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)

Студентка гр. 3388 _____ Титкова С.Д.

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|≤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

Задание 2:

Заданы две строки A (|A|≤5000000) и B (|B|≤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 (KMP) разработан для быстрого поиска всех мест в тексте T, где встречается заданная подстрока P. Он повышает эффективность поиска, минимизируя лишние сравнения символов благодаря использованию префикс-функции, которая позволяет пропускать уже обработанные участки текста при несовпадении.

Шаги алгоритма

1. **Проверка размеров**: сначала проверяются длины P и T. Если P пустая или её длина превышает длину T, возвращается пустой список.
2. **Вычисление префикс-функции π для P**:
   - Для каждого символа P[i] (где i от 1 до m−1, а m — длина P) вычисляется значение π[i].
   - Если символы P[k] и P[i] не совпадают, значение k уменьшается с использованием π[k−1], пока не будет найдено совпадение или k не станет равным 0.
   - Если P[k] совпадает с P[i], значение k увеличивается.
   - Итоговое значение π[i] равно k.
3. **Поиск вхождений P в T**:
   - Проходим по тексту T с индексом i, отслеживая количество совпавших символов q (из P).
   - Если P[q] не равно T[i], q уменьшается с использованием π[q−1].
   - Если P[q] равно T[i], q увеличивается.
   - Когда q достигает m (длина P), это означает полное совпадение, и позиция i−m+1 добавляется в список результатов. Затем q уменьшается с использованием π[q−1] для продолжения поиска.

Описание функций и структур:

- *vector<int> compute_prefix_function(const string& P)* – функция, которая вычисляет префикс-функцию для шаблона P.
- *vector<int> kmp_search(const string& T, const string& P)* – функция, которая ищет все вхождения P в T с использованием KMP

Оценка сложности алгоритма:

Временная сложность

Вычисление префикс-функции:

- Проход по P длиной m: O(m).
- Итог: O(m).

Поиск:

- Проход по T длиной n: O(n).
- Внутренний цикл while уменьшает q по pi, 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)

Пространственная сложность

Префикс-функция:

- pi: 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
ACGT 2 ACGTACGT CGTA	
AAAAA 2 A AA	1 1 1 2 2 1 2 2 3 1 3 2 4 1 4 2 5 1
ACGTACGT 3 AC CG GT	1 1 2 2 3 3 5 1 6 2 7 3
ACGTACGT 3 A AC ACG	1 1 1 2 1 3 5 1 5 2 5 3

Вывод

В ходе лабораторной работы были написаны программы с использованием алгоритма Ахо-Корасика. Также дополнительно было сделано: подсчёт вершин и определение пересечений.

Исходный код программы см. в ПРИЛОЖЕНИИ А.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ

KMP_1.cpp

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

using namespace std;
bool DEBUG = true;

vector<int> compute_prefix_function(const string& P) {
    int m = P.length();
    vector<int> pi(m, 0);
    int k = 0;

    for (int i = 1; i < m; ++i) {
        while (k > 0 && P[k] != P[i]) {
            k = pi[k - 1];
        }
        if (P[k] == P[i]) {
            k++;
        }
        pi[i] = k;

        if (DEBUG) {
            cout << "pi[" << i << "] = " << pi[i] << endl;
        }
    }

    return pi;
}

vector<int> kmp_search(const string& T, const string& P) {
    int n = T.length();
    int m = P.length();

    if (m == 0 || m > n) {
        return {};
    }

    vector<int> pi = compute_prefix_function(P);
    vector<int> occurrences;
    int q = 0;

    for (int i = 0; i < n; ++i) {
        while (q > 0 && P[q] != T[i]) {
            q = pi[q - 1];
        }
        if (P[q] == T[i]) {
            q++;
        }
        if (q == m) {
            int start_index = i - m + 1;
            occurrences.push_back(start_index);
            q = pi[q - 1];
```

```
            if (DEBUG) {
                cout << "Found occurrence at index: " << start_index
<< endl;
            }
        }
    }

    return occurrences;
}

int main() {
    string P, T;
    cin >> P >> T;

    if (DEBUG) {
        cout << "Pattern: " << P << endl;
        cout << "Text: " << T << endl;
    }

    vector<int> result = kmp_search(T, P);

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

    return 0;
}
```