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. 9382 _____ Кодуков А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы:

Изучить и использовать на практике алгоритм Кнута-Морисса-Пратта.

Задание 1:

Реализуйте алгоритм КМП и с его помощью для заданных шаблона PP ($|P| \leq 15000$) 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

Задание 2:

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 вляется 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

Описание алгоритма:

Расчет префикс-функции:

Префикс-функцией для позиции *i* строки *S* является длина наибольшего префикса подстроки S[1..i] равному постфиксу этой подстроки. Префикс-функция первой позиции считается равной нулю. Тогда, если $\pi(S, i) = k$, можно задать рекуррентный алгоритм для расчета префикс-функции для i+1 позиции:

- Если $S[i+1] = S[k+1] => \pi(S, i+1) = k+1$ (совпадает k первых и последних символов, и также совпал k + 1 символ)
- Иначе
  - $k = 0 => \pi(S, i+1) = 0$
  - $k := \pi(S, k)$, вернуться к п. 1

(индексация велась от 1)

Алгоритм Кнута-Морисса-Пратта:

Вычисляется префикс-функция для образца, который необходимо найти в тексте. Затем симулируется расчет префикс функции для склеенной строки, состоящей из образца и текста, учитывая уже рассчитанные префикс функции. Каждый раз при получении префикс-функции для позиции *j* равной длине образца, сохраняем текущую позицию, вычитая длину образца, в ответ, так как по определению префикс функции это означает, что подстрока оказалась равна префиксу, который и явлвяется образцом.

Сложность алгоритмов

Префикс-функция образца вычисляется в худшем случае за *O(2 * <длина образца>)*, так как случаев, где *k = 0* и не изменяется, и случаев, где *k* увеличивается на 1, не более <длина строки> - 1 штук. Так как k может увеличиваться только на 1, то суммарно уменьшений может быть не больше <длина строки> - 2 штук. Получаем *O(2 * <длина строки>)* по времени. Сам алгоритм проводит столько же итераций над текстом, поэтому итоговая сложность по времени *O(2 * (<длина текста> + <длина образца))*.

Хранить требуется только префикс-функцию для образца, поэтому сложность по памяти *O(<длина образца>)*.

Функции и структуры данных:

Реализованные функции:

Алгоритм Кнута-Морисса-Прата

Сигнатура: `std::vector<long long> KMP(const std::string &str, const std::string &pattern)`

Аргументы:

- str – тексе
- pattern – фрагмент

Возвращаемое значение:

- `std::vector<long long>` массив индексов вхождений фрагмента в текст

Алгоритм:

- Рассчитать префикс-функцию для паттерна
- Найти вхождения как суффикс к паттерну в начале строки с текстом

Тестирование:

Задание 1:

№	Входные данные	Вывод
1	ab abab	0, 2
2	zxc zxctuchkapauzazxc	0, 14
3	asd asd	0
4	bl AXAXAXAXAXA	-1

5	asdfg zxcasdfa	-1

Задание 2:

№	Входные данные	Вывод
1	defabc abcdef	3
2	asdfgh fgaasd	-1
3	a asd	-1
4	XAXAXAXAXA AXAXAXAXAX	1
5	aaaaaabaaaaa aaaabaaaaaaa	10

Вывод:

В результате выполнения работы был изучен и реализован алгоритм Кнута-Морисса-Пратта.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

first_step.cpp

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

//#define DEBUG

// Knuth–Morris–Pratt algorithm
std::vector<long long> KMP(const std::string &str, const std::string &pattern) {
  long long i, k, len;
  len = pattern.length();
  std::vector<long long> d, res;
  d.resize(len + 1);

  // Prefix-function of pattern
#ifdef DEBUG
  std::cout << "***Computing prefix-function for pattern***\n\n";
#endif
  d[0] = 0;  // pi(str, 0) = 0 by definition
  for (i = 1, k = 0; i < len; i++) {
#ifdef DEBUG
    std::cout << "i = " << i << " k = " << k << " ";
    std::cout << "S[" << i << "] = " << pattern[i];
#endif
    // decrease k back if symbols aren't equal
    while (k > 0 && pattern[k] != pattern[i]) {
#ifdef DEBUG
      std::cout << " != "
                << "S[" << k << "] = " << pattern[k]
                << " -> pf[i] = k = pf[k - 1]\n";
#endif
      k = d[k - 1];
    }
    // increase k if symbols are equal
    if (pattern[k] == pattern[i]) {
#ifdef DEBUG
      std::cout << " == "
                << "S[" << k << "] = " << pattern[k] << " -> pf[i] = k + 1\n";
#endif
      k++;
    } else {
#ifdef DEBUG
      std::cout << "\n";
#endif
    }
    // save prefix-function for position i
    d[i] = k;
  }

#ifdef DEBUG
  std::cout << "***Searching pattern in text***\n\n";
#endif
  // Search in text
  for (i = 0, k = 0; i < str.length(); i++) {
    // decrease k back if symbols aren't equal
#ifdef DEBUG
    std::cout << "i = " << i << " k = " << k << " ";
    std::cout << "S[" << i << "] = " << str[i];
```

```
#endif
    while (k > 0 && pattern[k] != str[i]) {
#ifdef DEBUG
      std::cout << " != "
                << "S[" << k << "] = " << pattern[k]
                << " -> pf[i] = k = pf[k - 1]\n";
#endif
      k = d[k - 1];
    }
    // increase k if symbols are equal
    if (pattern[k] == str[i]) {
#ifdef DEBUG
      std::cout << " == "
                << "S[" << k << "] = " << pattern[k] << " -> pf[i] = k + 1\n";
#endif
      k++;
    } else {
#ifdef DEBUG
      std::cout << "\n";
#endif
    }

    // suffix equal to pattern was found
    if (k == len) {
#ifdef DEBUG
      std::cout << "k == length of pattern, entry was found\n";
#endif
      res.push_back(i - k + 1);
    }
  }
  return res;
}

int main() {
  std::string str, pattern;

  // get strings
  std::getline(std::cin, pattern);
  std::getline(std::cin, str);
  #ifdef DEBUG
    std::cout << "Text:    " << str << "\n";
    std::cout << "Pattern: " << pattern << "\n\n";
  #endif
  // search pattern in text
  std::vector<long long> res = KMP(str, pattern);
  if (res.size() > 0) {
    for (int i = 0; i < res.size() - 1; i++) std::cout << res[i] << ",";
    std::cout << res.back();
  } else
    std::cout << "-1";
  return 0;
}
```

second_step.cpp

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

//#define DEBUG

// Knuth-Morris-Pratt algorithm
long long KMP(const std::string &str, const std::string &pattern) {
  long long i, k, len;
```

```
  len = pattern.length();

  std::vector<long long> d, res;
  d.resize(len + 1);

  // Prefix-function of pattern
#ifdef DEBUG
  std::cout << "***Computing prefix-function for pattern***\n\n";
#endif
  d[0] = 0; //pi(str, 0) = 0 by definition
  for (i = 1, k = 0; i < len; i++) {
#ifdef DEBUG
    std::cout << "i = " << i << " k = " << k << " ";
    std::cout << "S[" << i << "] = " << pattern[i];
#endif
    // decrease k back if symbols aren't equal
    while (k > 0 && pattern[k] != pattern[i]) {
#ifdef DEBUG
      std::cout << " != " << "S[" << k << "] = " << pattern[k] << " -> pf[i] = k
= pf[k - 1]\n";
#endif
      k = d[k - 1];
    }
    // increase k if symbols are equal
    if (pattern[k] == pattern[i]) {
#ifdef DEBUG
      std::cout << " == " << "S[" << k << "] = " << pattern[k] << " -> pf[i] = k
+ 1\n";
#endif
      k++;
    } else {
#ifdef DEBUG
      std::cout << "\n";
#endif
    }
    // save prefix-function for position i
    d[i] = k;
  }

#ifdef DEBUG
  std::cout << "***Searching cyclic shift***\n\n";
#endif
  // Search in text
  for (i = 0, k = 0; i < str.length(); i++) {
    // decrease k back if symbols aren't equal
#ifdef DEBUG
    std::cout << "i = " << i << " k = " << k << " ";
    std::cout << "S[" << i << "] = " << str[i];
#endif
    while (k > 0 && pattern[k] != str[i]) {
#ifdef DEBUG
      std::cout << " != "
                << "S[" << k << "] = " << pattern[k]
                << " -> pf[i] = k = pf[k - 1]\n";
#endif
      k = d[k - 1];
    }
    // increase k if symbols are equal
    if (pattern[k] == str[i]) {
#ifdef DEBUG
      std::cout << " == "
                << "S[" << k << "] = " << pattern[k] << " -> pf[i] = k + 1\n";
#endif
```

```
      k++;
    } else {
#ifdef DEBUG
      std::cout << "\n";
#endif
    }
    // suffix equal to pattern was found
    if (k == len) {
#ifdef DEBUG
      std::cout << "k = " << k << " == length of pattern, cyclic shift was
found\n";
#endif
      return i - k + 1;
    }
  }
  return -1;
}

int main() {
  std::string str, pattern;

  // get strings
  std::getline(std::cin, pattern);
  std::getline(std::cin, str);
  // cyclic shift must be same length
  if (str.length() != pattern.length()) {
    std::cout << "-1";
  } else {
    // search pattern in doubled text to find cyclic shift
    long long res = KMP(str + str, pattern);
    std::cout << res;
  }
  return 0;
}
```