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: Кнут-Моррис-Пратт

Студентка гр. 3343 _____ Какира Умар.

Преподаватель _____ жангиров Т.

Санкт-Петербург

2025

Цель работы.

Изучить и реализовать алгоритм Кнута-Морриса-Пратта для поиска всех подстрок по шаблону. Реализовать алгоритм проверки на циклический сдвиг.

Постановка задачи.

1 пункт. 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

2 пункт. 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.

Описание алгоритма.

Математически определение префикс-функции можно записать следующим образом: $\pi[] = \underset{n=0\ldots i}{}\{ : [0 \ldots -1] = [- +1 \ldots]\}$, где s[0 … n-1] - данная строка.

Алгоритм Кнута-Морриса-Пратта, который находит позиции всех вхождений строки P в текст T, работает следующим образом.

Построим строку S=P#T, где # - любой символ, не входящий в алфавит P и T. Посчитаем на ней значение префикс-функции p. Если в какой-то позиции i выполняется условие p[i]=|P|, то в этой позиции начинается очередное вхождение образца в цепочку.

Алгоритм поиска циклического сдвига отличается только тем, что ведется в удвоенной первой строке, так как при сложении строк первая будет содержать в себе вторую строку, если она является циклическим сдвигом.

Алгоритм Кнута-Морриса-Пратта имеет сложность O(P+T) по времени и памяти.

Описание структур и функций.

- vector < int > CalculatingPrefixFunction (string Line) - формирует вектор значений префикс функции символов входной строки и возвращает его;
- void KnuthMorrisPratt (string FirstLine, string SecondLine, vector <int>&Result) - функция реализует алгоритм Кнута-Морриса-Пратта;
- int IsCyclicShift (string FirstLine, string SecondLine) - функция проверяет является ли строка FirstLine, циклическим сдвигом строки SecondLine;

Код разработанных программ см. в приложении А.

Тестирование.

Таблица 1 - результаты тестирования lb4_1.cpp

Тест	Входные данные	результат работы алгоритма
№1	ab abab	0,2
№2	nnfjfkc	-1

	kgykg	
№3	asasas asasas	0

Таблица 2 - результаты тестирования lb4_2.cpp

Тест	Входные данные	результат работы алгоритма
№1	qwerty tyqwer	4
№2	htj uky	-1
№3	qweryt tyqwer	-1

Вывод.

В ходе выполнения работы были изучены алгоритм Кнута - Морриса - Пратта для поиска всех подстрок и префикс функция. Также алгоритм Кнута - Морриса - Пратта был оптимизирован для решения задачи поиска циклического сдвига. Данные алгоритмы были реализованы на языке программирования C++.

ПРИЛОЖЕНИЕ А
Исходный код программы

Название файла: **lb4_1.cpp**

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

using namespace std;

// Computes the prefix function (partial match table) for the KMP algorithm
// Input: String line (pattern to preprocess)
// Output: Vector of prefix lengths for each position in the pattern
vector<int> CalculatingPrefixFunction(string line)
{
    int lineLength = line.length();
    vector<int> prefixes(lineLength);  // Initialize prefix table
    prefixes[0] = 0;   // Base case: first character has prefix length 0

    // Build prefix table for each position in the string
    for (int i = 1; i < lineLength; i++)
    {
        // Start with prefix length of previous character
        int ActualLineLength = prefixes[i - 1];

        // While we have a partial match and current characters don't match,
        // backtrack using the prefix table
        while (ActualLineLength > 0 fifi (line[ActualLineLength] != line[i]))
            ActualLineLength = prefixes[ActualLineLength - 1];

        // If characters match, extend the prefix length
        if (line[ActualLineLength] == line[i])
            ActualLineLength++;

        // Store the computed prefix length for current position
        prefixes[i] = ActualLineLength;
    }
    return prefixes;
}

// KMP pattern matching algorithm to find all occurrences of FirstLine in SecondLine
// Input:
//   FirstLine - pattern to search for
//   SecondLine - text to search in
// Output:
//   Result - vector containing starting indices of all matches
void KnuthMorrisPratt(string FirstLine, string SecondLine, vector<int>fi Result)
{
    // Compute prefix function for pattern with special delimiter '#'
    vector<int> p = CalculatingPrefixFunction(FirstLine + "#");
    int FirstLineStep = 0;   // Current position in pattern

    // Iterate through each character in the text
    for (int SecondLineStep = 0; SecondLineStep < SecondLine.size(); ++SecondLineStep)
    {
        // While mismatch occurs, use prefix table to skip ahead
        while (FirstLineStep > 0 fifi FirstLine[FirstLineStep] !=
SecondLine[SecondLineStep])
            FirstLineStep = p[FirstLineStep - 1];

        // If characters match, move to next character in pattern
        if (FirstLine[FirstLineStep] == SecondLine[SecondLineStep])
            FirstLineStep++;

        // If entire pattern matched, record the starting position
        if (FirstLineStep == FirstLine.size())
            Result.push_back(SecondLineStep - FirstLine.size() + 1);
    }
```

```
}

int main()
{
    vector<int> Result; // Stores starting positions of all matches
    string Firstline, Secondline;

    // Read input strings
    cin >> Firstline;    // Pattern to search for
    cin >> Secondline; // Text to search in

    // Perform KMP search
    KnuthMorrisPratt(Firstline, Secondline, Result);

    // Output results
    if (!Result.size())
        cout << -1;   // No matches found
    else
    {
        // Print comma-separated list of match positions
        string separator;
        for (auto entry : Result)
        {
            cout << separator << entry;
            separator = ",";    // Only add commas after first element
        }
    }
    return 0;

 }
```

Название файла: **lb4_2.cpp**

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

using namespace std;

// Function to compute the prefix function (partial match table) for KMP algorithm
// This helps in skipping unnecessary comparisons during string matching
vector < int > CalculatingPrefixFunction(string line)
{
    int linelength = line.length();
    vector < int >prefixes(linelength);     // Create prefix table of same length as
string
    prefixes[0] = 0;   // First character always has prefix value 0

    // Compute prefix values for each position in the string
    for (int i = 1; i < linelength; i++)
    {
        // Start with prefix length of previous character
        int Actuallinelength = prefixes[i - 1];

        // While we have a partial match and characters don't match,
        // backtrack using the prefix table
        while (Actuallinelength > 0
            fifi (line[Actuallinelength] != line[i]))
            Actuallinelength = prefixes[Actuallinelength - 1];

        // If characters match, extend the prefix length
        if (line[Actuallinelength] == line[i])
            Actuallinelength++;

        // Store the computed prefix length
        prefixes[i] = Actuallinelength;
    }
    return prefixes;
```

```
}

// Function to check if Secondline is a cyclic shift of Firstline
// Uses KMP algorithm on Firstline concatenated with itself
void IsCyclicShift(string Firstline, string Secondline, vector < int >fi Result)
{
    // Create doubled version of Firstline to check all possible cyclic shifts
    Firstline = Firstline + Firstline;

    // Compute prefix function for Secondline (with special delimiter '#')
    vector<int> Prefixes = CalculatingPrefixFunction(Secondline + "#");
    int FirstlineStep = 0;    // Tracks position in Secondline (pattern)

    // Search through doubled Firstline (text)
    for (int SecondlineStep = 0; SecondlineStep < Firstline.size(); ++SecondlineStep)
    {
        // While mismatch occurs, use prefix table to skip ahead
        while (FirstlineStep > 0 fifi Secondline[FirstlineStep] !=
Firstline[SecondlineStep])
            FirstlineStep = Prefixes[FirstlineStep - 1];

        // If characters match, move to next character in pattern
        if (Secondline[FirstlineStep] == Firstline[SecondlineStep])
            FirstlineStep++;

        // If full pattern matched, store the starting position
        if (FirstlineStep == Secondline.size())
        {
            Result.push_back(SecondlineStep - Secondline.size() + 1);
            if (true) break;   // Exit after first match found
        }
    }
}

int main()
{
    vector < int >Result;   // Stores starting positions of matches
    string Firstline, Secondline;
    cin >> Firstline;
    cin >> Secondline;

    // Only check if strings are same length (prerequisite for cyclic shift)
    if (Secondline.size() == Firstline.size()) IsCyclicShift(Firstline, Secondline,
Result);

    // Output results
    if (!Result.size())
        cout << -1;    // No match found
    else
    {
        // Print comma-separated list of match positions
        string separator;
        for (auto entry : Result)
        {
            cout << separator << entry;
            separator = ",";
        }
    }
    return 0;

 }
```