**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.

|  |  |  |
| --- | --- | --- |
| Студент гр. 3344 |  | Тукалкин В.А. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

**Цель работы.**

Изучить алгоритм поиска набора подстрок в строке. Разработать программу, находящую все вхождения подстрок в строку с помощью алгоритма Ахо-Корасика.

**Задание.**

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

**Задание 6.1.**

Разработайте программу,  решающую задачу точного поиска набора образцов.  
  
**Вход:**  
Первая строка содержит текст (*T*, 1 ≤ ∣*T*∣ ≤ 100000 ).  
Вторая - число *n* (1 ≤ *n* ≤ 3000), каждая следующая из *n* строк содержит шаблон из набора *P* = {*p*1​,…,*pn*​}1 ≤ ∣*pi* ​∣ ≤ 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 6.2.**

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 Р в текст Т.  
  
Например, образец а*b*??с? 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 ≤ ∣*T*∣ ≤ 100000 )  
Шаблон (*P*, 1 ≤ ∣*P*∣≤ 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

**Описание алгоритма.**

Алгоритм Ахо-Корасика — это эффективный алгоритм поиска множества подстрок в тексте. Он сочетает в себе идеи использования борa (префиксного дерева) и автомата.

Этапы алгоритма (используется в заданиях 6.1 и 6.2):

1. Построение бора (Trie). Создаём бор для всех искомых строк:

1.1. Корень соответствует пустой строке.

1.2. Для каждого символа строки переходим по соответствующему ребру (если его нет — создаем).

1.3. В конце помечаем терминальную вершину (храним длину образца или его идентификатор).

2. Построение суффиксных и конечных ссылок:

2.1. Строим суффиксные ссылки (перебираем узлы-детей и их символы):

2.1.1. Суффиксная ссылка корня и суффиксная ссылка его потомков указывает на сам корень.

2.1.2. Иначе идем по суффиксным ссылкам родителя, пока не найдем узел-ребенка с текущим символом.

2.2. Построение конечных ссылок:

2.2.1. Если суффиксная ссылка ведет в терминальный узел, она является терминальной

2.2.2. Иначе продолжаем идти по суффиксным ссылкам, пока не найдем терминальный узел.

3. Поиск шаблонов в тексте:

3.1. Начинаем с корня.

3.2. Для каждого символа в тексте:

3.2.1. Пока символ и символ в рассматриваемом узле не совпали, идем по суффиксным ссылкам.

3.2.2. Если символы совпали, переходим в дочерний узел.

3.3. Проверяем является ли данный узел терминальным и идем по конечным ссылкам.

3.4. Сохраняем найденные вхождения.

4. В итоге получаем найденные вхождения шаблонов.

Поиск при использовании шаблона с символом-джокером (используется в задании 6.2):

1. Делим шаблон на подшаблоны игнорируя символы-джокеры.

2. Ищем вхождения подшаблонов алгоритмом Ахо-Корасика.

3. Ищем вхождения шаблона:

3.1. Создается массив, который хранит количество найденных подстрок для каждой позиции текста.

3.2. Перебираем позиции найденных вхождений подшаблонов. Увеличиваем значение ячейки массива с индексов позиции вхождения.

3.3. Если число в ячейки массива соответствует числу подшаблонов, индекс ячейки – начало вхождения полного шаблона.

Описание алгоритма индивидуализации:

1. Ищем вхождения шаблонов алгоритмом Ахо-Корасика.

2. Рассматриваем найденный массив вхождений. Рассматриваем каждое вхождения, с остальными, с которыми еще не рассматривалось это вхождение ранее.

3. Вычисляем концы рассматриваемых вхождений и проверяем не оканчивается ли хотя бы одно из них раньше, чем начинается другое.

3.1. Если оканчивается рассматриваем следующую пару

3.2. Иначе находим точку пересечения и заносим информацию о вхождениях и точке пересечения в массив найденных пересечений

4. В конце получаем массив найденных пересечений.

|  |  |  |
| --- | --- | --- |
|  | Сложность по времени | Cложность по памяти |
| Построение бора | O(L) | O(L) |
| Построение ссылок | O(L\*A) |  |
| Поиск | O(T+ L + n) |  |
| Деление шаблона на подшаблоны | O(P) |  |
| Поиск пересечений | O(n2) |  |

, где L = Σ длин всех шаблонов (размер алфавита \* общая длина слов в словаре), T – длина текста, A – число символов в алфавите (число разных символов среди всех шаблонов), P – длина шаблона с джокерами, n – число вхождений

Исходные коды реализующие алгоритмы из заданий 6.1 и 6.2 представлены в приложении (приложении 1 и 2 соответственно).

**Описание функций и структур данных.**

Структуры данных:

1. *TrieNode* – структура, представляющая из себя узел бора.

Содержит поля:

*id* - уникальный идентификатор узла (для вывода)

*isTerminal* - флаг, является ли вершина терминалом

*patternIndices* - номера шаблонов, заканчивающихся в этом узле

*patternLength* - длина шаблона (для определения позиции)

*childrens* - указатели на дочерние узлы

*suffixLink* - суффиксная ссылка

*finalLink* - конечная ссылка

2. *AhoCorasicAlgorithm* – класс, который хранит автомат (бор с ссылками) реализует методы для работы с ним, а также сам алгоритм Ахо-Корасика.

3. *C = [0] \* (len(text) + 1)* - массив, где C[i] - количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i.

Функции и методы:

1. *def \_\_init\_\_(self, patterns: list):* метод класса *AhoCorasicAlgorithm*, инициализирует алгоритм, создает автомат.

Содержит:

*root* – корень бора (тип *TrieNode*)

*nodeCount* – счетчик числа узлов в боре (тип *int*)

Параметры:

*patterns* – массив шаблонов, добавляемых в бор (тип *list*).

Сложность по времени: O(L), где L = Σ длин всех шаблонов

2. *def \_\_add(self, pattern: str, index: int):* метод класса *AhoCorasicAlgorithm*, добавляет новый шаблон в бор.

Параметры:

*pattern* – массив шаблон, добавляемый в бор (тип *str*).

*index* – индекс добавляемого шаблона.

Сложность по времени: O(L), где L = Σ длин всех шаблонов

3. *def \_\_makeLinks(self):* метод класса *AhoCorasicAlgorithm*, cоздает суффиксные и конечные ссылки.

Сложность по времени: O(A \* L), где L = Σ длин всех шаблонов, A – число символов в алфавите (число разных символов среди всех шаблонов)

4. *def search(self, text):* метод класса *AhoCorasicAlgorithm*, ищет все вхождения шаблонов в тексте.

Параметры:

*text* – текст, в котором осуществляется поиск (тип *str*).

Возвращает:

results – отсортированный по позициям в текст массив найденных вхождений (элемент массива имеет вид (позиция начала вхождения, индекс шаблона) – (*int, int*))

Сложность по времени: O(T + L + n), где L = Σ длин всех шаблонов, T – длина текста, n – число вхождений

5. *def getNodeCount(self):* метод класса *AhoCorasicAlgorithm*, позволяет получить число узлов в автомате.

Возвращает:

*nodeCount* – счетчик числа узлов в боре (тип *int*)

Сложность по времени: O(1)

6. *def \_\_printAutomat(self):* метод класса *AhoCorasicAlgorithm*, выводит информацию о построенном автомате.

Сложность по времени: O(L), где L = Σ длин всех шаблонов

7. *def findIntersectingPatterns(results: list, patterns: list):* ищет пересечения вхождений.

Параметры:

*results* –массив найденных вхождений (тип *list*).

*patterns* – массив шаблонов

Возвращает:

intersectionPairs – массив с найденными пересечениями(тип *list*).

Сложность по времени: O(n2), где n – число элементов массива *results.*

8. *def findSubPatterns(pattern: str, jokerSymbol: str):* делит шаблон с символом-джокером на отдельные части.

Параметры:

*pattern* – шалон с джокер-символами, который требуется разделить (тип *str*).

*jokerSymbol* – джокер-символ (тип *str*).

Возвращает:

*subPatterns, positions* – список подшаблонов и список их позиций (типы *list*)

Сложность по времени: O(P), где P – длина шаблона с джокерами.

**Тестирование.**

Тестирование программ представлено на таблице.

Таблица 1. Задание 6.1

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | NTAG  3  TAGT  TAG  T | 2 2  2 3 | Верно |
| 2 | shershe  4  he  she  his  hers | 1 2  2 1  2 4  5 2  6 1 | Верно |
| 3 | TTTTTTTTTT  2  NAN  GNG |  | Верно |
| 4 | TATTATAT  3  TATT  TAT  TATTAT | 1 1  1 2  1 3  4 2  6 2 | Верно |
| 5 | ABBBAABAB  3  ABB  ABB  ABA | 1 1  1 2  6 3 | Верно |

Таблица 2. Задание 6.2

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | ACTANCA  A$$A$  $ | 1 | Верно |
| 2 | ABBBACBBABBBC  A$BB$  $ | 1  5  9 | Верно |
| 3 | ABBBACBBABBBC  $C$  $ | 5 | Верно |
| 4 | ABBA  C$  $ |  | Верно |

Таблица 3. Индивидуализация

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | NTAG  3  TAGT  TAG  T | TAG и T пересекаются в точке 2 | Верно |
| 2 | shershe  4  he  she  his  hers | she и he пересекаются в точке 2  she и her пересекаются в точке 2  he и her пересекаются в точке 2  her и she пересекаются в точке 5  she и he пересекаются в точке 6 | Верно |
| 3 | ABBBACBBABBBC  A$BB$  $ | A$BB$ и A$BB$ пересекаются в точке 5  A$BB$ и A$BB$ пересекаются в точке 9 | Верно |
| 4 | ABCMMMCAB  2  ABC  CAB | Пересечений нет | Верно |

Примеры работ программ с выводом промежуточных результатов при введённом значении указаны в приложении (приложение 3 и приложение 4 для программ 1 и 2 соответственно)

**Выводы.**

Был изучен алгоритм Ахо-Корасика поиска всех вхождений набора подстрок в строку. Разработана программа, реализующая алгоритм Ахо-Корасика, а также алгоритм поиска шаблона с джокерами.

**Приложение**

Приложение 1

from collections import deque

class TrieNode:

"""Класс узла бора"""

def \_\_init\_\_(self, nodeId: int):

self.id = nodeId # уникальный идентификатор узла (для вывода)

self.isTerminal = False # флаг, является ли вершина терминалом

self.patternIndices = [] # номера шаблонов, заканчивающихся в этом узле

self.patternLength = 0 # длина шаблона (для определения позиции)

self.childrens = {} # указатели на дочерние узлы

self.suffixLink = None # суффиксная ссылка

self.finalLink = None # конечная ссылка

class AhoCorasicAlgorithm:

"""Класс, реализующий алгоритм Ахо-Корасика"""

def \_\_init\_\_(self, patterns: list):

"""Инициализирует алгоритм, создает автомат"""

self.root = TrieNode(0) # корневой узел

self.root.suffixLink = self.root # суффиксная ссылка корня (ведет на себя)

self.nodeCount = 1 # счетчик узлов

print("\nНачинаем построение бора:")

# строим бор

for index in range(len(patterns)):

print(f"\nДобавляем шаблон '{patterns[index]}' (индекс {index}) в бор:")

self.\_\_add(patterns[index], index) # добавляем шаблон

print("Бор построен\n")

# создаем ссылки, строим автомат (добавление суффиксных и конечных ссылок)

print("Начинаем построение автомата (добавление суффиксных и конечных ссылок):")

self.\_\_makeLinks()

# выводим автомат

print("\nВывод информации об автомате:")

self.\_\_printAutomat()

def \_\_add(self, pattern: str, index: int):

"""Добавляет новый шаблон в бор"""

currentNode = self.root # начинаем с корня

print(f" Начинаем с корня")

# перебераем символы шаблона

for char in pattern:

if char not in currentNode.childrens: # создаем новый узел при необходимости

newNode = TrieNode(self.nodeCount) # создаем новый узел

currentNode.childrens[char] = newNode

self.nodeCount += 1 # увеличиваем счетчик узлов в автомате

print(f" Создаем новый узел {newNode.id} для символа '{char}'")

else:

print(f" Переходим в существующий узел {currentNode.childrens[char].id} для символа '{char}'")

currentNode = currentNode.childrens[char] # переходим к дочернему узлу с определенным символом

currentNode.isTerminal = True # помечаем терминальную вершину

currentNode.patternIndices.append(index) # сохраняем индекс шаблона

currentNode.patternLength = len(pattern) # сохраняем длину шаблона

print(f" Помечаем узел {currentNode.id} как терминальный для шаблона '{pattern}' (индекс {index})")

def \_\_makeLinks(self):

"""Создает суффиксные и конечные ссылки"""

queue = deque() # инициализируем очередь для поиска в ширину

queue.append(self.root) # добавляем корень в очередь

print(" Добавляем корень в очередь для обработки")

while queue:

currentNode = queue.popleft() # берем следующий узел

print(f"\n Обрабатываем узел {currentNode.id}")

for char, childNode in currentNode.childrens.items(): # перебераем всех детей текущего узла

print(f" Обрабатываем ребенка с символом '{char}' (узел {childNode.id})")

queue.append(childNode) # добавляем ущел-ребенка в очередь

print(f" Добавляем узел {childNode.id} в очередь")

# для детей корня суффиксная ссылка ведет в корень

if currentNode == self.root:

childNode.suffixLink = self.root # устанавливаем суффиксную-ссылку на корень

print(f" Это ребенок корня - устанавливаем суффиксную ссылку на корень (0)")

else:

# ищем первую возможную суффиксную ссылку

temp = currentNode.suffixLink

print(f" Ищем суффиксную ссылку. Начинаем с узла {temp.id}")

while (temp != self.root) and (char not in temp.childrens):

temp = temp.suffixLink

print(f" Переходим по суффиксной ссылке к узлу {temp.id}")

# устанавливаем найденную ссылку или ссылку на корень

if char in temp.childrens:

childNode.suffixLink = temp.childrens[char]

print(f" Устанавливаем суффиксную ссылку на узел {temp.childrens[char].id}")

else:

childNode.suffixLink = self.root

print(f" Устанавливаем суффиксную ссылку на корень (0)")

# построение конечной ссылки

if childNode.suffixLink.isTerminal:

# если суффиксная ссылка ведет прямо в терминальный узел,

childNode.finalLink = childNode.suffixLink # его конечная ссылка это его суффиксная ссылка

print(f" Устанавливаем конечную ссылку на узел {childNode.suffixLink.id} (терминальный)")

else:

# если суффиксная ссылка ведет в нетерминальный узел,

childNode.finalLink = childNode.suffixLink.finalLink # берем его конечную ссылку

if childNode.finalLink:

print(f" Устанавливаем конечную ссылку на узел {childNode.finalLink.id}")

else:

print(f" Конечная ссылка не установлена (нет терминальных узлов в цепочке)")

def search(self, text):

"""Ищет все вхождения шаблонов в тексте"""

print("Начинаем поиск")

results = [] # массив с результом поиска

currentNode = self.root # начинаем с корня

for position in range(len(text)):

char = text[position] # текущий символ

print(f"\nПозиция {position + 1}, символ '{char}':")

# используем суффиксные ссылки при отсутствии перехода

while (currentNode != self.root) and (char not in currentNode.childrens):

print(f" Нет перехода по '{char}' из узла {currentNode.id}, переходим по суффиксной ссылке к узлу {currentNode.suffixLink.id}")

currentNode = currentNode.suffixLink

# переходим по символу, если переход существует

if char in currentNode.childrens:

print(f" Переход по '{char}' найден, переходим в узел {currentNode.childrens[char].id}")

currentNode = currentNode.childrens[char]

else:

print(f" Переход по '{char}' не найден, остаемся в корне")

# проверяем терминальные узлы

if currentNode.isTerminal:

print(f" Узел {currentNode.id} терминальный, шаблоны: {currentNode.patternIndices}")

for patternIndex in currentNode.patternIndices:

startPosition = position - currentNode.patternLength + 2 # вычисляем позицию начала вхождения

results.append((startPosition, patternIndex + 1)) # добавляем найденное вхождение к результату

print(f" Найдено вхождение шаблона {patternIndex + 1} на позиции {startPosition}")

# проверяем конечные ссылки для нахождения всех вложенных шаблонов

temp = currentNode.finalLink

while temp:

print(f" Переход по конечной ссылке в узел {temp.id}, шаблоны: {temp.patternIndices}")

for patternIndex in temp.patternIndices: # перебираем индексы шалонов в узле

startPosition = position - temp.patternLength + 2 # вычисляем позицию начала вхождения

results.append((startPosition, patternIndex + 1)) # добавляем найденное вхождение к результату

print(f" Найдено вхождение шаблона {patternIndex + 1} на позиции {startPosition} (по конечной ссылке)")

temp = temp.finalLink # переходим по конечной ссылке

print("\nПоиск завершен")

return sorted(results) # сортируем результаты по позиции в тексте

def getNodeCount(self):

"""Позволяет получить число узлов в автомате"""

return self.nodeCount

def \_\_printAutomat(self):

"""Выводит информацию о построенном автомате"""

queue = deque([self.root]) # инициализируем очередь для поиска в ширину

visited = set() # формируем сэт посещенных узлов

visited.add(self.root) # начинаем с корня

print(f"Структура автомата (число узлов {self.getNodeCount()}):")

print("Узел N [узлы-дети] -> [суф.ссылка, конеч.ссылка] [терминальный?] [шаблоны]")

while queue:

node = queue.popleft()

# Формируем информацию о узле

childrens = ", ".join([f"'{char}':{node.id}" for char, node in node.childrens.items()]) # информация о узлах\_детях

suffixLinkId = node.suffixLink.id if node.suffixLink else -1 # информация о суффиксной ссылке

finalLinkId = node.finalLink.id if node.finalLink else -1 # информация о конечной ссылке

isTerminal = "T" if node.isTerminal else " " # информация о терминальности узла

patterns = node.patternIndices if node.isTerminal else [] # информация о шаблонах

# печатаем информацию об узле автомата

print(f"Узел {node.id:2} [{childrens:15}] -> [{suffixLinkId:2}, {finalLinkId:2}] [{isTerminal}] {patterns}")

# добавляем детей текущего узла в очередь

for \_, child in node.childrens.items():

if child not in visited:

visited.add(child) # добавляем узел-ребенка в сэт посещенных

queue.append(child) # добавляем в очередь

def findIntersectingPatterns(results: list, patterns: list):

"""Ищет пересечения вхождений"""

intersectionPairs = [] # массив с найденными пересечениями

print("\nПоиск пересекающихся вхождений:")

# перебираем вхождения

for i in range(len(results)):

startPosition1, patternIndex1 = results[i] # начальная позиция и индекс шаблона первого вхождения, с которым будем сравнивать остальные

endPosition1 = startPosition1 + len(patterns[patternIndex1 - 1]) # конечная позиция данного вхождения

print(f"\nСравниваем вхождение шаблона {patternIndex1} ('{patterns[patternIndex1 - 1]}') на позициях {startPosition1}-{endPosition1}")

for j in range(i + 1, len(results)):

startPosition2, patternIndex2 = results[j] # начальная позиция и индекс шаблона второго вхождения, который сравнивают

endPosition2 = startPosition2 + len(patterns[patternIndex2 - 1]) # конечная позиция такого вхождения

print(f" С вхождением шаблона {patternIndex2} ('{patterns[patternIndex2 - 1]}') на позициях {startPosition2}-{endPosition2}")

if startPosition1 <= endPosition2 and startPosition2 <= endPosition1: # концы должны быть не раньше начал сравниваемых вхождений

intersectionPoint = max(startPosition1, startPosition2) # вычисляем точку начала пересечения

intersectionPairs.append((patterns[patternIndex1 - 1], patterns[patternIndex2 - 1], intersectionPoint)) # добавляем в массив перечений

print(f" Найдено пересечение в точке {intersectionPoint}")

return intersectionPairs

def main():

# ввод данных

text = input("Введите текст: ")

num = int(input("Введите количество шаблонов: "))

patterns = [input(f"Введите шаблон {i + 1}: ") for i in range(num)]

ahoCorasicAlgorithm = AhoCorasicAlgorithm(patterns) # инициализация алгоритма (создание автомата)

results = ahoCorasicAlgorithm.search(text) # поиск вхождений

# вывод результатов

print("\nНайденные вхождения:")

for startPosition, patternIndex in results:

print(f"Позиция {startPosition}: шаблон {patternIndex} ('{patterns[patternIndex - 1]}')")

# вывод пересекабщихся частей

intersectionPairs = findIntersectingPatterns(results, patterns)

if len(intersectionPairs) == 0:

print("\nПересечений нет")

else:

print("\nПересечения:")

for firstPattern, secondPattern, intersectionPoint in intersectionPairs:

print(f"{firstPattern} и {secondPattern} пересекаются в точке {intersectionPoint}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Приложение 2

from collections import deque

class TrieNode:

"""Класс узла бора"""

def \_\_init\_\_(self, nodeId: int):

self.id = nodeId # уникальный идентификатор узла (для вывода)

self.isTerminal = False # флаг, является ли вершина терминалом

self.patternIndices = [] # номера шаблонов, заканчивающихся в этом узле

self.patternLength = 0 # длина шаблона (для определения позиции)

self.childrens = {} # указатели на дочерние узлы

self.suffixLink = None # суффиксная ссылка

self.finalLink = None # конечная ссылка

class AhoCorasicAlgorithm:

"""Класс, реализующий алгоритм Ахо-Корасика"""

def \_\_init\_\_(self, patterns: list):

"""Инициализирует алгоритм, создает автомат"""

self.root = TrieNode(0) # корневой узел

self.root.suffixLink = self.root # суффиксная ссылка корня (ведет на себя)

self.nodeCount = 1 # счетчик узлов

print("\nНачинаем построение бора:")

# строим бор

for index in range(len(patterns)):

print(f"\nДобавляем шаблон '{patterns[index]}' (индекс {index}) в бор:")

self.\_\_add(patterns[index], index) # добавляем шаблон

print("Бор построен\n")

# создаем ссылки, строим автомат (добавление суффиксных и конечных ссылок)

print("Начинаем построение автомата (добавление суффиксных и конечных ссылок):")

self.\_\_makeLinks()

# выводим автомат

print("\nВывод информации об автомате:")

self.\_\_printAutomat()

def \_\_add(self, pattern: str, index: int):

"""Добавляет новый шаблон в бор"""

currentNode = self.root # начинаем с корня

print(f" Начинаем с корня")

# перебераем символы шаблона

for char in pattern:

if char not in currentNode.childrens: # создаем новый узел при необходимости

newNode = TrieNode(self.nodeCount) # создаем новый узел

currentNode.childrens[char] = newNode

self.nodeCount += 1 # увеличиваем счетчик узлов в автомате

print(f" Создаем новый узел {newNode.id} для символа '{char}'")

else:

print(f" Переходим в существующий узел {currentNode.childrens[char].id} для символа '{char}'")

currentNode = currentNode.childrens[char] # переходим к дочернему узлу с определенным символом

currentNode.isTerminal = True # помечаем терминальную вершину

currentNode.patternIndices.append(index) # сохраняем индекс шаблона

currentNode.patternLength = len(pattern) # сохраняем длину шаблона

print(f" Помечаем узел {currentNode.id} как терминальный для шаблона '{pattern}' (индекс {index})")

def \_\_makeLinks(self):

"""Создает суффиксные и конечные ссылки"""

queue = deque() # инициализируем очередь для поиска в ширину

queue.append(self.root) # добавляем корень в очередь

print(" Добавляем корень в очередь для обработки")

while queue:

currentNode = queue.popleft() # берем следующий узел

print(f"\n Обрабатываем узел {currentNode.id}")

for char, childNode in currentNode.childrens.items(): # перебераем всех детей текущего узла

print(f" Обрабатываем ребенка с символом '{char}' (узел {childNode.id})")

queue.append(childNode) # добавляем ущел-ребенка в очередь

print(f" Добавляем узел {childNode.id} в очередь")

# для детей корня суффиксная ссылка ведет в корень

if currentNode == self.root:

childNode.suffixLink = self.root # устанавливаем суффиксную-ссылку на корень

print(f" Это ребенок корня - устанавливаем суффиксную ссылку на корень (0)")

else:

# ищем первую возможную суффиксную ссылку

temp = currentNode.suffixLink

print(f" Ищем суффиксную ссылку. Начинаем с узла {temp.id}")

while (temp != self.root) and (char not in temp.childrens):

temp = temp.suffixLink

print(f" Переходим по суффиксной ссылке к узлу {temp.id}")

# устанавливаем найденную ссылку или ссылку на корень

if char in temp.childrens:

childNode.suffixLink = temp.childrens[char]

print(f" Устанавливаем суффиксную ссылку на узел {temp.childrens[char].id}")

else:

childNode.suffixLink = self.root

print(f" Устанавливаем суффиксную ссылку на корень (0)")

# построение конечной ссылки

if childNode.suffixLink.isTerminal:

# если суффиксная ссылка ведет прямо в терминальный узел,

childNode.finalLink = childNode.suffixLink # его конечная ссылка это его суффиксная ссылка

print(f" Устанавливаем конечную ссылку на узел {childNode.suffixLink.id} (терминальный)")

else:

# если суффиксная ссылка ведет в нетерминальный узел,

childNode.finalLink = childNode.suffixLink.finalLink # берем его конечную ссылку

if childNode.finalLink:

print(f" Устанавливаем конечную ссылку на узел {childNode.finalLink.id}")

else:

print(f" Конечная ссылка не установлена (нет терминальных узлов в цепочке)")

def search(self, text):

"""Ищет все вхождения шаблонов в тексте"""

print("Начинаем поиск")

results = [] # массив с результом поиска

currentNode = self.root # начинаем с корня

for position in range(len(text)):

char = text[position] # текущий символ

print(f"\nПозиция {position + 1}, символ '{char}':")

# используем суффиксные ссылки при отсутствии перехода

while (currentNode != self.root) and (char not in currentNode.childrens):

print(f" Нет перехода по '{char}' из узла {currentNode.id}, переходим по суффиксной ссылке к узлу {currentNode.suffixLink.id}")

currentNode = currentNode.suffixLink

# переходим по символу, если переход существует

if char in currentNode.childrens:

print(f" Переход по '{char}' найден, переходим в узел {currentNode.childrens[char].id}")

currentNode = currentNode.childrens[char]

else:

print(f" Переход по '{char}' не найден, остаемся в корне")

# проверяем терминальные узлы

if currentNode.isTerminal:

print(f" Узел {currentNode.id} терминальный, шаблоны: {currentNode.patternIndices}")

for patternIndex in currentNode.patternIndices:

startPosition = position - currentNode.patternLength + 2 # вычисляем позицию начала вхождения

results.append((startPosition, patternIndex + 1)) # добавляем найденное вхождение к результату

print(f" Найдено вхождение шаблона {patternIndex + 1} на позиции {startPosition}")

# проверяем конечные ссылки для нахождения всех вложенных шаблонов

temp = currentNode.finalLink

while temp:

print(f" Переход по конечной ссылке в узел {temp.id}, шаблоны: {temp.patternIndices}")

for patternIndex in temp.patternIndices: # перебираем индексы шалонов в узле

startPosition = position - temp.patternLength + 2 # вычисляем позицию начала вхождения

results.append((startPosition, patternIndex + 1)) # добавляем найденное вхождение к результату

print(f" Найдено вхождение шаблона {patternIndex + 1} на позиции {startPosition} (по конечной ссылке)")

temp = temp.finalLink # переходим по конечной ссылке

print("\nПоиск завершен")

return sorted(results) # сортируем результаты по позиции в тексте

def getNodeCount(self):

"""Позволяет получить число узлов в автомате"""

return self.nodeCount

def \_\_printAutomat(self):

"""Выводит информацию о построенном автомате"""

queue = deque([self.root]) # инициализируем очередь для поиска в ширину

visited = set() # формируем сэт посещенных узлов

visited.add(self.root) # начинаем с корня

print(f"Структура автомата (число узлов {self.getNodeCount()}):")

print("Узел N [узлы-дети] -> [суф.ссылка, конеч.ссылка] [терминальный?] [шаблоны]")

while queue:

node = queue.popleft()

# Формируем информацию о узле

childrens = ", ".join([f"'{char}':{node.id}" for char, node in node.childrens.items()]) # информация о узлах\_детях

suffixLinkId = node.suffixLink.id if node.suffixLink else -1 # информация о суффиксной ссылке

finalLinkId = node.finalLink.id if node.finalLink else -1 # информация о конечной ссылке

isTerminal = "T" if node.isTerminal else " " # информация о терминальности узла

patterns = node.patternIndices if node.isTerminal else [] # информация о шаблонах

# печатаем информацию об узле автомата

print(f"Узел {node.id:2} [{childrens:15}] -> [{suffixLinkId:2}, {finalLinkId:2}] [{isTerminal}] {patterns}")

# добавляем детей текущего узла в очередь

for \_, child in node.childrens.items():

if child not in visited:

visited.add(child) # добавляем узел-ребенка в сэт посещенных

queue.append(child) # добавляем в очередь

def findSubPatterns(pattern: str, jokerSymbol: str):

"""Делит шаблон с символом-джокером на отдельные части"""

subPatterns = [] # список для хранения найденных подшаблонов

positions = [] # список для хранения стартовых позиций подшаблонов

current = "" # текущий формируемый подшаблон

startPosition = 0 # начальная позиция текущего подшаблона

print(f"\nНачинаем разбиение шаблона '{pattern}' с джокером '{jokerSymbol}':")

for i, char in enumerate(pattern):

print(f"\nОбрабатываем символ {i + 1} ('{char}'):")

# если символ не джокер - добавляем к текущему подшаблону

if char != jokerSymbol:

current += char

print(f" Символ не джокер, добавляем к текущему подшаблону: '{current}'")

# если встретили джокер - сохраняем текущий подшаблон (если он не пустой)

else:

if current:

print(f" Встретили джокер, сохраняем подшаблон '{current}' (позиция {startPosition})")

subPatterns.append(current) # добавляем подшаблон

positions.append(startPosition) # добавляем позицию подшаблона

current = ""

else:

print(f" Встретили джокер, но текущий подшаблон пуст - пропускаем")

# следующий подшаблон начнется после этого джокера

startPosition = i + 1

print(f" Новая стартовая позиция для следующего подшаблона: {startPosition}")

# после цикла сохраняем последний подшаблон (если он не пустой)

if current:

print(f"\nЗавершение обработки - сохраняем последний подшаблон '{current}' (позиция {startPosition})")

subPatterns.append(current)

positions.append(startPosition)

else:

print("\nЗавершение обработки - последний подшаблон пуст")

# выводим итоговый результат разбиения

print("\nИтоговые подшаблоны и их позиции:")

for sub, pos in zip(subPatterns, positions):

print(f" Подшаблон '{sub}' начинается на позиции {pos}")

return subPatterns, positions

def findIntersectingPatterns(results: list, patterns: list):

"""Ищет пересечения вхождений"""

intersectionPairs = [] # массив с найденными пересечениями

print("\nПоиск пересекающихся вхождений:")

# перебираем вхождения

for i in range(len(results)):

startPosition1, patternIndex1 = results[i] # начальная позиция и индекс шаблона первого вхождения, с которым будем сравнивать остальные

endPosition1 = startPosition1 + len(patterns[patternIndex1 - 1]) # конечная позиция данного вхождения

print(f"\nСравниваем вхождение шаблона {patternIndex1} ('{patterns[patternIndex1 - 1]}') на позициях {startPosition1}-{endPosition1}")

for j in range(i + 1, len(results)):

startPosition2, patternIndex2 = results[j] # начальная позиция и индекс шаблона второго вхождения, который сравнивают

endPosition2 = startPosition2 + len(patterns[patternIndex2 - 1]) # конечная позиция такого вхождения

print(f" С вхождением шаблона {patternIndex2} ('{patterns[patternIndex2 - 1]}') на позициях {startPosition2}-{endPosition2}")

if startPosition1 <= endPosition2 and startPosition2 <= endPosition1: # концы должны быть не раньше начал сравниваемых вхождений

intersectionPoint = max(startPosition1, startPosition2) # вычисляем точку начала пересечения

intersectionPairs.append((patterns[patternIndex1 - 1], patterns[patternIndex2 - 1], intersectionPoint)) # добавляем в массив перечений

print(f" Найдено пересечение в точке {intersectionPoint}")

return intersectionPairs

def main():

# ввод данных

text = input("Введите текст: ")

pattern = input("Введите шаблон: ")

jokerSymbol = input("Введите джокер-символ: ")

# делим шаблон с символами-джокерами на подшаблоны

subPatterns, positions = findSubPatterns(pattern, jokerSymbol)

ahoCorasicAlgorithm = AhoCorasicAlgorithm(subPatterns) # инициализация алгоритма (создание автомата)

ahoCorasicresults = ahoCorasicAlgorithm.search(text) # поиск вхождений

C = [0] \* (len(text) + 1) # массив, где C[i] - количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i

# заполняем массив C

for startPosition, patternIndex in ahoCorasicresults:

startPositionInText = startPosition - positions[patternIndex - 1] # определяем позицию подшаблона в тексте

if startPositionInText >= 0:

C[startPositionInText] += 1

# вывод результатов

print("\nНайденные вхождения:")

results = [] # итоговые результаты поиска вхождений шаблона

for i in range(len(text)):

if C[i] == len(subPatterns) and i + len(pattern) - 1 <= len(text): # если число подшаблонов на позиции i равно общему количеству подшаблонов и шаблон полностью в тексте

print(i)

results.append((i, 1)) # добавляем позицию данного вхождения в результаты

# вывод пересекабщихся частей

intersectionPairs = sorted(set(findIntersectingPatterns(results, [pattern])))

if len(intersectionPairs) == 0:

print("\nПересечений нет")

else:

print("\nПересечения:")

for firstPattern, secondPattern, intersectionPoint in intersectionPairs:

print(f"{firstPattern} и {secondPattern} пересекаются в точке {intersectionPoint}")

if \_\_name\_\_ == "\_\_main\_\_":

main()