**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.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Синицкая Д.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

## Цель работы.

Реализация по алгоритму Кнута-Морриса-Пратта (КМП) эффективного алгоритма поиска подстроки в строке.

## Задание.

4.1 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

Sample Input:

ab

abab

Sample Output:

0,2

4.2 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 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

## Выполнение работы.

Основные шаги алгоритма:

1. Проверка на корректность ввода:

Функция *isLatin*() проверяет, что строки содержат только латинские буквы (заглавные или строчные).

2. Построение префикс-функции (функция *computePrefix*):

Для шаблона P создается массив *prefix*, где *prefix[i]* — длина наибольшего суффикса, совпадающего с префиксом подстроки P[0..i].

3. Основной поиск (функция *KMPSearch*):

С помощью массива *prefix* осуществляется поиск подстроки P в T: При совпадении символов указатели i и j сдвигаются. При полном совпадении шаблона фиксируется позиция вхождения. При несовпадении происходит «откат» по массиву *prefix*.

4. Вывод результата:

Если найдено хотя бы одно вхождение, программа выводит все позиции начала вхождений шаблона. В противном случае — -1.

Описание функций:

1. *bool isLatin(const string &text)* — функция проверяет, состоит ли строка только из латинских букв.

Параметры:

*const string &text* — ссылка на строку, которую нужно проверить.

Возвращаемое значение:

*true*, если строка состоит только из латиницы.

*false* — если содержит другие символы (включая пробелы, цифры, символы и т.д.).

2. *vector<int> computePrefix(const string &P) —* функция вычисляет префикс-функцию (массив *prefix*), которая определяет длину наибольшего правильного префикса, совпадающего с суффиксом для каждой позиции строки P.

Параметры:

*const string &P* — строка-образец, для которой строится префикс-функция.

Возвращаемое значение:

*vector<int>* — массив *prefix*, где *prefix[i]* показывает длину совпадения префикса и суффикса подстроки P[0..i].

3. *vector<int> KMPSearch(const string &P, const string &T)* — функция находит все вхождения строки P в строку T с помощью алгоритма Кнута-Морриса-Пратта.

Параметры:

*const string &P* — строка-образец.

*const string &T* — строка-текст.

Возвращаемое значение:

*vector<int>* — массив позиций начала каждого вхождения P в T.

4. *int main()* - основная точка входа.

Выполняет:

Считывание строк P и T с клавиатуры. Проверку на латиницу. Вызов поиска. Вывод результата.

Переменные:

*string P, T* — вводимые строки.

*vector<int> positions* — массив индексов вхождений P в T.

Вывод:

Позиции вхождений через запятую. Или -1, если вхождений нет.

Оценка сложности алгоритма:

Временная сложность:

Построение префикс-функции:

O(m) — где m — длина шаблона P.

Каждый символ обрабатывается максимум дважды (вперёд и назад).

Поиск подстроки:

O(n) — где n — длина текста T. Алгоритм использует заранее вычисленные префиксы, поэтому избегает лишних сравнений.

Общая временная сложность: O(n + m)

Сложность по памяти.

1. Массив префиксов (*prefix*):

Для хранения массива префикс-функции нам нужно O(m) памяти, где m — длина строки P.

2. Массив для хранения позиций вхождений (*result*):

Массив для хранения всех позиций вхождений подстроки P в строку T. Максимальный размер этого массива — O(k), где k — количество вхождений.

Общая сложность: O(m + k).

## Тестирование.

Результаты тестирования представлены в таблицах 1-2.

Таблица 1 – Результаты тестирования задания 4.1

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | ab  abab | 0,2 | ab встречается дважды — на позициях 0 и 2. Результат соответствует ожидаемому. |
| 2. | abc  abc | 0 | Строка полностью совпадает. Результат соответствует ожидаемому. |
| 3. | xyz  qwerty | -1 | Шаблон не найден. Результат соответствует ожидаемому. |
| 4. | a  aaaaa | 0,1,2,3,4 | По одному совпадению на каждую позицию.  Результат соответствует ожидаемому. |
| 5. | aaa  aaaaaa | 0,1,2,3 | Перекрывающиеся вхождения. Результат соответствует ожидаемому. |
| 6. | abcdef  abc | -1 | P больше, чем T. Результат соответствует ожидаемому. |
| 7. | xyz  abcxyz | 3 | Совпадение в конце. Результат соответствует ожидаемому. |
| 8. | bn1  sss | Ошибка: строки должны содержать только латинские буквы. | Нелатинские символы в P. Результат соответствует ожидаемому. |

Таблица 2 – Результаты тестирования задания 4.2

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | defabc  abcdef | 3 | Нормальный сдвиг. Результат соответствует ожидаемому. |
| 2. | abcdef  abcdef | 0 | Без сдвига. Результат соответствует ожидаемому. |
| 3. | abcdeg  abcdef | -1 | Нет сдвига.  Результат соответствует ожидаемому. |
| 4. | abc  abcdef | -1 | A короче, чем B. Результат соответствует ожидаемому. |
| 5. | abcdef  abc | -1 | B короче, чем A. Результат соответствует ожидаемому. |
| 6. | aaaaa  aaaaa | 0 | Все символы одинаковые. Все циклические сдвиги равны. Результат соответствует ожидаемому. |
| 7. | a  a | 0 | Один символ, одинаковый. Результат соответствует ожидаемому. |
| 8. | a  b | -1 | Один символ, разные.  Результат соответствует ожидаемому. |

## Выводы.

В лабораторной работе были проведены анализ и реализация алгоритма поиска подстроки Кнута–Морриса–Пратта (КМП). Алгоритм позволяет эффективно находить вхождения шаблона в тексте, используя предварительное построение таблицы префикс-функций.

# Приложение А Исходный код программы

Название файла: lr\_4\_1.cpp

#include <iostream>

#include <vector>

#include <string>

#include <cctype>

using namespace std;

// функция проверки состоит ли строка только из латиницы

bool isLatin(const string &text) {

for (char ch : text) {

if (!isalpha(ch) || (ch < 'a' && ch > 'Z')) {

return false;

}

}

return true;

}

// функция вычисления префиксного массива

vector<int> computePrefix(const string &P) {

int m = P.size(); // размер строки P

vector<int> prefix(m, 0); // инициализация массива prefix нулями

int len = 0; // длина предыдущего наибольшего префикса

int i = 1; // текущая позиция в строке P

// проход по строке P для заполнения массива prefix

while (i < m) {

if (P[i] == P[len]) { // символы совпадают

prefix[i++] = ++len; // увеличение длины и сохранение в prefix

} else {

// символы не совпадают

if (len != 0) {

len = prefix[len - 1]; // переход к предыдущему префиксу

} else {

prefix[i++] = 0; // нет префикса, установка 0

}

}

}

return prefix;

}

// функция поиска всех вхождений строки P в строке T

vector<int> KMPSearch(const string &P, const string &T) {

int m = P.size(), n = T.size(); // размеры строк P и T

vector<int> prefix = computePrefix(P); // вычисление массива prefix

vector<int> result; // массив для хранения позиций вхождений

int i = 0, j = 0; // i - индекс в T, j - индекс в P

while (i < n) {

if (P[j] == T[i]) { // символы совпадают

i++, j++; // переход к следующему символу в обеих строках

}

if (j == m) { // все символы P совпали с T

result.push\_back(i - j); // сохранение позиции начала вхождения

j = prefix[j - 1]; // обновление j согласно prefix

} else if (i < n && P[j] != T[i]) { // символы не совпадают

if (j != 0) {

j = prefix[j - 1]; // обновление j согласно prefix, если он не нулевой

} else {

i++; // j == 0, переходим к следующему символу в T

}

}

}

return result;

}

int main() {

string P, T; // строки для поиска и текст

cin >> P >> T;

// проверка обе строки состоят только из латиницы

if (!isLatin(P) || !isLatin(T)) {

cout << "Ошибка: строки должны содержать только латинские буквы." << endl;

return 1;

}

vector<int> positions = KMPSearch(P, T); // поиск вхождения P в T

// проверка найдены ли позиции

if (positions.empty()) {

cout << "-1";

} else {

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

if (i > 0) cout << ",";

cout << positions[i];

}

}

cout << endl;

return 0;

}

Название файла: lr\_4\_2.cpp

#include <iostream>

#include <string>

#include <vector>

using namespace std;

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

int m = pattern.size();

vector<int> lps(m, 0);

int j = 0;

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

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

j = lps[j - 1];

}

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

++j;

lps[i] = j;

}

}

return lps;

}

int KMPSearch(const string &text, const string &pattern) {

vector<int> lps = computeKMPTable(pattern);

int n = text.size(), m = pattern.size();

int j = 0;

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

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

j = lps[j - 1];

}

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

++j;

}

if (j == m) {

return i - m + 1;

}

}

return -1;

}

int main() {

string A, B;

cin >> A >> B;

if (A.size() != B.size()) {

cout << -1 << endl;

return 0;

}

string doubleA = A + A;

int index = KMPSearch(doubleA, B);

cout << index << endl;

return 0;

}