**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: Кнут-Моррис-Пратт

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Никишин С.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы.**

Изучить и реализовать алгоритм Кнута-Морриса-Пратта.

**Задание 1.**

Реализуйте алгоритм КМП и с его помощью для заданных шаблона *P* (∣*P*∣ ≤ 15000) 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

**Задание 2.**

Заданы две строки *A* (∣*A*∣ ≤ 5000000) и *B* (∣*B*∣ ≤ 5000000∣).  
Определить, является ли А циклическим сдвигом В (это значит, что А и В имеют одинаковую длину и А состоит из суффикса В, склеенного с префиксом В). 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

**Sample Output:**

3

**Описание алгоритма.**

Алгоритм Кнута — Морриса — Пратта (КМП) — это эффективный алгоритм поиска подстроки в строке. Основное преимущество алгоритма КМП заключается в том, что он позволяет избежать повторной обработки уже проверенных символов, что делает его значительно быстрее, чем наивный метод поиска подстроки, особенно для больших текстов.

Алгоритм реализован в функции *kmpAlgorithm(pattern: str, text: str)*. Основные этапы алгоритма при решении задач 1 и 2 одинаковы. Отличались лишь входные и выходные данные.

Основные этапы алгоритма:

1. Обработка образца с помощью префикс функции:

Функция *computePrefixFunction(string: str)* вычисляет префикс-функцию для образца. Т.е мы получаем массив, где для каждого символа образца хранится длина наибольшего собственного префикса, который одновременно является суффиксом для подстроки, заканчивающейся на этом символе.

Принцип работы префикс-функции *computePrefixFunction(string: str)* описан в разделе “Описание функций и структур данных”.

2. Поиск образца в тексте:

Инициализируется массив *positions* для хранения индексов начала вхождений образца в тексте (используется при решении задания 1). Переменная *k* хранит длину текущего совпавшего префикса.

1. Проходим по тексту, начиная с первого символа.

2.1. Если символы не совпадают (pattern[k] != text[i]), уменьшаем k до значения pi[k - 1].

2.2. Если символы совпадают (pattern[k] == text[i]), увеличиваем k на 1.

2.3. Если *k* становится равным длине образца (*k == patternLength*), значит, найдено вхождение.

2.3.1 Если рассматривается задание 1, записываем индекс начала вхождения в массив positions и сдвигаем образец, используя префикс-функцию (k = pi[k - 1]). После окончания цикла, возвращаем массив positions.

2.3.2 Если рассматривается задание 2, просто возвращаем первый индекс начала вхождения.

Отличия при решении задач 1 и 2 незначительные:

1. Меняется вывод (массив индексов вхождений для 1 задачи и первый индекс вхождения для 2 задачи),

2. Для задачи 2 устанавливается условие, что образец и текст одинаковы по длине, и текст увеличивается в 2 раза для определения наличия циклического сдвига.

Алгоритм имеет сложность по памяти O(m + n) (Худший случай, если текст состоит из повторяющегося образца)

Функции имеет сложность по времени:

|  |  |
| --- | --- |
| Функция | Сложность |
| computePrefixFunction(string: str) | O(m) |
| kmpAlgorithm(pattern: str, text: str) | O(m + n) |

,где m – длина образца, n – длина текста

Исходные коды обоих программ указаны в приложении (приложение 1 и приложение 2 соответственно)

**Описание функций и структур данных.**

Структуры данных:

1. pi – список значений префикс-функции

Функции:

1. *def computePrefixFunction(string: str)*: функция, вычисляющая префикс функцию.

Принцип работы:

1. Инициализируется нулями массив pi длиной, равной длине образца.

2. Проходим по образцу, начиная со второго символа.

3.1. Если символы не совпадают (string[k] != string[i]), уменьшаем k до значения pi[k - 1].

3.2. Если символы совпадают (string[k] == string[i]), увеличиваем k на 1.

4. Записываем значение k в pi[i].

5. В конце получаем массив значений, вычисленных префикс-функцией.

Параметры:

*string* – строка, по которой вычисляется префикс функция (тип *str*).

Возвращает:

*pi* — массив значений префикс-функции (тип *list*).

Сложность по времени: O(m)

2. *def kmpAlgorithm(pattern: str, text: str)*: функция, реализующая алгоритм Кнута-Морриса-Пратта для поиска всех вхождений образца в тексте.

Параметры:

*pattern* – строка образца (тип *str*).

*text* – строка текста (тип *str*).

Возвращает:

1. В задаче 1, возвращает массив индексов начал вхождений (тип list).

2. В задаче 2, возвращает первый индекс начала вхождения (тип int).

Сложность по времени: O(m + n)

,где m – длина образца, n – длина текста

**Тестирование.**

Тестирование программ представлено на таблицах.

Тестирование задание 1

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | ab  abab | 0,2 | Верно |
| 2 | faf  abba | -1 | Верно |
| 3 | hah  ahahahahahah | 1,3,5,7,9 | Верно |

Тестирование задание 2

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | defabc  abcdef | 3 | Верно |
| 2 | abc  abc | 0 | Верно |
| 3 | abc  bac | -1 | 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 реализован алгоритм Кнута-Морриса-Пратта (КМП). Реализованный алгоритм был использован для задач на поиск подстроки в строке и на проверку строк на циклический сдвиг. Полученные при тестировании ответы подтвердили корректность работы алгоритма.

**Приложение**

Приложение 1

import sys

def compute\_prefix\_function(pattern):

"""Вычисляет префикс-функцию для шаблона"""

m = len(pattern)

prefix = [0] \* m

k = 0

for q in range(1, m):

while k > 0 and pattern[k] != pattern[q]:

k = prefix[k - 1]

if pattern[k] == pattern[q]:

k += 1

prefix[q] = k

return prefix

def kmp\_search(text, pattern):

"""Находит все вхождения шаблона в текст с помощью алгоритма КМП"""

if not pattern or not text:

return []

n = len(text)

m = len(pattern)

if m > n:

return []

prefix = compute\_prefix\_function(pattern)

occurrences = []

j = 0 # индекс в шаблоне

for i in range(n): # индекс в тексте

while j > 0 and text[i] != pattern[j]:

j = prefix[j - 1]

if text[i] == pattern[j]:

j += 1

if j == m:

# Найдено вхождение

occurrences.append(i - m + 1)

j = prefix[j - 1] # продолжаем поиск следующих вхождений

return occurrences

def main():

print("=== Алгоритм Кнута-Морриса-Пратта (KMP) ===")

print("Поиск всех вхождений шаблона P в тексте T")

print()

print("Введите шаблон P (длина ≤ 25000):")

pattern = sys.stdin.readline().rstrip('\n')

print("Введите текст T (длина ≤ 5000000):")

text = sys.stdin.readline().rstrip('\n')

print(f"\nПоиск шаблона '{pattern}' в тексте длиной {len(text)} символов...")

# Поиск вхождений

occurrences = kmp\_search(text, pattern)

# Формирование вывода

print("\nРезультат поиска:")

if occurrences:

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

result = ','.join(map(str, occurrences))

print(f"Индексы начал вхождений: {result}")

else:

print("Вхождений не найдено: -1")

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

main()

Приложение 2

import sys

def compute\_lps(B):

"""Вычисляет префикс-функцию (LPS - Longest Prefix Suffix) для строки B"""

m = len(B)

if m == 0:

return []

lps = [0] \* m

j = 0

for i in range(1, m):

while j > 0 and B[i] != B[j]:

j = lps[j - 1]

if B[i] == B[j]:

j += 1

lps[i] = j

else:

lps[i] = 0

return lps

def kmp\_search(A, B, lps):

"""Ищет вхождение B в удвоенной строке A с помощью алгоритма КМП"""

j = 0

n = len(A)

for i in range(2 \* n):

while j > 0 and A[i % n] != B[j]:

j = lps[j - 1]

if A[i % n] == B[j]:

j += 1

if j == len(B):

return i - j + 1

return -1

def main():

# Надписи при вводе

print("Введите строку A:")

A = sys.stdin.readline().rstrip('\n')

print("Введите строку B:")

B = sys.stdin.readline().rstrip('\n')

# Проверка условий

if len(A) != len(B):

print("\nРезультат: -1 (строки разной длины)")

return

if len(A) == 0:

print("\nРезультат: 0 (пустые строки)")

return

# Вычисление префикс-функции и поиск

lps = compute\_lps(B)

result = kmp\_search(A, B, lps)

# Вывод результата

if result != -1:

print(f"\nРезультат: {result} (A является циклическим сдвигом B)")

print(f"Сдвиг: первые {result} символов B становятся суффиксом A")

else:

print(f"\nРезультат: -1 (A не является циклическим сдвигом B)")

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

main()