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 Кнута-Морриса-Пратта

Студент гр. 7383 _____ Рудоман В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Исследовать и реализовать задачу поиска вхождения подстроки в строке, используя алгоритм Кнута-Морриса-Пратта.

Формулировка задачи.

Необходимо разработать программу, которая реализует алгоритм Кнута-Морриса-Пртта и с его помощью для заданных шаблона P ($|P|\leq 15000$) и текста T ($|T|\leq 5000000$) найти 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 .

Реализация задачи.

void KMPSearch – реализация алгоритма Кнута-Морриса-Пратта.

void computeLPSArray – префикс-функция

Алгоритм вычисления префикс-функции: функция находит наибольшую длину наибольшего собственного суффикса подстроки, совпадающего с ее префиксом. Значение для первого символа всегда принимается за 0.

Алгоритм Кнута-Морриса-Пратта: когда вычислены все значения префикс-функций для шаблона, начинается поиск вхождения этого шаблона в текст: начинается сравнение символов с начала в обеих строках, если символы равны - продолжается сравнение пока не находится первый символ, не совпадающий с символом в первой строке (шаблоне) или пока не встречается конец этой строки. Если дошли до конца первой строки, значит первый индекс вхождения первой строки (шаблона) во вторую (текст) найден и вычисляется этот индекс, запоминается этот символ (сохраняется в массив find) и продолжается операция сравнения для следующего символа во второй строке и для последнего символа, лежащего по адресу, хранящемуся для данного символа в массиве префикс-функций, в первой строке.

Исследование.

В начале работы программа вычисляет значения префикс-функций для каждого символа первой строки. Примем за P длину первой строки (шаблона). После начинается сравнение каждого символа второй строки с символами первой строки. Примем за T длину второй строки. Тогда сложность алгоритма по времени будет равна O(P+T).

По памяти сложность алгоритма составляет O(P), так как вычисляются значения префикс-функций только для первой строки.

Вывод.

В ходе выполнения лабораторной работы была решена задача поиска подстроки в строке с помощью алгоритма Кнута-Морриса-Пратта на языке C++, и исследован алгоритм Кнута-Морриса-Пратта. Полученный алгоритм имеет линейную сложность как по памяти, так и по времени.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <thread>
#include <vector>

void KMPSearch(std::string &pat, std::string &txt, int* lps, int
begin, int end, std::vector<int> &positions) {
    int M = pat.size();

    // values for pattern
    int i = begin; // index for txt[]
    int j = 0; // index for pat[]
    while(i < end) {
        if(pat[j] == txt[i]) {
            j++;
            i++;
        }

        if(j == M) {
            positions.emplace_back(i-j);
            j = lps[j - 1];
        }
        // mismatch after j matches
        else if(i < end && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            (j != 0) ? j = lps[j - 1] : i = i + 1;
        }
    }
}
```

```
// Fills lps[] for given patttern pat[0..M-1]
void computeLPSArray(std::string &pat, int M, int* lps) {
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else { // (pat[i] != pat[len])
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];
            }
            else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int main() {
    std::string txt;
    std::string pat;
```

```
    std::cin >> pat >> txt;

    // create lps[] that will hold the longest prefix suffix
    int pSize = pat.size();
    int tSize = txt.size();

    if(pSize > tSize){
        std::cout << -1;
        return 0;
    }

    int* lps = new int[pSize];
    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, pSize, lps);

    unsigned concurentThreadsSupported =
std::thread::hardware_concurrency();
    int part = (tSize-pSize+1)/concurentThreadsSupported + pSize - 1 ;
    std::vector<int> positions;

    #pragma omp parallel for
    for(size_t i = 0; i < concurentThreadsSupported; i++) {
        KMPSearch(pat, txt, lps, i*part, (i+1)*part, positions);
    }

    #pragma omp parallel for
    for(size_t i = 1; i < concurentThreadsSupported; i++) {
        if(i*part+(pSize-1) > tSize)
            KMPSearch(pat, txt, lps, i*part-(pSize-1), tSize,
positions);
        else
            KMPSearch(pat, txt, lps, i*part-(pSize-1), i*part+(pSize-
1), positions);
    }
```

```
    if(!positions.size()){
        std::cout << -1;
        return 0;
    }

    sort(positions.begin(), positions.end());

    for(auto &elem : positions){
        std::cout << "Pattern matched at " << elem << " position: ";
        //colored output
        for(int i = 0; i < tSize; ++i)
           if(i == elem){
                std::cout << "\033[0;32m" << pat << "\033[0m";
                i += pSize-1;
           }
           else
                std::cout << txt[i];
        std::cout << std::endl;
    }

    return 0;
}
```