

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. 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 в несколько потоков (их количество задаётся пользователем). Количество потоков обозначим за k . На каждый поток выделяется $n = \text{длина строки} / \text{количество потоков}$ 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 $\text{длина строки} - n * k$ 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 двух вложенных циклов в реализации. Это происходит потому, что итоговое количество определяется количеством итераций

внутреннего цикла. Которе, в свою очередь определяется максимально возможным уменьшением счётчика, которое не превосходит $O(p + s)$.

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 задаче. 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 реализации алгоритма не использовались.

Используемые функции (методы)

- *static int[] PrefixFunction(string str)* - используется для вычисления префикс-функции для строки
string str - строка для поиска;
- *static List FindPatternsOccurrences(string str, string pattern)* - 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;
- *static List FindPatternsOccurrencesInPrefix(string str, int patternLength, IReadOnlyList prefix, int start, int end)* - используется для параллельного нахождения необходимых значений(индексов вхождений) в префикс-функции на заданном промежутке
string str - строка для поиска;
int patternLength - длина строки образца;
ReadOnlyList prefix - массив значений префикс-функции;
int start - начальный индекс диапазона для поиска;
int end - конечный индекс диапазона для поиска;

Нахождение циклического сдвига

Структуры данных Пользовательские классы, использующиеся в реализации 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(индекса 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

Calc prefix-function for ab#abab

Step => 1

str[i=1] => b

k value => 0 at prefix[0]

str[k] value => a

Prefix value => 0000000

Step => 2

str[i=2] => #

k value => 0 at prefix[1]

str[k] value => a

Prefix value => 0000000

Step => 3

str[i=3] => a

k value => 0 at prefix[2]

str[k] value => a

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

Prefix value => 0001000

Step => 4

str[i=4] => b

k value => 1 at prefix[3]

str[k] value => b

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

Prefix value => 0001200

Step => 5

str[i=5] => a

k value => 2 at prefix[4]

str[k] value => #

k index was decreased to => 0

str[k] value => a

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

Prefix value => 0001210

Step => 6

str[i=6] => b

k value => 1 at prefix[5]

str[k] value => b

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

Prefix value => 0001212

Count of indexes per thread => 4

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 : ab

abab

2

Pattern value => ab

String value => abab

Count of threads value => 2

Calc prefix-function for ab#abab

Step => 1

str[i=1] => b

k value => 0 at prefix[0]

str[k] value => a

Prefix value => 0000000

Step => 2

str[i=2] => #

k value => 0 at prefix[1]

str[k] value => a

Prefix value => 0000000

Step => 3

str[i=3] => a

k value => 0 at prefix[2]

str[k] value => a

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

Prefix value => 0001000

Step => 4

str[i=4] => b

k value => 1 at prefix[3]

str[k] value => b

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

Prefix value => 0001200

Step => 5

str[i=5] => a

k value => 2 at prefix[4]

str[k] value => #

k index was decreased to => 0

str[k] value => a

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

Prefix value => 0001210

Step => 6

str[i=6] => b

k value => 1 at prefix[5]

str[k] value => b

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

Prefix value => 0001212

Count of indexes per thread => 2

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 2

Output: 0,2

Тест 4: 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 => 00000000000000

i value => 2

Step => 2

str[i=2] => 3

str[k=0] => 1

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

Prefix value => 00000000000000

i value => 3

Step => 3

str[i=3] => 4
str[k=0] => 1
k value => 0 at prefix[i-1=2]
Prefix value => 00000000000000

i value => 4
Step => 4
str[i=4] => #
str[k=0] => 1
k value => 0 at prefix[i-1=3]
Prefix value => 00000000000000

i value => 5
Step => 5
str[i=5] => 2
str[k=0] => 1
k value => 0 at prefix[i-1=4]
Prefix value => 00000000000000

i value => 6
Step => 6
str[i=6] => 3
str[k=0] => 1
k value => 0 at prefix[i-1=5]
Prefix value => 00000000000000

i value => 7
Step => 7
str[i=7] => 4
str[k=0] => 1
k value => 0 at prefix[i-1=6]
Prefix value => 00000000000000

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 => 0 at 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+111]

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 программ 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;
using System.Threading;

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
    /// </summary>
    /// <param name="str">String</param>
    /// <returns>Value of prefix-function as array</returns>
    static int[] PrefixFunction(string str)
    {
        Logger.Log($"Calc prefix-function for {str}",
            Logger.LogLevel.Debug);
        var retArray = new int[str.Length]; // returnable array
        retArray[0] = 0;

        for (var i = 1; i < str.Length; ++i)
        {
            Logger.Log("", Logger.LogLevel.Debug);
            Logger.Log($"Step => {i}", Logger.LogLevel.Debug);
            Logger.Log($"str[i]={i}] => {str[i]}", Logger.LogLevel.Debug);
            var k = retArray[i - 1]; // take last k
            Logger.Log($"k value => {k} at prefix[{i - 1}]",
                Logger.LogLevel.Debug);
            Logger.Log($"str[k] value => {str[k]}", Logger.LogLevel.Debug);
            while (k > 0 && str[i] != str[k])
            {
                k = retArray[k - 1]; // decrease k
                Logger.Log($"tk index was decreased to => {k}",
                    Logger.LogLevel.Debug);
                Logger.Log($"tstr[k] value => {str[k]}",
                    Logger.LogLevel.Debug);
            }
        }
    }
}

```

```

    }

    if (str[i] == str[k])
    {
        ++k; // increase k (don't tell the elf)
        Logger.Log($"Found equals chars."
            "k index was increased to => {k}",
            Logger.LogLevel.Debug);
    }

    retArray[i] = k; // save k
    Logger.Log($"Prefix value => {string.Join("", retArray)}",
        Logger.LogLevel.Debug);
}

return retArray; // return (again, don't tell the elf)
}

/// <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)
{
    var retArray = new List<int>();
    var prefix = PrefixFunction($"{pattern}#{str}"); // calc prefix
    Logger.Log("", Logger.LogLevel.Debug);
    var countsPerThread = (int)(Math.Floor((double)str.Length / threadsCount));
    Logger.Log($"Count of indexes per thread => {countsPerThread}",
        Logger.LogLevel.Debug);
}

```

```

    Logger.Log("", Logger.LogLevel.Debug);

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

    var upperBound = countsPerThread * threadsCount; // calc upper bound
    retArray.AddRange(FindPatternsOccurrencesInPrefix(str,
        pattern.Length,
        prefix,
        upperBound,
        str.Length));

    return retArray; // return (and again, don't tell the elf)
}

/// <summary>
/// Find indexes of patterns occurrences in prefix-function
/// On specified range
/// </summary>
/// <param name="str">String</param>
/// <param name="patternLength">Length of pattern string</param>
/// <param name="prefix">Prefix-function</param>
/// <param name="start">Start of range for search</param>
/// <param name="end">End of range for search</param>
/// <returns></returns>

```

```

static List<int> FindPatternsOccurrencesInPrefix(string str,
    int patternLength,
    IReadOnlyList<int> prefix,
    int start,
    int end)
{
    var retArray = new List<int>();
    for (var i = start; i < end; ++i)
    {
        if (prefix[patternLength + i + 1] == patternLength)
        {
            Logger.Log($"Found value with {patternLength} at => "
                + "{i - patternLength + 1}",
                Logger.LogLevel.Debug);
            retArray.Add(i - patternLength + 1);
        }
    }

    return retArray; // return (and again, don't tell the elf)
}

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, threadsCount);
    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);
#ifdef 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
#ifdef NDEBUG
        std::cout << "Found equals chars."
            << " k index was increased to => "
            << k << std::endl;
#endif
    }

    retVec[i] = k; // save k

#ifdef 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
        {
#ifdef 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

    int result = stringA.length() == stringB.length()
        ? findIndexOfCyclicOffset(stringB, stringA)
        : -1;

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

```