**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. (КМП)**

Студент гр. 3388 _____ Кулач Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

**Цель работы:**

Изучить принцип работы алгоритма Кнута-Морриса-Пратта для нахождения подстрок в строке. Решить с его помощью задачи.

**Задание 1:**

Реализуйте алгоритм КМП и с его помощью для заданных шаблона $P$ ($|P|\leq 15000$) 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

**Задание 2:**

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 является 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

## Реализация

Описание алгоритма Кнута-Морриса-Пратта:

Алгоритм Кнута-Морриса-Пратта (KMP) разработан для быстрого поиска всех мест в тексте T, где встречается заданная подстрока P. Он повышает эффективность поиска, минимизируя лишние сравнения символов благодаря использованию префикс-функции, которая позволяет пропускать уже обработанные участки текста при несовпадении.

**Шаги алгоритма**

1. **Проверка размеров**: сначала проверяются длины P и T. Если P пустая или её длина превышает длину T, возвращается пустой список.
2. **Вычисление префикс-функции π для P**:
   - Для каждого символа P[i] (где i от 1 до m−1, а m — длина P) вычисляется значение π[i].
   - Если символы P[k] и P[i] не совпадают, значение k уменьшается с использованием π[k−1], пока не будет найдено совпадение или k не станет равным 0.
   - Если P[k] совпадает с P[i], значение k увеличивается.
   - Итоговое значение π[i] равно k.
3. **Поиск вхождений P в T**:
   - Проходим по тексту T с индексом i, отслеживая количество совпавших символов q (из P).
   - Если P[q] не равно T[i], q уменьшается с использованием π[q−1].
   - Если P[q] равно T[i], q увеличивается.
   - Когда q достигает m (длина P), это означает полное совпадение, и позиция i−m+1 добавляется в список результатов. Затем q уменьшается с использованием π[q−1] для продолжения поиска.

Описание функций и структур:

• `list[int] compute_prefix_function_verbose(const str& P)` – функция, которая вычисляет префикс-функцию для шаблона P. Возвращает список, в котором каждый элемент pi[i] показывает длину наибольшего префикса, совпадающего с суффиксом подстроки P[0..i]. В процессе работы выводит подробные комментарии (отладочную информацию) по каждому шагу.

• `list[int] kmp_search_verbose(const str& P, const str& T)` – функция, которая ищет все вхождения строки P в строке T с использованием алгоритма Кнута–Морриса–Пратта (KMP). Возвращает список индексов, с которых начинаются вхождения шаблона в тексте. Пошагово выводит ход поиска.

• `int kmp_search_shift_verbose(const str& P, const str& T)` – функция, которая проверяет, является ли строка P циклическим сдвигом строки T. Возвращает индекс, начиная с которого строка T при сдвиге совпадает с P, или -1, если сдвига не существует. Использует KMP и выводит подробные шаги.

• `void main()` – основная функция, которая запрашивает у пользователя выбор задачи:

- если введено 1, выполняется поиск подстроки P в тексте T с помощью `kmp_search_verbose`;
- если введено 2, проверяется, является ли строка B циклическим сдвигом строки A с помощью `kmp_search_shift_verbose`;
- если введён некорректный номер задачи, выводится соответствующее сообщение.

Оценка сложности алгоритма:

**Временная сложность**

Вычисление префикс-функции:

- Проход по P длиной m: O(m).
- Итог: O(m).

Поиск:

- Проход по T длиной n: O(n).
- Внутренний цикл while уменьшает q по pi, но общее число шагов равно O(n), так как каждое уменьшение компенсируется предыдущим увеличением.
- Добавление позиций: O(z), где z — число вхождений, но $z \leq n$.
- Итог: O(n).

Общая: O(m+n)

**Пространственная сложность**

Префикс-функция:

- pi: O(m) для массива длиной m.

Поиск:

- occurrences: O(z) для хранения позиций, где $z \leq n$.

Итого: O(m + z)

## Тестирование

Таблица 1. Тестирование.

| Входные данные | Выходные данные |
|---|---|
| alksdmlaksmdlkamdlksamdlkamdlkamdaldmlakdmlakdmalkmd amdaldmlakdmlakdmalkmdalksdmlaksmdlkamdlksamdlkamdlk | 30 |
| aba<br>abababababa | 0,2,4,6 |
| bab<br>bbaababbabaabab | 4,7,12 |

## Вывод

В ходе лабораторной работы были написаны программы с использованием алгоритма Ахо-Корасика. Также дополнительно было сделано: подсчёт вершин и определение пересечений.

**Исходный код программы см. в ПРИЛОЖЕНИИ А.**

# ПРИЛОЖЕНИЕ А.
# ИСХОДНЫЙ КОД ПРОГРАММЫ

main.py

```
def compute_prefix_function_verbose(P):
    print("Построение pi-функции:")
    n = len(P)
    pi = [0] * n
    j = 0
    for i in range(1, n):
        while j > 0 and P[i] != P[j]:
            print(f"q={i}: {P[i]}!={P[j]} и k={j}; переход к pi[{j -
1}]={pi[j - 1]}")
            j = pi[j - 1]
        if P[i] == P[j]:
            j += 1
            print(f"q={i}: {P[i]}=={P[j-1]}; pi[{i}]={j}")
        else:
            print(f"q={i}: {P[i]}!={P[j]} и k=0; pi[{i}]=0")
        pi[i] = j
    print(f"Результат pi: {pi}\n")
    return pi

def kmp_search_verbose(P, T):
    print("Поиск вхождений:")
    pi = compute_prefix_function_verbose(P)
    result = []
    j = 0
    for i in range(len(T)):
        while j > 0 and T[i] != P[j]:
            print(f"i={i}: {T[i]}!={P[j]}, q={j} -> q={pi[j - 1]}")
            j = pi[j - 1]
        if T[i] == P[j]:
            j += 1
            print(f"i={i}: {T[i]} совпало, q-> {j}")
        else:
            print(f"i={i}: {T[i]}!={P[j]}, q={j}")
        if j == len(P):
            print(f"Найдено вхождение с {i - len(P) + 1}")
            result.append(i - len(P) + 1)
            j = pi[j - 1]
    return result

def kmp_search_shift_verbose(P, T):
    print("Проверка циклического сдвига:")
    pi = compute_prefix_function_verbose(P)
    j = 0
    for i in range(len(T)):
        while j > 0 and T[i] != P[j]:
            print(f"i={i}: {T[i]}!={P[j]}, q={j} -> q={pi[j - 1]}")
            j = pi[j - 1]
        if T[i] == P[j]:
            j += 1
            print(f"i={i}: {T[i]} совпало, q-> {j}")
        else:
            print(f"i={i}: {T[i]}!={P[j]}, q={j}")
        if j == len(P):
            print(f"Найден сдвиг {i - len(P) + 1}")
```

```
            return i - len(P) + 1
    return -1

def main():
    task = input("").strip()

    if task == "1":
        P = input().strip()
        T = input().strip()
        matches = kmp_search_verbose(P, T)
        if matches:
            print(",".join(map(str, matches)))
        else:
            print(-1)

    elif task == "2":
        A = input().strip()
        B = input().strip()
        if len(A) != len(B):
            print(-1)
        else:
            A2 = A + A
            result = kmp_search_shift_verbose(B, A2)
            print(result)

    else:
        print("Некорректный выбор. Введите 1 или 2.")

if __name__ == "__main__":
    main()
```