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
Вариант 2

Студент гр. 3388 _____ Лексин М.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Разработка и реализация алгоритма Ахо-Корасик для множественного поиска образцов в тексте, а также его модификация для решения задачи поиска образца с джокером. Изучение понятий бор и суффиксные ссылки, сжатые суффиксные ссылки, их принципов работы и применения для оптимизации поиска подстрок.

Задание.

Задание 1:

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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 $\{A,C,G,T,N\}$

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?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 $\{A,C,G,T,N\}$

Вход:

Текст (T,$1 \le |T| \le 100000$)

Шаблон (P,$1 \le |P| \le 40$)

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 2:

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

Описание алгоритма.

1. Задача точного поиска набора образцов

Алгоритм Ахо-Корасик используется для эффективного поиска набора образцов в тексте за один проход. Он позволяет находить все вхождения каждого образца из заданного набора. Алгоритм комбинирует использование структуры данных бор и дополнительные переходы для эффективного поиска без возвратов.

Основные этапы алгоритма:

1. Построение бора (префиксного дерева):
   - Создаётся корневая вершина бора
   - Каждый образец добавляется в бор, создавая путь от корня к листовой вершине
   - Вершины, соответствующие концам образцов, помечаются как терминальные с сохранением номера образца
   - Для каждой вершины сохраняется информация о родителе и символе, по которому происходит переход к вершине

2. Построение суффиксных ссылок:
    - Для каждой вершины бора вычисляется суффиксная ссылка, указывающая на вершину, соответствующую наибольшему собственному суффиксу текущей вершины
    - Построение производится с использованием ленивых вычислений, когда суффиксная ссылка для вершины определяется только при необходимости и затем сохраняется
3. Построение сжатых суффиксных ссылок:
    - Для оптимизации поиска создаются сжатые суффиксные ссылки, которые указывают на ближайшую терминальную вершину в цепочке переходов по обычным суффиксным ссылкам
    - Эти ссылки позволяют за один шаг перейти к терминальной вершине, минуя промежуточные нетерминальные вершины
4. Поиск образцов в тексте:
    - Текст обрабатывается посимвольно
    - Для каждого символа выполняется переход в автомате с использованием функции перехода, которая учитывает как прямые переходы в боре, так и переходы по суффиксным ссылкам
    - После каждого перехода проверяется, является ли текущая вершина терминальной, и если так, то фиксируется вхождение соответствующего образца
    - Затем выполняются переходы по сжатым суффиксным ссылкам для проверки вхождений более коротких образцов
5. Сортировка результатов: найденные вхождения сортируются сначала по позиции в тексте, затем по номеру образца.

Оценка сложности:

Временная сложность алгоритма складывается из следующих составляющих:

- Построение бора. Если обозначить суммарную длину всех образцов как m, то построение бора выполняется за O(m) времени. Каждый символ каждого образца обрабатывается ровно один раз, добавляясь в соответствующую позицию бора.
- Построение суффиксных ссылок. Благодаря ленивым вычислениям и мемоизации, каждая суффиксная ссылка вычисляется не более одного раза. Суммарное время на построение всех суффиксных ссылок – O(m).
- Обработка текста. Если n – длина текста, а k – число образцов, то процесс обработки текста занимает в худшем случае O(n*k) времени. Это происходит потому, что для каждого символа текста может потребоваться обход цепочки сжатых суффиксных ссылок, который в худшем случае может включать до k переходов.
- Сортировка результатов. Если z – общее количество найденных вхождений, то сортировка занимает O(z*log(z)) времени.

Общая временная сложность алгоритма составляет O(m+n*k+z*log(z)).

Пространственная сложность определяется используемыми структурами данных:

- Бор. Содержит O(m) вершин, где m – суммарная длина образцов. Каждая вершина хранит константное количество информации. Переходы в каждой вершине хранятся в виде словаря, что экономит память по сравнению с хранением всей матрицы переходов.
- Суффиксные и сжатые суффиксные ссылки. Для каждой вершины бора хранится по одной суффиксной и сжатой суффиксной ссылке, что требует O(m) дополнительной памяти.
- Очередь для обхода в ширину. При построении суффиксных и сжатых ссылок используется очередь, размер которой в худшем случае может достигать O(m).
- Структуры для поиска и хранения результатов. Требуется O(z) памяти для хранения найденных вхождений, где z – общее количество вхождений.

Общая сложность по памяти составляет O(m+z).

2. Задача точного поиска для одного образца с джокером

Алгоритм должен найти все вхождения образца, содержащего специальный символ-джокер, который может соответствовать любому символу в тексте. Образец разбивается на подстроки без джокеров, которые затем ищутся в тексте, после чего проверяется полное соответствие образца с учётом джокеров. Основные этапы алгоритма:

1. Разбиение образца на подстроки:
   - Образец анализируется посимвольно, при встрече джокера текущая накопленная подстрока добавляется в список подстрок вместе с её начальной позицией образце
   - Этот процесс продолжается до конца образца, формируя набор подстрок без джокеров
2. Построение бора для подстрок:
   - Для всех подстрок без джокеров строится бор
   - В боре каждая терминальная вершина хранит информацию о номере подстроки и её начальной позиции в образце
3. Построение суффиксных и сжатых ссылок:
   - Механизм построения аналогичен использующемуся в первом задании: для каждой вершины вычисляются суффиксные и сжатые суффиксные ссылки, используя ленивые вычисления для эффективности
4. Поиск подстрок в тексте:
   - С помощью автомата Ахо-Корасик ищутся все вхождения всех подстрок в текст
   - Для каждой найденной подстроки вычисляется потенциальная начальная позиция образца в тексте, и эта информация сохраняется в специальном массиве
5. Проверка полных совпадений: для каждой потенциальной позиции начала образца проверяется

- Найдены ли все подстроки образца
- Соответствуют ли позиции найденных подстрок их позициям в образце
- Точное совпадение образца с учётом джокеров

6. Сортировка результатов: найденные позиции вхождений образца сортируются по возрастанию.

Оценка сложности:

Временная сложность алгоритма определяется следующими этапами:

- Разбиение образца на подстроки. Если p – длина образца, то разбиение выполняется за O(p) времени, требуя одного прохода по образцу.
- Построение бора, суффиксных и сжатых суффиксных ссылок. Аналогично первому заданию, занимает O(p) времени, поскольку суммарная длина всех подстрок не превышает длину образца.
- Поиск подстрок в тексте. Если n – длина текста, а s – количество подстрок образца (s <= p), то поиск выполняется за O(n*s) времени. Это происходит потому, что для каждого символа текста может потребоваться обход цепочки сжатых суффиксных ссылок, включающий до s переходов.
- Проверка полных совпадений. Для каждой потенциальной позиции начала образца (которых может быть до n) требуется проверка на полное соответствие за O(p) времени, что даёт в худшем случае O(n*p).

Общая временная сложность составляет O(p+n*s+n*p), что упрощается до O(p+n*p), так как s<=p.

Пространственная сложность определяется следующими компонентами:

- Бор и ссылки. Структуры бора, суффиксные и сжатые суффиксные ссылки занимают O(p) памяти, где p – длина образца.
- Очередь для обхода в ширину. При построении суффиксных и сжатых ссылок используется очередь, размер которой в худшем случае может достигать O(p).

- Хранение промежуточных результатов. Массив occurrences хранит для каждой позиции в тексте список найденных подстрок, что в худшем случае требует O(n*s) памяти, где n – длина текста, а s – количество подстрок
- Хранение результатов. Требуется O(r) памяти для хранения найденных вхождений образца, где r – количество вхождений.

Общая сложность по памяти составляет O(p+n*s+r), что в худшем случае можно упростить до O(p+n*p), так как s<=p и r<=n.

3. Индивидуализация

В рамках индивидуализации были реализованы два дополнительных задания: подсчёт вершин в автомате и поиск образцов, имеющих пересечения в строке поиска.

1. Подсчёт вершин в автомате: алгоритм выполняет обход в ширину всех вершин автомата, начиная с корня, и подсчитывает их количество. Основные этапы:
    - Инициализация пустого множества посещённых вершин и очереди с корнем бора
    - Последовательное извлечение вершин из очереди и добавление непосещённых вершин в множество
    - Добавление всех непосещённых дочерних вершин в очередь
    - Возврат размера множества посещённых вершин как результата
2. Поиск образцов с пересечениями: алгоритм анализирует найденные вхождения образцов и определяет, какие из них имеют пересечения друг с другом. Для первого задания ищутся пересечения между разными образцами, для второго – между разными вхождениями одного образца. Основные этапы:
    - Создание интервалов для каждого вхождения каждого образца (начало, конец, номер образца)
    - Сортировка интервалов по начальной позиции

- Последовательный анализ интервалов: если начало следующего интервала меньше или равно концу текущего, фиксируется пересечение
- Для первого задания: пересечения учитываются только между разными образцами
- Для второго задания: учитываются пересечения между разными вхождениями одного образца
- Возврат отсортированного списка образцов с пересечениями

Оценка сложности:

Временная сложность:

- Подсчёт вершин автомата выполняется за O(m) времени для первого задания, где m – суммарная длина образцов, и за O(p) времени для второго задания, где p – длина образца. Это обусловлено необходимостью обхода всех вершин автомата по одному разу.
- Поиск образцов с пересечениями имеет сложность $O(z^2)$ для первого задания, где z – общее количество вхождений всех образцов, и $O(r^2)$ для второго задания, где r – количество вхождений образца. Это связано с необходимостью попарного сравнения всех интервалов вхождений.

Пространственная сложность:

- Подсчёт вершин требует O(m) памяти для первого задания и O(p) для второго, что включает в себя множество посещённых вершин и очередь для обхода.
- Поиск пересечений требует O(z) памяти для первого задания и O(r) для второго, что определяется размером массива интервалов вхождений.

Общая дополнительная сложность по памяти для индивидуализации не превышает сложность основных алгоритмов и составляет O(m+z) или O(p+r) соответственно.

Тестирование.

Алгоритм был протестирован на различных наборах входных данных. Тестирование первой задачи:

Табл.1

Входные данные	Выходные данные
AAAA 1 A	=== РЕЗУЛЬТАТЫ === 1 1 2 1 3 1 4 1 Количество вершин в автомате: 2 Образцы с пересечениями: []
ACGACGACG 3 ACG ACGACG ACGACGACG	=== РЕЗУЛЬТАТЫ === 1 1 1 2 1 3 4 1 4 2 7 1 Количество вершин в автомате: 10 Образцы с пересечениями: [1, 2, 3]
ACTG 1 CGG	=== РЕЗУЛЬТАТЫ === Количество вершин в автомате: 4 Образцы с пересечениями: []
ACGT 2 AC AC	=== РЕЗУЛЬТАТЫ === 1 1 1 2 Количество вершин в автомате: 3 Образцы с пересечениями: [1, 2]
ACGT 1 ACGT	=== РЕЗУЛЬТАТЫ === 1 1 Количество вершин в автомате: 5 Образцы с пересечениями: []

Тестирование второй задачи:

Табл.2

Входные данные	Выходные данные
ACGT ACGT $	=== РЕЗУЛЬТАТЫ === 1 Количество вершин в автомате: 5 Образцы с пересечениями: []

ACGTACGT A$$T $	=== РЕЗУЛЬТАТЫ === 1 5 Количество вершин в автомате: 3 Образцы с пересечениями: []
ACGT $$$Z $	=== РЕЗУЛЬТАТЫ === Количество вершин в автомате: 2 Образцы с пересечениями: []
AAAA A$$ $	=== РЕЗУЛЬТАТЫ === 1 2 Количество вершин в автомате: 2 Образцы с пересечениями: [1]
ACGTACGT CTCT $	=== РЕЗУЛЬТАТЫ === 1 Количество вершин в автомате: 3 Образцы с пересечениями: []

Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм Ахо-Корасик для точного множественного поиска образцов в тексте. Также была разработана модификация данного алгоритма для решения задачи поиска образца с джокером, для обоих алгоритмов дополнительно был выполнен подсчет количества вершин в автомате и поиск образцов с пересечениями. Проведено тестирование корректности работы обоих алгоритмов.

ПРИЛОЖЕНИЕ

Код.

Файл: main1.py

```
class Node:
    def __init__(self):
        self.sons = {}  # Словарь сыновей (символ -> узел)
        self.go = {}    # Словарь переходов (символ -> узел) для ленивой
рекурсии
        self.parent = None  # Родительский узел
        self.suffLink = None  # Суффиксная ссылка
        self.up = None  # Сжатая суффиксная ссылка
        self.charToParent = None  # Символ, ведущий к родителю
        self.isLeaf = False  # Флаг терминала
        self.leafPatternNumbers = [] # Номера строк, за которые отвечает
терминал
        self.id = None  # Идентификатор для вывода

def addString(root, s, patternNumber):
    # Добавляет строку в бор
    print(f"\n---> Добавление строки '{s}' (образец #{patternNumber}) в бор:")
    cur = root
    path = "Корень"  # Для вывода пути

    for i, c in enumerate(s):
        if c not in cur.sons:
            # Создаем новую вершину
            cur.sons[c] = Node()
            cur.sons[c].parent = cur
            cur.sons[c].charToParent = c
            cur.sons[c].id = f"{s[:i+1]}"  # Используем префикс строки как id
            print(f"  Создана новая вершина: {cur.sons[c].id} (добавлен переход
'{c}' из '{path}')")
        else:
            print(f"  Вершина для перехода '{c}' из '{path}' уже существует:
{cur.sons[c].id}")

        cur = cur.sons[c]
        path = cur.id  # Обновляем путь для вывода

    cur.isLeaf = True
    cur.leafPatternNumbers.append(patternNumber)
    print(f"  Вершина {cur.id} помечена как терминал для образца
#{patternNumber}")

def getSuffLink(v, root):
    # Вычисляет суффиксную ссылку для вершины v
    if v.suffLink is None:
        # Если суффиксная ссылка еще не вычислена
        if v == root or v.parent == root:
            v.suffLink = root
            if v != root:
                print(f"  Суффиксная ссылка для {v.id}: → Корень (родитель
вершины - корень)")
        else:
            v.suffLink = getLink(getSuffLink(v.parent, root), v.charToParent,
root)
            print(f"  Суффиксная ссылка для {v.id}: → {v.suffLink.id}")
    return v.suffLink
```

```
def getLink(v, c, root):
    # Вычисляет переход из вершины v по символу c
    if c not in v.go:
        # Если переход еще не вычислен
        v_id = "Корень" if v == root else v.id

        if c in v.sons:
            v.go[c] = v.sons[c]
            print(f"  Переход из {v_id} по символу '{c}': → {v.sons[c].id}
(прямой переход)")
        elif v == root:
            v.go[c] = root
            print(f"  Переход из Корня по символу '{c}': → Корень (нет перехода,
остаемся в корне)")
        else:
            v.go[c] = getLink(getSuffLink(v, root), c, root)
            dest_id = "Корень" if v.go[c] == root else v.go[c].id
            print(f"  Переход из {v_id} по символу '{c}': → {dest_id} (через
суффиксную ссылку)")
    return v.go[c]

def getUp(v, root):
    # Вычисляет сжатую суффиксную ссылку для вершины v
    if v.up is None:
        v_id = "Корень" if v == root else v.id
        suffLink = getSuffLink(v, root)
        suffLink_id = "Корень" if suffLink == root else suffLink.id

        if suffLink.isLeaf:
            v.up = suffLink
            print(f"  Сжатая суфф. ссылка для {v_id}: → {suffLink_id}
(терминальная вершина)")
        elif suffLink == root:
            v.up = root
            print(f"  Сжатая суфф. ссылка для {v_id}: → Корень (суфф. ссылка -
корень)")
        else:
            v.up = getUp(suffLink, root)
            up_id = "Корень" if v.up == root else v.up.id
            print(f"  Сжатая суфф. ссылка для {v_id}: → {up_id} (через сжатую
суфф. ссылку родителя)")
    return v.up

def countVertices(root):
    # Подсчитывает количество вершин в автомате
    if not root:
        return 0

    print("\n---> Подсчет вершин в автомате:")
    visited = set()
    queue = [root]

    # Обход в ширину для подсчета вершин
    while queue:
        node = queue.pop(0)
        if node in visited:
            continue

        visited.add(node)
        node_id = "Корень" if node == root else node.id
        print(f"  Посещена вершина: {node_id}")

        for c, child in node.sons.items():
            if child not in visited:
                queue.append(child)
```

```
                child_id = "Корень" if child == root else child.id
                print(f"    Добавлен в очередь потомок: {child_id} (по символу
'{c}')")

    print(f"  Всего вершин: {len(visited)}")
    return len(visited)

def processText(root, text, patterns):
    # Обрабатывает текст и находит все вхождения образцов
    print("\n---> Обработка текста:")
    result = []
    patternOccurrences = {}  # Словарь для хранения вхождений образцов

    cur = root
    print(f"  Начинаем с корня бора")

    for i, c in enumerate(text):
        # Переход по символу текста
        prev_state = "Корень" if cur == root else cur.id
        cur = getLink(cur, c, root)
        cur_state = "Корень" if cur == root else cur.id
        print(f"  Символ текста[{i}] = '{c}': {prev_state} → {cur_state}")

        # Проверяем текущий узел на наличие терминалов
        if cur.isLeaf:
            print(f"    Вершина {cur_state} является терминалом:")
            for patternNumber in cur.leafPatternNumbers:
                pattern = patterns[patternNumber-1]
                pos = i - len(pattern) + 2
                print(f"      Найден образец #{patternNumber} '{pattern}' на
позиции {pos}")
                result.append((pos, patternNumber))

                # Сохраняем информацию о вхождении для поиска пересечений
                if patternNumber not in patternOccurrences:
                    patternOccurrences[patternNumber] = []
                patternOccurrences[patternNumber].append(pos)

        # Проходим по сжатым суффиксным ссылкам
        node = getUp(cur, root)
        up_followed = False
        while node != root:
            up_followed = True
            node_state = "Корень" if node == root else node.id
            print(f"    Переход по сжатой суфф. ссылке: {cur_state} →
{node_state}")

            if node.isLeaf:
                print(f"      Вершина {node_state} является терминалом:")
                for patternNumber in node.leafPatternNumbers:
                    pattern = patterns[patternNumber-1]
                    pos = i - len(pattern) + 2
                    print(f"        Найден образец #{patternNumber} '{pattern}'
на позиции {pos}")
                    result.append((pos, patternNumber))

                    if patternNumber not in patternOccurrences:
                        patternOccurrences[patternNumber] = []
                    patternOccurrences[patternNumber].append(pos)

            node = getUp(node, root)

        if not up_followed:
            print(f"    Нет переходов по сжатым суфф. ссылкам или переход ведет
в корень")
```

```
    # Находим образцы с пересечениями
    intersectingPatterns = findIntersectingPatterns(patternOccurrences,
patterns)

    return result, intersectingPatterns

def findIntersectingPatterns(patternOccurrences, patterns):
    # Находит образцы, имеющие пересечения с другими образцами в строке поиска
    print("\n---> Поиск образцов с пересечениями:")
    intersectingPatterns = set()

    # Создаем интервалы для каждого вхождения образца
    intervals = []
    for patternNumber, positions in patternOccurrences.items():
        patternLength = len(patterns[patternNumber-1])
        for pos in positions:
            intervals.append((pos, pos + patternLength - 1, patternNumber))

    # Сортируем интервалы по начальной позиции
    intervals.sort()
    print(f"  Созданы интервалы вхождений образцов: {intervals}")

    # Проверяем пересечения
    for i in range(len(intervals)):
        start1, end1, pattern1 = intervals[i]
        for j in range(i+1, len(intervals)):
            start2, end2, pattern2 = intervals[j]

            # Если начало второго интервала за концом первого, то пересечений
нет
            if start2 > end1:
                break

            # Если есть пересечение и это разные образцы
            if pattern1 != pattern2:
                print(f"  Найдено пересечение образцов #{pattern1} и
#{pattern2}:")
                print(f"    Образец #{pattern1}: позиции {start1}-{end1}")
                print(f"    Образец #{pattern2}: позиции {start2}-{end2}")
                intersectingPatterns.add(pattern1)
                intersectingPatterns.add(pattern2)

    if not intersectingPatterns:
        print("  Не найдено образцов с пересечениями")
    else:
        print(f"  Образцы с пересечениями:
{sorted(list(intersectingPatterns))}")

    return sorted(list(intersectingPatterns))

def ahoCorasick(text, patterns):
    # Основная функция алгоритма Ахо-Корасик
    print("\n=== АЛГОРИТМ АХО-КОРАСИК ===")
    print(f"Текст: '{text}'")
    print(f"Образцы: {patterns}")

    # Инициализация корня бора
    root = Node()
    root.id = "Корень"
    print("\n--- Шаг 1: Построение бора ---")

    # Построение бора
    for i, pattern in enumerate(patterns):
        addString(root, pattern, i+1)
```

```
    print("\n--- Шаг 2: Построение суффиксных ссылок ---")
    # Вычисляем суффиксные ссылки для всех вершин
    # Выполним обход бора в ширину для построения всех суффиксных ссылок
    queue = [root]
    visited = set([root])

    while queue:
        node = queue.pop(0)
        # Вычисляем суффиксную ссылку для текущей вершины
        if node != root:
            getSuffLink(node, root)

        # Добавляем потомков в очередь
        for child in node.sons.values():
            if child not in visited:
                queue.append(child)
                visited.add(child)

    print("\n--- Шаг 3: Построение сжатых суффиксных ссылок ---")
    # Вычисляем сжатые суффиксные ссылки для всех вершин
    queue = [root]
    visited = set([root])

    while queue:
        node = queue.pop(0)
        # Вычисляем сжатую суффиксную ссылку для текущей вершины
        if node != root:
            getUp(node, root)

        # Добавляем потомков в очередь
        for child in node.sons.values():
            if child not in visited:
                queue.append(child)
                visited.add(child)

    # Подсчет количества вершин в автомате
    verticesCount = countVertices(root)

    print("\n--- Шаг 4: Поиск образцов в тексте ---")
    result, intersectingPatterns = processText(root, text, patterns)

    # Сортировка результатов
    result.sort()
    print(f"\nНайденные вхождения (всего {len(result)}): {result}")

    return result, verticesCount, intersectingPatterns

text = input().strip()
n = int(input().strip())
patterns = []
for _ in range(n):
    patterns.append(input().strip())

result, verticesCount, intersectingPatterns = ahoCorasick(text, patterns)
print("\n=== РЕЗУЛЬТАТЫ ===")
for pos, patternNumber in result:
    print(pos, patternNumber)

print(f"Количество вершин в автомате: {verticesCount}")
print(f"Образцы с пересечениями: {intersectingPatterns}")
```

Файл: main2.py

```
class Node:
    def __init__(self):
        self.sons = {}  # Словарь сыновей
        self.go = {}    # Словарь переходов
        self.parent = None  # Родительский узел
        self.suffLink = None  # Суффиксная ссылка
        self.up = None  # Сжатая суффиксная ссылка
        self.charToParent = None  # Символ, ведущий к родителю
        self.isLeaf = False  # Флаг терминала
        self.leafPatternNumbers = []  # Номера строк и их стартовые позиции
        self.id = None  # Уникальный идентификатор для вывода

def addString(root, s, patternNumber, startPos):
    # Добавляет строку в бор с информацией о стартовой позиции в шаблоне
    print(f"\n---> Добавление подстроки '{s}' (образец #{patternNumber}, поз.
{startPos}) в бор:")
    cur = root
    path = "Корень"  # Для вывода пути

    for i, c in enumerate(s):
        if c not in cur.sons:
            # Создаем новую вершину
            cur.sons[c] = Node()
            cur.sons[c].parent = cur
            cur.sons[c].charToParent = c
            cur.sons[c].id = f"П{patternNumber}_{startPos}_{s[:i+1]}"  #
Используем префикс строки как id
            print(f"  Создана новая вершина: {cur.sons[c].id} (добавлен переход
'{c}' из '{path}')")
        else:
            print(f"  Вершина для перехода '{c}' из '{path}' уже существует:
{cur.sons[c].id}")

        cur = cur.sons[c]
        path = cur.id  # Обновляем путь для вывода

    cur.isLeaf = True
    cur.leafPatternNumbers.append((patternNumber, startPos))
    print(f"  Вершина {cur.id} помечена как терминал для подстроки
#{patternNumber} с позиции {startPos}")

def getSuffLink(v, root):
    # Вычисляет суффиксную ссылку для вершины v
    if v.suffLink is None:
        # Если суффиксная ссылка еще не вычислена
        if v == root or v.parent == root:
            v.suffLink = root
            if v != root:
                print(f"  Суффиксная ссылка для {v.id}: → Корень (родитель
вершины - корень)")
        else:
            v.suffLink = getLink(getSuffLink(v.parent, root), v.charToParent,
root)
            print(f"  Суффиксная ссылка для {v.id}: → {v.suffLink.id}")
    return v.suffLink

def getLink(v, c, root):
    # Вычисляет переход из вершины v по символу c
    if c not in v.go:
        # Если переход еще не вычислен
        v_id = "Корень" if v == root else v.id

        if c in v.sons:
            v.go[c] = v.sons[c]
```

```
            print(f"  Переход из {v_id} по символу '{c}': → {v.sons[c].id}
(прямой переход)")
        elif v == root:
            v.go[c] = root
            print(f"  Переход из Корня по символу '{c}': → Корень (нет перехода,
остаемся в корне)")
        else:
            v.go[c] = getLink(getSuffLink(v, root), c, root)
            dest_id = "Корень" if v.go[c] == root else v.go[c].id
            print(f"  Переход из {v_id} по символу '{c}': → {dest_id} (через
суффиксную ссылку)")
    return v.go[c]

def getUp(v, root):
    # Вычисляет сжатую суффиксную ссылку для вершины v
    if v.up is None:
        v_id = "Корень" if v == root else v.id
        suffLink = getSuffLink(v, root)
        suffLink_id = "Корень" if suffLink == root else suffLink.id

        if suffLink.isLeaf:
            v.up = suffLink
            print(f"  Сжатая суфф. ссылка для {v_id}: → {suffLink_id}
(терминальная вершина)")
        elif suffLink == root:
            v.up = root
            print(f"  Сжатая суфф. ссылка для {v_id}: → Корень (суфф. ссылка -
корень)")
        else:
            v.up = getUp(suffLink, root)
            up_id = "Корень" if v.up == root else v.up.id
            print(f"  Сжатая суфф. ссылка для {v_id}: → {up_id} (через сжатую
суфф. ссылку родителя)")
    return v.up

def countVertices(root):
    # Подсчитывает количество вершин в автомате
    if not root:
        return 0

    print("\n---> Подсчет вершин в автомате:")
    visited = set()
    queue = [root]

    # Обход в ширину для подсчета вершин
    while queue:
        node = queue.pop(0)
        if node in visited:
            continue

        visited.add(node)
        node_id = "Корень" if node == root else node.id
        print(f"  Посещена вершина: {node_id}")

        for c, child in node.sons.items():
            if child not in visited:
                queue.append(child)
                child_id = "Корень" if child == root else child.id
                print(f"    Добавлен в очередь потомок: {child_id} (по символу
'{c}')")

    print(f"  Всего вершин: {len(visited)}")
    return len(visited)

def findPatternWithJoker(text, pattern, joker):
```

```
    # Находит все вхождения шаблона с джокером в текст
    print("\n=== АЛГОРИТМ ПОИСКА ОБРАЗЦА С ДЖОКЕРОМ ===")
    print(f"Текст: '{text}'")
    print(f"Шаблон: '{pattern}'")
    print(f"Джокер: '{joker}'")

    # Разбиваем шаблон на подстроки без джокеров
    print("\n--- Шаг 1: Разбиение шаблона на подстроки без джокеров ---")
    substrings = []
    currentSubstring = ""
    currentStart = 0

    for i, c in enumerate(pattern):
        if c == joker:
            if currentSubstring:
                substrings.append((currentSubstring, currentStart))
                print(f"  Найдена подстрока: '{currentSubstring}' (начинается с
позиции {currentStart})")
                currentSubstring = ""
            currentStart = i + 1
            print(f"  Встречен джокер на позиции {i}, следующая подстрока
начнется с позиции {currentStart}")
        else:
            if not currentSubstring:
                currentStart = i
                print(f"  Начинаем собирать новую подстроку с позиции {i}")
            currentSubstring += c

    if currentSubstring:
        substrings.append((currentSubstring, currentStart))
        print(f"  Найдена подстрока: '{currentSubstring}' (начинается с позиции
{currentStart})")

    # Если нет подстрок (только джокеры), завершаем работу
    if not substrings:
        print("  Шаблон состоит только из джокеров, поиск невозможен")
        return [], 0, []

    print(f"  Итого подстрок без джокеров: {len(substrings)}")

    # Строим автомат Ахо-Корасик для подстрок
    print("\n--- Шаг 2: Построение бора для подстрок ---")
    root = Node()
    root.id = "Корень"
    for i, (substring, pos) in enumerate(substrings):
        addString(root, substring, i, pos)

    print("\n--- Шаг 3: Построение суффиксных ссылок ---")
    # Вычисляем суффиксные ссылки для всех вершин
    queue = [root]
    visited = set([root])

    while queue:
        node = queue.pop(0)
        # Вычисляем суффиксную ссылку для текущей вершины
        if node != root:
            getSuffLink(node, root)

        # Добавляем потомков в очередь
        for child in node.sons.values():
            if child not in visited:
                queue.append(child)
                visited.add(child)

    print("\n--- Шаг 4: Построение сжатых суффиксных ссылок ---")
```

```
    # Вычисляем сжатые суффиксные ссылки для всех вершин
    queue = [root]
    visited = set([root])

    while queue:
        node = queue.pop(0)
        # Вычисляем сжатую суффиксную ссылку для текущей вершины
        if node != root:
            getUp(node, root)

        # Добавляем потомков в очередь
        for child in node.sons.values():
            if child not in visited:
                queue.append(child)
                visited.add(child)

    # Подсчет количества вершин в автомате
    verticesCount = countVertices(root)

    print("\n--- Шаг 5: Поиск подстрок в тексте ---")
    # Обрабатываем текст
    occurrences = processText(root, text, substrings)

    print("\n--- Шаг 6: Проверка полных совпадений шаблона ---")
    # Находим позиции, где все подстроки найдены
    result = []
    patternOccurrences = {}  # Для хранения информации о пересечениях

    for i in range(len(text) - len(pattern) + 1):
        # Проверяем, найдены ли все подстроки на данной позиции
        found_count = len(occurrences[i]) if i < len(occurrences) else 0

        if found_count == len(substrings):
            print(f"  Позиция {i}: найдены все {len(substrings)} подстроки")

            # Проверяем, найдены ли все уникальные подстроки
            foundSubstrings = set(occurrences[i])
            if len(foundSubstrings) == len(substrings):
                print(f"    Все уникальные подстроки найдены")

                # Проверяем точное соответствие шаблону (с учетом джокеров)
                match = True
                for j in range(len(pattern)):
                    if i + j >= len(text):
                        match = False
                        print(f"    Выход за пределы текста на позиции {i+j}")
                        break
                    if pattern[j] != joker and pattern[j] != text[i + j]:
                        match = False
                        print(f"    Несовпадение на позиции {i+j}: шаблон
'{pattern[j]}', текст '{text[i+j]}'")
                        break

                if match:
                    pos = i + 1  # +1 для нумерации с 1
                    print(f"    ПОЛНОЕ СОВПАДЕНИЕ: шаблон найден на позиции
{pos}")
                    result.append(pos)

                    # Сохраняем информацию о вхождениях для поиска пересечений
                    patternOccurrences[1] = patternOccurrences.get(1, []) +
[pos]
        elif i < len(occurrences) and occurrences[i]:
            print(f"  Позиция {i}: найдено {len(occurrences[i])} подстрок из
{len(substrings)}")
```

```
    # Поскольку у нас только один шаблон, пересечения возможны только между его
вхождениями
    print("\n--- Шаг 7: Поиск пересечений шаблонов ---")
    intersectingPatterns = findIntersectingPatterns(patternOccurrences,
[pattern])

    return result, verticesCount, intersectingPatterns

def processText(root, text, substrings):
    # Обрабатывает текст и находит вхождения подстрок
    # Создаем список словарей для каждой позиции в тексте
    # В списке будем хранить найденные подстроки
    occurrences = [[] for _ in range(len(text) + 1)]

    cur = root
    print(f"  Начинаем с корня бора")

    for i, c in enumerate(text):
        # Переход по символу текста
        prev_state = "Корень" if cur == root else cur.id
        cur = getLink(cur, c, root)
        cur_state = "Корень" if cur == root else cur.id
        print(f"  Символ текста[{i}] = '{c}': {prev_state} → {cur_state}")

        # Проверяем текущий узел на наличие терминалов
        if cur.isLeaf:
            print(f"    Вершина {cur_state} является терминалом:")
            for substringIdx, startInPattern in cur.leafPatternNumbers:
                substring, _ = substrings[substringIdx]
                # Вычисляем позицию начала подстроки в тексте
                substringStart = i - len(substring) + 1
                # Вычисляем позицию начала шаблона
                patternStart = substringStart - startInPattern
                if patternStart >= 0:
                    print(f"      Найдена подстрока #{substringIdx}
'{substring}' с началом шаблона на позиции {patternStart}")
                    occurrences[patternStart].append(substringIdx)

        # Проходим по сжатым суффиксным ссылкам
        node = getUp(cur, root)
        up_followed = False
        while node != root:
            up_followed = True
            node_state = "Корень" if node == root else node.id
            print(f"    Переход по сжатой суфф. ссылке: {cur_state} →
{node_state}")

            if node.isLeaf:
                print(f"      Вершина {node_state} является терминалом:")
                for substringIdx, startInPattern in node.leafPatternNumbers:
                    substring, _ = substrings[substringIdx]
                    substringStart = i - len(substring) + 1
                    patternStart = substringStart - startInPattern
                    if patternStart >= 0:
                        print(f"        Найдена подстрока #{substringIdx}
'{substring}' с началом шаблона на позиции {patternStart}")
                        occurrences[patternStart].append(substringIdx)

            node = getUp(node, root)

        if not up_followed:
            print(f"    Нет переходов по сжатым суфф. ссылкам или переход ведет
в корень")
```

```
    return occurrences

def findIntersectingPatterns(patternOccurrences, patterns):
    # Находит образцы, имеющие пересечения с другими образцами в строке поиска
    print("\n---> Поиск образцов с пересечениями:")
    intersectingPatterns = set()

    # Если только один шаблон, проверяем пересечения между его вхождениями
    if len(patternOccurrences) == 1:
        patternNumber = list(patternOccurrences.keys())[0]
        positions = patternOccurrences[patternNumber]
        patternLength = len(patterns[patternNumber-1])

        # Создаем интервалы для вхождений
        intervals = [(pos, pos + patternLength - 1) for pos in positions]
        print(f"  Интервалы вхождений шаблона: {intervals}")

        # Сортируем интервалы
        intervals.sort()

        # Проверяем пересечения
        for i in range(len(intervals) - 1):
            _, end1 = intervals[i]
            start2, _ = intervals[i+1]

            if start2 <= end1:
                # Есть пересечение
                print(f"  Найдено пересечение между вхождениями шаблона:")
                print(f"    Вхождение на позиции {intervals[i][0]}: до позиции
{end1}")
                print(f"    Вхождение на позиции {start2}: начинается до
окончания предыдущего")
                intersectingPatterns.add(patternNumber)
                break

    if not intersectingPatterns:
        print("  Не найдено образцов с пересечениями")
    else:
        print(f"  Образцы с пересечениями:
{sorted(list(intersectingPatterns))}")

    return sorted(list(intersectingPatterns))

text = input().strip()
pattern = input().strip()
joker = input().strip()

result, verticesCount, intersectingPatterns = findPatternWithJoker(text,
pattern, joker)
print("\n=== РЕЗУЛЬТАТЫ ===")
for pos in result:
    print(pos)

print(f"Количество вершин в автомате: {verticesCount}")
print(f"Образцы с пересечениями: {intersectingPatterns}")
```