**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

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Песчатский С. Д. |
| Преподаватель |  | Виноградова Е. В. |

Санкт-Петербург

2025

## Задание

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*T* разделенных запятой, если P не входит в *T*, то вывести −1.

**Sample Input:**

ab

abab

**Sample Output:**

0,2

Заданы две строки A (∣A∣≤5000000) и B (∣B∣≤5000000).  
Определить, является ли А циклическим сдвигом ВВ (это значит, что А и В имеют одинаковую длину и А состоит из суффикса В, склеенного с префиксом ВВ). Например, defabc является циклическим сдвигом abcdef.  
Вход:  
Первая строка - A  
Вторая строка - B  
Выход:  
Если *A* является циклическим сдвигом B*B*, индекс начала строки B в A, иначе вывести −1. Если возможно несколько сдвигов вывести первый индекс.

**Sample Input:**

defabc

abcdef

**Sample Output:**

3

**Описание алгоритма**

Алгоритм Кнута-Морриса-Пратта (KMP) - это эффективный алгоритм поиска подстроки в строке. Он работает за линейное время O(n+m), где n - длина текста, m - длина образца.

Основные компоненты алгоритма:

1. Префикс-функция (LPS-массив):

- Вычисляется для образца заранее

- Хранит длины максимальных совпадающих префиксов и суффиксов

- Для "aba": [0,0,1] (a:0, ab:0, aba:1)

2. Процесс поиска:

- Построчно сравнивает текст с образцом

- При несовпадении использует LPS для "умного" сдвига

- Пропускает уже проверенные символы

**Преимущества алгоритма Кнута-Морриса-Пратта (KMP):**

1. **Эффективность:**
   * Алгоритм действительно работает за линейное время O(n + m), где n — длина текста, а m — длина образца.
   * Это достигается за счёт препроцессинга (построения LPS-массива), который позволяет избегать повторных сравнений символов.
2. **Экономия памяти:**
   * KMP требует O(m) дополнительной памяти для хранения LPS-массива (Longest Prefix Suffix).
   * Это делает его более эффективным по памяти по сравнению с некоторыми другими алгоритмами поиска подстрок (например, алгоритмом Ахо-Корасик, который требует больше памяти).
3. **Оптимизация за счёт LPS:**
   * Алгоритм избегает избыточных сравнений, используя информацию о совпадениях и несовпадениях из предыдущих шагов.
   * Это позволяет ему пропускать заведомо бесполезные проверки, ускоряя поиск.

**Анализ временной сложности KMP**

1. **Построение LPS-массива (префикс-функции)**
   * Сложность: O(m)
   * Каждый символ образца обрабатывается константное число раз благодаря эффективному использованию предыдущих вычислений.
   * Даже при рекурсивных откатах (когда len > 0 и происходит пересчёт len = lps[len - 1]), каждый символ участвует в сравнениях не более двух раз.
2. **Поиск вхождений образца в тексте**
   * Сложность: O(n)
   * Каждый символ текста обрабатывается один раз.
   * Благодаря LPS-массиву, сдвиг образца при несовпадении выполняется за O(1) (без возврата в тексте).

**Итоговая временная сложность:**

O(m+n)

(Линейная сложность, что является ключевым преимуществом KMP перед наивным алгоритмом с O(n·m).

**Анализ пространственной сложности KMP**

1. **LPS-массив**
   * **Память: O(m)**
   * Требуется массив длиной m (длина образца) для хранения значений префикс-функции.
2. **Хранение результатов (позиций вхождений)**
   * **Память: O(k)**, где k — количество вхождений.
   * В **худшем случае** (например, текст a^n и образец a), k = O(n).

**Итоговая пространственная сложность:**

O(m+k)O(m+k)

(Если не хранить все вхождения, а просто выводить их, то дополнительная память — только O(m))

**Вывод**

Ваш анализ корректен, но важно подчеркнуть:

1. KMP гарантирует **линейное время** даже на сложных примерах.
2. Память зависит от необходимости сохранять результаты (если выводить позиции на лету — O(m)).
3. LPS-массив — ключевая оптимизация, позволяющая избежать возвратов в тексте.

**Описание функций и структур данных**

Функции

Функция computePrefixFunction:

Сигнатура: vector<int> computePrefixFunction(const string& pattern);

Назначение: Вычисляет префикс-функцию (LPS-массив) для строки-образца.

Аргументы:

pattern — строка-образец, для которой вычисляется LPS-массив. Это строка, которую мы хотим найти в тексте.

Возвращаемое значение: Вектор pi, содержащий значения префикс-функции для каждого символа строки-образца. Каждый элемент pi[i] — это длина наибольшего собственного префикса строки pattern[0..i], который также является суффиксом этой подстроки.

Функция kmpSearch:

Сигнатура: vector<int> KMPSearch(const string& text, const string& pattern);

Назначение: ищет все вхождения образца в тексте с использованием алгоритма Кнута-Морриса-Пратта (KMP).

Аргументы:

pattern — строка-образец, которую нужно найти.

text — строка-текст, в которой выполняется поиск.

Возвращаемое значение: Вектор с позициями вхождений образца в тексте. Если вхождений нет, возвращается пустой вектор.

Функция main:

Сигнатура: int main();

Назначение: основная функция, считывающая входные данные и выводящая вхождения.

Возвращаемое значение: Нет (результат выводится в консоль).

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | ab  abab | 0,2 | Верный вывод |
|  | adasdqwdasd  sdqwdasdada | 8 | Верный вывод |
|  | aba  ababababa | 0,2,4,6 | Верный вывод |
|  | defabc  abcdef | 3 | Верный вывод |

# Приложение А Исходный код программы

Название файла: main.cpp

#include <iostream>

#include <vector>

#include <string>

using namespace std;

//Функция, высчитывающая префикс-функцию подстроки

vector<int> computePrefixFunction(const string& pattern) {

    int m = pattern.size();

    vector<int> pi(m, 0);//вектор для хранения значение префикс-функции

    for (int i = 1, j = 0; i < m; ++i) {

        while (j > 0 && pattern[i] != pattern[j]) {

            j = pi[j - 1];

        }

        if (pattern[i] == pattern[j]) {

            j++;

        }

        pi[i] = j;

    }

    return pi;

}

//функция нахождения вхождения подстроки в строки методом КМП

vector<int> KMPSearch(const string& text, const string& pattern) {

    vector<int> result;

    int n = text.size();

    int m = pattern.size();

    if (m == 0 || n < m) {//если длина шаблона 0 или шаблон длинее такста, возвращается пустой вектор

        return result;

    }

    vector<int> pi = computePrefixFunction(pattern);//значения префикс-функции для подстроки

    for (int i = 0, j = 0; i < n; ++i) {

        while (j > 0 && text[i] != pattern[j]) {

            j = pi[j - 1];

        }

        if (text[i] == pattern[j]) {

            j++;

        }

        if (j == m) {

            result.push\_back(i - m + 1);

            j = pi[j - 1];

        }

    }

    return result;

}

int main() {

    bool opt = true; //переменная опции, для определения рабоыт программы

    if (opt){

        string P, T;//поиск вхождений подстроки в строку

        //Ввод подстроки и строки

        getline(cin, P);

        getline(cin, T);

        //запуск поиска

        vector<int> occurrences = KMPSearch(T, P);

        //вывод результата

        if (occurrences.empty()) {

            cout << -1 << endl;

        } else {

            for (size\_t i = 0; i < occurrences.size(); ++i) {

                if (i != 0) {

                    cout << ",";

                }

                cout << occurrences[i];

            }

            cout << endl;

        }

    }

    else{

    string P, T;//поиск смещения между двумя равными строками

    //Ввод строки и подстроки

    getline(cin, T);

    getline(cin, P);

    if(T.length() != P.length()){//если их длины не равны, программа завершается

        cout<<-1;

        return 0;

    }

    T.append(T);

    //запусик поиска

    vector<int> occurrences = KMPSearch(T, P);

    //вывод результата

    if (occurrences.empty()) {

        cout << -1 << endl;

    } else {

        cout<<occurrences[0];

    }

    }

    return 0;

}