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

Студент гр. 3343 _____ Пивоев Н. М.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Изучить принцип работы алгоритма Ахо-Корасик. Написать две программы, решающие задачи поиска набора образцов в тексте и одного образца с джокером.

Задание №1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Входные данные:

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

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

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

Выходные данные:

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 a*b*??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*}

Входные данные:

Текст (T, $1 \le |T| \le 100000$)

Шаблон (P,$1 \le |P| \le 40$)

Символ джокера

Выходные данные:

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

Описание алгоритма Ахо-Корасик

Алгоритм заключается в построении префиксного дерева (бора) с суффиксными ссылками на основе поданных паттернов с последующим непрерывным обходом по переданному тексту. Изначально мы знаем только паттерны.

Сначала строится бор, который состоит из букв паттернов. Затем в него добавляются суффиксные и улучшенные суффиксные (терминальные) ссылки. Суффиксные ссылки показывают для каждой вершины максимальный суффикс текущей последовательности, который также есть в боре. Корень указывает на самого себя, как исключительный случай. Эти ссылки нужны для реализации ситуации, когда при обработке очередного символа текста дальнейшего перехода нет (либо из-за полного нахождения паттерна, либо из-за отсутствия следующей вершины в боре без ссылок). Помимо суффиксных ссылок есть улучшенные суффиксные (терминальные), которые указывают на вершины, хранящие в себе все паттерны, которые присутствовали в последовательности бора до данной вершины.

После построения бора с ссылками идёт анализ текста. В данном случае бор является непрерывным автоматом, потому что каждый анализируемый символ текста меняет текущее состояние в боре, осуществляется переход либо по самому бору, либо по ссылкам. За счёт существования ссылок эти переходы

могут происходить бесконечно, потому что даже при достижении конца очередного паттерна в боре следующий переход по суффиксной ссылке вернёт либо в корень бора, либо в вершину, стоящую раньше. Терминальные ссылки помогают оптимизировать процесс обработки и не пропустить ни один паттерн. Таким образом, мы получаем все вхождения паттернов в тексте.

Сложность по времени:

Сложность алгоритма по времени – $O(\sum|p_i| + |T|)$, где берётся сумма размеров всех паттернов и текста. Построение бора и обход текста занимают соответственно $O(\sum|p_i|)$ и $O(|T|)$, так как мы сначала обходим паттерны для построения бора, а затем обходим текст для поиска вхождений.

Сложность по памяти:

Сложность алгоритма по памяти - $O(\sum|p_i| * k)$, где k – размер алфавита, поскольку каждая вершина хранит множество своих детей, и в худшем случае его дети – все символы данного алфавита.

Описание модифицированного алгоритма

Алгоритм по построению бора и ссылок такой же, в качестве паттернов берутся подстроки шаблона, разделённые джокером. Логика поиска подстрок в тексте тоже практически совпадает. После поиска подстрок (совпадений) мы получаем список индексов и подстрок, которые возможно являются вхождениями в текст. Для каждой такой подстроки определяется позиция, с которой должен начаться весь шаблон, чтобы эта часть попала на своё место. Для каждой из позиций есть счётчик, который увеличивается при попадании туда начала шаблона. Если этот счётчик совпадёт с числом подстрок, то нужная позиция найдена.

Сложность по времени для модифицированного алгоритма:

Сложность алгоритма по времени - $O(\sum|p_i| + |T|)$ совпадает с оригинальным алгоритмом.

Сложность по памяти для модифицированного алгоритма:

Сложность алгоритма по памяти - $O(\sum|p_i| * k + |T|)$ совпадает с оригинальным алгоритмом, но добавляется массив длины T, хранящий счётчик потенциальных начал шаблона.

Описание функций

1. *build_trie* – создает префиксное дерево (бор) на основе переданных слов, добавляя узлы для каждого символа и отмечая терминальные узлы.
2. *build_links(self)* – создает суффиксные и терминальные ссылки для всех узлов, используя обход дерева в ширину.
3. *print_trie(self, node=None, level=0)* – выводит бор.
4. *search(self, text)* – осуществляет поиск в тексте всех вхождений паттерна, используя бор с суффиксными и терминальными ссылками.
5. *analyze_patterns(self, results, text)* – анализирует бор, вычисляя количество узлов в нём, и находит пересечения паттернов в тексте.
6. *count_nodes(node)* – подсчитывает число узлов в боре.
7. *find_matches(self, text)* – находит в тексте подстроки паттерна, разделённые джокерами, в формате (индекс, подстрока).
8. *find_wildcard_matches(text, pattern, wildcard)* – реализует алгоритм по поиску вхождения паттерна с джокерами в текст.

Тестирование.

Таблица 1 – Тестирование алгоритма Ахо-Корасик

Входные данные	Выходные данные
NTAG 3	2 2 2 3

TAGT TAG T	
ACCGTACA 2 AC GT	1 1 4 2 6 1
ACGT 3 ACGT CG GT	1 1 2 2 3 3

Таблица 2 – Тестирование алгоритма поиска с джокером

Входные данные	Выходные данные
ACTANCA A$$A$ $	1
ACACAA ACXA X	3
ACGANGAAAT A$G $	1 4

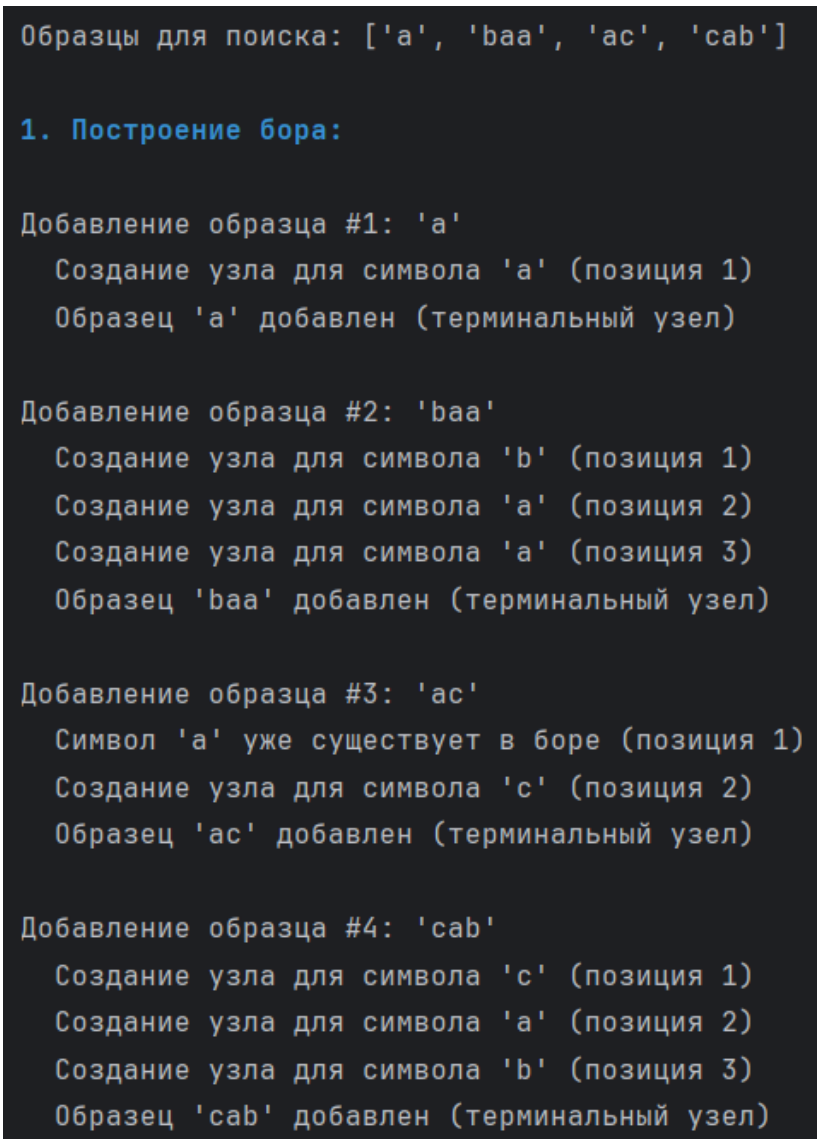

Рисунок 1 – Пример вывода построения бора

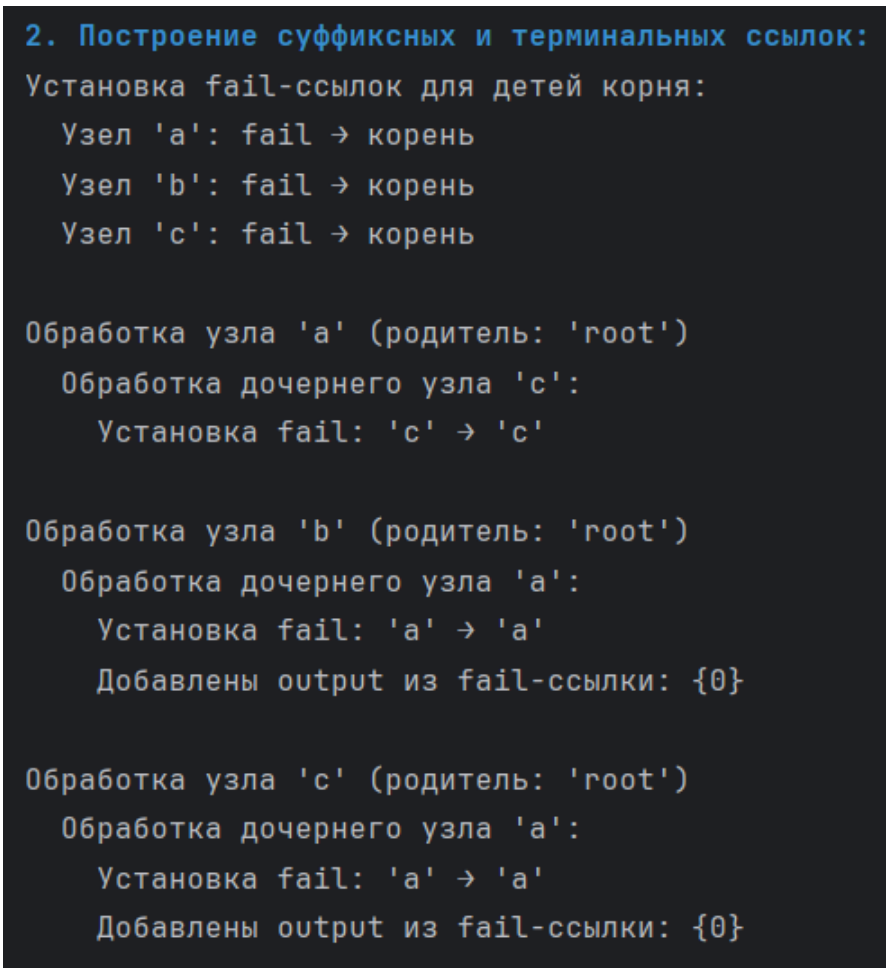

Рисунок 2 – Пример вывода создания суффиксных и терминальных ссылок

```
4. Поиск в тексте 'abacaba':

Символ 'a' (позиция 1):
  Переход в узел 'a'
  Найдены подстроки {0} через терминальные ссылки
    Образец #1 'a' начинается на позиции 1

Символ 'b' (позиция 2):
  Нет перехода по 'b', переход по fail: 'a' → 'None'
  Переход в узел 'b'

Символ 'a' (позиция 3):
  Переход в узел 'a'
  Узел 'a' унаследовал output через fail-ссылку: {0}
  Найдены подстроки {0} через терминальные ссылки
    Образец #1 'a' начинается на позиции 3

Символ 'c' (позиция 4):
  Нет перехода по 'c', переход по fail: 'a' → 'a'
  Переход в узел 'c'
  Найдены подстроки {2} через терминальные ссылки
    Образец #3 'ac' начинается на позиции 3
```

Рисунок 3 – Пример вывода поиска в тексте

```
3. Поиск подстрок в тексте:
[(1, 0), (4, 1), (5, 1), (7, 0), (9, 0), (11, 1)]
Найдено совпадений: 6

4. Подсчет возможных начал шаблона:
Подстрока 'ab' на позиции 1 → возможное начало шаблона: 1 (C[1] = 1)
Подстрока 'c' на позиции 4 → возможное начало шаблона: 0 (C[0] = 1)
Подстрока 'c' на позиции 5 → возможное начало шаблона: 1 (C[1] = 2)
Подстрока 'ab' на позиции 7 → возможное начало шаблона: 7 (C[7] = 1)
Подстрока 'ab' на позиции 9 → возможное начало шаблона: 9 (C[9] = 1)
Подстрока 'c' на позиции 11 → возможное начало шаблона: 7 (C[7] = 2)

Найденные позиции: [2, 8]
2
8
```

Рисунок 4 – Пример вывода поиска модифицированного алгоритма

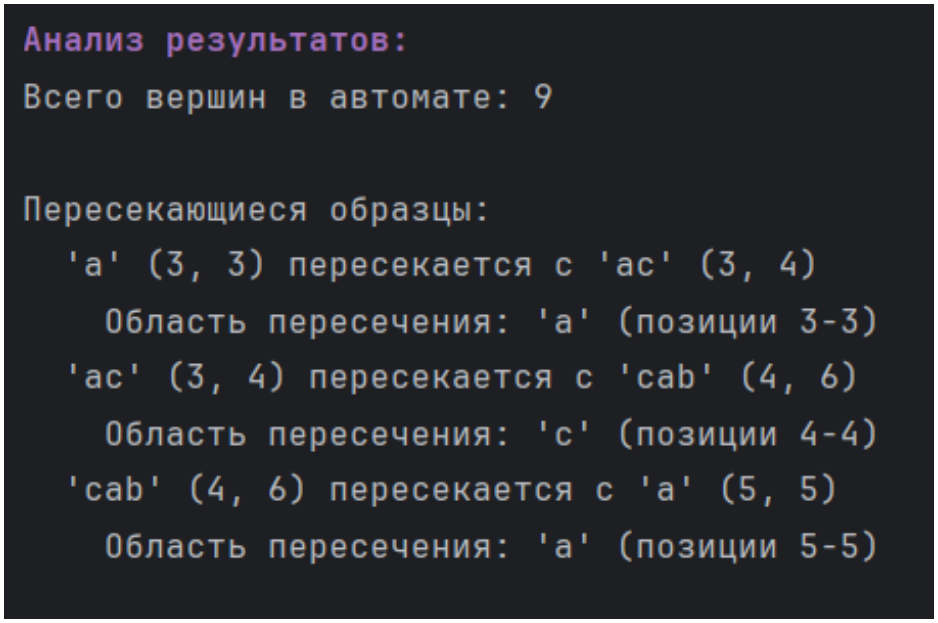

Рисунок 5 – Пример вывода подсчёта вершин и пересечений

Исследование.

Для проведения исследования, оценим рост временной сложности создания бора и поиска в тексте и сравним с теоретическими данными.

Таблица 3 – Исследование эффективности алгоритма по времени

Суммарный размер паттернов	Время создания бора
10	0.000025
100	0.000046
1000	0.000181
10000	0.001732
100000	0.017246
Размер текста	Время поиска вхождений
100	0.000235
1000	0.003478
10000	0.024324

100000	0.216399
1000000	1.840552

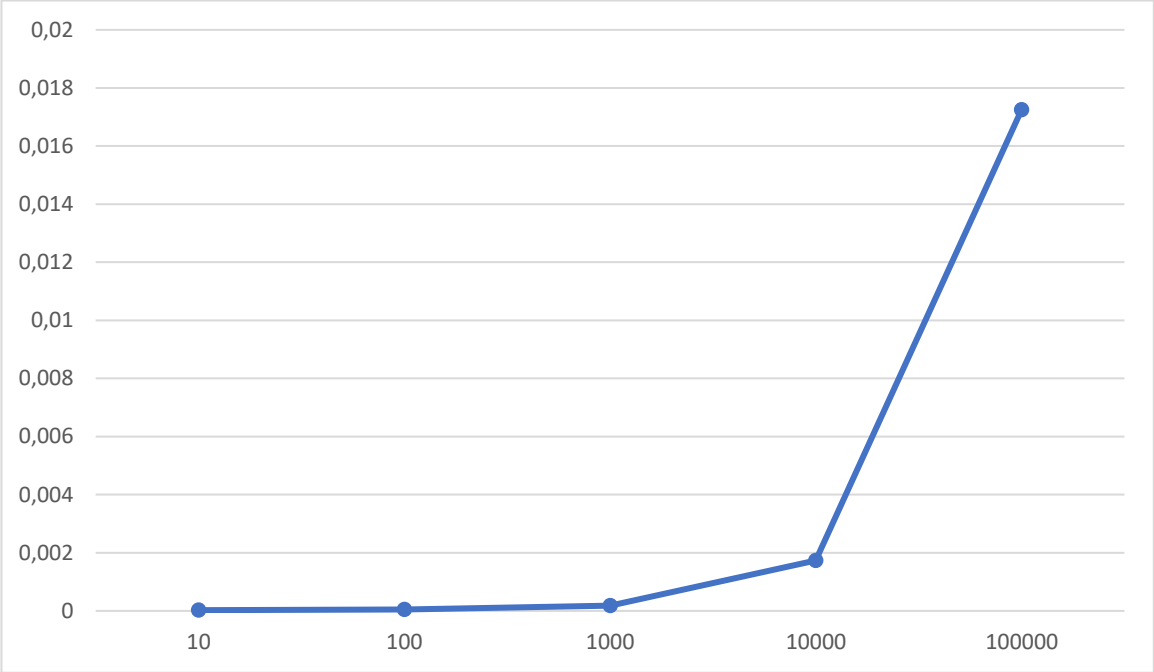

Рисунок 6 – Зависимость времени создания бора от суммарного размера паттернов

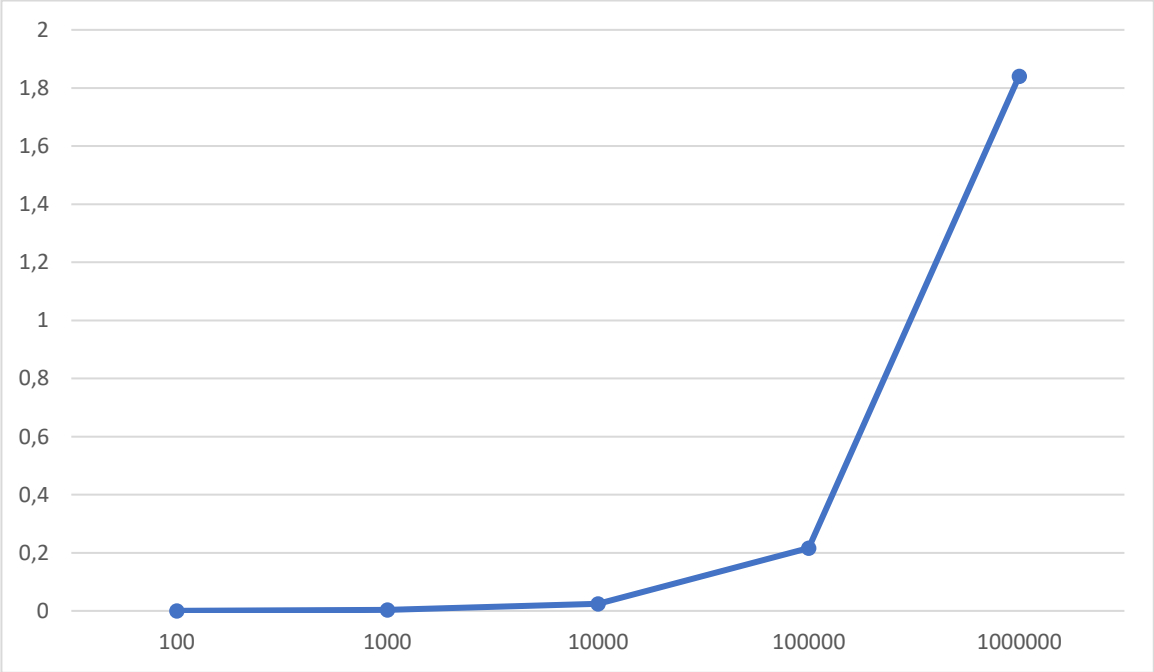

Рисунок 7 – Зависимость времени поиска от размера текста

Можно сделать вывод, что время построения бора и поиска в тексте линейно и зависит соответственно от размеров паттернов и текста, что подтверждает теоретическую оценку. Это можно увидеть в таблице или на графике.

Выводы.

Изучен принцип работы алгоритма Ахо-Корасик. Написаны программы, решающие задачу поиска паттернов в тексте и поиска подстроки с джокером.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Aho-Corasick.py

```
from collections import deque


class TrieNode:
    def __init__(self, char=None):
        self.char = char
        self.parent = None
        self.children = {}
        self.fail = None
        self.output = set()
        self.is_terminal = False


class AhoCorasick:
    def __init__(self, patterns):
        self.root = TrieNode()
        self.patterns = patterns
        self.build_trie()
        self.build_links()

    def build_trie(self):
        print("\033[1;34m1. Построение бора:\033[0m")
        for index, pattern in enumerate(self.patterns):
            print(f"\nДобавление образца #{index + 1}: '{pattern}'")
            node = self.root
            for i, char in enumerate(pattern):
                if char not in node.children:
                    print(f"   Создание узла для символа '{char}'
(позиция {i + 1})")
                    node.children[char] = TrieNode(char)
                    node.children[char].parent = node
                else:
                    print(f"  Символ '{char}' уже существует в боре
(позиция {i + 1})")
                node = node.children[char]
            node.output.add(index)
            node.is_terminal = True
            print(f"   Образец  '{pattern}'  добавлен  (терминальный
узел)")

    def build_links(self):
        print("\n\033[1;34m2. Построение суффиксных и терминальных
ссылок:\033[0m")
        queue = deque()
        self.root.fail = self.root
        print("Установка fail-ссылок для детей корня:")
        for char, child in self.root.children.items():
            child.fail = self.root
```

```
                queue.append(child)
                print(f"  Узел '{char}': fail → корень")

            while queue:
                current_node = queue.popleft()
                print(f"\nОбработка      узла      '{current_node.char}'
(родитель:   '{"root"   if   current_node.parent.char   is   None   else
current_node.parent.char}')")

                for char, child in current_node.children.items():
                    fail_node = current_node.fail
                    print(f"  Обработка дочернего узла '{char}':")

                    while  fail_node  !=  self.root  and  char  not  in
fail_node.children:
                        print(f"    Переход по fail: '{fail_node.char}'
→ '{fail_node.fail.char}'")
                        fail_node = fail_node.fail

                    if char in fail_node.children:
                        child.fail = fail_node.children[char]
                        print(f"         Установка  fail:  '{char}'  →
'{child.fail.char}'")
                    else:
                        child.fail = self.root
                        print("    Установка fail: → root")
                    child.output.update(child.fail.output)
                    if child.fail.output:
                        print(f"      Добавлены  output  из  fail-ссылки:
{child.fail.output}")

                    queue.append(child)

        def print_trie(self, node=None, level=0):
            if node is None:
                node = self.root
                print("\n\033[1;34m3. Визуализация бора:\033[0m")

            prefix = "    " * level
            char = node.char if node.char else "root"
            term = " [TERM]" if node.is_terminal else ""
            fail  =  f"  (fail:  '{node.fail.char  if  node.fail  and
node.fail.char else 'root'}')" if node.fail else ""
            out = f" [output: {node.output}]" if node.output else ""
            print(f"{prefix}└── {char}{term}{fail}{out}")

            for child in node.children.values():
                self.print_trie(child, level + 1)

        def search(self, text):
            print(f"\n\033[1;34m4. Поиск в тексте '{text}':\033[0m")
            current_node = self.root
            results = []

            for pos, char in enumerate(text):
                print(f"\nСимвол '{char}' (позиция {pos + 1}):")
```

```
                while current_node != self.root and char not in
current_node.children:
                    print(f"  Нет перехода по '{char}', переход по fail:
'{current_node.char}' → '{current_node.fail.char}'")
                    current_node = current_node.fail

                if char in current_node.children:
                    current_node = current_node.children[char]
                    print(f"  Переход в узел '{current_node.char}'")
                    if    current_node.fail    !=    self.root    and
current_node.fail.output:
                        print(f"  Узел '{current_node.char}' унаследовал
output через fail-ссылку: {current_node.fail.output}")
                else:
                    current_node = self.root
                    print("  Возврат в корень")

                if current_node.output:
                    print(f"   Найдены подстроки {current_node.output}
через терминальные ссылки")
                    for pattern_index in current_node.output:
                        pattern = self.patterns[pattern_index]
                        start = pos - len(pattern) + 2
                        results.append((start,   pattern_index   +   1,
pattern))
                        print(f"        Образец  #{pattern_index  +  1}
'{pattern}' начинается на позиции {start}")

            return results

        def analyze_patterns(self, results, text):
            total_nodes = count_nodes(self.root)
            print(f"\n\033[1;35mАнализ результатов:\033[0m")
            print(f"Всего вершин в автомате: {total_nodes}")

            overlapping = []
            results_sorted = sorted(results, key=lambda x: x[0])

            for i in range(len(results_sorted)):
                start_i, _, pattern_i = results_sorted[i]
                end_i = start_i + len(pattern_i) - 1

                for j in range(i + 1, len(results_sorted)):
                    start_j, _, pattern_j = results_sorted[j]
                    end_j = start_j + len(pattern_j) - 1

                    if start_j <= end_i:
                        overlapping.append((pattern_i,       pattern_j,
(start_i, end_i), (start_j, end_j)))
                    else:
                        break

            if overlapping:
                print("\nПересекающиеся образцы:")
                for pattern1, pattern2, pos1, pos2 in overlapping:
                    print(f"     '{pattern1}'   {pos1}   пересекается   с
'{pattern2}' {pos2}")
```

```
                    overlap_start = max(pos1[0], pos2[0])
                    overlap_end = min(pos1[1], pos2[1])
                    overlap_text = text[overlap_start - 1:overlap_end]
                    print(f"     Область пересечения: '{overlap_text}'
(позиции {overlap_start}-{overlap_end})")
            else:
                print("\nПересекающихся образцов не найдено")

            return total_nodes, overlapping


    def count_nodes(node):
        count = 1
        for child in node.children.values():
            count += count_nodes(child)
        return count


    print("Введите текст:")
    text = input()

    print("Введите паттерны:")
    patterns = input().split()

    print("\033[1;34mАлгоритм Ахо-Корасик\033[0m")
    print(f"Текст для поиска: '{text}'")
    print(f"Образцы для поиска: {patterns}\n")

    aho = AhoCorasick(patterns)
    aho.print_trie()

    results = aho.search(text)

    print("\n\033[1;34mРезультаты поиска\033[0m")
    for start_pos, pattern_idx, pattern in sorted(results):
        print(f"На позиции {start_pos} найден образец #{pattern_idx}
'{pattern}'")

aho.analyze_patterns(results, text)
```

Название файла: joker.py

```
from collections import deque



class TrieNode:
    def __init__(self, char=None):
        self.char = char
        self.parent = None
        self.children = {}
        self.fail = None
```

```
        self.output = set()
        self.is_terminal = False


class AhoCorasick:
    def __init__(self, patterns):
        self.root = TrieNode()
        self.patterns = patterns
        self.build_trie()
        self.build_links()

    def build_trie(self):
        for index, pattern in enumerate(self.patterns):
            print(f"\nДобавление образца #{index + 1}: '{pattern}'")
            node = self.root
            for i, char in enumerate(pattern):
                if char not in node.children:
                    print(f"  Создание узла для символа '{char}' (позиция
{i + 1})")
                    node.children[char] = TrieNode(char)
                    node.children[char].parent = node
                else:
                    print(f"   Символ  '{char}'  уже  существует  в  боре
(позиция {i + 1})")
                node = node.children[char]
            node.output.add(index)
            node.is_terminal = True
            print(f"  Образец '{pattern}' добавлен (терминальный узел)")

    def build_links(self):
        print("\n\033[1;34mПостроение    суффиксных    и    терминальных
ссылок:\033[0m")
        queue = deque()
        self.root.fail = self.root
        print("Установка fail-ссылок для детей корня:")
        for char, child in self.root.children.items():
            child.fail = self.root
```

```
            queue.append(child)
            print(f"  Узел '{char}': fail → корень")

        while queue:
            current_node = queue.popleft()
            print(f"\nОбработка узла '{current_node.char}' (родитель: '{"root" if current_node.parent.char is None else current_node.parent.char}')")

            for char, child in current_node.children.items():
                fail_node = current_node.fail
                print(f"  Обработка дочернего узла '{char}':")

                while fail_node != self.root and char not in fail_node.children:
                    print(f"    Переход по fail: '{fail_node.char}' → '{fail_node.fail.char}'")
                    fail_node = fail_node.fail

                if char in fail_node.children:
                    child.fail = fail_node.children[char]
                    print(f"          Установка fail: '{char}' → '{child.fail.char}'")
                else:
                    child.fail = self.root
                    print("    Установка fail: → root")

                child.output.update(child.fail.output)
                if child.fail.output:
                    print(f"        Добавлены output из fail-ссылки: {child.fail.output}")

                queue.append(child)

    def print_trie(self, node=None, level=0):
        if node is None:
            node = self.root
            print("\n\033[1;34mВизуализация бора:\033[0m")
```

```
        prefix = "    " * level
        char = node.char if node.char else "root"
        term = " [TERM]" if node.is_terminal else ""
        fail = f" (fail: '{node.fail.char if node.fail and node.fail.char else 'root'}')" if node.fail else ""
        out = f" [output: {node.output}]" if node.output else ""
        print(f"{prefix}└── {char}{term}{fail}{out}")

        for child in node.children.values():
            self.print_trie(child, level + 1)

    def find_matches(self, text):
        current_node = self.root
        results = []
        for pos, char in enumerate(text):
            while current_node != self.root and char not in current_node.children:
                current_node = current_node.fail
            if char in current_node.children:
                current_node = current_node.children[char]
            else:
                current_node = self.root
            for pattern_index in current_node.output:
                pattern = self.patterns[pattern_index]
                start_pos = pos - len(pattern) + 1
                results.append((start_pos, pattern_index))
        print(results)
        return results


def find_wildcard_matches(text, pattern, wildcard):
    print("\n\033[1;34mШаблон с джокерами\033[0m")
    print(f"Текст: '{text}'")
    print(f"Шаблон: '{pattern}' (джокер: '{wildcard}')")

    print("\n\033[1;34m1. Разбиение шаблона на подстроки:\033[0m")
```

```
    subpatterns, positions, current = [], [], []
    start_pos = 0

    for i, char in enumerate(pattern):
        if char == wildcard:
            if current:
                subpatterns.append(''.join(current))
                positions.append((start_pos, i - 1))
                print(f"Найдена подстрока: '{''.join(current)}' (позиции
в шаблоне: {start_pos}-{i-1})")
                current = []
            start_pos = i + 1
        else:
            current.append(char)

    if current:
        subpatterns.append(''.join(current))
        positions.append((start_pos, len(pattern) - 1))
        print(f"Найдена подстрока: '{''.join(current)}' (позиции в
шаблоне: {start_pos}-{len(pattern)-1})")

    if not subpatterns:
        print("Нет подстрок для поиска")
        return []

    print("\n\033[1;34m2. Построение автомата Ахо-Корасик:\033[0m",
end="")
    aho = AhoCorasick(subpatterns)
    print(f"Добавлено подстрок: {len(subpatterns)}")

    aho.print_trie()

    print("\n\033[1;34m3. Поиск подстрок в тексте:\033[0m")
    matches = aho.find_matches(text)
    print(f"Найдено совпадений: {len(matches)}")

    print("\n\033[1;34m4. Подсчет возможных начал шаблона:\033[0m")
```

```
    C = [0] * (len(text) + len(pattern)) # C[i] = количество подстрок - вероятных начал шаблона
    for pos, pattern_idx in matches:
        possible_start = pos - positions[pattern_idx][0]
        C[pos - (positions[pattern_idx][0])] += 1
        print(f"Подстрока '{subpatterns[pattern_idx]}' на позиции {pos} → возможное начало шаблона: {possible_start} (C[{possible_start}] = {C[possible_start]})")

    required = len(subpatterns)
    possible_starts = [i for i in range(len(C)) if C[i] == required]

    valid_starts = []
    for start in possible_starts:
        if start + len(pattern) <= len(text):
            valid_starts.append(start + 1)
        else:
            print(f"Отброшена позиция {start} (выходит за границы текста)")

    print(f"\n\033[1;32mНайденные позиции: {sorted(valid_starts)}\033[0m")
    return sorted(valid_starts)



print("Введите текст:")
text = input()

print("Введите паттерн с джокерами:")
pattern = input()

print("Введите символ джокера:")
wildcard = input()

matches = find_wildcard_matches(text, pattern, wildcard)
for match in sorted(matches):
    print(match)
```