**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

**отчет**

**по лабораторной работе №4**

**по дисциплине «Построение и анализ алгоритмов»**

**Тема: Поиск подстроки в строке**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Лексин М.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы.**

Исследование и решение двух классических задач в области обработки строк с использованием эффективных алгоритмов. Первая задача состоит в разработке метода для быстрого обнаружения всех мест, где указанный образец встречается внутри большого текстового массива данных. Вторая задача направлена на определение специфического структурного отношения между двумя строками — является ли одна из них циклической перестановкой другой.

**Задание.**

Первое задание:

Реализуйте алгоритм КМП и с его помощью для заданных шаблона P (|P| ≤ 25000) и текста T (|T| ≤ 5000000) найдите все вхождения P в T.

Вход:

Первая строка - P

Вторая строка - T

Выход:

индексы начал вхождений P в T, разделенных запятой, если P не входит в T, то вывести –1

Sample Input:

ab

abab

Sample Output:

0,2

Второе задание:

Заданы две строки A (|A| ≤ 5000000) и B (|B| ≤ 5000000).

Определить, является ли A циклическим сдвигом B (это значит, что A и B имеют одинаковую длину и A состоит из суффикса B, склеенного с префиксом B). Например, defabc является циклическим сдвигом abcdef.

Вход:

Первая строка - A

Вторая строка - B

Выход:

Если A вляется циклическим сдвигом B, индекс начала строки B в A, иначе вывести –1. Если возможно несколько сдвигов вывести первый индекс.

Sample Input:

defabc

abcdef

Sample Output:

3

**Описание алгоритма.**

Задание 1.

Алгоритм КМП состоит из двух основных фаз: вычисление префикс-функции и поиск в тексте.

Фаза 1: вычисление префикс функции. На этом этапе анализируется только строка-шаблон pattern. Цель — построить вспомогательную структуру данных, называемую префикс-функцией.

Префикс функция pi для шаблона pattern длиной pattern\_length представляет собой массив целых чисел размером pattern\_length. Значение pi[i] определяется как длина наибольшего собственного префикса подстроки pattern[0…i] (то есть префикса, неравного самой подстроке pattern[0…i]), который одновременно является суффиксом этой же подстроки pattern[0…i].

Значение pi[i] используется в фазе поиска для определения величины «умного» сдвига шаблона вдоль текста при возникновении несовпадения символов. Если несовпадение произошло при сравнении text[i] с pattern[match\_len], то следующее сравнение будет происходить с pattern[pi[match\_len-1]], без необходимости возвращаться назад по тексту.

Префикс-функция вычисляется за линейное время O(pattern\_length). Происходит это следующим образом:

* Массив pi инициализируются нулями.
* Используется переменная border\_len, которая хранит длину текущей границы (префикса-суффикса).
* Для каждого индекса i от 1 до pattern\_length-1:
  + Пока border\_len > 0 и символы pattern[border\_len] и pattern[i] не совпадают, значение border\_len уменьшается до pi[border\_len -1].
  + Если pattern[border\_len] == pattern[i], увеличиваем border\_len на 1.
  + Записываем pi[i] = border\_len.

Фаза 2: поиск в тексте. На этом этапе происходит сканирование текста text слева направо с использованием вычисленной префикс-функции pi. Используются два указателя: i для текущей позиции в тексте text и match\_len для текущей позиции в шаблоне pattern. Переменная match\_len также представляет длину текущего префикса шаблона, который совпадает с подстрокой текста, заканчивающейся на позиции i-1. Для хранения найденных позиций вхождения используется список occurrences.

Процесс поиска следующий:

1. Инициализация и итерация по тексту: match\_len = 0, occurrences = [], цикл по i от 0 до text\_length-1.
2. Обработка несовпадений: Пока match\_len > 0 и текущий символ текста text[i] не совпадает с символом шаблона pattern[match\_len], происходит «сдвиг» шаблона путем присваивания match\_len = pi[match\_len-1]. Это позволяет продолжить сравнение с более коротким префиксом шаблона, который гарантировано совпадает с суффиксом уже просмотренного участка текста, без смещения указателя i назад.
3. Обработка совпадений: Если text[i] == pattern[match\_len], это означает, что совпадение текущих символов найдено, и мы продолжаем по шаблону: match\_len = match\_len + 1.
4. Обнаружение полного вхождения: Если match\_len достигает значения pattern\_length, это означает, что найдено полное вхождения шаблона pattern. Оно заканчивается на текущей позиции i в тексте text. Начальный индекс этого вхождения равен i-pattern\_length+1.
5. Поиск последующих вхождений: После обнаружения вхождения, чтобы найти возможные последующие вхождения, которые могут перекрываться, можно не сбрасывать match\_len в 0, а снова использовать префикс функцию: match\_len = pi[patter\_length-1]. Это эффективно сдвигает шаблон на максимально возможную величину, пропуская уже проверенные символы, и подготавливает к поиску следующего возможного совпадения.
6. Если список occurrences пуст, возвращается [-1], иначе возвращается список найденных позиций.

Задание 2.

Решение основано на сведении задачи к поиску подстроки с использованием алгоритма КМП. Ключевой идеей является использование конкатенации первой строки с самой собой.

Если предварительные проверки пройдены (строки непустые, имеют одинаковую длину N, но не идентичны), создается новая строка doubled\_str1, равная конкатенации строки str1 с самой собой.

Вызывается алгоритм КМП для поиска первого вхождения строки str2 (шаблон) в строке doubled\_str1 (текст). Алгоритм КМП выполняет:

1. Вычисление префикс функции pi для str2.
2. Сканирование строки doubled\_str1 с использованием префикс-функции для эффективного поиска str2.

В этой версии функция возвращает индекс первого найденного вхождения, а не список вхождений:

* При обнаружении полного совпадения (match\_len == pattern\_length) функция сразу возвращает индекс начала вхождения: i-pattern\_length+1
* Если вхождения не найдено, возвращается -1.

Если str2 является циклическим сдвигом str1, то str2 обязательно будет найдена как подстрока в doubled\_str1. Индекс, возвращаемый функцией kmp\_matcher, и будет искомой величиной циклического сдвига.

**Оценка сложности.**

Задание 1.

Фаза 1. Основной цикл вычисления префикс-функции pi выполняется pattern\_length-1 раз. Внутри цикла находится while, который уменьшает значение border\_len. Переменная border\_len увеличивается не более чем на 1 на каждой итерации внешнего цикла for. Суммарное увеличение border\_len за все время работы не превышает pattern\_length. Поскольку border\_len неотрицательно, общее число выполнения операций уменьшения border\_len не может превышать общее число увеличений border\_len. Следовательно, суммарное время работы внутреннего цикла while по всем итерациям внешнего цикла for является O(pattern\_length). Общее время работы фазы 1 составляет O(m), где m - длина шаблона.

Фаза 2. Основной цикл поиска выполняется text\_length раз. Переменная match\_len увеличивается не более чем на 1 на каждой итерации внешнего цикла for. Суммарное увеличение match\_len не превышает text\_length. Каждая итерация while уменьшает match\_len. Общее число уменьшений match\_len не может превысить общее число увеличений match\_len. Следовательно, суммарное время работы внутреннего цикла while по всем итерациям внешнего цикла for является O(text\_length). Добавление элементов в список occurrences происходит не более чем text\_length/pattern\_length раз. Общее время работы фазы 2 составляет O(n), где n - длина текста.

Таким образом, общая временная сложность алгоритма КМП для решения данной задачи составляет O(n+m).

Оценим объем памяти, используемой алгоритмом помимо памяти для хранения исходных строк pattern и text, которые требуют O(m) и O(n) памяти соответственно.

* Массив префикс-функций pi хранит m целых чисел и требует O(m) памяти.
* Список occurrences в худшем случае (когда pattern состоит из одного символа, который встречается везде в тексте) может содержать до n элементов, что требует O(n) памяти.

Объем памяти, необходимый для работы алгоритма КМП: O(n+m).

Задание 2.

Предварительные проверки (сравнение длин, проверка на идентичность строк) - O(N), где N - длина строк. Конкатенация строк занимает время, пропорциональное суммарной длине результирующей строки - 2N => O(N).

Внутри КМП:

* Вычисление функции для str2 - O(N).
* Поиск в тексте doubled\_str1: O(2N) = O(N).

В этой версии kmp\_matcher возвращает только первое вхождение, поэтому алгоритм останавливается сразу при нахождении совпадения, что в худшем случае все равно требует O(2N) времени.

Итого, общая сложность алгоритма для проверки циклического сдвига составляет O(N).

Оценим объем памяти, используемой алгоритмом, помимо O(N) для хранения исходных строк str1, str2.

* Строка doubled\_str1 требует O(2N) = O(N) памяти.
* Внутри КМП:
  + Массив префикс-функции = O(N) памяти
  + Дополнительные переменные (индексы, счетчики) = O(1) памяти.

Итого получаем O(N) памяти.

**Тестирование.**

Задание 1.

Табл.1

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| aba  ababa | 0,2 |
| aabaaab  aabaaacaabaac | -1 |
| test  test string testing another test | 0,12,28 |

Задание 2.

Табл.2

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| defabc  abcdef | 3 |
| cdeab  abcde | 3 |
| hello  hello | 0 |

**Выводы.**

Исследованы и решены две задачи обработки строк. Для быстрого поиска образца в тексте реализован эффективный алгоритм КМП с линейной сложностью. Определение циклического сдвига строк выполнено путем сведения к поиску подстроки (также с КМП) в удвоенной строке, с корректным вычислением требуемого индекса отношения.

**Приложение**

**Код.**

Файл kmp.py

def kmp\_matcher(text: str, pattern: str) -> list[int]:

text\_length = len(text)

pattern\_length = len(pattern)

occurrences = []

print("=" \* 60)

print("АЛГОРИТМ КНУТА-МОРРИСА-ПРАТТА")

print("=" \* 60)

print(f"Текст: '{text}' (длина: {text\_length})")

print(f"Паттерн: '{pattern}' (длина: {pattern\_length})")

print()

pi = compute\_prefix\_function(pattern)

print(f"Префикс-функция pi: {pi}")

print()

print("ПОИСК ПАТТЕРНА В ТЕКСТЕ:")

print("-" \* 60)

match\_len = 0

for i in range(text\_length):

print(f"\nПозиция i={i}, символ text[{i}]='{text[i]}'")

while match\_len > 0 and pattern[match\_len] != text[i]:

old\_match\_len = match\_len

match\_len = pi[match\_len - 1]

print(f" Несовпадение: pattern[{old\_match\_len}]='{pattern[old\_match\_len]}' != text[{i}]='{text[i]}'")

print(f" Откат: match\_len = pi[{old\_match\_len - 1}] = {match\_len}")

if pattern[match\_len] == text[i]:

print(f" Совпадение: pattern[{match\_len}]='{pattern[match\_len]}' == text[{i}]='{text[i]}'")

match\_len += 1

print(f" Увеличиваем match\_len до {match\_len}")

else:

print(f" Несовпадение: pattern[{match\_len}]='{pattern[match\_len]}' != text[{i}]='{text[i]}'")

print(f" match\_len остается 0")

if match\_len == pattern\_length:

occurrence\_pos = i - pattern\_length + 1

occurrences.append(occurrence\_pos)

print(f" \*\*\* НАЙДЕНО ВХОЖДЕНИЕ на позиции {occurrence\_pos} \*\*\*")

print(f" Текст: {text}")

print(f" Паттерн: {' ' \* occurrence\_pos}{pattern}")

match\_len = pi[pattern\_length - 1]

print(f" Продолжаем поиск: match\_len = pi[{pattern\_length - 1}] = {match\_len}")

print("\n" + "=" \* 60)

if not occurrences:

print("Вхождений не найдено")

return [-1]

else:

print(f"Найдено вхождений: {len(occurrences)}")

print(f"Позиции: {occurrences}")

return occurrences

def compute\_prefix\_function(pattern: str) -> list[int]:

pattern\_length = len(pattern)

pi = [0 for \_ in range(pattern\_length)]

border\_len = 0

print("\nВЫЧИСЛЕНИЕ ПРЕФИКС-ФУНКЦИИ:")

print("-" \* 60)

print(f"Паттерн: '{pattern}'")

print(f"Индексы: {' '.join(str(i) for i in range(pattern\_length))}")

print(f"Символы: {' '.join(pattern)}")

print()

print("pi[0] = 0 (по определению)")

for i in range(1, pattern\_length):

print(f"\nВычисляем pi[{i}] для символа '{pattern[i]}':")

while border\_len > 0 and pattern[border\_len] != pattern[i]:

print(f" pattern[{border\_len}]='{pattern[border\_len]}' != pattern[{i}]='{pattern[i]}'")

old\_border = border\_len

border\_len = pi[border\_len - 1]

print(f" Откат: border\_len = pi[{old\_border - 1}] = {border\_len}")

if pattern[border\_len] == pattern[i]:

print(f" pattern[{border\_len}]='{pattern[border\_len]}' == pattern[{i}]='{pattern[i]}'")

border\_len += 1

print(f" Увеличиваем border\_len до {border\_len}")

else:

print(f" pattern[{border\_len}]='{pattern[border\_len]}' != pattern[{i}]='{pattern[i]}'")

print(f" border\_len остается 0")

pi[i] = border\_len

print(f" pi[{i}] = {border\_len}")

print(f" Текущий массив pi: {pi[:i+1]}")

print(f"\nИтоговая префикс-функция: {pi}")

return pi

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

print("Введите паттерн:")

pattern = input()

print("Введите текст:")

text = input()

if len(text) < len(pattern):

print(-1)

else:

result = kmp\_matcher(text, pattern)

print("\nРЕЗУЛЬТАТ:")

print(\*result, sep=',')

Файл shift.py

def kmp\_matcher(text: str, pattern: str) -> int:

text\_length = len(text)

pattern\_length = len(pattern)

print("\nАЛГОРИТМ КМП ДЛЯ ПОИСКА ПАТТЕРНА:")

print("-" \* 60)

print(f"Текст: '{text}' (длина: {text\_length})")

print(f"Паттерн: '{pattern}' (длина: {pattern\_length})")

print()

pi = compute\_prefix\_function(pattern)

print(f"Префикс-функция pi: {pi}")

print()

print("ПОИСК ПАТТЕРНА В ТЕКСТЕ:")

print("-" \* 60)

match\_len = 0

for i in range(text\_length):

print(f"\nПозиция i={i}, символ text[{i}]='{text[i]}'")

while match\_len > 0 and pattern[match\_len] != text[i]:

old\_match\_len = match\_len

match\_len = pi[match\_len - 1]

print(f" Несовпадение: pattern[{old\_match\_len}]='{pattern[old\_match\_len]}' != text[{i}]='{text[i]}'")

print(f" Откат: match\_len = pi[{old\_match\_len - 1}] = {match\_len}")

if pattern[match\_len] == text[i]:

print(f" Совпадение: pattern[{match\_len}]='{pattern[match\_len]}' == text[{i}]='{text[i]}'")

match\_len += 1

print(f" Увеличиваем match\_len до {match\_len}")

else:

print(f" Несовпадение: pattern[{match\_len}]='{pattern[match\_len]}' != text[{i}]='{text[i]}'")

print(f" match\_len остается 0")

if match\_len == pattern\_length:

occurrence\_pos = i - pattern\_length + 1

print(f" \*\*\* НАЙДЕНО ПОЛНОЕ СОВПАДЕНИЕ на позиции {occurrence\_pos} \*\*\*")

print(f" Текст: {text}")

print(f" Паттерн: {' ' \* occurrence\_pos}{pattern}")

return occurrence\_pos

print("\nПолное совпадение не найдено")

return -1

def compute\_prefix\_function(pattern: str) -> list[int]:

pattern\_length = len(pattern)

pi = [0 for \_ in range(pattern\_length)]

border\_len = 0

print("\nВЫЧИСЛЕНИЕ ПРЕФИКС-ФУНКЦИИ:")

print("-" \* 40)

print(f"Паттерн: '{pattern}'")

print(f"Индексы: {' '.join(f'{i:2}' for i in range(pattern\_length))}")

print(f"Символы: {' '.join(f'{c:2}' for c in pattern)}")

print()

print("pi[0] = 0 (по определению)")

for i in range(1, pattern\_length):

print(f"\nВычисляем pi[{i}] для символа '{pattern[i]}':")

while border\_len > 0 and pattern[border\_len] != pattern[i]:

print(f" pattern[{border\_len}]='{pattern[border\_len]}' != pattern[{i}]='{pattern[i]}'")

old\_border = border\_len

border\_len = pi[border\_len - 1]

print(f" Откат: border\_len = pi[{old\_border - 1}] = {border\_len}")

if pattern[border\_len] == pattern[i]:

print(f" pattern[{border\_len}]='{pattern[border\_len]}' == pattern[{i}]='{pattern[i]}'")

border\_len += 1

print(f" Увеличиваем border\_len до {border\_len}")

else:

print(f" pattern[{border\_len}]='{pattern[border\_len]}' != pattern[{i}]='{pattern[i]}'")

print(f" border\_len остается 0")

pi[i] = border\_len

print(f" pi[{i}] = {border\_len}")

print(f" Текущий массив pi: {pi[:i+1]}")

print(f"\nИтоговая префикс-функция: {pi}")

return pi

def check\_cyclic\_shift(str1: str, str2: str) -> int:

print("=" \* 80)

print("ПРОВЕРКА ЦИКЛИЧЕСКОГО СДВИГА")

print("=" \* 80)

print(f"Строка 1 (str1): '{str1}'")

print(f"Строка 2 (str2): '{str2}'")

print()

if len(str1) != len(str2):

print("РЕЗУЛЬТАТ: Строки имеют разную длину!")

print(f"Длина str1: {len(str1)}, длина str2: {len(str2)}")

return -1

print(f"Длины строк совпадают: {len(str1)}")

if str1 == str2:

print("РЕЗУЛЬТАТ: Строки идентичны! Сдвиг = 0")

return 0

print("\nСтроки не идентичны, проверяем циклический сдвиг...")

print("\nМЕТОД: Удваиваем первую строку и ищем вторую строку в ней")

doubled\_str1 = str1 + str1

print(f"Удвоенная str1: '{doubled\_str1}'")

print()

print("Визуализация возможных циклических сдвигов str1:")

for i in range(len(str1)):

shifted = str1[i:] + str1[:i]

print(f" Сдвиг на {i:2}: '{shifted}'")

if shifted == str2:

print(f" ^ Совпадает с str2!")

print("\nЗапускаем поиск str2 в удвоенной str1...")

result = kmp\_matcher(doubled\_str1, str2)

print("\n" + "=" \* 80)

if result != -1:

print(f"РЕЗУЛЬТАТ: str2 является циклическим сдвигом str1 на {result} позиций!")

print(f"Проверка: str1[{result}:] + str1[:{result}] = '{str1[result:] + str1[:result]}' == str2")

else:

print("РЕЗУЛЬТАТ: str2 НЕ является циклическим сдвигом str1")

return result

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

str2 = input()

str1 = input()

result = check\_cyclic\_shift(str2, str1)

print(f"\nИТОГОВЫЙ ОТВЕТ: {result}")