**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: Алгоритм А**хо-Корасика**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Сочков И.С. |
| Преподаватель |  | Шевелева А.М. |

Санкт-Петербург

2023

**Задание 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={p1, …, 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 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 аb??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

**Выполнение работы.**

Класс Vertex представляет вершину бора, который является древовидной структурой данных для хранения и обработки множества строк.

Атрибуты класса Vertex:

* next\_vertexes: словарь, хранящий следующие вершины в боре и связующие символы;
* move: словарь, хранящий функцию перехода, указывающую на следующую вершину, которая соответствует заданному символу;
* flag: флаг, указывающий на то, является ли текущая вершина концом какого-либо шаблона;
* pattern\_index: индекс шаблона в списке шаблонов Bor, который заканчивается в этой вершине;
* suffix\_link: индекс вершины в боре, которая соответствует самому длинному суффиксу текущей строки;
* good\_suffix\_link: индекс вершины в боре, которая соответствует суффиксу текущей строки и имеет флаг;
* parent: индекс вершины-родителя;
* symbol: символ, который связывает текущую вершину с ее родительской вершиной.

Был создан класс Bor, предназначенный для поиска заданных шаблонов в строке. Содержит следующие атрибуты:

* bor: список объектов класса Vertex, представляющих узлы бора.
* patterns: список строк, представляющих добавленные в бор образцы (паттерны).
* result: список кортежей, представляющих найденные в тексте сочетания образцов и позиции, в которых они встретились.

Класс Bor имеет следующие методы:

1. \_\_init\_\_(self): конструктор класса, инициализирующий пустой бор, список шаблонов и результат поиска.
2. add\_to\_bor(self, pattern: str) -> None: добавляет шаблон в бор. Для этого метод последовательно проходит по символам шаблона, создавая новые вершины и связи в боре, если таких еще нет. После прохода по всем символам метод помечает последнюю вершину, соответствующую концу шаблона, как финальную.
3. get\_suffix\_link(self, vertex: int) -> int: возвращает суффиксную ссылку для заданной вершины. Суффиксная ссылка - это ссылка на самую длинную подстроку шаблона, совпадающую с префиксом данной вершины, за исключением этого префикса. Для вычисления суффиксной ссылки метод использует рекурсивный алгоритм, переходя от данной вершины к ее родителю и далее по бору, пока не найдет первую вершину, имеющую связь с текущим символом шаблона.
4. get\_good\_suffix\_link(self, vertex: int) -> int: возвращает хорошую суффиксную ссылку для заданной вершины. Хорошая суффиксная ссылка - это ссылка на первую вершину с флагом (последнюю вершину шаблона), находящуюся по пути суффиксных ссылок от данной вершины. Если такой вершины нет, то ссылка устанавливается на вершину, находящуюся на один уровень выше в иерархии бора.
5. get\_move(self, vertex: int, symbol: str) -> int: возвращает вершину, соответствующую переходу из заданной вершины по заданному символу. Если такой вершины нет, метод вычисляет суффиксную ссылку для данной вершины и выполняет переход из нее по тому же символу.
6. find(self, text: str) -> None: находит все вхождения шаблонов в заданной строке и сохраняет их позиции в результирующем списке. Для каждого символа входной строки он проходит от текущей вершины бора вниз по дереву, переходя от вершины к вершине по символам строки. Если текущая вершина является финальной для какого-то шаблона, то соответствующее вхождение добавляется в список результатов. Затем он переходит к следующему символу входной строки и продолжает поиск с новой текущей вершины.
7. get\_answer(self, text: str) -> List[Tuple[int, int]]: принимает строку text и вызывает метод find, чтобы заполнить список результатов. Затем он сортирует список результатов и возвращает его. Каждый элемент списка представляет собой кортеж из индекса начала вхождения в строку text и индекса шаблона в списке добавленных шаблонов.

Код решения первой задачи представлен в приложении A, в файле task\_1.py.

Для решения второй задачи потребовалось модифицировать функцию get\_resultи создать функцию build\_bor, добавляющую подстроки шаблона в бор, разделяя её части по символу-джокеру.

Был модифицирован метод findкласса Bor – теперь он сохраняет позиции вхождения шаблонов в тексте, а также позицию этих строк в строке-паттерне.

Итоговый код также представлен в task\_2.py (см. в Приложении А).

**Тестирование.**

Результаты тестирования представлены в таблице 1.

Таблица 1 — Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Входные данные | Выходные данные | Комментарии |
| NTAG  3  TAGT  TAG  T | 2 2  2 3 | Работа программы task\_1.py |
| ACTANCA  A$$A$  $ | 1 | Работа программы task\_2.cpp |

**Выводы.**

В рамках проведенной лабораторной работы были разработаны программы, основанные на алгоритме Ахо-Корасик, для решения двух задач: точный поиск набора образцов в строке и поиск всех вхождений строки, содержащей символ-джокер, в тексте. Для разработки программ использовался язык программирования Python. Программы были успешно протестированы на платформе Stepik. В результате тестирования было подтверждено, что программы работают корректно и эффективно выполняют свои задачи. Этот опыт помог улучшить понимание алгоритмов поиска в строках и структур данных в целом, а также улучшить навыки программирования на Python.

Приложение А

КОД ДЛЯ решения задач

task\_1.py:

from typing import Dict, List, Tuple

'''

Класс Vertex представляет вершину бора, который является древовидной

структурой данных для хранения и обработки множества строк.

Атрибуты класса Vertex:

next\_vertexes: словарь, хранящий следующие вершины в боре и связующие символы

move: словарь, хранящий функцию перехода, указывающую на следующую вершину,

которая соответствует заданному символу

flag: флаг, указывающий на то, является ли текущая вершина концом какого-либо шаблона

pattern\_index: индекс шаблона в списке шаблонов Bor, который заканчивается в этой вершине

suffix\_link: индекс вершины в боре, которая соответствует самому длинному суффиксу текущей строки

good\_suffix\_link: индекс вершины в боре, которая соответствует суффиксу текущей строки и имеет флаг

parent: индекс вершины-родителя

symbol: символ, который связывает текущую вершину с ее родительской вершиной

'''

class Vertex:

def \_\_init\_\_(self, parent:int, symbol:str) -> None:

self.next\_vertexes: Dict[str, int] = {}

self.move: Dict[str, int] = {}

self.flag = False

self.pattern\_index = -1

self.suffix\_link = -1

self.good\_suffix\_link = -1

self.parent = parent

self.symbol = symbol

'''

Класс Bor предназначен для поиска заданных шаблонов в строке.

Содержит следующие атрибуты:

bor: список объектов класса Vertex, представляющих узлы бора.

patterns: список строк, представляющих добавленные в бор образцы (паттерны).

result: список кортежей, представляющих найденные в тексте сочетания образцов и позиции,

в которых они встретились.

'''

class Bor:

#конструктор класса, инициализирующий пустой бор, список шаблонов и результат поиска

def \_\_init\_\_(self) -> None:

self.bor: List[Vertex] = [Vertex(0, '$')]

self.patterns: List[str] = []

self.result: List[Tuple[int, int]] = []

#метод добавления шаблона в бор

def add\_to\_bor(self, pattern: str) -> None:

number = 0

for symbol in pattern:

if symbol not in self.bor[number].next\_vertexes:

self.bor.append(Vertex(number, symbol))

self.bor[number].next\_vertexes[symbol] = len(self.bor) - 1

number = self.bor[number].next\_vertexes[symbol]

self.bor[number].flag = True

self.patterns.append(pattern)

self.bor[number].pattern\_index = len(self.patterns) - 1

#метод поиска суффиксной ссылки для заданной вершины бора

def get\_suffix\_link(self, vertex: int) -> int:

if self.bor[vertex].suffix\_link == -1:

if vertex == 0 or self.bor[vertex].parent == 0:

self.bor[vertex].suffix\_link = 0

else:

self.bor[vertex].suffix\_link = self.get\_move(self.get\_suffix\_link(self.bor[vertex].parent), self.bor[vertex].symbol)

return self.bor[vertex].suffix\_link

#метод поиска суффиксной ссылки для заданной вершины бора

def get\_good\_suffix\_link(self, vertex: int) -> int:

if self.bor[vertex].good\_suffix\_link == -1:

current\_suffix\_link = self.get\_suffix\_link(vertex)

if current\_suffix\_link == 0:

self.bor[vertex].good\_suffix\_link = 0

else:

self.bor[vertex].good\_suffix\_link = current\_suffix\_link if self.bor[current\_suffix\_link].flag else self.get\_good\_suffix\_link(current\_suffix\_link)

return self.bor[vertex].good\_suffix\_link

#метод получения вершины, соответствующего переходу из заданной вершины по заданному символу

def get\_move(self, vertex: int, symbol: str) -> int:

if symbol not in self.bor[vertex].move:

if symbol in self.bor[vertex].next\_vertexes:

self.bor[vertex].move[symbol] = self.bor[vertex].next\_vertexes[symbol]

else:

self.bor[vertex].move[symbol] = 0 if vertex == 0 else self.get\_move(self.get\_suffix\_link(vertex), symbol)

return self.bor[vertex].move[symbol]

#метод поиска всех вхождений шаблонов в заданной строке и сохранения их позиций в результирующем списке

def find(self, text: str) -> None:

vertex\_number = 0

for text\_index, symbol in enumerate(text):

vertex\_number = self.get\_move(vertex\_number, symbol)

vertex = vertex\_number

while vertex != 0:

if self.bor[vertex].flag:

self.result.append((text\_index + 2 - len(self.patterns[self.bor[vertex].pattern\_index]), self.bor[vertex].pattern\_index + 1))

vertex = self.get\_good\_suffix\_link(vertex)

#метод решения задачи

def get\_answer(self, text: str) -> List[Tuple[int, int]]:

self.find(text)

self.result.sort(key=lambda x: (x[0], x[1]))

return self.result

#Запуск программы

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

bor = Bor()

text = input()

patternQuantity = int(input())

for \_ in range(patternQuantity):

pattern = input()

bor.add\_to\_bor(pattern)

result = bor.get\_answer(text)

for i in range(len(result)):

print(\*result[i])

task\_2.py:

from typing import List, Tuple

'''

Класс Vertex представляет вершину бора, который является древовидной

структурой данных для хранения и обработки множества строк.

Атрибуты класса Vertex:

next\_vertexes: словарь, хранящий следующие вершины в боре и связующие символы

move: словарь, хранящий функцию перехода, указывающую на следующую вершину,

которая соответствует заданному символу

flag: флаг, указывающий на то, является ли текущая вершина концом какого-либо шаблона

pattern\_index: индекс шаблона в списке шаблонов Bor, который заканчивается в этой вершине

suffix\_link: индекс вершины в боре, которая соответствует самому длинному суффиксу текущей строки

good\_suffix\_link: индекс вершины в боре, которая соответствует суффиксу текущей строки и имеет флаг

parent: индекс вершины-родителя

symbol: символ, который связывает текущую вершину с ее родительской вершиной

'''

class Vertex:

def \_\_init\_\_(self, parent: int, symbol: str):

self.next\_vertexes = {}

self.move = {}

self.flag = False

self.pattern\_indexes = []

self.suffix\_link = -1

self.good\_suffix\_link = -1

self.parent = parent

self.symbol = symbol

'''

Класс Bor предназначен для поиска заданных шаблонов в строке.

Содержит следующие атрибуты:

bor: список объектов класса Vertex, представляющих узлы бора.

patterns: список строк, представляющих добавленные в бор образцы (паттерны).

result: список кортежей, представляющих найденные в тексте сочетания образцов и позиции,

в которых они встретились.

pattern\_positions: список позиций каждого образца в строке, где образец встретился впервые.

occurrence: список количества вхождений каждого образца в строке.

found\_patterns: список строк, найденных в тексте в процессе поиска образцов.

'''

class Bor:

#конструктор класса, инициализирующий пустой бор, список шаблонов и результат поиска

def \_\_init\_\_(self):

self.bor = [Vertex(0, '$')]

self.patterns = []

self.result = []

self.pattern\_positions = []

self.occurrence = []

self.found\_patterns = []

#метод добавления шаблона в бор

def add\_to\_bor(self, pattern: str) -> None:

number = 0

for symbol in pattern:

if symbol not in self.bor[number].next\_vertexes:

self.bor.append(Vertex(number, symbol))

self.bor[number].next\_vertexes[symbol] = len(self.bor) - 1

number = self.bor[number].next\_vertexes[symbol]

self.bor[number].flag = True

self.patterns.append(pattern)

self.bor[number].pattern\_indexes.append(len(self.patterns) - 1)

#метод поиска суффиксной ссылки для заданной вершины бора

def get\_suffix\_link(self, vertex: int) -> int:

if self.bor[vertex].suffix\_link == -1:

if vertex == 0 or self.bor[vertex].parent == 0:

self.bor[vertex].suffix\_link = 0

else:

self.bor[vertex].suffix\_link = self.get\_move(self.get\_suffix\_link(self.bor[vertex].parent), self.bor[vertex].symbol)

return self.bor[vertex].suffix\_link

#метод поиска суффиксной ссылки для заданной вершины бора

def get\_good\_suffix\_link(self, vertex: int) -> int:

if self.bor[vertex].good\_suffix\_link == -1:

current\_suffix\_link = self.get\_suffix\_link(vertex)

if current\_suffix\_link == 0:

self.bor[vertex].good\_suffix\_link = 0

else:

self.bor[vertex].good\_suffix\_link = current\_suffix\_link if self.bor[current\_suffix\_link].flag else self.get\_good\_suffix\_link(current\_suffix\_link)

return self.bor[vertex].good\_suffix\_link

#метод получения вершины, соответствующего переходу из заданной вершины по заданному символу

def get\_move(self, vertex: int, symbol: str) -> int:

if symbol not in self.bor[vertex].move:

if symbol in self.bor[vertex].next\_vertexes:

self.bor[vertex].move[symbol] = self.bor[vertex].next\_vertexes[symbol]

else:

if vertex == 0:

self.bor[vertex].move[symbol] = 0

else:

self.bor[vertex].move[symbol] = self.get\_move(self.get\_suffix\_link(vertex), symbol)

return self.bor[vertex].move[symbol]

#метод поиска всех вхождений шаблонов в заданной строке и сохранения их позиций в

#результирующем списке

def find(self, text: str) -> None:

vertex\_number = 0

for text\_index in range(len(text)):

vertex\_number = self.get\_move(vertex\_number, text[text\_index])

vertex = vertex\_number

while vertex != 0:

if self.bor[vertex].flag:

for number in self.bor[vertex].pattern\_indexes:

position\_in\_text = text\_index + 1 - len(self.patterns[number])

self.found\_patterns.append((position\_in\_text, number))

vertex = self.get\_good\_suffix\_link(vertex)

#Функция построения бора шаблона с символом-джокером

def build\_bor(pattern: str, joker: str, bor: Bor):

pattern\_part = ""

for pattern\_index in range(len(pattern)):

if pattern[pattern\_index] != joker:

pattern\_part += pattern[pattern\_index]

if pattern\_index == len(pattern) - 1 or pattern[pattern\_index+1] == joker:

bor.add\_to\_bor(pattern\_part)

bor.pattern\_positions.append(pattern\_index - len(pattern\_part) + 2)

pattern\_part = ""

#Функция поиска шаблона в тексте с учетом символа-джокера

def get\_result(bor: Bor, text: str, pattern: str):

bor.find(text)

bor.occurrence = [0] \* len(text)

for i in range(len(bor.found\_patterns)):

text\_position = bor.found\_patterns[i][0]

substring\_index = bor.pattern\_positions[bor.found\_patterns[i][1]]

index = text\_position - substring\_index + 1

if index + len(pattern) > len(text):

continue

if text\_position >= substring\_index - 1:

bor.occurrence[index] += 1

for i in range(len(bor.occurrence)):

if bor.occurrence[i] == len(bor.patterns):

print(i + 1)

#Запуск программы

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

bor = Bor()

text = input()

pattern = input()

joker = input()

build\_bor(pattern, joker, bor)

get\_result(bor, text, pattern)