МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

По лабораторной работе № 4

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Кнута-Морриса-Пратта

Студент гр. 0303 _____ Болкунов В.О.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2022

Цель работы.

Изучение и практическое применение алгоритма поиска подстроки в строке – алгоритма Кнута-Морриса-Пратта.

Задание 1.

Реализуйте алгоритм КМП и с его помощью для заданных шаблона P ($|P| \leq 15000$) и текста T ($|T| \leq 5000000$) найдите все вхождения P в T.

Входные данные

Первая строка - P

Вторая строка – T

Выходные данные

индексы начал вхождений P в T, разделенных запятой, если P не входит в T, то вывести -1

Пример входных данных

ab

abab

Соответствующие выходные данные

0, 2

Задание 2.

Заданы две строки A ($|A| \leq 5000000$) и B ($|B| \leq 5000000$). 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.

Входные данные

Первая строка - *A*

Вторая строка – *B*

Выходные данные

Если *A* является циклическим сдвигом *B*, индекс начала строки *B* в *A*, иначе вывести -1. Если возможно несколько сдвигов вывести первый индекс.

Пример входных данных

defabc

abcdef

Соответствующие выходные данные

3

Основные теоретические положения.

Префикс-суффикс (бордер) строки **–** какое-либо количество (возможно нулевое) символов суффикса, полностью совпадающих с префиксом.

Префикс-функция от строки — массив длин наибольших (но меньше длины самой строки) префикс- суффиксов (бордеров) для каждой позиции этой строки. То есть для каждого среза строки от [0:1] до [0:n] - максимальное количество символов суффикса, совпадающее с префиксом.

Алгоритм Кнута — Морриса — Пратта — алгоритм поиска подстроки в строке. Дан текст *T* шаблон *P*, можно построить строку *P*|*T* где ‘|’ - разделитель не входящий в алфавит. Тогда при взятии префикс-функции от полученной строки на позициях массива П.Ф. равных длине |*P*| будет полное совпадение текущего суффикса с префиксом – т.е. со строкой *P*, что и будет вхождением подстроки *P* в строку *T*. Очевидно значений больше, чем |*P*| быть не может, так как символ-разделитель не встречается более нигде.

Выполнение работы

Алгоритм:

1. Префикс функция:

Сначала создаётся массив заполненный нулями, который и будет значением П.Ф. С первого символа начинается цикл вычисления остальных значений максимальных **П**рефикс-**С**уффиксов (очевидно для нулевого символа оно будет 0). Далее стандартный алгоритм: берётся значение максимального п.с. на предыдущем шаге (то есть сколько символов уже совпало), и далее сравниваются очередные символы префикса и суффикса, и если они равны, то текущее значение максимального п.с. будет на единицу больше предыдущего. Иначе же итеративно (а именно – значение п.с. на позиции равной значению очередного максимального п.с. начиная с предыдущего п.с.) ищется ближайшее предыдущее значение максимального п.с., для которого будет либо совпадение следующего символа префикса с текущим символом суффикса, либо пока значение п.с. не дойдёт до нуля. В итоге полученный таким путём следующий символ префикса сравнивается с текущим символом суффикса, и если совпадение есть, то снова значение максимального п.с. на данном шаге будет на единицу больше предыдущего найденного (возможно и нулевого). Иначе же значение п.с. остаётся нулевым, так как массив был заполнен нулями.

2. Алгоритм Кнута-Морриса-Пратта:

В решении используется классический алгоритм, при котором строится некая строка P|T (где P – шаблон, а T – текст), разделённые символом, не входящим в алфавит. От полученной строки вычисляется значение реализованной префикс-функции. Далее в цикле производится поиск максимальных длин префикс-суффиксов равных длине первой строки – что и будет являться вхождением строки P в T. Очередное найденное вхождение добавляется в массив решений.

3. Циклический сдвиг строки A от B

Сначала проверяется предполагаемый сдвиг вправо. При решении данной задачи используется аналогичный подход с использованием префикс-функции. Вычисляется префикс-функция от строки B|A. Но теперь нас интересует значение только на последнем символе, оно будет равно максимальной длине суффикса A, совпавшего с префиксом B. Имея это значение достаточно сравнить оставшиеся части строк: префикс A с суффиксом B.

Если сдвиг вправо не найден, выполняется проверка сдвига влево, аналогичным способом, просто с заменой строк местами. В таком случае ответом будет разница между длиной строк и полученным сдвигом.

Оценка сложности по времени

- Вычисление префикс-функции:

На каждом шаге значение максимального префикс-суффикса увеличивается не более чем на 1. Т.е. максимальное его значение будет n-1 (где n – длина строки). Во внутреннем цикле поиска совпадений это значение только уменьшается, а значит внутренний цикл отработает не более чем n итераций. Внешний цикл – всегда n операций.

Итого получаем линейную сложность O(2n) = **O(n)**

- КМП:

Вычисление П.Ф.: O(|P| + 1 + |T|) = O(|P| + |T|) (где P и T – переданные строки шаблона и текста)

Цикл поиска вхождений: O(|T|)

Итоговая асимптотическая сложность **O(|P| + |T|)**

- Поиск циклического сдвига

Вычисление П.Ф.: O(|A| + 1 + |B|) = O(|A| + |B|) = O(|A| + |B|) (где A и B – переданные строки)

Сравнение префикса A с суффиксом B: O(|A|)

Итоговая сложность: **O(|A| + |B|)**

Оценка сложности по памяти

- Вычисление префикс-функции:

Выделение памяти под массив решения: **O(n)**

- КМП:

Создание новой строки O(|P| + 1 + |T|) = O(|P| + |T|) (где P и T – переданные строки шаблона и текста)

Вызов префикс-функции от полученной строки: O(|P| + |T|)

Массив вхождений (в худшем случае): O(|T|)

Итоговая сложность: O(2|P| + 3|T|) = **O(|P| + |T|)**

- Поиск циклического сдвига

Создание новой строки O(|A| + 1 + |B|) = O(|A| + |B|) (где A и B – переданные строки)

Вызов префикс-функции от полученной строки: O(|A| + |B|)

Итоговая сложность: O(2|P| + 2|T|) = **O(|P| + |T|)**

Реализованные структуры данных и функции

Реализованный программный код см. в приложении А.

- Префикс функция:

```
std::vector<int> prefFunc(const std::string &str)
```

Принимаемые значения:

str – строка от которой вычисляется П.Ф.

Возвращаемое значение: массив максимальных бордеров для каждой позиции в строке.

- Алгоритм КМП

std::vector<int> kmp(const std::string &s1, const std::string &s2, char delimiter = '|')

Принимаемые значения:

s1 – шаблон поиска

s2 – строка в которой ищется шаблон

delimiter – разделитель строки для применения к ней префикс-функции (по умолчанию ‘|’)

Возвращаемое значение: массив индексов вхождений строки s1 в s2

- Определение циклического сдвига

  int cycleShift(const std::string &s1, const std::string &s2, char delimiter = '|')

  Принимаемые значения:

  s1 – строка предполагаемая циклическим сдвигом s2

  s2 – искомая строка

  delimiter – разделитель строки для применения к ней префикс-функции (по умолчанию ‘|’)

  Возвращаемое значение: индекс в s1 с которого начинается строка s2, (если s1 не циклический сдвиг то -1).

- Для отладочных логов используется класс Logger, который содержит перечисление режимов MODE (*DEBUG* и, *RELEASE*), статические поля *mode* и *tab*, соответственно режим и отступ лога, и следующие статические методы:

  static void setMode(MODE m);

  - установка режима m из перечисления режимов

  static MODE getMode();

  - возвращение текущего режима

  static void log(std::string str, bool ignoreTab = false);

- вывод строки лога str (при включенном флаге ignoreTab, отступы ставиться не будут)

static void incTab();

- увеличение отступа лога

static void decTab();

- уменьшение отступа

Для оптимизации были написаны следующие макросы, которые сначала проверяют режим, и если тот равен отладочному, то вызывают уже соответствующие методы у логгера. Таким образом строки лишний раз не копируются и не вызываются методы.

```
#define _log(...)
#define _incTab()
#define _decTab()
#define _setMode(...)
```

Тестирование

Таблица 1 – Результаты тестирования

Алгоритм КМП			
№ п/п	Входные данные	Выходные данные	Комментарии
1.	ab abab	0,2	
2.	abc wee-wee	-1	
3.	dolor Lorem_ipsum_dolor_sit_amet	12	
4.	AC GTCACGTAAACACAGTT	3, 9, 11	
Циклический сдвиг			
1.	defabc abcdef	3	
2.	ronijab jabroni	4	

3.	wecanas aswecan	5	
4.	jarnanhalast halastja	-1	

Выводы:

В ходе выполнения работы был исследован строковый алгоритм Кнута-Морриса-Пратта и подходы для его реализации. Реализован сам алгоритм поиска подстроки в строке, и алгоритм определения циклического сдвига одной строки от другой.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл Logger.h

```
#ifndef LOGGER_H
#define LOGGER_H

#include <iostream>
#include <cstdarg>

class Logger {
public:
    // Режимы логов (в release - не выводятся)
    enum MODE {
        DEBUG,
        RELEASE
    };

private:
    static MODE mode;
    // Отступ
    static int tab;

public:
    // Установка режима
    static void setMode(MODE m);
    // Получение режима
    static MODE getMode();

    // Вывод лога (ignoreTab - не печатать отступы)
    static void log(std::string str, bool ignoreTab = false);

    // Увеличить/Уменьшить отступ
    static void incTab();
    static void decTab();
};

// Макросы для оптимизации (чтобы строки лишний раз не складывались и не вызыва-
лась функция log)
#define _log(...) if (Logger::getMode() == Logger::DEBUG)
Logger::log(__VA_ARGS__);
#define _incTab() if (Logger::getMode() == Logger::DEBUG) Logger::incTab();
#define _decTab() if (Logger::getMode() == Logger::DEBUG) Logger::decTab();
#define _setMode(...) Logger::setMode(__VA_ARGS__);


#endif //LOGGER_H
```

Файл Logger.cpp

```
#include "Logger.h"

Logger::MODE Logger::mode = RELEASE;
int Logger::tab = 0;

void Logger::setMode(Logger::MODE m) {
    mode = m;
}
```

```
void Logger::log(std::string str, bool ignoreTabs) {
    if (mode == DEBUG) {
        if (!ignoreTabs)
            for (int i = 0; i < tab; ++i) {
                std::cout << '\t';
            }
        std::cout << str;
    }
}

void Logger::incTab() {
    tab++;
}

void Logger::decTab() {
    tab--;
}

Logger::MODE Logger::getMode() {
    return mode;
}
```

Файл prefFunc.h

```
#ifndef PREFIXFUNC_H
#define PREFIXFUNC_H

#include <vector>
#include <string>
#include "Logger.h"

std::vector<int> prefFunc(std::string str);

#endif //PREFIXFUNC_H
```

Файл prefFunc.cpp

```
#include "prefixFunc.h"

// Вычисление префикс-функции
std::vector<int> prefFunc(const std::string &str) {
    _log("Calculation of Prefix-Function of string: \"" + str + "\"\n");
    _incTab();

    int n = str.size(), j;
    // Итоговый массив П.Ф. (заполнен нулями)
    std::vector<int> pf(n, 0);

    _log("Current prefix: (len:0)" + str.substr(0, 1) + "\n");
    _log("Value of border at 0th symbol = 0\n\n");
    for (int i = 1; i < n; ++i) {
        _log("Current prefix: (len:" + std::to_string(i) + ") " + str.substr(0,
i + 1) + "\n");
        // Значение П.Ф. на предыдущей позиции.
        j = pf[i - 1];
        _log("Previos border: " + std::to_string(j) + "\n");
        // Сравниваем очередной символ в строке с символом на позиции какого-
либо значения П.Ф.
```

```
        // до тех пор пока не найдётся совпадение, либо значение П.Ф. не дойдёт
до нуля
        _incTab();
        _log("Comparing symbol at " + std::to_string(i) + " pos - '" + str[i] +
"' with symbol at border value - '" + str[j] + "'\n");
        while (j > 0 && str[i] != str[j]) {
            j = pf[j - 1];
            _log("Occurence is not found, next border to compare: " +
std::to_string(j) + "\n");
            _log("Comparing symbol at " + std::to_string(i) + " pos - '" +
str[i] + "' with symbol at index of border value - '" + str[j] + "'\n");
        }
        // Если совпадение есть, очередное значение П.Ф. равно предыдущему + 1
        // Иначе i-ое значение остаётся нулевым
        if (str[i] == str[j]) {
            pf[i] = j + 1;
            _log("Occurency is found, value of border: " + std::to_string(j + 1)
+ "\n");
        } else {
             _log("Occurency is not found, value of border: 0 \n");
        }
               _decTab();

        _log("Value of border at " + std::to_string(i) + "th symbol = " +
std::to_string(pf[i]) + "\n\n");
    }
    _decTab();
    _log("Final P.F.: [");
    for (int i = 0; i < pf.size(); ++i) {
        _log(std::to_string(pf[i]), true);
        if (i != pf.size() - 1)
            _log(", ", true);
    }
    _log("]\n\n", true);

    return pf;
}
```

Файл kmp.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "prefixFunc.h"

// Поиск вхождений s1 в s2 (алгоритм КМП)
std::vector<int> kmp(const std::string &s1, const std::string &s2, char delim-
iter = '|') {
    _log("Finding occurrences of \"" + s1 + "\" in \"" + s2 + "\"\n");
    _incTab();

    // Длины переданных строк
    int l1 = s1.size(), l2 = s2.size();
    // Объединённая разделителем строка
    std::string S = s1 + delimiter + s2;
    // Вычисление П.Ф. от объединённой строки
    std::vector<int> pf = prefFunc(S);
    // Массив вхождений s1 в s2
    std::vector<int> pos;

    _log("Iterate over prefix-function:\n");
    _incTab();
```

```
    for (int i = l1 + 1; i < l1 + 1 + l2; ++i) {
        _log("Current substring: " + s2.substr(0, i - l1) + "\n");

        // Если П.Ф. на i-ом элемента равна длине первой строки, значит i-ый
элемент
        // - последний элемент вхождения s1 в s2.
        if (pf[i] == l1) {
            // А индекс начала вхождения будет вычисляться
            // (i - l1 + 1) - начало вхождения s1 в строке S
            // (- l1 - 1) - вычесть длину присоединённой s1 и разделителя
            // Итого (i - 2 * l1)
            pos.push_back(i - 2 * l1);
            _log("Occurrence is found at symbol " + std::to_string(i - l1 - 1) +
                 " (beginning of occurrence at " + std::to_string(i - 2 * l1) +
")\n");
        }
        _log("\n", true);
    }
    _decTab();
    _decTab();
    _log("Final occurrences = [");
    for (int i = 0; i < pos.size(); ++i) {
        _log(std::to_string(pos[i]));
        if (i != pos.size() - 1)
            _log(", ");
    }
    _log("]\n\n");

    return pos;
}

int main() {
    _setMode(Logger::DEBUG);

    // P - шаблон, T - текст
    std::string P, T;
    std::cin >> P >> T;

    // Массив вхождений
    std::vector<int> pos = kmp(P, T);
    for (int i = 0; i < pos.size(); ++i) {
        std::cout << pos[i];
        if (i != pos.size() - 1)
            std::cout << ',';
    }
    // Если пустой -> -1
    if (!pos.size())
        std::cout << -1;

    return 0;
}
```

Файл cycle.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "prefixFunc.h"
#include "Logger.h"

// Вычисление индекса сдвига (s1 - сдвиг s2)
int cycleShift(std::string s1, std::string s2, char delimiter = '|') {
    _log("Detection of possible cycle shift from \"" + s2 + "\" to \"" + s1 +
```

```
"\"\n");
    _incTab();

    int l1 = s1.size();
    // Если длины строк различны, дальше сравнивать бессмысленно
    if (l1 != s2.size()) {
        _decTab();
        _log("Strings sizes are not equal => -1");
        return -1;
    }

    // Используем подход аналогичный КМП, конкатенируем строки с разделителем
    // (только теперь s2 будет искаться в s1)
    std::string S = s2 + delimiter + s1;
    // П.Ф. от полученной строки
    std::vector<int> pf = prefFunc(S);
    // Нас интересует значение П.Ф. только на последнем символе, оно будет равно
    // количеству совпавших символов из начала второй строки с концом первой
строки
    int j = pf.back();

    _log("Value of P.F. at last symbol = " + std::to_string(j) + "\n");

    // Остаётся сравнить оставшиеся части: начало первой строки и конец второй
    _decTab()
    _log("Compare remain parts : \"" + s1.substr(0, l1 - j) + "\" and \"" +
s2.substr(j) + "\"");
    if (s1.substr(0, l1 - j) == s2.substr(j)) {
        _log(" (equal) => " + std::to_string(j) + " is shift index\n\n", true);
        return l1 - j;
    }
    else {
        _log(" (not equal) => s1 is not cycle shift of s2\n\n", true);
        return -1;
    }
}

int main() {
    _setMode(Logger::DEBUG);
    // B - строка, A - её предполагаемый циклический сдвиг
    std::string A, B;
    std::cin >> A >> B;
    std::cout << cycleShift(A, B);
}
```