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. 3388 _____ Ижболдин А.В.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Изучить и реализовать алгоритм Кнута–Морриса–Пратта (КМП) для поиска подстроки в строке. Получить практические навыки построения префикс-функции и анализа эффективности алгоритмов строкового поиска. Ознакомиться с преимуществами КМП по сравнению с наивными методами поиска.

Задание

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

Вход:

- Первая строка — P P

- Вторая строка — TT

Выход:

индексы начал вхождений P P в TT, разделённые запятой; если PP не входит в TT, то вывести -1.

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

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

Вход:

- Первая строка — AA

- Вторая строка — BB

Выход: Если AA является циклическим сдвигом BB, то индекс начала строки BB в AA; иначе вывести -1. Если возможно несколько сдвигов, вывести первый индекс.

Выполнение работы

Функция **prefix_function** для вычисления префикс-функции проходит по всем позициям входной строки, начиная со второй, и для каждой вычисляет длину максимального собственного префикса, который одновременно является суффиксом префикса до этой позиции.

На каждом шаге она берёт значение, уже найденное для предыдущего индекса, и, пока текущий символ не совпадёт с символом в позиции, равной этому значению, откатывается к более короткому префиксу, чей размер хранится в массиве.

Если же символы совпадают, значение возрастает на единицу, и в итоговый массив заносится новая длина совпадающего префикса-суффикса.

Функция kmp_optimized. Сначала вычисляет префикс-функцию для образца, что позволяет при поиске по тексту не возвращаться к началу при каждом несовпадении, а «перескакивать» на ту позицию образца, на которой всё ещё может сохраняться совпавшая часть.

Затем он пробегает по всем символам текста, поддерживая указатель на текущую позицию в образце. При несовпадении возвращается к предыдущей возможной длине совпавшего фрагмента через массив префикс-функции, а при совпадении продвигает указатель вперёд.

Когда указатель достигает конца образца, это означает обнаружение полного вхождения, и его начальная позиция добавляется в список результатов.

Для задачи проверки циклического сдвига функция **cyclic** используется приём, допускающий представить бесконечное удвоение первой строки путём взятия её символов по модулю длины. В этой «развёрнутой» строке выполняется поиск второго слова с помощью того же механизма префикс-функции, которое рассчитывается один раз для второго слова.

При совпадении длина совпавшего фрагмента доходит до размера искомой строки, после чего вычисляется фактический сдвиг как остаток от деления пройденного расстояния на длину строки.

Это необходимо для того, чтобы не тратить память на создание новых строк и оптимизировать использование памяти.

Асимптотика алгоритма

Префик функция. Стоит отметить, что j увеличивается на каждом шаге не более чем на единицу, значит максимально возможное значение $j=n-1$. Поскольку внутри цикла while значение j лишь уменьшается, получается, что j не может суммарно уменьшиться больше, чем $n-1$ раз. Значит цикл while в итоге выполнится не более n раз, что дает итоговую оценку времени алгоритма O(n). Память O(n), для массива префикс функции.

КМП. Т.к. префикс. функцию можно вычислить за лин. сложность, то асимптотика КМП будет равно O(n + p), где n - длина текста, а p - длина паттерна.

В функции нахождения цикл. смещения аналогично, т.к. там используется тот же самый КМП, но он находиться прямо в этой функции, чтобы избежать лишнего копирования строк. Откуда временная сложность будет линейная aka O(n) и память, будет также O(n) так как новых строк не создается, максимум массив префикс. суммы.

Вывод

В ходе выполнения лабораторной работы были реализованы и протестированы алгоритмы, основанные на префикс-функции, включая алгоритм Кнута–Морриса–Пратта (КМП) для поиска подстроки в строке, а также алгоритм определения циклического сдвига строк. Реализация префикс-функции позволила добиться линейной временной сложности при

сравнении строк, что делает данные методы эффективными при работе с большими объемами текста.

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
def prefix_function(text, debug=False):
    n = len(text)
    pref = [0] * n

    if debug:
        print(f"Вычисление префикс-функции для строки: '{text}'")

    for i in range(1, n):
        j = pref[i - 1]
        if debug:
            print(f"i={i}, символ='{text[i]}', начальное j={j}")

        while j > 0 and text[i] != text[j]:
            if debug:
                        print(f"   Несовпадение: '{text[i]}' !=
'{text[j]}', переход к j={pref[j-1]}")
            j = pref[j - 1]

        if text[i] == text[j]:
            j += 1
            if debug:
                          print(f"   Совпадение: '{text[i]}' ==
'{text[j-1]}', увеличиваем j до {j}")

        pref[i] = j

    if debug:
        print(f"Итоговая префикс-функция: {pref}")

    return pref

def kmp_optimized(pattern, text, debug=False):
    p = len(pattern)
    n = len(text)

    if debug:
            print(f"KMP_OPTIMIZED: поиск pattern='{pattern}' в
text='{text}'")

    if p > n:
        if debug:
            print("Шаблон длиннее текста, совпадений нет")
        return [-1]

    pref = prefix_function(pattern, debug)

    matches = []
    j = 0

    if debug:
        print("Поиск совпадений:")
```

```
        for i in range(n):
            if debug:
                    print(f"i={i}, символ text[{i}]='{text[i]}', j =
{j}")

            while j > 0 and text[i] != pattern[j]:
                if debug:
                                print(f"   Несовпадение: '{text[i]}' !=
'{pattern[j]}', переход к j={pref[j-1]}")
                j = pref[j - 1]

            if text[i] == pattern[j]:
                j += 1
                if debug:
                                print(f"   Совпадение: '{text[i]}' ==
'{pattern[j-1]}', увеличиваем j до {j}")

            if j == p:
                match_pos = i - p + 1
                if debug:
                      print(f"  Найдено полное совпадение на позиции
{match_pos}")
                matches.append(match_pos)
                j = pref[j - 1]

        if debug:
              print(f"Все найденные совпадения: {matches if matches
else [-1]}")

        return matches if matches else [-1]

    def cyclic(a, b, debug=False):
        n = len(a)
        m = len(b)

        if debug:
            print(f"CYCLIC: поиск сдвига между a='{a}' и b='{b}'")

        if n != m:
            if debug:
                  print("Строки разной длины, циклического сдвига не
существует")
            return -1

        pref = prefix_function(b, debug)
        l = 0

        if debug:
              print(f"Поиск сдвига с использованием префикс-функции
b:")

        for i in range(n * 2):
            c = a[i % n]
            if debug:
                print(f"i={i}, символ a[{i%n}] = '{c}', l={l}")
```

```
            while l > 0 and c != b[l]:
                if debug:
                        print(f"   Несовпадение: '{c}' != '{b[l]}',
переход к l={pref[l-1]}")
                l = pref[l - 1]

            if c == b[l]:
                l += 1
                if debug:
                        print(f"   Совпадение: '{c}' == '{b[l-1]}',
увеличиваем l до {l}")

            if l == m:
                result = (i - m + 1) % n
                if debug:
                    print(f"  Найден циклический сдвиг: {result}")
                return result

        if debug:
            print("Циклический сдвиг не найден")
        return -1

    def find_rot(debug=False):
        a = input()
        b = input()
        print(cyclic(a, b, debug))

    def find_substr(debug=False):
        pattern = input()
        text = input()
                print(','.join(map(str,(kmp_optimized(pattern,   text,
debug)))))

    if __name__ == '__main__':
        # find_substr(debug=False)
        find_rot(debug=False)
```