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 _____ Наумкин А.Д,

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Изучить принцип работы алгоритма Кнута-Морриса_Пратта. Написать функцию, вычисляющую для каждого элемента строки максимальное значение длины префикса и с помощью данной функции решить поставленные задачи. А именно написать программу, осуществляющую поиск вхождений подстроки в строку, а также программу, определяющую, являются ли строки циклическим сдвигом друг друга, найти индекс начала вхождения второй строки в первую.

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* 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

Префикс-функция:

Алгоритм начинается с инициализации трех переменных:

- пустой список prefixes, заполненный нулями длиной строки, для которой нужно найти префикс функцию.

- i – индекс, для прохождения по строке

- j – переменная, хранящая в себе текущую длину совпадений суффикса с префиксом.

Далее, пока не достигнут конец строки, проверяем максимальное число совпадений символов, попутно увеличивая счетчики i,j. Как только мы нашли первые неравные символы, появляется два исхода: 1) совпадений не было, т. е. Можем просто продолжить перебор, оставив текущий префикс нулевым. 2) Совпадения были и нам требуется вернуть значение j на prefixes[j-1]. Откат на prefixes[j-1] символов позволяет нам эффективно продолжать поиск с максимально возможной позиции в подстроке, не повторяя уже выполненных проверок.

Таким образом, формируется список, состоящих из максимальных длин префиксов. Далее данный список prefixes возвращается.

Алгоритм Кнута-Морриса-Пратта:

Принцип работы алгоритма Кнута-Морриса-Пратта (КМП) заключается в эффективном использовании префикс-функции. Вместо того чтобы возвращаться к уже проверенным символам при несоответствии, алгоритм использует префикс-функцию для сдвига подстроки на максимально возможную позицию.

Алгоритм состоит из двух основных шагов:

1. Построение префикс-функции для подстроки.
2. Поиск всех вхождений подстроки в строку с использованием префикс-функции.

Шаг 1. Подробное описание построения префикс-функции приведено в разделе "Префикс-функция".

Шаг 2. Для поиска всех вхождений подстроки в строку, мы одновременно идем по строке и подстроке, сравнивая символы на каждой позиции. Если символы совпадают, переходим к следующей позиции. В случае несовпадения применяем префикс-функцию, чтобы определить, на какую позицию нужно сдвинуть подстроку вправо. Это позволяет продолжить сравнение с максимально возможной позицией, не теряя информации о возможных совпадениях. Сдвиг осуществляется на значение prefixes[j-1], где j — позиция, на которой произошло несовпадение.

Алгоритм продолжает сравнение символов до тех пор, пока не будут найдены все вхождения подстроки в строку. Если по завершению строки вхождений не найдено, возвращается пустой массив, а затем выводится значение -1.

Таким образом, алгоритм КМП эффективно находит все вхождения подстроки, минимизируя количество ненужных сравнений благодаря использованию префикс-функции.

Оценка сложности алгоритма по памяти и операциям.

1. Сложность алгоритма поиска подстроки.

Сложность по времени линейная O(n+m), где m – длина подстроки, n – длина строки. Так как за O(m) осуществляется построение префикс-функции, а также за O(n) осуществляется проход по строке, чтобы найти индексы вхождения. Сложность по памяти O(m), так как нужно хранить вектор префиксов данной длины.

2. Сложность алгоритма поиска циклического сдвига. Сложность по времени O(m+2n), где n – длина строки. Так как за O(m) осуществится построение префикс функции, а за O(2n) дважды будет осуществлен проход строки.

Сложность по памяти O(m).

Описание функций.

В процессе выполнения работы были написаны следующие функции:

`compute_lps(pattern, verbose=False)`

Функция, принимающая на вход строку и вычисляющая значения максимальных длин префиксов для каждого элемента. Результат записывает в контейнер std::vector и возвращает его.

`kmp_search(text, pattern, verbose=False)`

Функция, принимающая на вход подстроку pattern, вхождение которой будем искать в строке text. Возвращается строка, содержащая информацию об индексах начала вхождений подстроки в строку.

Также были созданы файлы для измерения времени выполнения КМП с различными входными данными, реализован алгоритм наивного поиска для сравнения и программа на python, создающая графики из выборки.

Тестирование.

Проведем тестирование.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ab avdabhjab	Результат: 3, 7	Тест к первому заданию. Верно найдены индексы вхождения подстроки в строку.
2.	abc avdabhjab	Результат: -1	Тест ко первому заданию, когда нет ни одного вхождения подстроки в строку. Результатом в данном случае будет -1.
3.	ababab bababa	Результат: 1	Тест ко второму заданию. Верно определен индекс первого вхождения циклического сдвига (тут их несколько).
4.	abcedf abcdef	Результат: -1	Тест к второму заданию. Верно определено то, что нет циклического сдвига, результат = -1.
5.	abcedfabccaab	0 0 0 0 0 0 1 2 3 0 1 1 2	Протестирована функция prefixFunction(), верно вычислен результат.

Результат работы программы с отладочным выводом для первого задания (см. рис 1, 2, 3).

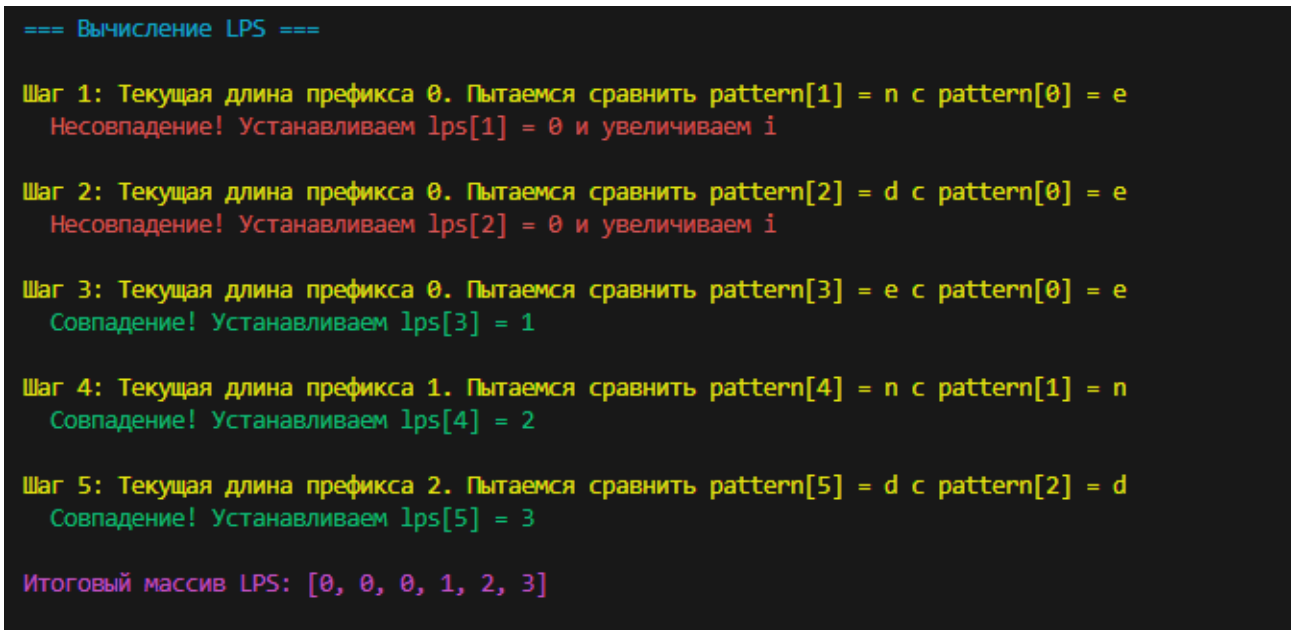

Рисунок 1 – вычисление префикс функции

```
=== Поиск КМР ===

Шаг 0: Сравниваем text[0] = T с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 1

Шаг 1: Сравниваем text[1] = h с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 2

Шаг 2: Сравниваем text[2] = i с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 3

Шаг 3: Сравниваем text[3] = s с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 4

Шаг 4: Сравниваем text[4] =   с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 5

Шаг 5: Сравниваем text[5] = s с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 6

Шаг 6: Сравниваем text[6] = e с pattern[0] = e
  Совпадение! Переходим к следующему символу: i = 7, j = 1

Шаг 7: Сравниваем text[7] = n с pattern[1] = n
  Совпадение! Переходим к следующему символу: i = 8, j = 2
  Несовпадение! Переходим на j = 0 согласно lps

Шаг 8: Сравниваем text[8] = t с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 9

Шаг 9: Сравниваем text[9] = e с pattern[0] = e
  Совпадение! Переходим к следующему символу: i = 10, j = 1

Шаг 10: Сравниваем text[10] = n с pattern[1] = n
  Совпадение! Переходим к следующему символу: i = 11, j = 2
  Несовпадение! Переходим на j = 0 согласно lps

Шаг 11: Сравниваем text[11] = c с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 12

Шаг 12: Сравниваем text[12] = e с pattern[0] = e
  Совпадение! Переходим к следующему символу: i = 13, j = 1
  Несовпадение! Переходим на j = 0 согласно lps

Шаг 13: Сравниваем text[13] =   с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 14

Шаг 14: Сравниваем text[14] = e с pattern[0] = e
  Совпадение! Переходим к следующему символу: i = 15, j = 1

Шаг 15: Сравниваем text[15] = n с pattern[1] = n
  Совпадение! Переходим к следующему символу: i = 16, j = 2

Шаг 16: Сравниваем text[16] = d с pattern[2] = d
  Совпадение! Переходим к следующему символу: i = 17, j = 3
  Несовпадение! Переходим на j = 0 согласно lps

Шаг 17: Сравниваем text[17] = s с pattern[0] = e
  Несовпадение! Увеличиваем i: i = 18
```

Рисунок 2 – вывод KMP

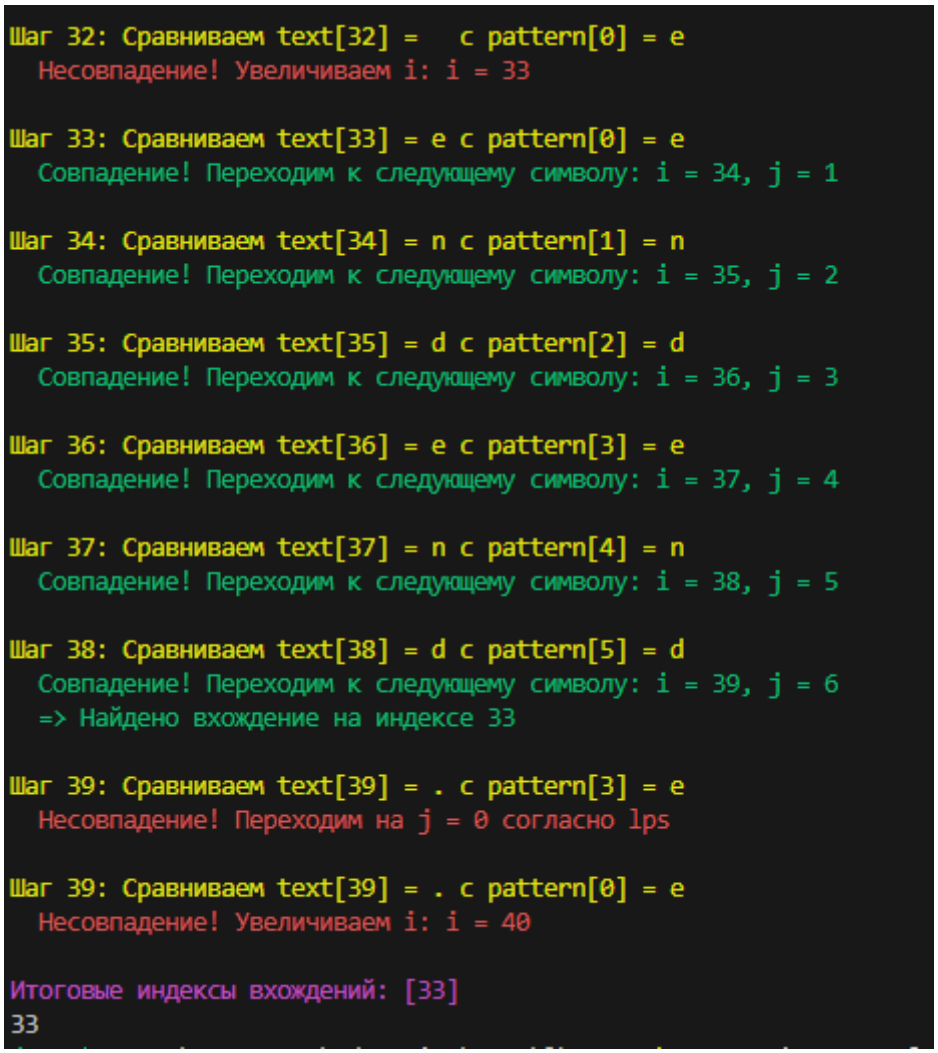

Рисунок 3 – вывод KMP

Исследование.

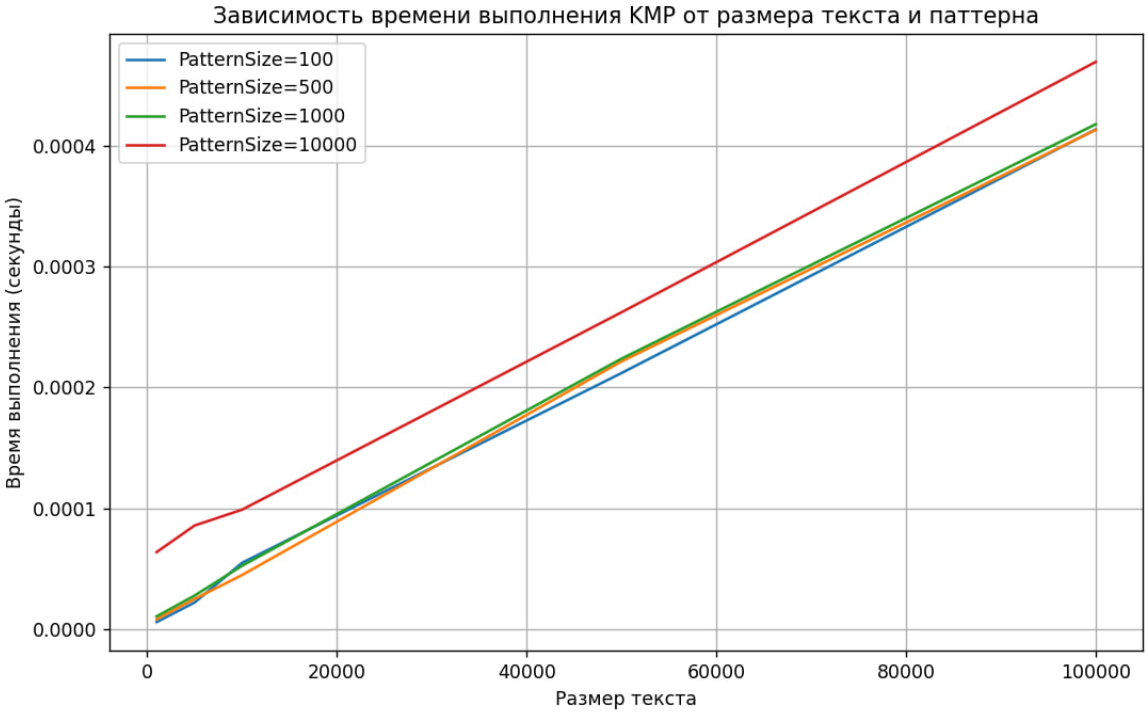

Рисунок 4 – Тестирование КМП

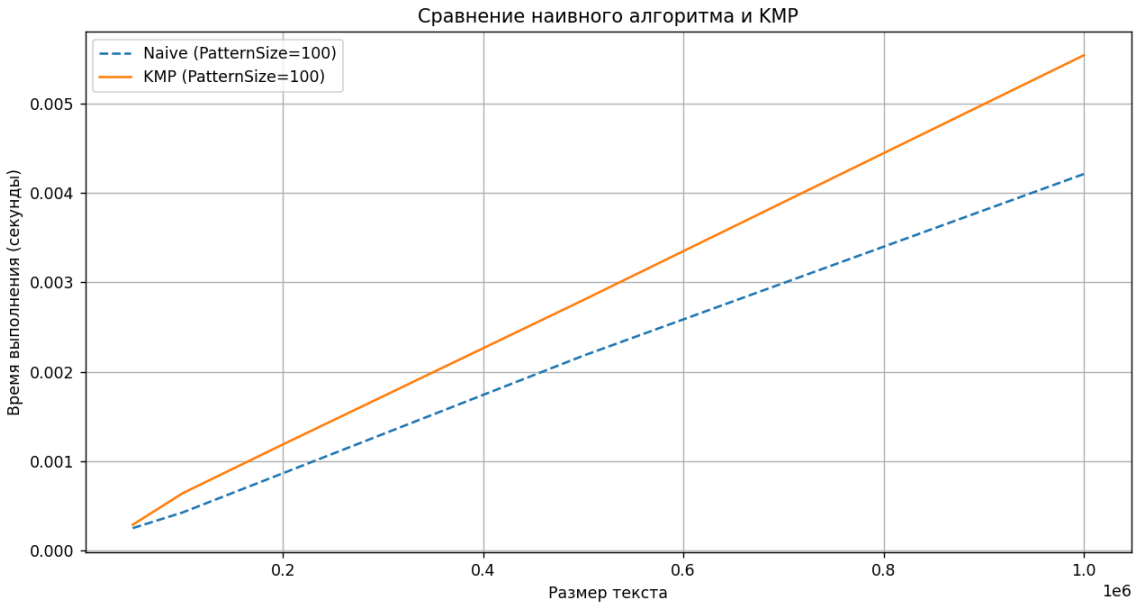

Рисунок 5 – Сравнение КМП и прямого обхода

Можно сделать вывод, что КМП выполняется быстрее, чем наивный алгоритм, показатели могут быть лучше на выборках, которые содержат много последовательностей символов входящих в подстроку.

Выводы.

Изучен принцип работы алгоритма Кнута-Морриса-Пратта. Написаны программы, корректно решающие поставленные задачи с помощью функции вычисления максимальной длины префикса для каждого символа.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: kmp.py

```
from colorama import Fore, Back, Style, init
init(autoreset=True)

def compute_lps(pattern, verbose=False):
    """
    Вычисляет массив LPS (Longest Prefix Suffix).
    LPS[i] хранит длину наибольшего собственного префикса, который является суффиксом строки pattern[:i+1].
    """
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1

    if verbose:
        print(Fore.CYAN + "\n=== Вычисление LPS ===")

    while i < m:
        if verbose:
            print(Fore.YELLOW + f"\nШаг {i}: Текущая длина префикса {length}. Пытаемся сравнить pattern[{i}] = {pattern[i]} с pattern[{length}] = {pattern[length]}")

        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
            if verbose:
                print(Fore.GREEN + f"  Совпадение! Устанавливаем lps[{i-1}] = {length}")
        elif length:
            length = lps[length - 1]
            if verbose:
```

```
                    print(Fore.RED + f"  Несовпадение! Уменьшаем длину
префикса до lps[{length}]")
            else:
                lps[i] = 0
                i += 1
                if verbose:
                      print(Fore.RED + f"  Несовпадение! Устанавливаем
lps[{i-1}] = 0 и увеличиваем i")

        if verbose:
            print(Fore.MAGENTA + f"\nИтоговый массив LPS: {lps}\n")

        return lps

    def kmp_search(text, pattern, verbose=False):
        """
        Реализация алгоритма Кнута-Морриса-Пратта для поиска подстроки в
строке.
        Возвращает список индексов начала всех вхождений pattern в text.
        """
        n, m = len(text), len(pattern)
        if m == 0:
            return list(range(n + 1))

        lps = compute_lps(pattern, verbose)
        indices = []
        i = j = 0

        if verbose:
            print(Fore.CYAN + "\n=== Поиск КМР ===")

        while i < n:
            if verbose:
                print(Fore.YELLOW + f"\nШаг {i}: Сравниваем text[{i}] =
{text[i]} с pattern[{j}] = {pattern[j]}")

            if text[i] == pattern[j]:
                i += 1
```

```
                j += 1
                if verbose:
                        print(Fore.GREEN + f"  Совпадение! Переходим к
следующему символу: i = {i}, j = {j}")

            if j == m:
                indices.append(i - j)
                if verbose:
                   print(Fore.GREEN + f"  => Найдено вхождение на индексе
{i - j}")
                j = lps[j - 1]

            elif i < n and text[i] != pattern[j]:
                if j:
                    j = lps[j - 1]
                    if verbose:
                         print(Fore.RED + f"  Несовпадение! Переходим на
j = {j} согласно lps")
                else:
                    i += 1
                    if verbose:
                          print(Fore.RED + f"  Несовпадение! Увеличиваем
i: i = {i}")

        if verbose:
                print(Fore.MAGENTA  +  f"\nИтоговые  индексы  вхождений:
{indices}")

        return indices

    from test import *
    if __name__ == "__main__":
        pattern = pattern5
        text = text5

        verbose = True

        result = kmp_search(text, pattern, verbose)
```

```
print(",".join(map(str, result)) if result else -1)
```

Название файла: cycle.py

```
from colorama import Fore, Back, Style, init
init(autoreset=True)

def kmp_search(text, pattern, verbose=False):
    """
    Реализует алгоритм КМП для поиска вхождения pattern в text.
     Возвращает индекс первого вхождения или -1, если вхождение не
найдено.
    """
    n = len(pattern)
    lps = [0] * n  # lps[i] — длина наибольшего собственного префикса,
совпадающего с суффиксом pattern[0:i+1]
    length = 0  # длина предыдущего совпадающего префикса
    i = 1

    # Промежуточный вывод для LPS
    if verbose:
        print(Fore.CYAN + "\n=== Вычисление массива LPS ===")

    while i < n:
        if verbose:
            print(Fore.YELLOW + f"Шаг {i}: Сравниваем pattern[{i}] =
{pattern[i]} с pattern[{length}] = {pattern[length]}")
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
            if verbose:
              print(Fore.GREEN + f"Совпадение! Устанавливаем lps[{i-
1}] = {length}")
        else:
            if length != 0:
                length = lps[length - 1]
                if verbose:
                    print(Fore.RED + f"Несовпадение! Уменьшаем длину
префикса до lps[{length}]")
            else:
                lps[i] = 0
                i += 1
                if verbose:
                     print(Fore.RED + f"Несовпадение! Устанавливаем
lps[{i-1}] = 0 и увеличиваем i")

    if verbose:
        print(Fore.MAGENTA + f"\nИтоговый массив LPS: {lps}\n")

    # Поиск pattern в text
    i = 0  # индекс для text
    j = 0  # индекс для pattern
    m = len(text)
```

```
        if verbose:
            print(Fore.CYAN + "\n=== Поиск совпадений в тексте ===")

        while i < m:
            if verbose:
                 print(Fore.YELLOW + f"Шаг {i}: Сравниваем text[{i}] =
{text[i]} с pattern[{j}] = {pattern[j]}")

            if pattern[j] == text[i]:
                i += 1
                j += 1
                if verbose:
                         print(Fore.GREEN + f"Совпадение! Переходим к
следующему символу: i = {i}, j = {j}")
                if j == n:
                    if verbose:
                      print(Fore.GREEN + f"Найдено вхождение! Возвращаем
индекс {i - j}")
                    return i - j  # найдено вхождение, возвращаем индекс
начала
            else:
                if j != 0:
                    j = lps[j - 1]
                    if verbose:
                        print(Fore.RED + f"Несовпадение! Переходим на j
= {j} согласно lps")
                else:
                    i += 1
                    if verbose:
                        print(Fore.RED + f"Несовпадение! Увеличиваем i:
i = {i}")

        if verbose:
            print(Fore.MAGENTA + "Вхождение не найдено.")
        return -1

    from test_cycle import *
    def main():
        # Считываем строки строго в порядке ввода:
        # первая строка - A, вторая строка - B.
        A = A1
        B = B1

        verbose = True  # Установите в True для вывода шагов

        # Проверка длины строк
        if len(A) != len(B):
            print(Fore.RED + -1)
            return

        # Если строки совпадают, циклический сдвиг равен 0.
        if A == B:
            print(Fore.GREEN + 0)
            return

        # Проверяем, является ли A циклическим сдвигом B.
        # Для этого ищем A в строке B+B.
```

```
        text = B + B
        pos = kmp_search(text, A, verbose)

        # Если найденное вхождение начинается в пределах длины строки B,
        # вычисляем сдвиг.
        if pos != -1 and pos < len(B):
             print(Fore.GREEN + f"Циклический сдвиг: {(len(B) - pos) %
len(B)}")
        else:
            print(Fore.RED + -1)

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