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: пусть π[j] — значение префикс-функции от строки S[0,m-1] для индекса j. Тогда после сдвига мы можем возобновить сравнения с места T[i+j] и S[π[j]] без потери возможного местонахождения образца. Можно показать, что таблица π может быть вычислена (амортизационно) за Θ[m] сравнений перед началом поиска. А поскольку строка T будет пройдена ровно один раз, суммарное время работы алгоритма будет равно Θ[m+n], где n — длина текста T.

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 — линейная от длины шаблона O(m)

Сложность по времени — линейная от суммы длинн шаблона и текста O(m+n), 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;
}
```