state] =
self.transitions[fail_state].get(char, root)

                if DEBUG:
                    print(f"Установлена суффиксная ссылка:
fail[{next_state}] = {self.fail[next_state]}")

                fail = self.fail[next_state]
                self.output_link[next_state] = fail if
self.output[fail] else self.output_link[fail]
                if DEBUG:
                    print(f"Установлена конечная ссылка:
output_link[{next_state}] = {self.output_link[next_state]}")

        if DEBUG:
            print("\nПостроенный автомат:")
        for state in range(self.state_count):
            trans = "{" + ", ".join(f"'{k}': {v}" for k, v in
self.transitions[state].items()) + "}"
            if DEBUG:
                print(f"Состояние {state}: transitions = {trans}, fail
= {self.fail[state]}, "
                      f"output = {self.output[state]}, output_link =
{self.output_link[state]}")

    def search(self, text):
        if DEBUG:
            print("\nПроцесс поиска в тексте:", text)
        current_state = 0
        results = []
        for i in range(len(text)):
            char = text[i]
            if DEBUG:
```

```
                print(f"\nПозиция {i + 1}: символ '{char}', текущее
состояние = {current_state}")
            while current_state != 0 and char not in
self.transitions[current_state]:
                if DEBUG:
                    print(f"Нет перехода по '{char}' из
{current_state}, переходим к fail = {self.fail[current_state]}")
                current_state = self.fail[current_state]

            if char in self.transitions[current_state]:
                next_state = self.transitions[current_state][char]
                if DEBUG:
                    print(f"Переход по '{char}' из {current_state} в
{next_state}")
                current_state = next_state
            else:
                if DEBUG:
                    print(f"Нет перехода по '{char}' из
{current_state}, переходим в корень (0)")
                current_state = 0

            temp_state = current_state
            visited = set()
            while temp_state != -1 and temp_state not in visited:
                visited.add(temp_state)
                if self.output[temp_state]:
                    for pattern_index in self.output[temp_state]:
                        pos = i - len(self.patterns[pattern_index]) +
2

                        if DEBUG:
                            print(f"Найдено вхождение шаблона
{pattern_index + 1} на позиции {pos}")
                        results.append((pos, pattern_index + 1))
                temp_state = self.output_link[temp_state]
        return sorted(results)

    def get_vertex_count(self):
        return self.state_count

    def find_overlapping_patterns(self, text, results):
        if not results:
            return set()

        occurrences = [(pos, pos + len(self.patterns[pattern_num - 1])
- 1, pattern_num)
                       for pos, pattern_num in results]
        overlap_patterns = set()

        for i in range(len(occurrences)):
            start1, end1, pattern1 = occurrences[i]
            for j in range(len(occurrences)):
                if i != j:
                    start2, end2, pattern2 = occurrences[j]
                    if start1 <= end2 and start2 <= end1:
                        overlap_patterns.add(pattern1)
                        overlap_patterns.add(pattern2)
        return overlap_patterns
```

```
def find_pattern_with_wildcards(T, P, wildcard):
    substrings = [s for s in P.split(wildcard) if s]
    if not substrings:
        return [], 0, set()

    k = len(substrings)
    start_positions = []
    pos = 0
    for i, sub in enumerate(P.split(wildcard)):
        if sub:
            start_positions.append(pos)
        pos += len(sub) + (1 if i < len(P.split(wildcard)) - 1 else 0)

    ac = AhoCorasick(substrings)
    matches = ac.search(T)

    n = len(T)
    C = [0] * n

    for pos, pattern_idx in matches:
        start_pos_in_P = start_positions[pattern_idx - 1]
        text_start = pos - start_pos_in_P - 1
        if text_start >= 0:
            C[text_start] += 1

    result = []
    for i in range(n):
        if C[i] == k and i + len(P) - 1 < n:
            valid = True
            for j in range(len(P)):
                text_pos = i + j
                if P[j] != wildcard and T[text_pos] != P[j]:
                    valid = False
                    break
            if valid:
                result.append(i + 1)

    vertex_count = ac.get_vertex_count()
    overlapping = ac.find_overlapping_patterns(T, matches)
    return sorted(result), vertex_count, overlapping


if __name__ == "__main__":
    T = input().strip()
    P = input().strip()
    wildcard = input().strip()

    occurrences, vertex_count, overlapping =
find_pattern_with_wildcards(T, P, wildcard)

    print(f"Количество вершин в автомате: {vertex_count}")
    if occurrences:
        for pos in occurrences:
            print(pos)
    else:
        print(-1)
    if overlapping:
```

```
        print("Шаблоны с пересечениями:", ", ".join(str(p) for p in
sorted(overlapping)))
    else:
        print("Шаблоны с пересечениями отсутствуют")
```