**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра МОЭВМ**

# ОТЧЕТ

**по лабораторной работе №4**

**по дисциплине «Построение и анализ алгоритмов»**

**Тема: Алгоритм Кнута-Морриса-Пратта**

Студентка гр. 8382 _____ Кузина А.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

**Цель работы.**

Ознакомиться с алгоритмом Кнута-Морриса-Пратта для поиска подстроки в строке.

**Постановка задачи.**

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.

Задача 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.

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

Задача 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.

Вход:

Первая строка - A

Вторая строка - B

Выход:

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

Sample input:

edfabc

abcdef

Sample output:

3

**Описание алгоритма.**

**Префикс-функция.**

Префикс-функция это функция, которая для каждого символа строки находит максимальную длину собственного суффикса равного некоторму префиксу этой же строки. Для каждого символа строки с номером *n*, кроме 0-го, для которого префикс-функция всегда равна 0, вычисляется значение *tmpLen* равное значению префикс-функции *n* - 1 символа строки. Если текущий символ n и символ под номером *tmpLen* совпадают, то для данного символа префикс-функция будет на единицу больше, чем для предыдущего символа. Если же они не совпадают, то *tmpLen* вычисляется заново как префикс-функция *tmpLen* – 1-го элемента. Для обновленного значения проверка повторяется. Такие проверки заканчиваются, если в какой-то момент текущий символ совпал с символом под номером *tmpLen* или если *tmpLen* стала равна нулю. В случае, когда *tmpLen* = 0, префикс-функция для текущего *n*-го элемента равна единице, если текущий символ совпадает с нулевым, или нулю в противном случае.

**Поиск вхождений образца в строку.**

На вход программе последовательно подается две строки: образец и строка поиска. Для образца вычисляется префикс-функция. Далее для

строки поиска выполняется алгоритм, схожий с вычислением префикс-функции, только в данном случае префикс берется из образца, а суффикс берется в строке поиска. Так как для стоки поиска не обязательно хранить все значения префикс-функции, сохраняется только последнее полученное. Если наступает момент, в котором значение префикс-функции для текущего символа совпадает с длиной образца значит, что найдено вхождение образца в строку поиска. На экран выводится индекс его начала: текущий индекс – длина строки + 1.

**Поиск циклического сдвига.**

На вход программе последовательно подается две строки: изначальная и та, которую программа проверяет на сдвиг. Если длины полученных строк не совпадают, значит одна из них точно не является сдвигом другой, программа выводит -1 и завершается. Иначе для первой строки вычисляется префикс-функция. Затем для второй строки выполняется алгоритм циклического обхода дважды, с вычислением текущего значения префикс-функции, где префикс берется из первой строки, а суффикс из второй с зацикливанием. Если наступает момент, в котором значение префикс-функции для текущего символа совпадает с длиной строки значит, что вторая строка является сдвигом первой строки. На экран выводится индекс начала сдвига: текущий индекс + 1.

**Сложность алгоритма.**

Пусть длина строки-образца равна n, а длина второй строки равна m. Для вычисления значений префикс-функции строки используется два вложенных цикла: внешний цикл обходит каждый элемент строки, внутренний цикл находит максимальную длину префикса для текущей подстроки. Внешний цикл совершает n итераций, при этом каждый раз значение префикс-функции не более чем на 1 превышает ее предыдущее значение. Внутренний цикл уменьшает значение префикс-функции на 1 или

больше, при этом оно не может опуститься ниже нуля. Количество уменьшений значения во внутреннем цикле не может превысить количество увеличений во внешнем, значит внутренний цикл также совершит не более чем n итераций. Значит сложность вычисления префикс-функции *O(2n).*

Сложность поиска образца оценивается также, только с учетом того, что внешний цикл проходит по строке большей длины m. Общая сложность алгоритма первого задания составит *O(2n + 2m)*.

Сложность поиска сдвига одной строки в другой при учете их одинаковой длины составит *O(2n + 4n) = O(6n)* за счет того, что вторая строка проходится алгоритмом в худшем случае два раза.

Сложность по памяти составляет O(n + m) для хранения строк и еще *O(n)* для хранения префикс-функции первой строки, что соответствует поставленной задаче.

**Описание структур данных и функций.**

**Структуры данных.**

Пользовательские структуры данных в данной работе не использовались.

**Используемые функции.**

- *int* prefix(string& text)* – используется для нахождения префикс-функции строки *text.* Возвращает массив вычисленных значений префикс-функции для строки *text.*
- *void Pattern(string& pattern, string& text)* – используется для поиска подстроки *pattern* в тексте *text.* Производит вычисления, описанные выше в пункте «Поиск вхождения образца в строку» раздела «Описание алгоритма». При этом функция выводит все промежуточные результаты в понятном виде. Также функция выводит полученный результат: индексы начала вхождения образца в текст или сообщение об их отсутствии.

- *void Shift(string& pattern, string& text)* – используется для проверки, является ли строка *text* сдвигом строки *pattern*. Производит вычисления, описанные выше в пункте «Поиск циклического сдвига» раздела «Описание алгоритма». При этом функция выводит все промежуточные результаты в понятном виде. Также функция выводит полученный результат: индекс сдвига или сообщение о том, что вторая строка не является сдвигом первой.
- В функции *int main()* производится считывание вводимых пользователем строк, настройка вывода: будут ли выводится подробные сообщения о ходе работы программы или только результат ее работы, и также вызов необходимой функции.

**Тестирование.**

Исходный код программы представлен в приложении А.

| Входные данные | Выходные данные |
|---|---|
| Поиск вхождений образца в текст | |
| ab<br>abab | Enter first text, then press enter-button and enter second text<br>Calculating prefix function for: ab<br>str[0] value => a; prefix = 0<br>str[1] value => b; prefix = 0<br><br>Serching for pattern: ab in the text: abab<br><br>Step 0:<br>Watching text[0] value => a; current prefix length = 0<br>pattern[0] value => a == text[0] value => a<br>Expend previous prefix, new prefix length = 1<br>Prefix for text[0] value => a = 1<br><br>Step 1:<br>Watching text[1] value => b; current prefix length = 1<br>pattern[1] value => b == text[1] value => b |

| | |
|---|---|
| | Expend previous prefix, new prefix length = 2<br>Prefix for text[1] value => b = 2<br><br>Find a pattern in the text at 0 position<br><br>Step 2:<br>Watching text[2] value => a; current prefix length = 2<br>pattern[2] value => != text[2] value => a<br>Can't expend previous prefix, new prefix length = 0<br>pattern[0] value => a == text[2] value => a<br>Expend previous prefix, new prefix length = 1<br>Prefix for text[2] value => a = 1<br><br>Step 3:<br>Watching text[3] value => b; current prefix length = 1<br>pattern[1] value => b == text[3] value => b<br>Expend previous prefix, new prefix length = 2<br>Prefix for text[3] value => b = 2<br><br>Find a pattern in the text at 2 position |
| abaa<br><br>ababaaabasa<br>abaabaa | 2, 11, 14 |
| abaa<br><br>abababasaba<br>ba | -1 |
| aba<br><br>abababa | 0, 2, 4 |
| Поиск циклического сдвига строки | |
| defabc<br>acbdef | Enter first text, then press enter-button and enter second text<br>Calculating prefix function for: defabc<br>str[0] value => d; prefix = 0<br>str[1] value => e; prefix = 0<br>str[2] value => f; prefix = 0<br>str[3] value => a; prefix = 0 |

```
        str[4] value => b; prefix = 0
        str[5] value => c; prefix = 0

Check is str2 "abcdef" shift of str1 "defabc"

  Step 0:
        Watching str2[0] value => a; current prefix length = 0
        Prefix for str2[0] value => a = 0

  Step 1:
        Watching str2[1] value => b; current prefix length = 0
        Prefix for str2[1] value => b = 0

  Step 2:
        Watching str2[2] value => c; current prefix length = 0
        Prefix for str2[2] value => c = 0

  Step 3:
        Watching str2[3] value => d; current prefix length = 0
          str1[0] value => d == str2[3] value => d
            Expend previous prefix, new prefix length = 1
        Prefix for str2[3] value => d = 1

  Step 4:
        Watching str2[4] value => e; current prefix length = 1
          str1[1] value => e == str2[4] value => e
            Expend previous prefix, new prefix length = 2
        Prefix for str2[4] value => e = 2

  Step 5:
        Watching str2[5] value => f; current prefix length = 2
          str1[2] value => f == str2[5] value => f
            Expend previous prefix, new prefix length = 3
        Prefix for str2[5] value => f = 3

  Step 6:
        Watching str2[0] value => a; current prefix length = 3
```

| | |
|---|---|
| | str1[3] value => a == str2[0] value => a<br>Expend previous prefix, new prefix length = 4<br>Prefix for str2[0] value => a = 4<br><br>Step 7:<br>Watching str2[1] value => b; current prefix length = 4<br>str1[4] value => b == str2[1] value => b<br>Expend previous prefix, new prefix length = 5<br>Prefix for str2[1] value => b = 5<br><br>Step 8:<br>Watching str2[2] value => c; current prefix length = 5<br>str1[5] value => c == str2[2] value => c<br>Expend previous prefix, new prefix length = 6<br>Prefix for str2[2] value => c = 6<br><br>Second string IS the first string shift! Shift = 3 |
| qwertyuio<br>wertyuioq | 8 |
| 123321<br>123123 | -1 |
| aba<br>ababa | -1 |
| Ab<br>Ab | 0 |

**Выводы.**

В ходе выполнения лабораторной работы были получены навыки применения алгоритма Кнута-Морриса-Пратта. Также была написана программа, выполняющая следующие функции: нахождение всех вхождений образца в текст, проверка является ли одна строка циклическим сдвигом другой строки, выводящая промежуточные данные, ход решения и ответ понятным образом на экран.

## ПРИЛОЖЕНИЕ А

В функции Main должна быть не закомментирована функция *shift*, если задача – проверить строку на циклический сдвиг или функция *pattern*, если задача – найти все вхождения паттерна в текст.

Переменная *output* определяет будет ли полный вывод или только результат работы программы.

## ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

```
#include <iostream>
using namespace std;
int output = 0;

int* prefix(string& text) {
	//calculate the prefix function for the input string
	if (output) {
		cout << "Calculating prefix function for: " << text << "\n"; }

	int* pi = (int*)malloc(text.length()*sizeof(int));
	pi[0] = 0; //array for prefix function values

	for (int i = 1; i < text.length(); i++){ //for each string elem:
		int tmpLen = pi[i - 1]; //current suffix length = prefix function of the pre-
vious element
		while (tmpLen > 0 && text[tmpLen] != text[i])
			tmpLen = pi[tmpLen - 1]; //can’t increase the current suffix - go to
the suffix of shorter length
		if (text[tmpLen] == text[i])
			tmpLen++; //can increase the current suffix - the elements does match
		pi[i] = tmpLen; //prefix function for the current item
	}

	for (int i = 0; i < text.length(); i++) {
		if (output) {
			cout << "\tstr[" << i << "] value => " << text[i] << "; prefix = " <<
pi[i] << "\n"; }
	}
	if (output) cout << "\n";
	return pi;
}

void Pattern(string& pattern, string& text) {
	int* pi = prefix(pattern); //prefix function values for pattern
	int n = text.length();
	int pLen = pattern.length();
	int tmpLen = 0;
	int test = 0;
	if (output) {
		cout << "Serching for pattern: " << pattern << " in the text: " << text <<
"\n"; }

	for (int i = 0; i < n; i++) { // for each element of second string
		//current suffix length = prefix function of the previous element
		if (output) {
			cout << "\n  Step " << i << ":"; }
		if (output) {
```

```
                        cout << "\n\tWatching text[" << i << "] value => " << text[i] << ";
current prefix length = " << tmpLen << "\n";
                }
                while (tmpLen > 0 && pattern[tmpLen] != text[i]) {
                        // can’t increase the current suffix - element does not match
                        if (output) {
                                cout << "\t  pattern[" << tmpLen << "] value => " << pat-
tern[tmpLen] << " != text[" << i << "] value => " << text[i] << "\n";
                        }
                        tmpLen = pi[tmpLen - 1];  // try suffix of shorter length
                        if (output) {
                                cout << "\t    Can't expend previous prefix, new prefix length =
" << tmpLen << "\n"; }
                }
                if (pattern[tmpLen] == text[i]) {//can increase the current suffix - the ele-
ments does match
                        if (output) {
                                cout << "\t  pattern[" << tmpLen << "] value => " << pat-
tern[tmpLen] << " == text[" << i << "] value => " << text[i] << "\n";
                        }
                        if (output) {
                                cout << "\t    Expend previous prefix, new prefix length = " <<
tmpLen + 1 << "\n"; }
                        tmpLen++;
                }
                if (output) {
                        cout << "\tPrefix for text[" << i << "] value => " << text[i] << " = "
<< tmpLen << "\n";
                }
                if (tmpLen == pLen) {//the suffix length macth with the length of the pattern
                        if (output) {
                                cout << "\nFind a pattern in the text at "; }
                        if (test > 0 && !output) cout << ",";
                        test++;
                        cout <<  i - tmpLen + 1;// the occurrence of the pattern in the text
was found
                        if (output) {
                                cout << " position\n"; }
                }
        }
        if (test == 0) {
                if (output) {
                        cout << "There is on pattern in the next!\n"; }
                else cout << -1;//No occurrence of the pattern in the text was found
        }
}

void Shift(string& pattern, string& text) {
        //if the lengths of the lines do not match, one is definitely not a shift of the
other
        if (text.length() != pattern.length()) {
                if (output) {
                        cout << "Two strings has different sizes! " << text.length() << " != "
<< pattern.length() << "\n"; }
                else cout << -1;
                return;
        }

        int* pi = prefix(pattern); //prefix function values for first string
        int textLength = text.length();
        int tmpLen = 0;
        if (output) {
                cout << "Check is str2 \"" << text << "\" shift of str1 \"" << pattern <<
"\"\n"; }
```

```
	for (int i = 0; i < textLength * 2; i++) { //loop around the second string twice
		if (output) {
			cout << "\n  Step " << i << ":"; }
		int j = i % textLength; //for looping around, when i = textLength -> j = 0 ...
		//current suffix length = prefix function of the previous element
		if (output) {
			cout << "\n\tWatching str2[" << j << "] value => " << text[j] << ";
current prefix length = " << tmpLen << "\n";
		}

		while (tmpLen > 0 && pattern[tmpLen] != text[j]) {//can't increase the current
suffix - element does not match
			if (output) {
				cout << "\t  str1[" << tmpLen << "] value => " << pattern[tmpLen]
<< " != str2[" << j << "] value => " << text[j] << "\n";
			}
			tmpLen = pi[tmpLen - 1];  // try suffix of shorter length
			if (output) {
				cout << "\t    Can't expend previous prefix, new prefix length =
" << tmpLen << "\n";
			}
		}

		if (pattern[tmpLen] == text[j]) {//can increase the current suffix - the ele-
ments does match
			if (output) {
				cout << "\t  str1[" << tmpLen << "] value => " << pattern[tmpLen]
<< " == str2[" << j << "] value => " << text[j] << "\n";
			}
			if (output){
				cout << "\t    Expend previous prefix, new prefix length = " <<
tmpLen + 1 << "\n";
			}
			tmpLen++;
		}
		if (output) {
			cout << "\tPrefix for str2[" << j << "] value => " << text[j] << " = "
<< tmpLen << "\n"; }
		if (tmpLen == textLength) {//the suffix length macth with the length of the
pattern
			if (output) {
				cout << "\nSecond string IS the first string shift! Shift = "; }
			cout << i-   textLength  + 1;  //found cyclic first string shift
			if (output) cout << "\n";
			return;
		}
	}
	if (output) {
		cout << "\nSecond string ISN'T the first string shift!\n"; }
	else cout << -1; //the second string is not a shift of the first string
}


int main() {
	string pattern;
	string text;
	output = 1; //If = 1 there will be a detailed conclusion of the course of the deci-
sion. if = 0 there will only be an answer
	if (output) {
		cout << "Enter first text, then press enter-button and enter second text\n"; }
	cin  >>  pattern >> text;//Input

	//You can use only one or both functions at once
	Pattern(pattern, text); //finds all occurrences of the first row in the second
	//Shift(pattern, text); //checks if the second line is a shift of the first
```

```
        return 0;
}
```