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 _____ Отмахов Д. В.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Изучить принцип работы алгоритма Кнута-Морриса-Пратта, разработать программу, реализующую поиск всех вхождений заданного шаблона в текст, а также определить, является ли одна строка циклическим сдвигом другой, используя модификацию данного алгоритма.

Задание 1

Реализуйте алгоритм КМП и с его помощью для заданных шаблона $P(|P| \leq 15000)$ и текста $T(|T| \leq 5.000.000)$ найдите все вхождения P в T.

Входные данные:

Первая строка – P

Вторая строка – T

Выходные данные:

Индексы начал вхождений P в T, разделенные запятой, если P не входит в T, то вывести -1.

Sample Input:

ab

abab

Sample Output:

0,2

Задание 2

Заданы две строки $A(|A| \leq 5.000.000)$ и $B(|B| \leq 5.000.000)$.

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.

Входные данные:

Первая строка – A

Вторая строка – B

Выходные данные:

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

Выполнение работы

Описание алгоритма

Алгоритм Кнута-Морриса-Пратта (КМП) — это эффективный алгоритм поиска подстроки в строке, основанный на использовании префикс-функции. Основная идея заключается в том, чтобы избежать избыточных сравнений символов при несовпадении шаблона с текстом, используя предварительно вычисленную информацию о самом шаблоне.

Этапы алгоритма:

1. Вычисление префикс-функции для шаблона:

Префикс-функция prefix[i] определяет длину наибольшего собственного суффикса подстроки pattern[0..i], который одновременно является её префиксом.

Это позволяет при несовпадении символов "перепрыгивать" по уже обработанной части шаблона, а не начинать сравнение с начала.

2. Поиск вхождений шаблона в текст:

Алгоритм последовательно сравнивает символы текста и шаблона, используя префикс-функцию для оптимизации сдвигов.

При несовпадении символов указатель в шаблоне сдвигается не на начало, а на значение префикс-функции в текущей позиции, что уменьшает количество сравнений.

3. Определение циклического сдвига:

Для проверки, является ли строка *text* циклическим сдвигом строки *pattern*, строка *pattern* удваивается, и в ней ищется вхождение *text* с помощью алгоритма КМП.

Если вхождение найдено, его позиция соответствует величине сдвига.

Оценка сложности

Временная сложность алгоритма линейна – $O(n + m)$, где n – длина текста, а m – длина паттерна. Это достигается за счёт однократного прохода по тексту и шаблону с использованием префикс-функции.

Пространственная сложность алгоритма - $O(m)$, так как требуется дополнительная память для хранения префикс-функции шаблона.

Код программы содержит реализацию класса *KMPMatcher*:

- *std::string pattern* – строка, содержащая образец (подстроку), который мы ищем в тексте*;*
- *std::vector<int> prefix* – массив значений префикс-функции для заданного образца;
- *static std::vector<int> computePrefix(const std::string& pattern)* – статический метод, вычисляющий префикс-функцию для заданного образца. Префикс-функция определяет длину наибольшего собственного суффикса, который совпадает с префиксом;
- *std::vector<int> searchAll(const std::string& text)* – метод, осуществляющий поиск всех вхождений образца в тексте и возвращающий вектор индексов начала этих вхождений;
- *int serchFirst(const std::string& text)* – метод, осуществляющий поиск первого вхождения образца в тексте и возвращающий индекс его начала или -1, если образец не найден;
- *static int cyclicShiftIndex(const std::string& pattern*, *const std::string& text)* – статический метод, определяющий, является ли строка text циклическим сдвигом строки pattern. Возвращает индекс начала сдвига или -1, если text не является циклическим сдвигом pattern.

Тестирование

Оба метода(*searchAll, cyclicShiftIndex)* были протестированы на различных входных данных.

Таблица 1. Тестирование метода *searchAll* (Задание 1).

Входные данные	Выходные данные
ab abab	0,2
aba ababaababa	0,2,5,7
xyz abcdef	-1
 abc	-1
abcdef abcdef	0

Таблица 2. Тестирование метода *cyclicShiftIndex* (Задание 1).

Входные данные	Выходные данные
defabc abcdef	3
hello world	-1
abcdef abcdef	0
abc abcd	-1
	-1

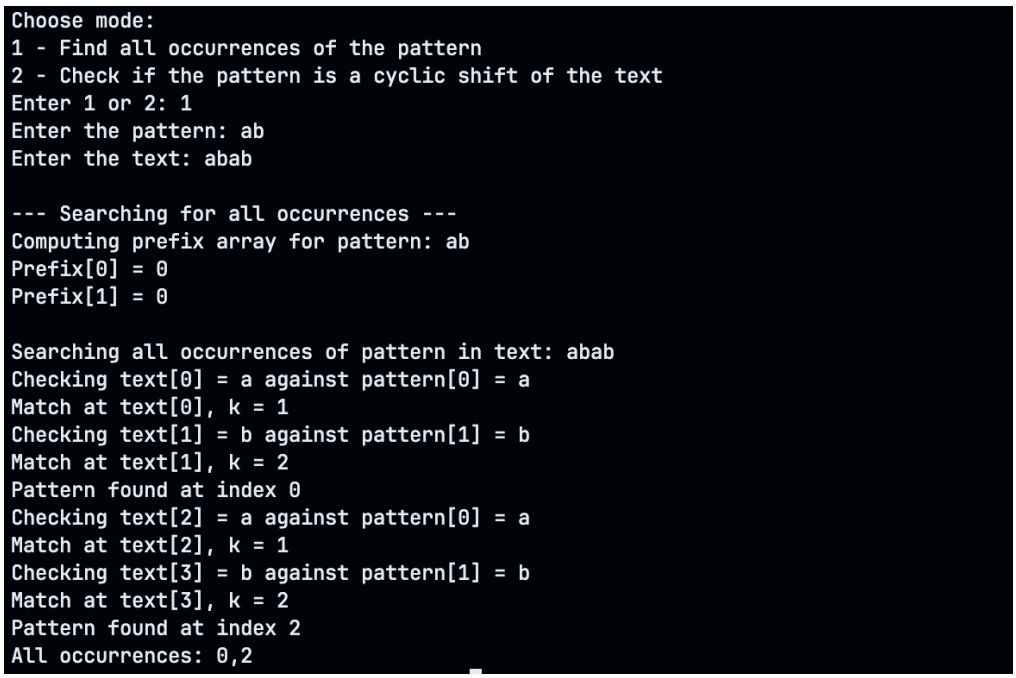

Рисунок 1 – Результат работы программы для нахождения всех вхождений подстроки в текст

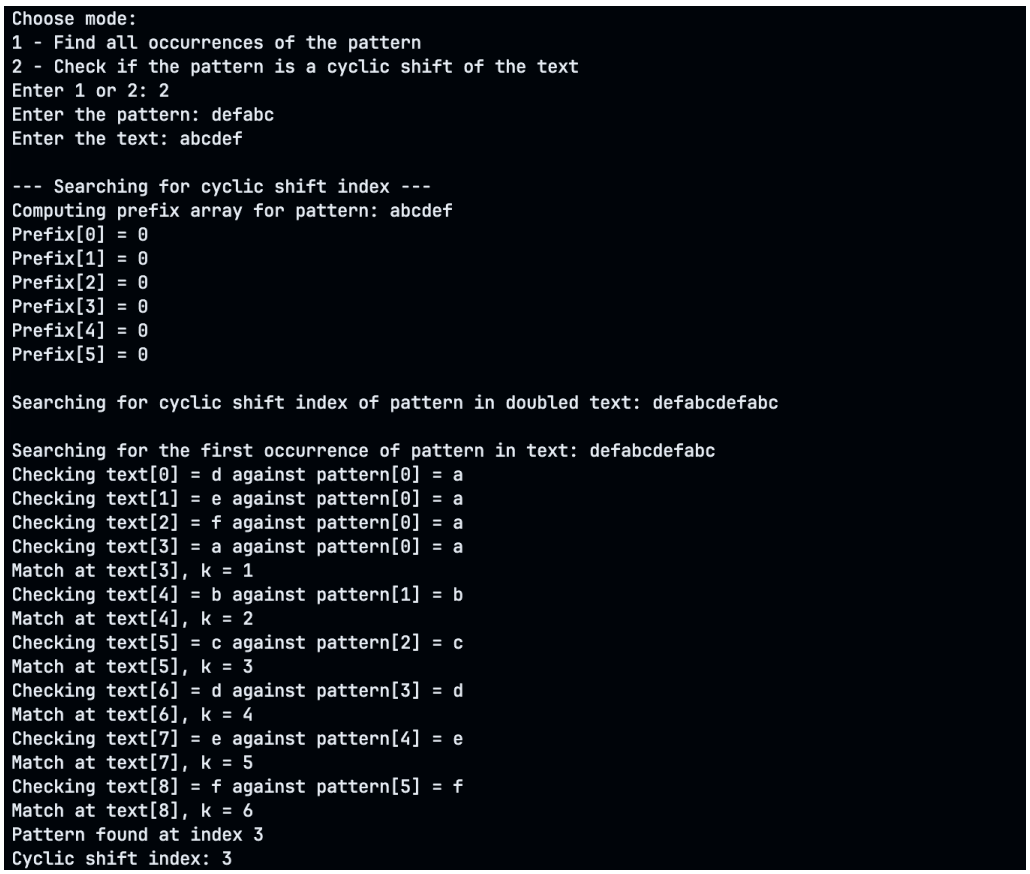

Рисунок 2 – Результат работы программы для определения циклического сдвига

Исследование

Исследуем эффективность алгоритма Кнута-Морриса-Пратта на различных объёмах входных данных.

Таблица 3. Исследование эффективности по времени.

Размер паттерна	Размер текста	Затраченное время, с
1	1.000	0.000279417
10	1.000	0.000308917
1	100.000	0.0179685
100	100.000	0.0161675
10.000	100.000	0.0134173
1	1.000.000	0.0638443
1.000	1.000.000	0.0694583
100.000	1.000.000	0.0725747
1	10.000.000	0.642375
100	10.000.000	0.765016
10.000	10.000.000	0.763974
1.000.000	10.000.000	0.794663

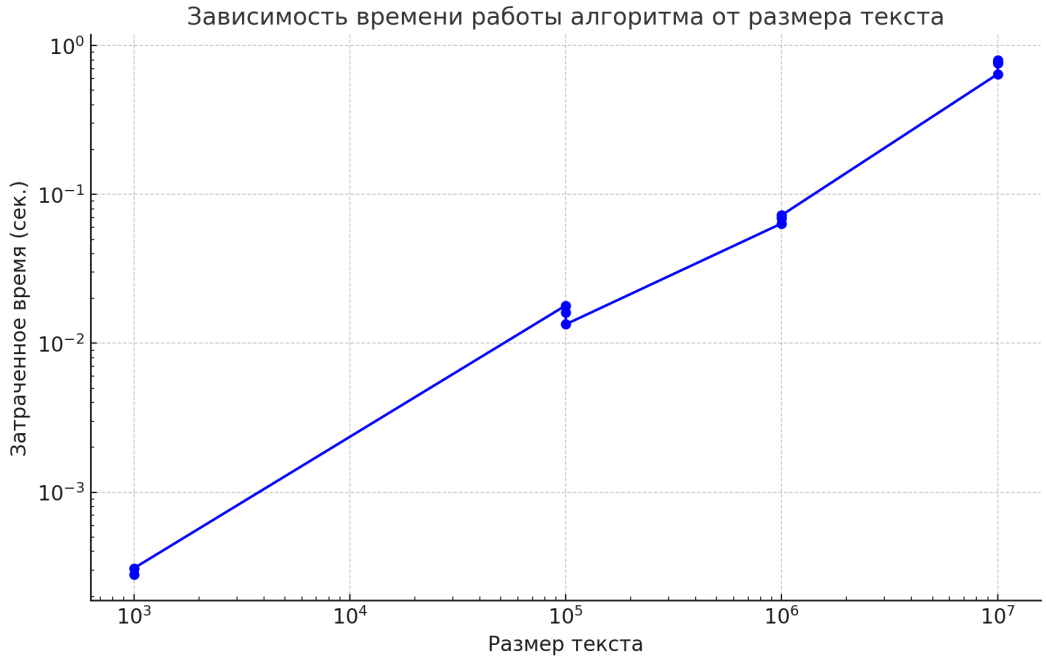

Рисунок 3 – График зависимости времени работы алгоритма от размера текста

Можно сделать следующие выводы по исследованию:

1. Время работы алгоритма прямо пропорционально размеру текста.
2. Время работы практически не зависит от размера паттерна, но при небольших размерах, результат вычисляется эффективнее.

Выводы

В ходе лабораторной работы был изучен и реализован алгоритм Кнута-Морриса-Пратта, позволяющий эффективно находить все вхождения подстроки в текст и определять циклические сдвиги строк, что подтверждено тестированием на различных входных данных.

ПРИЛОЖЕНИЕ
ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.cpp

```
#include <iostream>
#include <vector>
#include <string>

class KMPMatcher {
private:
    std::string pattern;
    std::vector<int> prefix;

    static std::vector<int> computePrefix(const std::string& pattern) {
        std::vector<int> prefix(pattern.size(), 0);
        int k = 0;

        std::cout << "Computing prefix array for pattern: " << pattern <<
std::endl;
        std::cout << "Prefix[0] = 0" << std::endl;

        for (size_t i = 1; i < pattern.size(); ++i) {
            while (k > 0 && pattern[k] != pattern[i]) {
                std::cout << "Mismatch at position " << i << " (k = " << k
<< "). Backtracking to " << prefix[k - 1] << std::endl;
                k = prefix[k - 1];
            }
            if (pattern[k] == pattern[i]) {
                k++;
                std::cout << "Match found at position " << i << " (k = "
<< k << ")." << std::endl;
            }
            prefix[i] = k;
            std::cout << "Prefix[" << i << "] = " << prefix[i] <<
std::endl;
        }

        return prefix;
    }

public:
```

```
        KMPMatcher(const std::string& pattern) : pattern(pattern),
prefix(computePrefix(pattern)) {}

        std::vector<int> searchAll(const std::string& text) {
            std::vector<int> result;
            int k = 0;

            std::cout << "\nSearching all occurrences of pattern in text: " <<
text << std::endl;

            for (size_t i = 0; i < text.size(); ++i) {
                std::cout << "Checking text[" << i << "] = " << text[i] << "
against pattern[" << k << "] = " << pattern[k] << std::endl;

                while (k > 0 && pattern[k] != text[i]) {
                    std::cout << "Mismatch at text[" << i << "], backtracking
to prefix[" << (k - 1) << "] = " << prefix[k - 1] << std::endl;
                    k = prefix[k - 1];
                }
                if (pattern[k] == text[i]) {
                    k++;
                    std::cout << "Match at text[" << i << "], k = " << k <<
std::endl;
                }
                if (k == (int)pattern.size()) {
                    result.push_back(i - k + 1);
                    std::cout << "Pattern found at index " << (i - k + 1) <<
std::endl;
                    k = prefix[k - 1];
                }
            }

            return result;
        }

        int searchFirst(const std::string& text) {
            int k = 0;

            std::cout << "\nSearching for the first occurrence of pattern in
text: " << text << std::endl;

            for (size_t i = 0; i < text.size(); ++i) {
```

```
                std::cout << "Checking text[" << i << "] = " << text[i] << "
against pattern[" << k << "] = " << pattern[k] << std::endl;

                while (k > 0 && pattern[k] != text[i]) {
                    std::cout << "Mismatch at text[" << i << "], backtracking
to prefix[" << (k - 1) << "] = " << prefix[k - 1] << std::endl;
                    k = prefix[k - 1];
                }
                if (pattern[k] == text[i]) {
                    k++;
                    std::cout << "Match at text[" << i << "], k = " << k <<
std::endl;
                }
                if (k == (int)pattern.size()) {
                    std::cout << "Pattern found at index " << (i - k + 1) <<
std::endl;
                    return i - k + 1;
                }
            }

            return -1;
        }

        static int cyclicShiftIndex(const std::string& pattern, const
std::string& text) {
            if (pattern.size() != text.size()) {
                return -1;
            }

            KMPMatcher matcher(text);
            std::string doubled = pattern + pattern;

            std::cout << "\nSearching for cyclic shift index of pattern in
doubled text: " << doubled << std::endl;

            int index = matcher.searchFirst(doubled);

            if (index != -1 && index < static_cast<int>(pattern.size())) {
                return index;
            }

            return -1;
        }
```

```
};

int main() {
    std::cout << "Choose mode:\n";
    std::cout << "1 - Find all occurrences of the pattern\n";
    std::cout << "2 - Check if the pattern is a cyclic shift of the
text\n";
    std::cout << "Enter 1 or 2: ";

    int mode;
    std::cin >> mode;
    std::cin.ignore();

    std::string pattern, text;
    std::cout << "Enter the pattern: ";
    std::getline(std::cin, pattern);
    std::cout << "Enter the text: ";
    std::getline(std::cin, text);

    if (pattern.empty() || text.empty()) {
        std::cout << "-1" << std::endl;
        return 0;
    }

    if (mode == 1) {
        std::cout << "\n--- Searching for all occurrences ---\n";
        if (pattern.size() > text.size()) {
            std::cout << "-1" << std::endl;
        } else {
            KMPMatcher matcher(pattern);
            std::vector<int> occurrences = matcher.searchAll(text);

            if (occurrences.empty()) {
                std::cout << "-1" << std::endl;
            } else {
                std::cout << "All occurrences: ";
                for (size_t i = 0; i < occurrences.size(); ++i) {
                    if (i != 0) std::cout << ",";
                    std::cout << occurrences[i];
                }
                std::cout << std::endl;
            }
        }
```

```
    } else if (mode == 2) {
        std::cout << "\n--- Searching for cyclic shift index ---\n";
        int shift = KMPMatcher::cyclicShiftIndex(pattern, text);
        std::cout << "Cyclic shift index: " << shift << std::endl;
    } else {
        std::cout << "Invalid mode selected." << std::endl;
    }

    return 0;
}
```