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 _____ Павлов А.Р.

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 (КМП) предназначен для эффективного поиска всех вхождений заданного шаблона P в текст T. Он оптимизирует наивный процесс поиска, избегая ненужных сравнений символов за счёт использования префикс-функции, которая позволяет пропускать уже проверенные части текста при несовпадении. Алгоритм применяется в задачах обработки строк, где требуется найти все позиции начала подстроки P в T.

Шаги алгоритма

Проверяется длина P и T: если P пуст или длиннее T, возвращается пустой список. Вычисляется префикс-функция prefix для P.

Для каждого символа P[i] (от 1 до m−1) определяется prefix[i]. Если P[k]≠P[i], k уменьшается по prefix[k−1] до совпадения или 0. Если P[k]=P[i], k увеличивается.

prefix[i]=k.

Далее делаем проход по T с индексом i и текущим совпадением q (число совпавших символов P). При P[q]≠T[i]: q уменьшается по prefix[q−1]. При P[q]=T[i]: q увеличивается. Если q=m (полное совпадение), позиция i−m+1 добавляется в результат, q сдвигается по prefix[q−1].

.

Описание функций и структур:

- *vector<size_t> prefix_function(const string& x)* – функция, которая вычисляет префикс-функцию для строки *x*.
- *vector<size_t> kmp_search(const string& haystack, const string& needle)* – функция, которая ищет все вхождения *needle* в *haystack* с использованием КМП.

Оценка сложности алгоритма:

Временная сложность

Вычисление префикс-функции:

- Проход по *needle* длиной m: $O(m)$.
- Итог: $O(m)$.

Поиск:

- Проход по *haystack* длиной n: $O(n)$.
- Внутренний цикл while уменьшает j по *needle*, 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: $O(m + n)$

Сложность по памяти

Префикс-функция:

- prefix: $O(m)$ 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
ACGTACGT AGCT	
AB ABAB	0,2
AC ACGTACAC	0,4,6

Вывод

В ходе лабораторной работы были написаны программы с использованием алгоритма Кнута-Морриса-Пратта. Также дополнительно был добавлен отладочный вывод для интерпретации результатов работы программы.

Исходный код программы см. в приложении А.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

task1.cpp

```
#include <iostream>
#include <string>
#include <vector>
#define DEBUG
using namespace std;


vector<size_t> prefix_function(const string& x) {
    size_t m = x.length();
    vector<size_t> lps(m, 0);
    size_t len = 0;
    size_t i = 1;

    while (i < m) {
        if (x[i] == x[len]) {        // если текущий символ совпадает
с символом на позиции len
            len++;                   // увеличиваем длину
префикса/суффикса
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {          // у нас есть предыдущий
префикс/суффикс
                len = lps[len - 1];
            } else {                 // нет предыдущего
префикса/суффикса
                lps[i] = 0;
                i++;
            }
        }
    }

#ifdef DEBUG
    for (size_t i = 0; i < m; i++)
        cout << "prefix[" << i << "] = " << lps[i] << endl;
#endif

    return lps;
}


vector<size_t> kmp_search(const string& needle, const string&
haystack) {
    size_t n = haystack.length();
    size_t m = needle.length();
    vector<size_t> lps = prefix_function(needle);
    vector<size_t> occurrences;
    size_t i = 0;
    size_t j = 0;

    while (i < n) { // перебираем текст
```

```
        if (needle[j] == haystack[i]) {
            j++;
            i++;
        }

        if (j == m) {                       // если мы достигли конца
шаблона, значит, нашли вхождение
#ifdef DEBUG
            cout << "Нашли совпадение по индексу: " << i - j << endl;
#endif
            occurrences.push_back(i - j);
            j = lps[j - 1];                 // переходим к длине
совпадающего префикса/суффикса, чтобы найти перекрывающиеся вхождения
        }
        // расхождение после j совпадений
        else if (i < n && needle[j] != haystack[i]) {
            if (j != 0) {       // у нас есть уже совпадающий
префикс/суффикс, можно перепрыгнуть
                j = lps[j - 1];
            } else {            // нет предыдущего префикса/суффикса
                i++;
            }
        }
    }
    return occurrences;
}


int main() {
    string needle, haystack;
    cin >> needle >> haystack;

    vector<size_t> occurrences = kmp_search(needle, haystack);

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

    return 0;
}
```

task2.cpp

```
#include <iostream>
#include <string>
#include <vector>
#define DEBUG
using namespace std;


vector<size_t> prefix_function(const string& x) {
    size_t m = x.length();
```

```
    vector<size_t> lps(m, 0);
    size_t len = 0;
    size_t i = 1;

    while (i < m) {
        if (x[i] == x[len]) {        // если текущий символ совпадает
с символом на позиции len
            len++;                   // увеличиваем длину
префикса/суффикса
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {          // у нас есть предыдущий
префикс/суффикс
                len = lps[len - 1];
            } else {                 // нет предыдущего
префикса/суффикса
                lps[i] = 0;
                i++;
            }
        }
    }

#ifdef DEBUG
    for (size_t i = 0; i < m; i++)
        cout << "prefix[" << i << "] = " << lps[i] << endl;
#endif

    return lps;
}


size_t kmp_first(const string& needle, const string& haystack) {
    size_t n = haystack.length();
    size_t m = needle.length();
    vector<size_t> lps = prefix_function(needle);
    size_t i = 0;
    size_t j = 0;

    while (i < n) { // перебираем текст
        if (needle[j] == haystack[i]) {
            j++;
            i++;
        }

        if (j == m) {                    // если мы достигли конца
шаблона, значит, нашли вхождение
            return i - j;                // в этом задании нам надо
только первое вхождение
        }
        // расхождение после j совпадений
        else if (i < n && needle[j] != haystack[i]) {
            if (j != 0) {       // у нас есть уже совпадающий
префикс/суффикс, можно перепрыгнуть
                j = lps[j - 1];
            } else {            // нет предыдущего префикса/суффикса
                i++;
```

```
            }
        }
    }

    return -1;
}


int main() {
    string a, b;
    cin >> a >> b;

    // при разной длине точно нельзя сдвигом получить другую строку
    if (a.length() != b.length()) {
        cout << -1 << endl;
        return 0;
    }

    size_t index = kmp_first(b, a + a);
    if (index == -1)
        cout << -1 << endl;
    else
        cout << index << endl;

    return 0;
}
```