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 _____ Щемель Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Задание

Вхождение образца в строку

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

Нахождение циклического сдвига

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:
defabc
abcdef

Sample Output:
3

Номер варианта: 1. Подготовка к распараллеливанию: работа по поиску разделяется на k равных частей, пригодных для обработки k потоками (при

этом длина образца гораздо меньше длины строки поиска).

Описание алгоритма

Вхождение образца в строку

Считываются две строки: образец и строка для поиска. После эти строки склеиваются между собой через уникальный символ, отсутствующий в обеих строках. Для получившейся строки считается префикс-функция. Это функция, которая для каждого символа строки сохраняет максимальную длину собственного суффикса, равного префиксу этой же строки. Поэтому и используется уникальный символ, чтобы префиксом считался только образец. Префикс-функция считается в несколько потоков. На каждый поток выделяется определённый диапазон значений. После этого? необходимо заново пересчитать префис-функцию, от конца потока до конца образца. По тем значениям префикс функции, которые совпадают с длиной образца можно найти индексы вхождения образца в строке. (Если вычесть длину образца + 1 на уникальный символ). Данные значения ищутся также ищутся в несколько потоков(их количество задаётся пользователем). Количетсво потоков обозначим за *k*. На каждый поток выделяется *n = длина строки / количество потоков* значений для обработки. В случае, если длина строки не делится нацело на заданное количество потоков: деление выполняется с округлением вверх, на *k-1* потоков распределяется по *n* значений для поиска, оставшиеся *длина строки - n*k* значений отводятся для последнего потока.

Нахождение циклического сдвига

Считываются две строки. После чего высчитывается префикс-функция для условной “склейки” второй строки, уникального символа и двух идущих подряд первых строк. (В реализации вместо создания новой строки используется функция-обёртка, позволяющая по индексу получить символ, который в этой склейке бы был). В результате в префикс-функции последнее значение, равное длине первой строки и будет требуемым результатом (индекс начала второй строки в первой).

Сложности алгоритма:

Вхождение образца в строку

Сложность по количеству операций: $O(p + s)$, где p - длина строки образца, а s - длина строки для поиска. Префикс-функция вычисляется за линейное время, несмотря на наличие 2ух вложенных циклов в реализации. Это происходит потому, что итоговое количество определяется количеством итераций внутреннего цикла. Которе, в своб очередь определяется максимально возможным уменьшением счётчика, которое не преводсходит $O(p + s)$.

Сложность по количеству памяти: $O(p + s)$, где p - длина строки образца, а s - длина строки для поиска. Нам необходимо хранить сами строки, их “склейку” и значение префикс-функцию для неё.

Нахождение циклического сдвига

Сложность по количеству операций: $O(p + s)$, где p - длина строки образца, а s - длина строки для поиска. Аналогично предыдущей задачи. Сложность по количеству памяти: $O(p + s)$, где p - длина строки образца, а s - длина строки для поиска. Аналогично предыдущей задаче.

Описание функций и структур данных

Вхождение образца в строку

Структуры данных Пользовательские классы, использующиеся в реалзиации алгоритма не использовались.

Используемые функции (методы)

- *static int[] MultiThreadsPrefixFunction(string s, int[] patternPrefix, int lowerBound, int upperBound)* - используется для нахождения значения префикс-функции на заданном промежутке.

  string s - строка для поиска

  int[] patternPrefix - вычисленное значение префикс-функции для образца

  int lowerBound - нижняя грацница для вычисления

  int upperBound - верхняя граница для вычисления

- *static int[] NativePrefixFunction(string str)* - используется для высчитывания префикс-функции для строки в один поток

  string str - строка для поиска;

- *static void FixPrefix(List prefix, int[] patternPrefix, string s, int threadsCount , int countsPerThread)* - используется для повторного вычисления префикс-функции там, где конец потока не совпадает с концом образца

  List prefix - вычисленное значение префикс-функции

  int[] patternPrefix - вычисленное значение префикс-функции для образца

  int threadsCount - количесво потоков

  int countsPerThread - количество орабатываемых символов на поток.

- *static List FindPatternsOccurrences(string str, string pattern, int threadsCount)* - используется для как точка входа для алгоритма и возвращает список индексов начала образца *pattern* в строке *str*

  string str - строка для поиска;

  string pattern - искомый образец;

  int threadsCount - количество потоков

- *static List FindPatternsOccurrencesInPrefix(string str, int patternLength, IReadOnlyList prefix, int lowerBound, int upperBound)* - используется для параллельного нахождения необходимых значений(идексов вхождений) в префикс-функции на заданном промежутке

  string str - строка для поиска;

  int patternLength - длина строки образца;

  ReadOnlyList prefix - миссив значений префикс-функции;

  int lowerBound - начальный индекс диапазона для поиска;

  int upperBound - конечный индекс диапазона для поиска;

Нахождение циклического сдвига

Структуры данных Пользовательские классы, использующиеся в реалзиации алгоритма не использовались.

Используемые функции (методы)

- *char charFromPatternOrStr(int i, const std::string& pattern, const std::string& str)* - принимает индекс и по этому индексу возвращает символ, который стоял бы на месте “склейки” двух строк

  int i - индекс получаемого символа;

  const std::string& pattern - первая строка;

  const std::string& str - вторая строка;

- *std::vector prefixFunction(const std::string& pattern, const std::string& str)* - используется для вычисления префикс-функции для двух строк

  const std::string& pattern - первая строка;

  const std::string& str - вторая строка;

- *int findPatternsOccurenciesInPrifix(int patternLength, const std::vector& prefix)* - используется для нахождения необходимого значения(идекса начала вхождения одной строки в другую)

  int patternLength - длина первой строки;

  const std::vector& prefix - вектор значений префикс-функции;

- *int findIndexOfCyclicOffset(const std::string& str, const std::string& pattern)* - точка входа для алгоритма, которая принимает две строки и ищет начало вхождения одной строки в другую

  const std::string& str - вторая строка;

  const std::string& pattern - первая строка;

Тестирование

Вхождение образца в строку

Тест 1:

Input:
ab
abab
1

Output : Pattern value => ab String value => abab Count of threads value => 1 Thread => 1 Concatenated string => ab#abab s[i=1] => b s[k=0] => a Current prefix function => 00 Prefix for pattern => 000 Count of indexes per thread => 4

Thread => 0 Bounds => [0;4] s[i=0] => a s[k=0] => a Current prefix function => 1000 s[i=1] => b s[k=1] => b Current prefix function => 1200 s[i=2] => # s[k=2] => # s[k=0] => a Current prefix function => 1210 s[i=3] => a s[k=1] => b Current prefix function => 1212

Thread => 0 Bounds => [0;4]

Found value with 2 at => 0 Found value with 2 at => 2

0,2

Тест 2:

Input:
ab
abab
2

Output : Pattern value => ab String value => abab Count of threads value => 2 Thread => 1 Concatenated string => ab#abab s[i=1] => b s[k=0] => a Current prefix function =>

00 Prefix for pattern => 00 Count of indexes per thread => 2

Thread => 0 Bounds => [0;2] s[i=0] => a s[k=0] => a Current prefix function => 10 s[i=1] => b s[k=1] => b Current prefix function => 12

Thread => 1 Bounds => [2;4] s[i=2] => # s[k=0] => a Current prefix function => 10 s[i=3] => a s[k=1] => b Current prefix function => 12

Thread => 1 s[i=1] => b s[k=0] => a Current prefix function => 00 s[i=2] => # s[k=0] => a Current prefix function => 01 New prefix for bounds[1;3] => 01 Current prefix function => 1212

Thread => 0 Bounds => [0;2]

Found value with 2 at => 0

Thread => 1 Bounds => [2;4]

Found value with 2 at => 2

0,2

Тест 3:

Input:
aba
ababa 1

Output: 0,2

Test 4: aba
ababa
2

Output: 0,2

Тест 5: Input:
ab
ba
1

Output: -1

Тест 6: Input:
ab

ba

1

Output: -1

Нахождение циклического сдвига

Тест 1:

Input:

1234

2341

Output : Calc prefix-function for => 1234#23412341

i value => 1

Step => 1

str[i=1] => 2

str[k=0] => 1

k value => 0 at prefix[i-1=0]

Prefix value => 0000000000000

i value => 2

Step => 2

str[i=2] => 3

str[k=0] => 1

k value => 0 at prefix[i-1=1]

Prefix value => 0000000000000

i value => 3

Step => 3

str[i=3] => 4

str[k=0] => 1

k value => 0 at prefix[i-1=2]

Prefix value => 0000000000000

i value => 4

Step => 4

str[i=4] => #

str[k=0] => 1

k value => 0 at prefix[i-1=3]

Prefix value => 0000000000000

i value => 5

Step => 5

str[i=5] => 2

str[k=0] => 1

k value => 0 at prefix[i-1=4]

Prefix value => 0000000000000

i value => 6

Step => 6

str[i=6] => 3

str[k=0] => 1

k value => 0 at prefix[i-1=5]

Prefix value => 0000000000000

i value => 7

Step => 7

str[i=7] => 4

str[k=0] => 1

k value => 0 at prefix[i-1=6]

Prefix value => 0000000000000

i value => 8

Step => 8

str[i=8] => 1

str[k=0] => 1

k value => 0 at prefix[i-1=7]

Found equals chars. k index was increased to => 1

Prefix value => 0000000010000

i value => 9

Step => 9

str[i=9] => 2

str[k=1] => 2

k value => 1 at prefix[i-1=8]

Found equals chars. k index was increased to => 2

Prefix value => 0000000012000

i value => 10

Step => 10

str[i=10] => 3

str[k=2] => 3

k value => 2 at prefix[i-1=9]

Found equals chars. k index was increased to => 3

Prefix value => 0000000012300

i value => 11

Step => 11

str[i=11] => 4

str[k=3] => 4

k value => 3 at prefix[i-1=10]

Found equals chars. k index was increased to => 4

Prefix value => 0000000012340

i value => 12

Step => 12

str[i=12] => 1

str[k=4] => #

k value => 4 at prefix[i-1=11]

k index was decreased to => 0at prefix[11]

str[k] value => 1

Found equals chars. k index was increased to => 1

Prefix value => 0000000012341

prefix => 0000000012341

Found answer at prefix[patternLength+i+1=11]

1

Тест 2: Input:

123

323

Output: -1

Тест 3: Input:

ab

ab

Output: 0

Тест 4: Input:

123

1234

Output: -1

Тест 4: Input:

1234

123

Output: -1

Вывод

В ходе выполнения лабораторной работы были получены навыки применения алгоритма Кнута-Морриса-Пратта на примере создания програм для решения следующих задач: нахождение индексов вхождения образца в строке и индекс циклического смещения одной строки в другой.

Приложение А. Исходный код

Вхождение образца в строку

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace lab4
{
    /// <summary>
    /// Class wrapper for printing
    /// </summary>
    public static class Logger
    {
        public enum LogLevel
        {
            Debug,
            Info
        }

        /// <summary>
        /// Log message with specify log level
        /// </summary>
        /// <param name="text"></param>
        /// <param name="level"></param>
        public static void Log(object text, LogLevel level = LogLevel.Debug)
        {
            int logLevel;
            var hasVariable = int
            .TryParse(Environment
            .GetEnvironmentVariable("LAB4_LOG_LEVEL"), out logLevel);
            logLevel = hasVariable ? logLevel : (int)level;

            if ((int)level >= logLevel)
            {
```

```
            Console.WriteLine(text.ToString());
        }
    }
}

class Program
{
    /// <summary>
    /// Calculate prefix-function for string for single thread
    /// </summary>
    /// <param name="s">String</param>
    static int[] NativePrefixFunction(string s)
    {
        // Init array with default value
        var retArray = new int[s.Length];
        retArray[0] = 0;

        for (var i = 1; i < s.Length; ++i)
        {
            Logger.Log($"s[i={i}] => {s[i]}");
            // Take last prefix value
            var k = retArray[i-1];
            Logger.Log($"s[k={k}] => {s[k]}");
            while (k > 0 && s[i] != s[k])
            {
                // Reduce k value to previous in prefix
                k = retArray[k - 1];
                Logger.Log($"s[k={k}] => {s[k]}");
            }

            if (s[i] == s[k])
            {
                ++k;
                Logger.Log($"s[k={k}] == s[i={i}]. K increased to {k}");
            }
```

```
            retArray[i] = k;
            Logger.Log($"Current prefix function =>"
             "{string.Join("", retArray)}");
        }

        return retArray;
    }

    /// <summary>
    /// Calculate prefix-function for string
    /// </summary>
    /// <param name="s">String</param>
    /// <param name="patternPrefix">Prefix for pattern</param>
    /// <param name="lowerBound">Lower bound for thread</param>
    /// <param name="upperBound">Upper bound for thread</param>
    /// <returns>Value of prefix-function as array</returns>
    static int[] MultiThreadsPrefixFunction(string s,
            int[] patternPrefix,
            int lowerBound,
            int upperBound)
    {
        // Init with empty array
        var retArray = new int[upperBound - lowerBound];

        for (var i = lowerBound; i < upperBound; ++i)
        {
            Logger.Log($"s[i={i}] => {s[i]}");
            // Get value from concatenated string at i index
            var valueAtIIndex = s[i+patternPrefix.Length+1];

            // Take previous prefix value if exist
            var k = i - lowerBound > 0 ? retArray[i-lowerBound-1] : 0;
            Logger.Log($"s[k={k}] => {s[k]}");
            while (k > 0 && valueAtIIndex != s[k])
```

```
            {
                // Reduce k value to previous in prefix
                k = patternPrefix[k - 1];
                Logger.Log($"s[k={k}] => {s[k]}");
            }

            if (valueAtIIndex == s[k])
            {
                ++k;
            }

            // Save k in local array (not for whole string)
            retArray[i-lowerBound] = k;
            Logger.Log($"Current prefix function => "
            "{string.Join("", retArray)}");
        }

        return retArray;
    }

    /// <summary>
    /// Fix prefix values between end of thread and rest of pattern
    /// </summary>
    /// <param name="prefix">Calculated prefix for string</param>
    /// <param name="patternPrefix">Calculated pattern for prefix</param>
    /// <param name="s">String for search</param>
    /// <param name="threadsCount">Count of threads</param>
    /// <param name="countsPerThread">Count of elements per thread</param>
    static void FixPrefix(List<int> prefix,
        int[] patternPrefix,
        string s,
        int threadsCount,
        int countsPerThread)
    {
        for (var i = 1; i < threadsCount; ++i)
```

```
        {
            Logger.Log($"Thread => {i}", Logger.LogLevel.Debug);

            // Calc borders for fix prefix
            var lowerBound = countsPerThread * i - 1;
            var upperBound = lowerBound
                + patternPrefix.Length;
            upperBound = upperBound < s.Length ? upperBound : s.Length;

            // Calc prefix in new borders
            var localPrefix = MultiThreadsPrefixFunction(s,
                            patternPrefix,
                            lowerBound,
                            upperBound);
            Logger.Log($"New prefix for bounds[{lowerBound};{upperBound}] =>"
                " {string.Join("", localPrefix)}");
            for (var j = lowerBound+1; j < upperBound; ++j)
            {
                prefix[j] = localPrefix[j - lowerBound];
            }
            Logger.Log($"Current prefix function =>"
             "{string.Join("", prefix)}");
            Logger.Log("", Logger.LogLevel.Debug);
        }
    }

    /// <summary>
    /// Find indexes of pattern occurrences in string
    /// </summary>
    /// <param name="str">String</param>
    /// <param name="pattern">Pattern for search</param>
    /// <param name="threadsCount">Count of threads</param>
    /// <returns>List of indexes</returns>
    static List<int> FindPatternsOccurrences(string str,
                string pattern,
```

```
            int threadsCount)
{
    // Init data
    Logger.Log($"Thread => 1", Logger.LogLevel.Debug);
    var retArray = new List<int>();
    var strForPrefix = $"{pattern}#{str}";
    Logger.Log($"Concatenated string => {strForPrefix}");
    var prefix = new List<int>();
    var patternPrefix = NativePrefixFunction(pattern);
    Logger.Log($"Prefix for pattern => "
        "{string.Join("", patternPrefix)}");
    var countsPerThread = (int)(Math
        .Ceiling((double)str.Length / threadsCount));
    Logger.Log($"Count of indexes per thread =>"
        "{countsPerThread}", Logger.LogLevel.Debug);

    // Calc prefix by threads
    for (var i = 0; i < threadsCount; ++i)
    {
        Logger.Log("", Logger.LogLevel.Debug);
        Logger.Log($"Thread => {i}", Logger.LogLevel.Debug);
        var lowerBound = (countsPerThread * i);
        var upperBound = countsPerThread * (i+1);
        upperBound = upperBound < str.Length
            ? upperBound
            : str.Length;
        Logger.Log($"Bounds => [{lowerBound};{upperBound}]",
            Logger.LogLevel.Debug);
        prefix.AddRange(MultiThreadsPrefixFunction(strForPrefix,
                patternPrefix,
                lowerBound,
                upperBound));
        Logger.Log("", Logger.LogLevel.Debug);
    }
```

```
        // Recalc prefix if in multithreads
        if (countsPerThread != prefix.Count)
        {
            FixPrefix(prefix,
                patternPrefix,
                strForPrefix,
                pattern.Length,
                countsPerThread);
        }

        // Collect result
        for (var i = 0; i < threadsCount; ++i)
        {
            Logger.Log("", Logger.LogLevel.Debug);
            Logger.Log($"Thread => {i}", Logger.LogLevel.Debug);
            var lowerBound = (countsPerThread * i);
            var upperBound = countsPerThread * (i+1);
            upperBound = upperBound < str.Length ? upperBound : str.Length;
            Logger.Log($"Bounds => [{lowerBound};{upperBound}]",
                Logger.LogLevel.Debug);
            Logger.Log("", Logger.LogLevel.Debug);
            retArray.AddRange(FindPatternsOccurrencesInPrefix(prefix,
                    pattern.Length,
                    lowerBound, upperBound));
            Logger.Log("", Logger.LogLevel.Debug);
        }

        Logger.Log("", Logger.LogLevel.Debug);
        return retArray;
    }

    /// <summary>
    /// Find indexes of patterns occurrences in prefix-function
    /// </summary>
    /// <param name="prefix">Prefix-function</param>
```

```
/// <param name="patternLength">Length of pattern string</param>
/// <param name="lowerBound">Lower bound</param>
/// <param name="upperBound">Upper bound</param>
/// <returns></returns>
static List<int> FindPatternsOccurrencesInPrefix(IReadOnlyList<int> prefix,
        int patternLength,
        int lowerBound,
        int upperBound)
{
    var retArray = new List<int>();
    for (var i = lowerBound; i < upperBound; ++i)
    {
        if (prefix[i] != patternLength) continue;
        var resolvedIndex = i - patternLength + 1;
         Logger.Log($"Found value with {patternLength} at =>"
          "{resolvedIndex}", Logger.LogLevel.Debug);
        retArray.Add(resolvedIndex);
    }

    return retArray;
}

static void Main(string[] args)
{
    var pattern = Console.ReadLine();
    var str = Console.ReadLine();
    var threadsCount = int.Parse(Console.ReadLine());
    Logger.Log($"Pattern value => {pattern}",
            Logger.LogLevel.Debug);
    Logger.Log($"String value => {str}",
            Logger.LogLevel.Debug);
    Logger.Log($"Count of threads value => {threadsCount}",
            Logger.LogLevel.Debug);
    var result = FindPatternsOccurrences(str, pattern, 1);
    Logger.Log(result.Any() ? string.Join(",", result) : "-1");
```

```
        }
    }
}
```

Нахождение циклического сдвига

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

/// Return char that appearance at {pattern#strstr}
/// \param i index
/// \param pattern first string
/// \param str second str
/// \return char
char charFromPatternOrStr(int i,
        const std::string& pattern,
        const std::string& str)
{
    if (i < pattern.length()) // pattern part
    {
        return pattern[i];
    }

    if (i == pattern.length()) // separator
    {
        return '#';
    }

    if (i < pattern.length() + str.length() + 1) // first string part
    {
        return str[i - pattern.length() - 1];
    }
```

```
    return str[i - pattern.length() - str.length() - 1]; // second string part
}

/// Calculate prefix-function for {pattern#strstr}
/// \param pattern first string
/// \param str second str
/// \return prefix-function as vector of int
std::vector<int> prefixFunction(const std::string& pattern,
            const std::string& str)
{
#ifndef NDEBUG
    std::cout << "Calc prefix-function for => "
              << pattern
              << "#"
              << str
              << str
              << std::endl;
#endif
    int arraySize = pattern.length() + str.length() + str.length() + 1;
    std::vector<int> retVec(arraySize);
    retVec[0] = 0;

    for (int i = 1; i < arraySize; ++i)
    {
        int k = retVec[i - 1]; // take k
        char iChar = charFromPatternOrStr(i, pattern, str);
        char kChar = charFromPatternOrStr(k, pattern, str);
#ifndef NDEBUG
        std::cout << std::endl;
        std::cout << "i value => " << i << std::endl;
        std::cout << "Step => " << i << std::endl;
        std::cout << "str[i=" << i << "] => " << iChar << std::endl;
        std::cout << "str[k=" << k << "] => " << kChar << std::endl;
        std::cout << "k value => "
                  << k
```

```
                << " at prefix[i-1="
                << i - 1
                << "]"
                << std::endl;
#endif
        while (k > 0 && iChar != kChar)
        {
            k = retVec[k - 1]; // decrease k
            kChar = charFromPatternOrStr(k, pattern, str);
#ifndef NDEBUG
            std::cout << "\tk index was decreased to => "
                      << k
                      << "at prefix["
                      << i - 1
                      << "]"
                      << std::endl;
            std::cout << "\tstr[k] value => " << kChar << std::endl;
#endif
        }

        if (iChar == kChar)
        {
            ++k; // increase k
#ifndef NDEBUG
            std::cout << "Found equals chars."
                         " k index was increased to => "
                      << k << std::endl;
#endif
        }

        retVec[i] = k; // save k

#ifndef NDEBUG
        std::cout << "Prefix value => ";
        for (int i = 0; i < retVec.size(); ++i)
```

```
        {
            std::cout << retVec[i];
        }
        std::cout << std::endl;
#endif
    }

    return retVec; // return array
}

/// Find first pattern length occurrence in specified range in prefix-function
/// \param patternLength length of pattern
/// \param prefix prefix-function
/// \return first pattern occurrence
int findPatternsOccurenciesInPrifix(int patternLength,
            const std::vector<int>& prefix)
{
    for (int i = 2 * patternLength; i >= 0; --i)
    {
        if (prefix[patternLength + i + 1] == patternLength) // index of offset
        {
#ifndef NDEBUG
            std::cout << "Found answer at "
                      << "prefix[patternLength+i+1"
                      << patternLength + i + 1
                      << "]" << std::endl;
#endif
            return i - patternLength + 1;
        }
    }

    return -1;
}

/// Find index of cyclic offset second string in first one
```

```
/// \param str first string
/// \param pattern second string
/// \return index of offset
int findIndexOfCyclicOffset(const std::string& str, const std::string& pattern)
{
    int result = -1;
    // calc prefix function
    std::vector<int> prefix = prefixFunction(pattern, str);
#ifndef NDEBUG
    std::cout << "prefix => ";
    for (int i : prefix)
    {
        std::cout << i;
    }
    std::cout << std::endl;
#endif

    result = findPatternsOccurenciesInPrifix(pattern.length(), prefix);

    return result != -1 ? str.length() - result : result;
}

int main()
{
    std::string stringA;
    std::cin >> stringA; // read str
    std::string stringB;
    std::cin >> stringB; // read str
    bool isStringsEqualsBySize = stringA.length() == stringB.length();

    if (!isStringsEqualsBySize)
    {
        std::cout << "String have different size" << stringA.length() << " != " <<
    }
```

```
    int result = isStringsEqualsBySize
        ? findIndexOfCyclicOffset(stringB, stringA)
        : -1;

    std::cout << result << std::endl; // print result
}
```