**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**

Студентка гр. 7383 _____ Маркова А. В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

# Содержание

## Цель работы

Исследовать и реализовывать задачу поиска вхождения подстроки в строке, используя алгоритм Кнута – Морриса – Пратта.

Формулировка задачи: необходимо разработать программу, которая реализует 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. Если P не входит в T, то вывести $-1$.

Также следует разработать программу для решения следующей задачи: 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. Если A является циклическим сдвигом B, индекс начала строки B в A, иначе вывести $-1$.

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.

Входные данные: в первой строке указывается строка шаблона P, а во второй текст, в котором ищем подстроки.

Выходные данные: индексы вхождения подстроки в строку.

**Реализация задачи**

В данной работе используются главная функция `main()` и дополнительные функции `void Prefix_function(std:: string &pattern)` и `void Algorithm_KMP(std:: string &pattern)`.

Параметр, который мы передаём обеим функциям `pattern`, является строкой шаблона.

В функции `main()` считывается строка `pattern`, а затем вызывается `Prefix_function`, в которой заполняется вектор `prefix` для строки шаблона. Далее начинает свою работу `Algorithm_KMP`, который посимвольно считываем строку и сравнивает её с шаблоном, используя полученные значения префикс – функции.

Для более понятной работы алгоритма рассмотрим пример работы программы для строк «abaabb» и «aba». Расчет значения префикс – функции представлен на рис. 1.

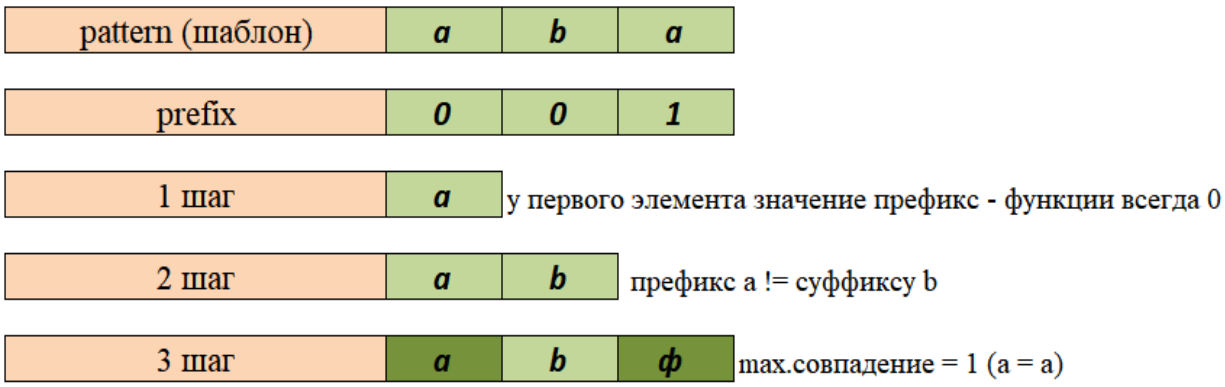


Рисунок 1 – `prefix`

Дальнейшие рассуждения показаны на рис. 2.

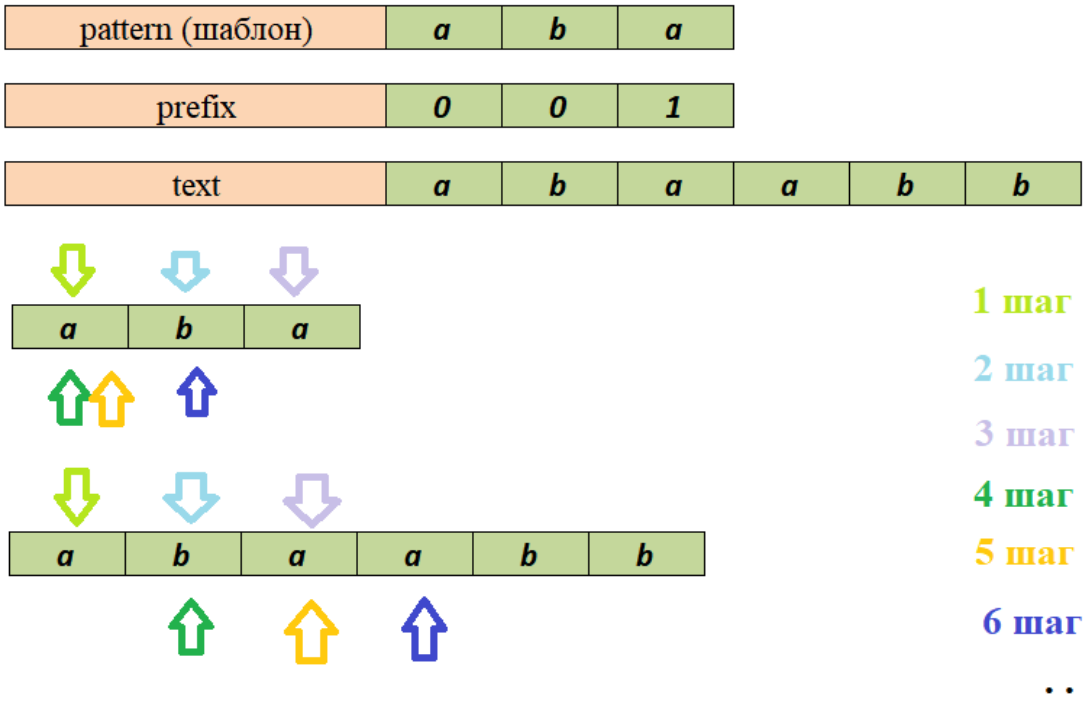


Рисунок 2 – ход алгоритма

Пока символы одинаковые оба указателя двигаем на единицу вперёд, если шаблон закончился, то возвращаемся по префикс – функции к а и опять сравниваем значения. Алгоритм работает до конца строки.

Для задания с циклическим сдвигом производим деление с остатком, тем самым переходя опять на начало шаблона, префикс – функция для данной задачи остаётся неизменной.

Исходный код программы представлен в приложении Б.

## Тестирование

Программа собрана в операционной системе Ubuntu 17.04, с использованием компилятора g++ версии 5.4.0 20160609. В других ОС и компиляторах тестирование не проводилось.

Программа может быть скомпилирована с помощью команды:

g++ <имя файла>.cpp

Тестовые случаи представлены в Приложении А.

Исходя из тестовых случаев можно заметить, что тестовые случаи не выявили неправильного поведения программы, что говорит о том, что по результатам тестирование было показано, поставленная задача была выполнена.

## Исследование

В начале работы программа вычисляет значения префикс функции для каждого символа первой строки. Пусть P – длина первой строки, тогда T – длина второй. В этом случае сложность алгоритма по времени будет составлять O(P + T).

По памяти сложность алгоритма составляет O(P).

## Выводы

В ходе лабораторной работы был изучен алгоритм поиска подстроки в строке, используя алгоритм Кнута – Морриса – Пратта. Был написан код на языке программирования C++, который применял этот метод для поставленной задачи. Полученный алгоритм имеет линейную сложность как по времени, так и по памяти.

# ПРИЛОЖЕНИЕ А

## Тестовые случаи

| Ввод | Вывод | Верно? |
|---|---|---|
| defabc<br>abcdef | 3 | Да |
| ab<br>abab | 0,2 | Да |
| aba<br>abaabb | 0 | Да |
| Aabaa<br>aabaabaabaabaaaaaaaaa | 0,3,6,9 | Да |
| sk<br>amxva | -1 | Да |

# ПРИЛОЖЕНИЕ Б

## Код программы

**lab4_1**

```
#include <iostream>
#include <string>
#include <vector>

std:: vector <int> prefix;

void Prefix_function(std:: string &pattern) {
    prefix.assign(pattern.size(), 0);
    prefix[0] = 0;
    int index = 1, current = 0;
    while (index < pattern.size()) {
        if(pattern[index] == pattern[current]) {
            prefix[index] = current + 1;
            index++;
            current++;
        }
        if(pattern[index] != pattern[current])
            if(current == 0) {
                prefix[index] = 0;
                index++;
            }
            else current = prefix[current - 1];
    }
}

void Algorithm_KMP(std:: string &pattern) {
    char symbol;
    bool check = false;
    std:: cin >> symbol;
    int text = 0, image = 0;
    Prefix_function(pattern);
   while(true) {
        if(symbol == pattern[image]) {
            text++;
            image++;
            if(image == pattern.size()){
```

```
                if(check)
                    std:: cout << "," << text - pattern.size();
                else {
                    check = true;
                    std:: cout << text - pattern.size();
                }
              image = prefix[image - 1];
              }
            if(!(std:: cin >> symbol)) break;
        }
        else {
            if(image == 0) {
                text++;
                if(!(std:: cin >> symbol)) break;
            }
            else image = prefix[image - 1];
       }
    }
    if(!check) std:: cout << -1;                    //нет ни одного
совпадения в строке
}

int main() {
    std:: string pattern;
    std:: cin >> pattern;
    Algorithm_KMP(pattern);
    return 0;
}
```

**lab4_2**

```
#include <iostream>
#include <string>
#include <vector>

#define N 5000000

std:: vector <int> prefix;

void Prefix_function(std:: string &pattern) {
    prefix.assign(pattern.size(), 0);
    prefix[0] = 0;
    int index = 1, current = 0;
```

```
    while (index < pattern.size()) {
        if(pattern[index] == pattern[current]) {
            prefix[index] = current + 1;
            index++;
            current++;
        }
        if(pattern[index] != pattern[current])
            if(current == 0) {
                prefix[index] = 0;
                index++;
            }
            else current = prefix[current - 1];
    }
}

int Algorithm_KMP(std:: string &pattern, std:: string &text) {
    Prefix_function(pattern);
   int first = 0, second = 0, check = 0;
   for (int i = 0; i < pattern.size(); i++) {
         while (pattern[first] == text[second]) {
               check++;
               first = (first + 1) % pattern.size();
               second = (second + 1) % pattern.size();
               if (check == pattern.size()) return (second %
pattern.size());
               i++;
         }
         check=0;
         if (pattern[first] != text[second])
               if (first != 0){
                     first =  prefix[first - 1];
                     i--;
               }
               else second++;
   }
   return -1;
}

int main(){
   std::string text_A,text_B;
   text_A.reserve(N);
   text_B.reserve(N);
   std::cin >> text_B >> text_A;
```

```
    if (text_A.size() - text_B.size()) {
          std::cout << -1;
          return 0;
    }
    std::cout << Algorithm_KMP(text_A,text_B);
     return 0;
}
```