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. 8382 Мирончик П.Д.

Преподаватель Фирсов М.А.

Санкт-Петербург
2020

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

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* вляется 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:

Вариант 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.

ОПИСАНИЕ СТРУКТУР

Задание 1.

void printPosition(bool& isFirst, int position) – выводит *position* и, если *isFirst* установлен в *false*, запятую перед *position*. Устанавливает *isFirst* в *false*.

string P – строка, содержащая образ.

int pi[P.length()] – массив префиксов для *P*.

Задание 2.

void printPosition(bool& isFirst, int position) – выводит *position* и, если *isFirst* установлен в *false*, запятую перед *position*. Устанавливает *isFirst* в *false*.

string P – строка, содержащая образ.

string T – строка, в которой производится поиск.

int pi[P.length()] – массив префиксов для *P*.

АЛГОРИТМ

Задание 1.

1. Для образа создается массив префиксов pi, соответствующий правилу

size=pi[k], P[0:size-1] == P[k-size+1:size], причем pi[k] – максимальное число, при котором выполняется вторая часть условия.

где *P[n,m]* – подстрока образа с *n* по *m* символы включительно.

Далее обрабатывается строка, в которой выполняется поиск. На каждом шаге имеются следующие параметры:

i – позиция курсора в строке поиска;

j – позиция курсора в образе.

Алгоритм шага можно описать следующим образом:

1. Если *j==P.length(),* то найдено вхождение *P*, выводим позицию и присваиваем *j=pi[j-1]*. Фактически этим действием достигается сдвиг образа на наименьшую дистанцию, после которого часть строки *P* левее *j* совпадает с подстрокой *T* такой же длины левее *i*. Иначе это можно представить как несовпадение последнего + 1 (стоящего за пределами) символа образа и текущего символа строки поиска.

2. Иначе, если *T[i]==P[j],* можно сдвинуть оба курсора вправо – т.е. найден очередной совпадающий символ.

3. Иначе, если *j==0* (т.е. длина текущего совпадения равна нулю и при этом текущие символы не совпали), сдвигаем курсор I вправо.

4. Иначе присваиваем *j=pi[j-1],* выполняя смещение образа на наименьшую дистанцию, после которого часть строки *P* левее *j* совпадает с подстрокой *T* такой же длины левее *i*.

Если после прохождения строки поиска не было найдено ни одно совпадение, то совпадений нет, выводится *-1*.

Задание 2.

Обработка образа *P* (создание массива префиксов) выполняется аналогично.

Второй алгоритм имеет два отличия от первого: во-первых необходимо пройти не до конца строки, а, фактически, пройти строку дважды (визуально это можно представить как поиск в двух склеенных строках поиска). Этим достигается просмотр всех возможных вариантов, т.к. просматриваются все возможные позиции для начала образа.

Во-вторых, из-за необходимости дважды проходить строку возникает необходимость хранить строку T в памяти.

В алгоритме для двойного прохода используется простое решение: курсор *i* изменяется от *0* до *T.length()*2,* а текущий символ можно посчитать как *T[I % T.length()]*. Этим достигается отсутствие необходимости хранить склеенную строку поиска.

При выполнении поиска цикл прерывается при нахождении первого же вхождения *P* в *T*, в отличие от первого алгоритма.

СЛОЖНОСТЬ

Сложности алгоритмов по времени можно описать как $O(m + n)$ для первого и $O(n*3)$ для второго, где n – длина P, m – длина T. Отличие сложности второго алгоритма от первого объясняется тем, что длины строк P и T совпадают, а проход по строке поиска производится дважды, в отличие от первого алгоритма.

Сложность по памяти вычисляется как $O(m)$ для первого и $O(2n)$ для второго.

ТЕСТИРОВАНИЕ

Задание 1

Номер теста	input	output
1	ab abab	0,2
2	abc abcadbf	0
3	aa aaaaaaaaaa	0,1,2,3,4,5,6,7,8

Задание 2

Номер теста	input	output
1	aaabbb bbaaab	4
2	abcdef abcdef	0
3	asdfg gfdsa	-1

Также приводится вывод для первого теста первого задания с параметром *IS_LOGGING*, установленным в *True*:

```
Suffix's size is 0 on 1
Symbols on T[0], P[0] are equals. Increase equals part size.
Symbols on T[1], P[1] are equals. Increase equals part size.
Found ingoing on position 0
0

Symbols on T[2], P[0] are equals. Increase equals part size.
Symbols on T[3], P[1] are equals. Increase equals part size.
Found ingoing on position 2
,2

End.
```

ЗАПУСК ПРОГРАММ

Для корректной работы необходимо, чтобы в системе был установлен компилятор *g++* и путь к нему был добавлен в переменные среды.

Для сборки и запуска программ необходимо запустить *main.bat* (или *main_second.bat* для второй программы) из консоли *cmd*, т.к. в *Windows PowerShell* кое-что не сработает. В качестве параметров необходимо передать номер теста, который необходимо выполнить.

Тесты первой программы хранятся в папке *tests*, второй – в *tests_second*. Тест *n* должен называться как записывается как *test**n**.txt* и лежать в соответствующей программе папке.

Пример запуска теста номер 1 для первой программы:

main 1

Для второй, соответственно, это будет

main_second 1

ВЫВОД

В процессе выполнения работы была решена задача поиска максимального потока в сети с использованием рекурсивной реализации алгоритма Кнута-Морриса-Пратта. При этом в решении использовались различные паттерны для упрощения кода, а также разнообразные методы оптимизации, необходимые для решения поставленных задач.

ПРИЛОЖЕНИЕ А.
КОД ПРОГРАММЫ 1

```
#include <iostream>
#include <cstring>

using namespace std;

bool TEST = true;

/**
 * Prints [msg] to cout, is [TEST] is true
 *
 * @param msg Message to print
 */
void log(const string& msg) {
    if (TEST)
        cout << msg << endl;
}

/**
 * Prints [position] and, if [ifFirst] is false, ',' before. Sets
[isFirst] to false
 *
 * @param isFirst Should or not ',' be printed
 * @param position Number to print
 */
void printPosition(bool& isFirst, int position) {
    if (!isFirst)
        cout << ",";

    isFirst = false;
    cout << (position);
    log("\n");
}

int main() {
    string P;
    getline(cin, P);

    // Array of prefixes
    int pi[P.length()];
    pi[0] = 0;
    // size - current prefix(postfix)'s length
    // i - current symbol (last postfix's symbol position)
    // j is used in the next cycle.
    int size = 0, i = 1, j;

    while (i < P.length()) {
        // If symbols are equals, move cursor and increase postfix's
size
        if (P[size] == P[i]) {
            log("Suffix's size increased. Suffix on " + to_string(i) +
" is " + to_string(size + 1));
            pi[i] = size + 1;
```

```
            size++;
            i++;
            continue;
        }

        // If symbols are not equals and postfix's length is 0,
maximum length of prefix and postfix
        // when they are equals is 0
        if (size == 0) {
            log("Suffix's size is 0 on " + to_string(i));
            pi[i] = 0;
            i++;
            continue;
        }

        log("Decrease suffix size from " + to_string(size) + " to " +
to_string(pi[size - 1]));
        // Decrease postfix's size until it's possible and prefix is
not equals suffix with that size
        size = pi[size - 1];
    }

    // Current symbol's index in string
    i = 0;

    // Current symbol's index in image
    j = 0;

    // True if there yet was not found any position, false otherwise
    bool isFirst = true;

    // If image is empty, there is no equals substrings (or any
position is available)
    if (P.length() == 0) {
        log("P's length is 0. End.");
        cout << "-1";
        return 0;
    }

    // Current symbol in string
    char c;
    cin >> c;

    // Read while we can
    while (!cin.eof()) {
        // If j is outside of image, we have found image in the
string.
        // We should print found position and decrease j on the
nearest available
        // position, so P[0:j-1] is equals string[i-j;i-1]
        if (j == P.length()) {
            log("Found ingoing on position " + to_string(i -
P.length()));
            printPosition(isFirst, (int) (i - P.length()));
            j = pi[j-1];
            continue;
        }
```

```
        // If current symbol in string is equals with current symbol
in image,
        // move cursors to the next positions
        if (c == P[j]) {
            log("Symbols on T[" + to_string(i) + "], P[" +
to_string(j) + "] are equals. Increase equals part size.");
            i++;
            j++;
            cin >> c;
            continue;
        }

        // If j is 0 and P[j] is not equals to c, just move cursor in
string
        if (j == 0) {
            log("There is no equals substring on " + to_string(i) + "
position. Increase T's cursor");
            i++;
            cin >> c;
            continue;
        }

        log("Decrease P's cursor.");
        // Decrease image's cursor while j is not 0 and P[j] is not
equals to c
        j = pi[j-1];
    }

    // If j==P.length(), there was found image position at the end of
string
    if (j == P.length()) {
        log("Found ingoing on position " + to_string(i - P.length()));
        printPosition(isFirst, (int) (i - P.length()));
    }

    // If isFirst is false (or if there was not found any substrings
in string
    // equals to P, print -1
    if (isFirst) {
        log("No ingoing found in T. End.");
        cout << "-1";
    } else {
        log("End.");
    }
}
```

ПРИЛОЖЕНИЕ Б.

КОД ПРОГРАММЫ 2

```
#include <iostream>
#include <cstring>

using namespace std;

bool TEST = true;

/**
 * Prints [msg] to cout, is [TEST] is true
 *
 * @param msg Message to print
 */
void log(const string& msg) {
    if (TEST)
        cout << msg << endl;
}

/**
 * Prints [position] and, if [ifFirst] is false, ',' before. Sets
[isFirst] to false
 *
 * @param isFirst Should or not ',' be printed
 * @param position Number to print
 */
void printPosition(bool& isFirst, int position) {
    if (!isFirst)
        cout << ",";

    isFirst = false;
    cout << (position);
    log("\n");
}

/**
 * Computes index of [real_position] in string with length [length]
 * @param real_position Real position of cursor
 * @param length Length of string
 * @return real_position % length - position of cursor in string
 */
int index(int real_position, int length) {
    return real_position % length;
}

int main() {
    // image
    string P;
    // string (where image will be searched. Or not:)
    string T;
    // read image and string
    getline(cin, T);
    getline(cin, P);

    // If P is not equals to T, P is not be created from prefix and
suffix of
```

```
    // T, as it required with task
    if (P.length() != T.length()) {
        cout << "-1";
        return 0;
    }

    // Array of P's prefixes. Memory allocates dynamically because
20Mb's stack is
    // not pass on Stepik :(
    int *pi = new int[P.length()];

    // like in 1st task
    int size = 0, i = 1, j;
    pi[0] = 0;

    // creating prefixes array like if 1st task
    while (i < P.length()) {
        if (P[size] == P[i]) {
            log("Suffix's size increased. Suffix on " + to_string(i) +
" is " + to_string(size + 1));
            pi[i] = size + 1;
            size++;
            i++;
            continue;
        }

        if (size == 0) {
            log("Suffix's size is 0 on " + to_string(i));
            pi[i] = 0;
            i++;
            continue;
        }

        log("Decrease suffix size from " + to_string(size) + " to " +
to_string(pi[size - 1]));
        size = pi[size - 1];
    }

    i = 0;
    j = 0;
    bool isFirst = true;

    // just one more useless check
    if (P.length() == 0) {
        log("P's length is 0. End.");
        cout << "-1";
        return 0;
    }

    i = 0;
    // length of T (will be used often, so it's just for usability)
    int Tlen = (int) T.length();

    // Iterating while i is less than Tlen*2. It's necessary, because
be can have lines like
    // AAAC, CAAA,
    // and we don't need more, because in that case search will start
anew
```

```
    while (i < T.length() * 2) {
        if (j == P.length()) {
            printPosition(isFirst, (int) (index(i - (int) P.length(),
Tlen)));
            log("Found ingoing on position " + to_string(i -
P.length()));
            j = pi[j-1];
            break;
        }

        if (T[index(i, Tlen)] == P[j]) {
            log("Symbols on T[" + to_string(i) + "], P[" +
to_string(j) + "] are equals. Increase equals part size.");
            i++;
            j++;
            continue;
        }

        if (j == 0) {
            log("There is no equals substring on " + to_string(i) + "
position. Increase T's cursor");
            i++;
            continue;
        }

        log("Decrease P's cursor.");
        j = pi[j-1];
    }

    if (isFirst)
        cout << "-1";
}
```