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: АЛГОРИТМ КМП

Студент гр. 8383 _____ Ларин А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить принцип работы алгоритма КМП для нахождения подстроки с втроке. Решить с его помощью задачи

Основные теоретические положения.

Алгоритм Кнута — Морриса — Пратта (КМП-алгоритм) — эффективный алгоритм, осуществляющий поиск подстроки в строке. Время работы алгоритма линейно зависит от объёма входных данных, то есть разработать асимптотически более эффективный алгоритм невозможно.

Рассмотрим сравнение строк на позиции i , где образец S[0,m-1] сопоставляется с частью текста T[i,i+m-1]. Предположим, что первое несовпадение произошло между T[i+j] и S[j], где 1<j<m. Тогда T[i, i+j-1] = S[0, j-1] = P и a = T[i+j] ≠ S[j] = b.

При сдвиге вполне можно ожидать, что префикс (начальные символы) образца S сойдется с каким-нибудь суффиксом (конечные символы) текста P. Длина наиболее длинного префикса, являющегося одновременно суффиксом, есть значение префикс-функции от строки S для индекса j.

Это приводит нас к следующему алгоритму:

- Считать значения префикс-функции π[i] будем по очереди: от i=1 к i=n-1 (значение π[0] просто присвоим равным нулю).
- Для подсчёта текущего значения π [i] мы заводим переменную j, обозначающую длину текущего рассматриваемого образца. Изначально j = π[i-1].
- Тестируем образец длины j, для чего сравниваем символы s[j] и s[i]. Если они совпадают — то полагаем π[i] = j+1 и переходим к следующему индексу i+1. Если же символы отличаются, то уменьшаем длину j, полагая её равной π[j-1], и повторяем этот шаг алгоритма с начала.

- Если мы дошли до длины j=0 и так и не нашли совпадения, то останавливаем процесс перебора образцов и полагаем π[i] = 0 и переходим к следующему индексу i+1.

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, разделенных запятой, если P не входит в T, то вывести −1
Sample Input:
ab
abab
Sample Output:
0,2

Вар. 2. Оптимизация по памяти: программа должна требовать O(m) памяти, где m - длина образца. Это возможно, если не учитывать память, в которой хранится строка поиска.

Реализация

Выделяется буфер под значения перфикс-функции длины, равной длине шаблона. Далее в цикле высчитывается значение префикс функции для каждой позиции в шаблоне. Значение заносятся в буфер. Затем в цикле высчитывается значение префикс-функции для каждой позиции уже в тексте, где постфикс считается от текущей позиции, а префикс от начала шаблона. Значение сравнивается с длиной шаблона, и при совпадении индекс начала постфикса в тексте заносится в отдельный массив — это найденное вхождение шаблона в тексте.

Массив индексов вхождений возвращается.

Сложность по памяти —O(m), где m - длина шаблона.

Сложность по времени — линейная от суммы длинн шаблона и текста O(m+n), где m - длина шаблона, n – длина текста, т.к. цикл проходит сначала шаблон, потом текст.

Описание функций и структур данных

Для хранения буфера используются структура данных vector из STL

`std::vector<size_t> prefix;`

`vector<size_t> KMP(const string& sample, const string& text)` - функция, реализующая основную логику алгоритма KMP. Принимает шаблон для поиска `sample` и текст для поиска `text`. Возвращает вектор индексов вхождений.

Тесты.

1.
aab
aabaabaaaabaabaaab
0,3,8,11,16
2.
ab
abab
0,2

Выводы.

В результате работы была написана полностью рабочая программа решающая поставленную задачу при использовании изученных теоретических материалов. Программа было протестирована, результаты тестов удовлетворительны.

ПРИЛОЖЕНИЕ А(ЛИСТИНГ ПРОГРАММЫ)

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

using namespace std;

vector<size_t> KMP(const string& sample, const string& text){
    vector<size_t>& prefix = *new vector<size_t>(sample.length());//O(m) memory usage
    vector<size_t> entries;//Entries array

    size_t last_prefix = prefix[0] = 0;//Init prefix value for zero position
    for (size_t i=1; i<sample.length(); i++) {//Hence going from secons position
        while (last_prefix > 0 && sample[last_prefix] != sample[i])
            last_prefix = prefix[last_prefix-1];

        if (sample[last_prefix] == sample[i])
            last_prefix++;

        prefix[i] = last_prefix;
    }

    last_prefix = 0;
    for (size_t i=0; i<text.length(); i++) {
        while (last_prefix > 0 && sample[last_prefix] != text[i])
            last_prefix = prefix[last_prefix-1];

        if (sample[last_prefix] == text[i])
            last_prefix++;

        if (last_prefix == sample.length()) {
            entries.push_back(i + 1 - sample.length());
        }
    }
    delete(&prefix);
    return entries;
}

int cycle(string A, string B){
    vector<size_t>& prefix = *new vector<size_t>(A.length());
    size_t max_prefix = 0;
    size_t last_prefix = prefix[0] = 0;
```

```
    FOR (SIZE_T I=1; I<A.LENGTH(); I++) {
        WHILE (LAST_PREFIX > 0 && A[LAST_PREFIX] != A[I])
            LAST_PREFIX = PREFIX[LAST_PREFIX-1];

        IF (A[LAST_PREFIX] == A[I])
            LAST_PREFIX++;

        PREFIX[I] = LAST_PREFIX;
    }

    LAST_PREFIX = 0;
    FOR (SIZE_T I=0; I<B.LENGTH(); I++) {
        WHILE (LAST_PREFIX > 0 && A[LAST_PREFIX] != B[I])
            LAST_PREFIX = PREFIX[LAST_PREFIX-1];

        IF (A[LAST_PREFIX] == B[I])
            LAST_PREFIX++;

        IF (LAST_PREFIX > MAX_PREFIX) {
            MAX_PREFIX = LAST_PREFIX;
        }
    }

    FOR(INT I = 0;I<A.LENGTH()-MAX_PREFIX;I++){
        IF(A[I+MAX_PREFIX]!=B[I]){
            DELETE(&PREFIX);
            RETURN -1;
        }
    }

    DELETE(&PREFIX);
    RETURN MAX_PREFIX;

}

INT MAIN() {
    STRING SAMPLE;
    STRING TEXT;
    CIN>>SAMPLE;
    CIN>>TEXT;
    COUT<<CYCLE(TEXT,SAMPLE);
    RETURN 0;
```

```
}
```