**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.**

**Алгоритм Кнута-Морриса-Пратта.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Иванов П. Д. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

## Цель работы

Ознакомиться с принципами работы алгоритма Кнута–Морриса–Пратта (КМП) и реализовать функцию для вычисления префикс-функции строки. На основе этой функции разработать:

1. Программу для поиска всех вхождений подстроки в строку;

2. Алгоритм для нахождения индекса начала вхождения одной строки в другую при условии циклического сдвига.

## 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.

№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 является циклическим сдвигом B, индекс начала строки B в A, иначе вывести −1. Если возможно несколько сдвигов вывести первый индекс.

## Выполнение работы

**Префикс-функция**

Функция *vector\_prefix* предназначена для построения префиксного вектора для заданной строки *s*. Этот вектор используется в алгоритме Кнута-Морриса-Пратта (КМП) для эффективного поиска подстроки в строке.

Принцип работы:

Инициализация:

* Определяется длина строки s и создаётся массив p той же длины, заполненный нулями.
* Переменная j используется для хранения текущей длины наибольшего общего префикса, совпадающего с суффиксом для текущей позиции.

Основной цикл:

* Цикл начинается с индекса i = 1 (так как для первого символа префиксная функция равна 0).
* Для каждого символа s[i] функция сравнивает его с символом s[j]:
* Если символы совпадают, значение j увеличивается на 1, и это значение записывается в p[i].
* Если символы не совпадают и j больше 0, происходит «откат»: j устанавливается равным p[j-1]. Этот шаг позволяет избежать повторного сравнения уже обработанных символов.
* Если после отката символы совпадают — опять увеличивается j и обновляется p[i].

**Алгоритм КМП**

Функция *kmp(A, B)* реализует модифицированный алгоритм Кнута-Морриса-Пратта для поиска строки *A* в строке *B*, рассматривая строку *B* как циклически сдвигаемую. Для имитации циклического сдвига строка *B* логически удваивается (через использование операции взятия по модулю), что позволяет искать совпадения, охватывающие конец и начало строки.

Принцип работы:

Инициализация:

* Вычисляются длины строк A (обозначается как m) и B (обозначается как n).
* Вызывается функция vector\_prefix(A) для вычисления префиксного вектора строки A. Этот вектор (p) используется для определения, на сколько можно сдвинуть индекс j при обнаружении несовпадения символов.
* Переменная j инициализируется нулём и служит индексом для строки A.

Поиск в удвоенной строке:

* Основной цикл проходит от i = 0 до i = 2 \* n - 1. При этом символ из строки B берётся по индексу i % n, что позволяет "обернуть" конец строки и продолжить сравнение с начала.

Для каждой итерации:

* + Сравнивается текущий символ ch из строки B с символом A[j].
  + Если происходит несовпадение и j > 0, алгоритм использует префиксный вектор для отката: j устанавливается равным p[j - 1]. Это позволяет избежать повторного сравнения уже проверенных префиксов.
  + Если символы совпадают, значение j увеличивается на 1.

Нахождение совпадения:

* Когда значение j достигает длины m (то есть j == m), это означает, что найдено полное совпадение строки A в удвоенной строке B + B.
* Вычисляется индекс начала совпадения: idx = i - m + 1.
* Если idx меньше n, это означает, что совпадение найдено в пределах одного цикла строки B, и функция возвращает этот индекс как корректный циклический сдвиг.
* Если совпадение выходит за пределы первоначальной строки (то есть idx ≥ n), происходит откат значения j и поиск продолжается.

Функция *vector\_kmp(sub\_str, search\_str)* используется для поиска всех вхождений подстроки *sub\_str* в строке *search\_str* с помощью алгоритма Кнута-Морриса-Пратта. Функция возвращает список индексов, с которых начинаются все найденные вхождения. Если вхождения отсутствуют, возвращается пустой список.

Принцип работы:

Подготовка строки для префикс-функции:

* Функция проверяет, что символ-разделитель "~" отсутствует как в sub\_str, так и в search\_str. Этот символ используется для объединения строк, чтобы корректно разделять подстроку и строку поиска.
* Создаётся строка full\_str, которая представляет собой конкатенацию: sub\_str + "~" + search\_str. Это позволяет вычислить префиксный вектор для объединённой строки и затем определить позиции, в которых полное совпадение подстроки встречается в строке поиска.

Вычисление префиксного вектора:

* Вызывается функция vector\_prefix для строки full\_str. Префиксный вектор позволяет определить длину наибольшего общего префикса, совпадающего с суффиксом для каждой позиции в строке.
* Длина подстроки сохраняется в переменной sub\_len.

Поиск вхождений:

* Цикл начинается с позиции i = sub\_len + 1 в строке full\_str (начиная сразу после символа-разделителя) и продолжается до конца строки.
* Для каждой позиции проверяется значение p[i] (префиксного вектора). Если p[i] равно sub\_len, это означает, что подстрока sub\_str полностью совпала с частью строки search\_str.
* Вычисляется индекс начала вхождения в строке поиска по формуле: *index = i - 2 \* sub\_len.*
* Найденный индекс добавляется в список matching\_indices.

**Оценка сложности алгоритмов**

* Алгоритм поиска подстроки в строке:
  + Время: O(m+n). Где m — длина подстроки, n — длина строки поиска. Основное время работы занимает префикс-функция, которая обрабатывает каждый символ за O(1), а значит вся строка обработается за O(m+n).
  + Память: O(m+n). Для формирования строки full\_str используется память O(m+n). Дополнительно используется массив p, размер которого соответствует длине full\_str — O(m+n).
* Алгоритм поиска при циклическом сдвиге:
  + Время**:** O(m+2n), где m – длина первой строки, n – длина второй строки.
  + Память**:** O(m) – хранение префиксного вектора для первой строки.

## Тестирование

Результаты тестирования представлены в таблице 1.

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | avarrdgghjidav | [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2] | Тест префикс-функции. Результат вычислен верно. |
| 2. | ava  avavaavagdsedavdvava | 0,2,5,17 | Тест для первого задания (подстрока содержится в поисковой строке). Результат вычислен верно. |
| 3. | asd  aswasgghrhfgbdsa | -1 | Тест для первого задания (подстрока не содержится в поисковой строке). Результат вычислен верно. |
| 4. | baa  aba | 2 | Тест для второго задания (строки являются циклическими сдвигами). Результат вычислен верно. |
| 5. | qwerty  asdfgh | -1 | Тест для второго задания (строки не являются циклическими сдвигами). Результат вычислен верно. |

Табл. 1. – Результаты тестирования

## Исследование

Были проведены два теста, которые проверяли: 1 - изменение скорости работы алгоритма КМП в зависимости от паттерна и размера текста(от 100 до 100000 символов) (Рис.1).

2 – сравнение скорости работы алгоритма КМП с прямым поиском (Рис.2).

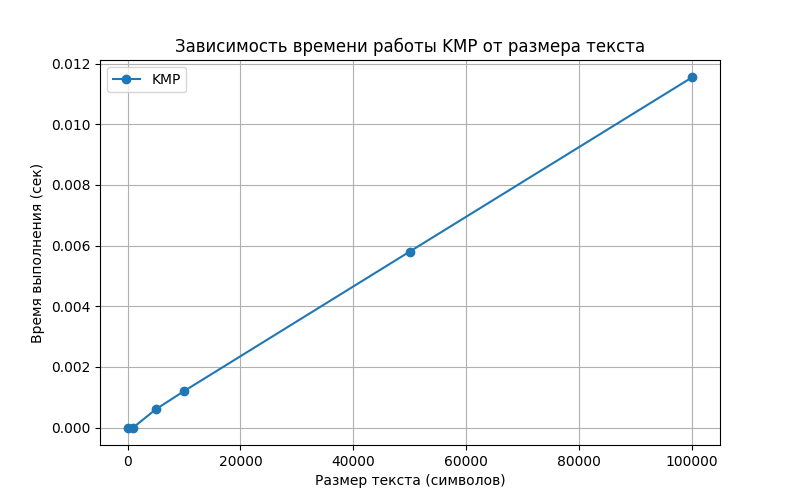


Рис. 1 – Скорость работы алгоритма в зависимости от размера текста

По графику хорошо видно, что скорость работы растет линейно от размера текста, что подтверждается оценкой временной сложности в O(n+m).

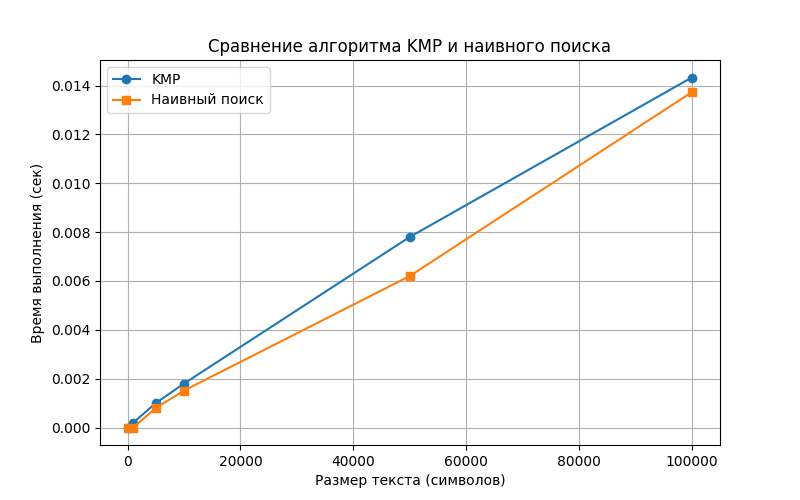


Рис. 2 – Сравнение скорости работы алгоритмов КМП и наивного поиска

Из данного графика видно, что алгоритм КМП работает быстрее (особенно на больших объемах текста), так как он более оптимизирован и не проводит лишних бессмысленных итераций.

## Выводы

Был детально изучен принцип работы алгоритма Кнута‑Морриса‑Пратта, что позволило разработать программы, корректно решающие поставленные задачи с использованием функции, вычисляющей максимальную длину префикса для каждого символа.

# Приложение А

1 – Файл kmp.py

DEBUG = False

def vector\_prefix(s):

if DEBUG:

print(f"\nСтроится префиксный вектор для строки: {s}")

n = len(s)

p = [0] \* n

j = 0

for i in range(1, n):

if DEBUG:

print(f"\nИтерация i = {i}; j = {j}:")

print(f"Текущий символ на позиции i: s[{i}] = '{s[i]}', текущее значение по j: s[{j}] = '{s[j]}'")

while j and s[i] != s[j]:

if DEBUG:

print(f"Несоответствие: s[{i}] = '{s[i]}' != s[{j}] = '{s[j]}'. Обновляем j: {j} -> {p[j-1]}")

j = p[j - 1]

if s[i] == s[j]:

j += 1

if DEBUG:

print(f"Совпадение: s[{i}] = '{s[i]}' == s[{j-1}] = '{s[j-1]}'. Увеличиваем j до {j}")

p[i] = j

if DEBUG:

print(f"Промежуточный префиксный вектор: {p}")

if DEBUG:

print(f"\nИтоговый префиксный вектор: {p}\n")

return p

def vector\_kmp(sub\_str, search\_str):

if "~" in search\_str or "~" in sub\_str:

raise ValueError("Символ разделитель присутствует в строке")

full\_str = sub\_str + "~" + search\_str

if DEBUG:

print(f"Сформированная строка для алгоритма КМП: {full\_str}")

p = vector\_prefix(full\_str)

sub\_len = len(sub\_str)

matching\_indices = []

if DEBUG:

print(f"\nИщем подстроку '{sub\_str}' в строке '{search\_str}'")

for i in range(sub\_len + 1, len(full\_str)):

if DEBUG:

print(f"Проверяем позицию i = {i} с префиксным значением {p[i]}")

if p[i] == sub\_len:

index = i - 2 \* sub\_len

if DEBUG:

print(f"Найдено вхождение (len({sub\_str}) = p[{i}] = {p[i]})). Индекс начала в поисковой строке: {index}")

matching\_indices.append(index)

if DEBUG:

if matching\_indices:

print(f"\nВсе найденные индексы вхождений: {matching\_indices}")

else:

print("\nВхождения не найдены.")

return matching\_indices

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

sub\_str = input()

search\_str = input()

res = vector\_kmp(sub\_str, search\_str)

print(','.join(map(str, res)) if res else -1)

2 – Файл cyclic\_shift.py

DEBUG = False

def vector\_prefix(s):

if DEBUG:

print(f"\nСтроится префиксный вектор для строки: {s}")

n = len(s)

p = [0] \* n

j = 0

for i in range(1, n):

if DEBUG:

print(f"\nИтерация i = {i}; j = {j}:")

print(f"Текущий символ на позиции i: s[{i}] = '{s[i]}', текущее значение по j: s[{j}] = '{s[j]}'")

while j and s[i] != s[j]:

if DEBUG:

print(f"Несоответствие: s[{i}] = '{s[i]}' != s[{j}] = '{s[j]}'. Обновляем j: {j} -> {p[j-1]}")

j = p[j - 1]

if s[i] == s[j]:

j += 1

if DEBUG:

print(f"Совпадение: s[{i}] = '{s[i]}' == s[{j-1}] = '{s[j-1]}'. Увеличиваем j до {j}")

p[i] = j

if DEBUG:

print(f"Промежуточный префиксный вектор: {p}")

if DEBUG:

print(f"\nИтоговый префиксный вектор: {p}\n")

return p

def kmp(A, B):

n = len(B)

m = len(A)

if DEBUG:

print(f"\nПоиск строки A = '{A}' в удвоенной строке B + B = '{B + B}'")

print(f"Длина A = {m}, длина B = {n}, перебор от i = 0 до i = {2 \* n - 1}")

p = vector\_prefix(A)

j = 0

for i in range(2 \* n):

ch = B[i % n]

a\_ch = A[j] if j < m else "-"

if DEBUG:

print(f"\nПроверка для i = {i} (B[{i % n}] = '{ch}'); j = {j} (A[{j}] = '{a\_ch}')")

while j > 0 and ch != A[j]:

if DEBUG:

print(f"Несовпадение: '{ch}' != '{A[j]}'. Откат j: {j} -> {p[j - 1]}")

j = p[j - 1]

if ch == A[j]:

j += 1

if DEBUG:

print(f"Совпадение: '{ch}' == '{A[j - 1]}'. Увеличиваем j -> {j}")

else:

if DEBUG:

print(f"Нет совпадения: '{ch}' != '{A[j]}' (j = {j})")

if j == m:

idx = i - m + 1

if DEBUG:

print(f"Подстрока найдена! Начало совпадения в удвоенной строке: {idx}")

if idx < n:

if DEBUG:

print(f"Индекс {idx} < длина строки {n}, это корректный сдвиг")

return idx

j = p[j - 1]

if DEBUG:

print(f"Продолжаем поиск, j откат -> {j}")

if DEBUG:

print("Совпадений не найдено")

return -1

def cyclic\_shift\_check(A, B):

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

if DEBUG:

print("Строки разной длины — циклический сдвиг невозможен.")

return -1

if not A and not B:

if DEBUG:

print("Обе строки пусты — сдвиг 0")

return 0

k = kmp(A, B)

if k == -1:

if DEBUG:

print("Циклический сдвиг не найден")

return -1

result = (len(B) - k) % len(B)

if DEBUG:

print(f"Циклический сдвиг найден. Сдвиг = {result}")

return result

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

first\_str = input()

second\_str = input()

print(cyclic\_shift\_check(first\_str, second\_str))

3 – Файл test.py

import random

import string

import time

import matplotlib.pyplot as plt

# Функция генерации случайной строки заданной длины

def generate\_random\_string(length):

return ''.join(random.choices(string.ascii\_lowercase, k=length))

# Реализация префиксного вектора для алгоритма КМП

def vector\_prefix(s):

n = len(s)

p = [0] \* n

j = 0

for i in range(1, n):

while j and s[i] != s[j]:

j = p[j - 1]

if s[i] == s[j]:

j += 1

p[i] = j

return p

# Алгоритм КМП поиска первого вхождения паттерна в текст

def kmp\_search(pattern, text):

# Формируем объединённую строку с разделителем

full\_str = pattern + "~" + text

p = vector\_prefix(full\_str)

pat\_len = len(pattern)

for i in range(pat\_len + 1, len(full\_str)):

if p[i] == pat\_len:

return i - 2 \* pat\_len # индекс вхождения

return -1

# Наивная реализация поиска подстроки

def naive\_search(pattern, text):

n = len(text)

m = len(pattern)

for i in range(n - m + 1):

if text[i:i + m] == pattern:

return i

return -1

# Функция для замера времени выполнения переданной функции поиска

def measure\_time(search\_func, pattern, text, iterations=5):

total\_time = 0

result = None

for \_ in range(iterations):

start = time.time()

result = search\_func(pattern, text)

end = time.time()

total\_time += (end - start)

return total\_time / iterations, result

# Исследование зависимости времени работы алгоритма КМП от размера текста

def experiment\_kmp\_text\_size():

text\_sizes = [100, 1000, 5000, 10000, 50000, 100000]

times = []

# Фиксируем паттерн (например, 10 символов)

pattern = generate\_random\_string(10)

print("Эксперимент KMP (размер текста):")

for size in text\_sizes:

text = generate\_random\_string(size)

t, res = measure\_time(kmp\_search, pattern, text)

times.append(t)

print(f"Размер текста: {size:6d} | Время: {t:.6f} сек | Результат: {res}")

plt.figure(figsize=(8, 5))

plt.plot(text\_sizes, times, marker='o', label='KMP')

plt.xlabel('Размер текста (символов)')

plt.ylabel('Время выполнения (сек)')

plt.title('Зависимость времени работы KMP от размера текста')

plt.legend()

plt.grid(True)

plt.savefig("kmp\_text\_size.png")

plt.show()

# Сравнение алгоритмов KMP и наивного поиска

def experiment\_kmp\_vs\_naive():

text\_sizes = [100, 1000, 5000, 10000, 50000, 100000]

kmp\_times = []

naive\_times = []

# Фиксируем паттерн (например, 10 символов)

pattern = generate\_random\_string(10)

print("\nСравнение KMP и наивного поиска:")

for size in text\_sizes:

text = generate\_random\_string(size)

t\_kmp, res\_kmp = measure\_time(kmp\_search, pattern, text)

t\_naive, res\_naive = measure\_time(naive\_search, pattern, text)

kmp\_times.append(t\_kmp)

naive\_times.append(t\_naive)

print(

f"Размер текста: {size:6d} | KMP: {t\_kmp:.6f} сек (результат: {res\_kmp}) | Наивный: {t\_naive:.6f} сек (результат: {res\_naive})")

plt.figure(figsize=(8, 5))

plt.plot(text\_sizes, kmp\_times, marker='o', label='KMP')

plt.plot(text\_sizes, naive\_times, marker='s', label='Наивный поиск')

plt.xlabel('Размер текста (символов)')

plt.ylabel('Время выполнения (сек)')

plt.title('Сравнение алгоритма KMP и наивного поиска')

plt.legend()

plt.grid(True)

plt.savefig("kmp\_vs\_naive.png")

plt.show()

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

random.seed(42) # для воспроизводимости

experiment\_kmp\_text\_size()

experiment\_kmp\_vs\_naive()