**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*∣≤15000) и текста *T*(∣*T*∣≤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

1. **пункт.** Заданы две строки *A*(∣*A*∣≤5000000) и *B*(∣*B*∣≤5000000). Определить, является ли А циклическим сдвигом В (это значит, что А и В имеют одинаковую длину и А состоит из суффикса В, склеенного с префиксом В). 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: π[] = n=0...i{ : [0 … − 1] = [ − + 1 … ]}, 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 С++.

**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 Įine (pattern to preprocess)

// Output: Vector of prefix lengths for each position in the pattern vector<int> CalculatingPrefixFunction(string Įine)

{

int ĮineĮength = Įine.length();

vector<int> prefixes(ĮineĮength); // 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 < ĮineĮength; i++)

{

// Start with prefix length of previous character int ActualĮineĮength = prefixes[i - 1];

// While we have a partial match and current characters don't match,

// backtrack using the prefix table

while (ActualĮineĮength > 0 fifi (Įine[ActualĮineĮength] != Įine[i])) ActualĮineĮength = prefixes[ActualĮineĮength - 1];

// If characters match, extend the prefix length if (Įine[ActualĮineĮength] == Įine[i])

ActualĮineĮength++;

// Store the computed prefix length for current position prefixes[i] = ActualĮineĮength;

}

return prefixes;

}

// KMP pattern matching algorithm to find all occurrences of FirstĮine in SecondĮine

// Input:

// FirstĮine - pattern to search for

// SecondĮine - text to search in

// Output:

// Result - vector containing starting indices of all matches

void KnuthMorrisPratt(string FirstĮine, string SecondĮine, vector<int>fi Result)

{

// Compute prefix function for pattern with special delimiter '' vector<int> p = CalculatingPrefixFunction(FirstĮine + "");

int FirstĮineStep = 0; // Current position in pattern

// Iterate through each character in the text

for (int SecondĮineStep = 0; SecondĮineStep < SecondĮine.size(); ++SecondĮineStep)

{

// While mismatch occurs, use prefix table to skip ahead while (FirstĮineStep > 0 fifi FirstĮine[FirstĮineStep] !=

SecondĮine[SecondĮineStep])

FirstĮineStep = p[FirstĮineStep - 1];

// If characters match, move to next character in pattern if (FirstĮine[FirstĮineStep] == SecondĮine[SecondĮineStep])

FirstĮineStep++;

// If entire pattern matched, record the starting position if (FirstĮineStep == FirstĮine.size())

Result.push\_back(SecondĮineStep - FirstĮine.size() + 1);

}

}

int main()

{

vector<int> Result; // Stores starting positions of all matches string FirstĮine, SecondĮine;

// Read input strings

cin >> FirstĮine; // Pattern to search for cin >> SecondĮine; // Text to search in

// Perform KMP search KnuthMorrisPratt(FirstĮine, SecondĮine, Result);

// Output results if (!Result.size())

cout << -1; // o 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 Įine)

{

int ĮineĮength = Įine.length();

vector < int >prefixes(ĮineĮength); // 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 < ĮineĮength; i++)

{

// Start with prefix length of previous character int ActualĮineĮength = prefixes[i - 1];

// While we have a partial match and characters don't match,

// backtrack using the prefix table while (ActualĮineĮength > 0

fifi (Įine[ActualĮineĮength] != Įine[i])) ActualĮineĮength = prefixes[ActualĮineĮength - 1];

// If characters match, extend the prefix length if (Įine[ActualĮineĮength] == Įine[i])

ActualĮineĮength++;

// Store the computed prefix length prefixes[i] = ActualĮineĮength;

}

return prefixes;

}

// Function to check if SecondĮine is a cyclic shift of FirstĮine

// Uses KMP algorithm on FirstĮine concatenated with itself

void IsCyclicShift(string FirstĮine, string SecondĮine, vector < int >fi Result)

{

// Create doubled version of FirstĮine to check all possible cyclic shifts FirstĮine = FirstĮine + FirstĮine;

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

// Search through doubled FirstĮine (text)

for (int SecondĮineStep = 0; SecondĮineStep < FirstĮine.size(); ++SecondĮineStep)

{

// While mismatch occurs, use prefix table to skip ahead while (FirstĮineStep > 0 fifi SecondĮine[FirstĮineStep] !=

FirstĮine[SecondĮineStep])

FirstĮineStep = Prefixes[FirstĮineStep - 1];

// If characters match, move to next character in pattern if (SecondĮine[FirstĮineStep] == FirstĮine[SecondĮineStep])

FirstĮineStep++;

// If full pattern matched, store the starting position if (FirstĮineStep == SecondĮine.size())

{

Result.push\_back(SecondĮineStep - SecondĮine.size() + 1); if (true) break; // Exit after first match found

}

}

}

int main()

{

vector < int >Result; // Stores starting positions of matches string FirstĮine, SecondĮine;

cin >> FirstĮine; cin >> SecondĮine;

// Only check if strings are same length (prerequisite for cyclic shift)

if (SecondĮine.size() == FirstĮine.size()) IsCyclicShift(FirstĮine, SecondĮine, Result);

// Output results if (!Result.size())

cout << -1; // o match found else

{

// Print comma-separated list of match positions string separator;

for (auto entry : Result)

{

cout << separator << entry; separator = ",";

}

}

return 0;

}