**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: Поиск набора подстрок в строке

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Иванов Д. М. |
| Преподаватель |  | Виноградова Е. В. |

Санкт-Петербург

2025

## Задание

Задание состоит из двух задач.

1) Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T, 1 <= |T| <= 100000).

Вторая – число n (1 <= n <= 3000), каждая следующая из n строк содержит шаблон из набора P = {p1, …, pn} 1<=|pi|<=75.

Все строки содержат символы из алфавита {A, C, G, T, N}.

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

2) Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения Р в текст Т.

Например, образец аb??с? с джокером ? встречается дважды в тексте xabvccbababcax.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита {A, C, G, T, N}.

Индивидуализация: Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

**Описание алгоритма**

Рассмотрим отдельно алгоритмы для каждого из заданий.

Задание 1

Для реализации алгоритма Ахо-Красика нужно создать Trie-дерево или так называемый бор. Это структура данных, которая позволяет компактно хранить строки. Узлами дерева будут все просматриваемые префиксы шаблонов и сами шаблоны. А ребрами – символы, которые осуществляют переход между узлами. Например, между узлом a и ab будет ребро b.

Реализуем бор на базе вектора. Каждый вектор будет хранить некоторые состояния (узел) бора. Состояние будет иметь: подстроку в узле, потомки узла, суффиксную и конечную ссылки, номер состояния в дереве и флаг для определения, является ли данный узел шаблоном.

Этап 1 – Построение дерева

Корнем дерева всегда будет пустая строка. От нее начинаем обход. Мы пробегаемся циклом по всем переданным шаблонам. И также посимвольно идем по шаблонам. Смотрим на потомков состояния. Если он имеет переход к одному из них по текущему символу, мы идем в него. Иначе мы создаем новое состояние и делаем туда переход по данному символу.

Как только дошли до конца шаблона, мы оказались в каком-то состоянии дерева. Его помечаем, как узел, относящийся к шаблону.

Этап 2 – Построение ссылок

Теперь надо из бора сделать конечный автомат, то есть научить бор корректно сбрасывать состояния при несовпадающей букве. Для этого помогут суффиксные ссылки. Суффиксная ссылка – это ссылка на узел, соответствующий самому длинному суффикса ключа текущего узла. Также для оптимизации подойдет для каждого состояния ссылка на ближайший по суффиксным ссылкам узел с допускающим состоянием, то есть принадлежащий словарю – конечная ссылка.

Для их построение будем использовать обход в ширины через очередь. Начинаем с корня. Для его потомков ссылками будет как раз этот корень. Эти узлы добавляются в очередь и идет обход по их потомкам. Через данный узел мы построим ссылки для его потомков. Пусть состояние переходит в потомок Y по символу ch. Для этого возьмем суффиксную ссылку этого состояния на узел X и посмотрим, имеет ли X переход по символу ch. Если да, то потомок X и будет ссылкой для изначального потомка Y. В противном случае мы уменьшаем размерность возможного суффикса и у X возьмем его ссылку и будем рассматривать ее. Если так и не нашелся суффикс, то суффиксной ссылкой Y будет корень дерева.

Конечная ссылка строится по суффиксной. Если узел A по суфф. ссылке относится к словарю, то есть флаг в состоянии помечен, то суфф. сылка будет равна конечной. Инача конечной ссылкой будет конечая ссылка A.

Этап 3 – поиск по тексту

Теперь наша программа получает текст, в котором нужно найти все вхождения шаблонов из словаря. Мы будем циклом проходиться по каждому символу текста, параллельно выполняя определенный обход по бору, начиная с корня.

Если из данного состояния мы можем сделать переход по текущему символу, то мы переходим в данный потомок. Иначе берем суффиксную ссылку состояния и проверяем ее.

Дальше мы заводим временную переменную для небольшого обхода по найденному узлу через конечную ссылку. Мы первым делом проверяем, что этот узел помечен и добавляем его номер и его позицию в тексте (считается как i – размер\_узла + 1, где i – текущий индекс в тексте). Дальше переходим к следующему узлу по конечной ссылке и проверяем также ее. Когда дошли до корня, мини-обход заканчивается.

Таким же образом продолжается обход, пока не будут пройдены все символы текста.

Сложность по времени:

Построение бора: O(M), где M – суммарная длина всех шаблонов. Так как каждый символ обрабатывается ровно 1 раз при добавлении в бор.

Построение ссылок: O(M \* |E|), где |E| - размер алфавита. Мы делаем обход в ширину по всем узлам, это будет примерно равно M. Для каждого узла в худшем случае мы можем потратить операции для нахождения суффиксных ссылок. Однако в нашей задаче алфавит фиксирован, так что |E| можно считать константой.

Поиск: O(N + Z), где N – длина текста, Z – количество найденных вхождений. В основном цикле мы проходим по всему тексту, отсюда получаем N. У нас есть переходы по суффиксным ссылкам, но там будет O(1). По той причине, что суффиксная ссылка ведет к уменьшению глубины (было, например, aab а мы перешли к ab). При этом добавление каждого символа ведет к увеличению глубины на 1. По итогу мы сможем сделать суммарно переходов по суффиксным ссылкам только N раз, так что для 1 операции это в среднем будет O(1). Каждое вхождение мы обрабатываем ровно 1 раз, так что суммарно за весь алгоримт выйдет Z обходов.

Итогова сложность: O(M + N + Z)

Сложность по памяти:

O(M \* |E|) – так как каждый узел может хранить в себе через словарь переходы на все символы алфавита. Это будет худший случай.

Задание 2

Для 2-ого задания будем искать индексы вхождения шаблона в текст, если шаблон имеет символ-джокер. Для начала мы разобьем наш шаблон на подстроки, разделенные масками (например, для a??bc? такими подстроками будут a и bc). И также находим индексы вхождения этих подстрок в шаблон.

Также инициализируем

**Описание функций и структур данных**

std::vector<int> prefixFunction(std::string stringTemplate) – Функция для вычисления префикс-функции для переданного шаблона. Значения длин бордеров для каждой позиции добавляются в prefixPositions. Данный вектор возвращается в качестве результата работы функции. Зполнение идет в соответствии с вышеописанным алгоритмом.

std::vector<int> KMP(std::string stringInput, std::string stringTemplate)- Основная функция поиска индексов вхождения шаблона в строку. Индексы будут добавляться в вектор result. Номера позиций для сравнения перебираются через перменную index. А переменная offset будет хранить смещение, показывающее, откуда начинать следующее сравнение шаблона с подстрокой. Функция содержит внешний цикл для перебора index, и внутренний для сравнения посимвольного шаблона с подстрокой.

**Выполнение поставленных заданий**

В функции main() будет прописана логика для каждого задания отдельно, используя вышенаписанные функции для алгоритма КМП.

Задание 1.

Необходимо через std::cin прочитать 2 строки: шаблон и текст. После этого вызвать фунция KMP с переданными в качестве аргумента этих двух строк. Затем через цикл пройтись по итоговому вектору и вывести значения через запятую.

Задание 2.

Также получаем на вход 2 строки. Первым делм проверяем, чтобы длины были одинаковые. Чтобы затем проверить циклический сдвиг, необходимо расширить первую строку: stringFirst = stringFirst + stringFirst. Так как цикл подразумевает под собой, что начало строки является продолжением конца. В этой удвоенной строке мы и находим вхождение stringSecond.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Номер задания | Входные данные | Выходные данные | Комментарии |
|  | 1 | ab  abab | 0,2 | Верный вывод |
|  | 1 | abcabd  abcabeabcabcabd | 9 | Верный вывод |
|  | 2 | defabc  abcdef | 3 | Верный вывод |
|  | 2 | defabc  abcde | -1 | Верный вывод |
|  | 2 | defabc  bcdefa | 4 | Верный вывод |

# Приложение А Исходный код программы

Название файла: main1.cpp

#include <iostream>

#include <string>

#include <vector>

// Функция для вычисления префикс-функции

std::vector<int> prefixFunction(std::string stringTemplate){

std::vector<int> prefixPositions(stringTemplate.length()); // инициализация вектора для хранения длины максимального бордера для каждой позиции

for (int i = 1; i < stringTemplate.length(); i++){

int k = prefixPositions[i - 1]; // получение значения префикс функции для предыдущей позиции

while (k > 0 && stringTemplate[i] != stringTemplate[k]){ // пытаемся найти длину следующего бордера путем сравнения конца суффикса с концом префикса

k = prefixPositions[k - 1]; // если концы не равны, то ищем следующую подстроку, беря бордер меньшей длины

}

if (stringTemplate[i] == stringTemplate[k])

k++; // если концы совпали, то значение префикс функции увеличивается на 1 по сравнению с длиной бордера для предыдущей позиции

prefixPositions[i] = k; // обновляем значение для новой позиции

}

return prefixPositions;

}

// Основная функция поиска индексов вхождения

std::vector<int> KMP(std::string stringInput, std::string stringTemplate){

std::vector<int> result; // инициализация вектора, куда будут добавляться итоговые индексы

int index = 0; // индекс строки, начиная с которой нужно делать сравнение

int offset = 0; // переменная смещения для пропуска лишней проверки префикса, равного предыдущему суффиксу

std::vector<int> prefixPositions = prefixFunction(stringTemplate);

/\*for (int i = 0; i < prefixPositions.size(); i++)

std::cout << stringTemplate[i] << " - " << prefixPositions[i] << std::endl;\*/

while (index + stringTemplate.length() <= stringInput.length()) // цикл по нахождению всех индексов совпадения шаблона со строкой

{

bool flag = true; // флаг для вывода результата сравнения

/\*std::cout << stringInput << std::endl;

for (int i = 0; i < index; i++)

std::cout << " ";

std::cout << stringTemplate << std::endl;

for (int i = 0; i < index + offset; i++)

std::cout << " ";

std::cout << "^" << std::endl;\*/

for (int i = offset; i < stringTemplate.length(); i++){ // проверка совпадения шаблона с частью строки, которая начинается с index

if (stringTemplate[i] != stringInput[index + i]){ // в случае несовпадения обновляем index

flag = false;

if (i == 0){ // если не совпал 1-ый элемент шаблона, делаем единичный сдвиг вправо

index++;

offset = 0;

break;

}

index = index + i - prefixPositions[i - 1]; // обновление index с учетом префикс-функции

offset = prefixPositions[i - 1]; // обновление смещения

break;

}

}

if (flag){ // если флаг равняется истине, индекс добавляется в итоговый массив

result.push\_back(index);

index = index + stringTemplate.length() - prefixPositions[stringTemplate.length() - 1]; // обновление index с учетом префикс-функции

offset = prefixPositions[stringTemplate.length() - 1]; // обновление смещения

}

}

if (result.size() == 0)

result.push\_back(-1); // если не нашлось совпадений, возвращаем -1

return result;

}

int main(){

std::string stringTemplate;

std::string stringInput;

std::cin >> stringTemplate >> stringInput; // ввод строки и шаблона

std::vector<int> result = KMP(stringInput, stringTemplate);

for (int i = 0; i < result.size(); i++){ // вывод результата через обход итогового вектора

if (i == result.size() - 1){

std::cout << result[i] << std::endl;

break;

}

std::cout << result[i] << ",";

}

return 0;

}

Название файла: main2.cpp

#include <iostream>

#include <string>

#include <vector>

// Функция для вычисления префикс-функции

std::vector<int> prefixFunction(std::string stringTemplate){

std::vector<int> prefixPositions(stringTemplate.length()); // инициализация вектора для хранения длины максимального бордера для каждой позиции

for (int i = 1; i < stringTemplate.length(); i++){

int k = prefixPositions[i - 1]; // получение значения префикс функции для предыдущей позиции

while (k > 0 && stringTemplate[i] != stringTemplate[k]){ // пытаемся найти длину следующего бордера путем сравнения конца суффикса с концом префикса

k = prefixPositions[k - 1]; // если концы не равны, то ищем следующую подстроку, беря бордер меньшей длины

}

if (stringTemplate[i] == stringTemplate[k])

k++; // если концы совпали, то значение префикс функции увеличивается на 1 по сравнению с длиной бордера для предыдущей позиции

prefixPositions[i] = k; // обновляем значение для новой позиции

}

return prefixPositions;

}

// Основная функция поиска индексов вхождения

std::vector<int> KMP(std::string stringInput, std::string stringTemplate){

std::vector<int> result; // инициализация вектора, куда будут добавляться итоговые индексы

int index = 0; // индекс строки, начиная с которой нужно делать сравнение

int offset = 0; // переменная смещения для пропуска лишней проверки префикса, равного предыдущему суффиксу

std::vector<int> prefixPositions = prefixFunction(stringTemplate);

/\*for (int i = 0; i < prefixPositions.size(); i++)

std::cout << stringTemplate[i] << " - " << prefixPositions[i] << std::endl;\*/

while (index + stringTemplate.length() <= stringInput.length()) // цикл по нахождению всех индексов совпадения шаблона со строкой

{

bool flag = true; // флаг для вывода результата сравнения

/\*std::cout << stringInput << std::endl;

for (int i = 0; i < index; i++)

std::cout << " ";

std::cout << stringTemplate << std::endl;

for (int i = 0; i < index + offset; i++)

std::cout << " ";

std::cout << "^" << std::endl;\*/

for (int i = offset; i < stringTemplate.length(); i++){ // проверка совпадения шаблона с частью строки, которая начинается с index

if (stringTemplate[i] != stringInput[index + i]){ // в случае несовпадения обновляем index

flag = false;

if (i == 0){ // если не совпал 1-ый элемент шаблона, делаем единичный сдвиг вправо

index++;

offset = 0;

break;

}

index = index + i - prefixPositions[i - 1]; // обновление index с учетом префикс-функции

offset = prefixPositions[i - 1]; // обновление смещения

break;

}

}

if (flag){ // если флаг равняется истине, индекс добавляется в итоговый массив

result.push\_back(index);

index = index + stringTemplate.length() - prefixPositions[stringTemplate.length() - 1]; // обновление index с учетом префикс-функции

offset = prefixPositions[stringTemplate.length() - 1]; // обновление смещения

break;

}

}

if (result.size() == 0)

result.push\_back(-1); // если не нашлось совпадений, возвращаем -1

return result;

}

int main(){

std::string stringFirst;

std::string stringSecond;

std::cin >> stringFirst >> stringSecond; // чтение строк

if (stringFirst.length() != stringSecond.length()){ // проверка равенства их длины

std::cout << -1 << std::endl;

return 0;

}

std::vector<int> result = KMP(stringFirst + stringFirst, stringSecond); // удлинение 1-ой строки и нахождения в ней 2-ой строки

std::cout << result[0] << std::endl; // вывод результата

return 0;

}