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) предназначен для эффективного поиска всех вхождений заданного шаблона P в текст T. Он оптимизирует 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 π для P.

Для каждого символа P[i] (от 1 до m−1) определяется π[i]. Если P[k]≠P[i], k уменьшается по π[k−1] до совпадения или 0. Если P[k]=P[i], k увеличивается. π[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 π[q−1]. При P[q]=T[i]: q увеличивается. Если q=m (полное совпадение), позиция i−m+1 добавляется в результат, q сдвигается по π[q−1].

.

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

Описание Кнута-Морриса-Пратта:

Данный алгоритм решает задачу определения, является ли строка A циклическим сдвигом строки B, и возвращает индекс начала B в A (0-based) или −1, если это не так. Он использует алгоритм Кнута-Морриса-Пратта (KMP) для поиска первого вхождения B в удвоенной строке AA=A+A, а также включает предварительные проверки для оптимизации.

Шаги алгоритма

Считываются строки A и B. Проверяется |A|=|B|, иначе возвращается −1. Далее производится проверка состава символов. Копии A и B сортируются. Если sortedA≠sortedB , возвращается −1.

Далее формируется AA=A+A, длина 2|A|. А затем производится поиск при помощи алгоритма Кнута-Морриса-Пратта. Вычисляется префикс-функция π для B. Выполняется поиск B в AA, возвращается первая позиция k или −1.

Впоследствии производится проверка результата. Если 0≤k<|A|, возвращается k, иначе −1.

Описание функций и структур:

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

Оценка сложности алгоритма:

Временная сложность:

Чтение и проверка:

- Чтение A и B: O(m + n) для ввода строк.
- Проверка |A|=|B|: O(1).
- Сортировка sorted_A sorted_B : O(nlogn)
- Сравнение sorted_A и s sorted_B : O(n).

Удвоение строки (AA=A+A):

- O(n) для конкатенации.

Функция compute_prefix_function:

- Проход по P длиной m: O(m).
- Внутренний цикл while ограничен m уменьшениями k , итого O(m).

Функция kmp_search:

- compute_prefix_function: O(m) O(m) O(m).
- Проход по T (|AA|=2n): O(2n) = O(n).
- Внутренний цикл while ограничен O(n) уменьшениями q.

Итог: O(nlogn) из-за сортировки, доминирующей над O(n+m)от KMP. Без сортировки: O(n+m).

Пространственная сложность

compute_prefix_function:

- pi: O(m) — вектор длиной m.
- k: O(1).

kmpsearch kmp_search kmpsearch:

- pi: O(m) — вектор длиной m.
- occurrences: O(1) — только одно вхождение.

main:

- A, B O(n) каждый.
- sorted_A ,sorted_B: O(n) каждый.
- AA: O(2n).

Итого: O(n).

Тестирование

Таблица 2. Тестирование.

Входные данные	Выходные данные
AAAAA A*A *	1 2 3
ACGTACGT *CG* *	1 5
ACTANCA A$$A$ $	1
NTAG T*G *	2

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;
}
```

KMP_2.cpp

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

using namespace std;

const 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;
}

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

    if (m == 0 || m > n) {
        if (DEBUG) {
            cout << "Substring is empty or longer than the text.
Returning -1." << endl;
        }
        return -1;
    }

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

    for (int i = 0; i < n; ++i) {
        if (DEBUG) {
            cout << "Checking T[" << i << "] = " << T[i] << " against
P[" << q << "] = " << P[q] << endl;
        }

        while (q > 0 && P[q] != T[i]) {
            q = pi[q - 1];
            if (DEBUG) {
                cout << "Mismatch. New q = " << q << endl;
            }
        }
        if (P[q] == T[i]) {
            q++;
            if (DEBUG) {
                cout << "Match. New q = " << q << endl;
            }
        }
        if (q == m) {
            if (DEBUG) {
                cout << "Full match found at index: " << i - m + 1 <<
endl;
            }
            return i - m + 1;
        }
    }

    if (DEBUG) {
        cout << "No match found. Returning -1." << endl;
    }
    return -1;
```

```
}

int main() {
    string A, B;
    cin >> A >> B;

    if (DEBUG) {
        cout << "Input strings: A = " << A << ", B = " << B << endl;
    }

    if (A.length() != B.length()) {
        if (DEBUG) {
            cout << "Lengths of A and B are different. Returning -1."
<< endl;
        }
        cout << -1 << endl;
        return 0;
    }

    string sorted_A = A;
    string sorted_B = B;
    sort(sorted_A.begin(), sorted_A.end());
    sort(sorted_B.begin(), sorted_B.end());

    if (sorted_A != sorted_B) {
        if (DEBUG) {
            cout << "A and B contain different characters. Returning -
1." << endl;
        }
        cout << -1 << endl;
        return 0;
    }

    string AA = A + A;
    if (DEBUG) {
        cout << "Constructed AA: " << AA << endl;
    }
    int index = kmp_search(AA, B);

    if (index >= 0 && index < A.length()) {
        if (DEBUG) {
            cout << "Valid shift found at index: " << index << endl;
        }
        cout << index << endl;
    } else {
        if (DEBUG) {
            cout << "No valid shift found. Returning -1." << endl;
        }
        cout << -1 << endl;
    }

    return 0;
}
```