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 ($|P| \leq 25000$) и текста T ($|T| \leq 5000000$) 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 ($|A| \leq 5000000$) и B ($|B| \leq 5000000$).

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}")
```