

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по лабораторной работе №5**  
**по дисциплине «Построение и анализ алгоритмов»**  
**Тема: Алгоритм Ахо-Корасик**

Студент гр. 8303

\_\_\_\_\_

Колосова М.П.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

## **Цель работы**

Изучить работу алгоритма Ахо-Корасик для решения задач точного поиска набора образцов и поиска образца с джокером (символом, совпадающим с любым из алфавита).

## **Задание 1**

Разработайте программу, решающую задачу точного поиска набора образцов.

### **Вход:**

Первая строка содержит текст ( $T$ ,  $1 \leq |T| \leq 100000$ )

Вторая - число ( $1 \leq n \leq 3000$ ), каждая следующая из строк содержит шаблон из набора  $= \{p_1, \dots, p_n\}$   $1 \leq |p_i| \leq 75$

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.

### **Sample Input:**

NTAG

3

TAGT

TAG

T

### **Sample Output:**

2 2

2 3

## Задание 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  $P$  в текст  $T$ .

Например, образец  $a???c?$  с джокером  $?$  встречается дважды в тексте .

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  $\{A,C,G,T,N\}$

### Вход:

Текст ( $T$ ,  $1 \leq |T| \leq 100000$ )

Шаблон ( $P$ ,  $1 \leq |P| \leq 40$ )

Символ джокера

### Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

### Sample Input:

ACTANCA

A\$\$\$A\$

\$

### Sample Output:

1

## **Индивидуализация**

### **Вариант 3**

Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

### **Бор**

Бор (префиксное дерево) — структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки. Представляет собой корневое дерево, каждое ребро которого помечено каким-то символом так, что для любого узла все рёбра, соединяющие этот узел с его сыновьями, помечены разными символами. Считается, что бор содержит данную строку-ключ тогда и только тогда, когда эту строку можно прочитать на пути из корня до некоторого (единственного для этой строки) терминального узла.

Из такой структуры данных возможно построить автомат путем добавления суффиксных ссылок.

### **Построение бора и автомата**

С помощью соответствующих методов вставки и создания автомата сначала бор в цикле заполняется всеми паттернами из словаря, а затем при обходе в ширину устанавливаются суффиксные ссылки следующим образом. Если у родителя существует суффиксная ссылка, то выполняется переход по ней как по началу собственного суффикса текущей позиции. Осуществляется переход по суффиксным ссылкам пока это возможно или пока не найден переход в символ обрабатываемой вершины. Затем устанавливаем суффиксную ссылку для обрабатываемой вершины. В случае, если переход найден не был, устанавливается ссылка на корень.

### **Описание алгоритма №1**

В программе реализован алгоритм Ахо-Корасик. По словарю шаблонов строится бор. Затем для каждого символа текста выполняется поиск по автомату. Переход в автомате осуществляется следующим образом. Если возможно, то

выполняется переход в потомка, в другом случае - по суффиксной ссылке. После перехода выполняется проверка на то, является ли вершина и всевозможные её суффиксы – терминальными. Если да, то возвращаем все такие найденные номера паттернов. Если символа в автомате не оказалось, то текущая вершина принимает значение корня – вхождение не найдено.

## **Описание алгоритма задания №2**

В качестве словаря шаблонов используются подстроки маски, разделенные символами джокера. Аналогично заданию №1 сначала строится бор из этих подстрок и выполняется поиск по автомату для каждого символа текста. Появления подстроки в тексте на позиции  $j$  означает возможное появление маски на позиции  $j-l+1$ , где  $l$  – индекс начала подстроки в маске. Далее, с помощью вспомогательного массива для таких позиций увеличиваем его значение на 1. Индексы, по которым хранятся значения равного  $n$  (количеству подстрок), являются вхождениями маски в текст.

## **Описание реализации индивидуализации**

Конечная ссылка ведёт на ближайшую по суффиксным ссылкам конечную вершину, поэтому после построения автомата с помощью обхода в ширину от каждой вершины происходит попытка построения конечной ссылки. Если суффиксная цепь ведет от текущей вершине к терминальной, то устанавливается ссылка от текущей к конечной вершине. Далее вычисляется длина каждой возможной цепи суффиксных ссылок и цепи конечных ссылок. В качестве результата выводятся максимальные значения.

## **Сложность алгоритма**

Построение бора выполняется за  $O(m)$ , где  $m$  – суммарная длина паттернов (для джокеров – суммарная длина подстрок, разделенных джокером). Суффиксные ссылки строятся с помощью обхода в ширину, сложность которого -  $O(V+E)$ . Число ребер имеет линейную зависимость от числа вершин. Отсюда -  $O(2m) = O(m)$ . Бор позволяет за один проход по тексту найти все

вхождения, поэтому сложность составляет  $O(n)$ , где  $n$  — длина текста. Значит, сложность по времени составляет  $O(m + n)$ .

Сложность по памяти для хранения бора —  $O(m)$ , где  $m$  — количество вершин (каждый символ представляет собой вершину бора), также на каждой позиции текста могут встретиться все шаблоны, что в свою очередь приводит к общей сложности по памяти  $O(n*k+m)$ , где  $n$  — длина текста,  $k$  — число шаблонов.

### **Описание функций и структур данных**

Class `TreeNode` — структура, для хранения данных о корневой вершине бора.

Поля класса:

`string dbgStr`; - строка, которую можно получить при переходе в текущую вершину по ребрам

`char value` — Значение ноды, символ, по которому был произведён переход;

`TreeNode* parent` — ссылка на родительскую вершину;

`TreeNode* suffixLink` — суффиксная ссылка;

`unordered_map <char, TreeNode*> children` — ассоциативный неупорядоченный контейнер потомков, ключом которого является символ, по которому можно перейти на потомка;

`size_t numOfPattern` — порядковый номер паттерна

`vector<pair<size_t, size_t>> substringEntries` — вектор, элементами которого является пара: индекс вхождения в маску и длина подстроки

### **Методы**

1) `TreeNode(char val)` — конструктор для заполнения поля значения значением символа, по которому перешли;

Принимает на вход значение `val`, по которому осуществлен переход

`TreeNode() : value(0)` — конструктор для создания корневой вершины

`value` — поле класса, содержащее информацию о том, по какому ребру произошел переход, для корня — это нулевой символ

2) `void insert(const string &str)` — метод для вставки строки в бор;

Метод принимает на вход строку (`const string &str`), которую необходимо вставить в бор. Результат работы метода — модифицированный бор.

Метод проверяет, был ли создан переход по текущему символу строки с помощью функции `find` в контейнере потомков узла и либо создает переход по этому символу (добавляет нового потомка), если он не существует на текущей позиции, либо спускается вниз по бору. После вставки строки увеличивается счетчик количества паттернов и изменяется поле класса `numOfPattern`, являющееся индикатором паттерна в словаре.

3) `auto find(const char c)` – метод для поиска подстроки в строке при помощи автомата, выполняет поиск, по заданному символу, в боре, в случае найденной терминальной вершины, возвращает либо вектор `size_t` (задание 1), либо вектор пар `size_t` (задание 2);

Метод принимает на вход символ, который необходимо рассмотреть и возвращает вектор номеров найденных терминальных вершин в 1 задании и вектор пар, состоящих из начала безмасочной подстроки в маске и ее длины, во 2 задании. Обходим всех потомков текущей позиции и переходим по суффиксной ссылке. Если среди потомков не было искомого символа, то переходим по суффиксной ссылке для дальнейшего поиска. Если символ потомка равен искомому, спускаемся в эту вершину и обходим его суффиксы, так как они тоже могут быть терминальными вершинами, то есть вхождениями, заполняем вектор этими терминальными вхождениями в 1 задании и парами во 2 задании и возвращаем вектор.

4) `void makeAutomaton()` – функция, которая модифицирует бор в автомат путём добавления суффиксных ссылок;

Метод возвращает `void`. Описание работы данного метода находится в описании построения бора и автомата в начале отчета. Результат работы — модифицированный бор с суффиксными ссылками, то есть автомат.

5) `void makeFinishLink()` - функция построения конечных ссылок на основе суффиксных

Метод возвращает `void` и не имеет аргументов, он модифицирует бор путем добавления конечных ссылок. С помощью обхода в ширину рассматривается

каждая позиция бора и выполняется переход по цепи суффиксных ссылок до ближайшей терминальной вершины. В конце получим значение конечной ссылки для каждой вершины, оно будет равняться либо ближайшей терминальной при переходе по суффиксным ссылкам, либо нулевому указателю.

6) `void findMaxLinkChain()` - функция поиска максимальной длины ссылочных цепей

Метод не принимает на вход никаких аргументов, не возвращает никакое значение и не модифицирует бор, в нем осуществляется обход автомата в ширину и переход по цепям сначала суффиксных ссылок, затем конечных из каждой вершины и рассчитывается длина каждой цепи. Результатом работы является вывод информации о максимальных длинах таких цепей.

#### **№1:**

`set<pair<size_t, size_t>> AhoCorasick(const string &text, const vector<string> &patterns)` – функция, возвращающая множество, состоящее из пары индекса вхождения в текст и номера паттерна, который был найден в нём.

Принимает на вход строку текста(`const string &text`) и вектор шаблонов(`const vector <string> &patterns`), точнее — их ссылки. В начале создается и инициализируется бор, затем из него строится автомат, автомат заполняется конечными ссылками, находятся максимальные длины цепей суффиксных и конечных ссылок и выполняется поиск вхождений. Результат работы функции — вектор пар номеров вхождений и их индексов в тексте.

#### **№2:**

`vector <size_t> AhoCorasick(const string &text, const string &mask, const char joker)`— функция, возвращающая вектор индексов вхождения маски в текст.

Принимает на вход строку-текст, строку-маску и символ джокера. Результат работы - строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер). Функция работает аналогично №1, за исключением инициализации бора паттернами и специфичного вывода.



## Тестирование

### №1

Ввод	Вывод
ABCASDTEAD 5 ABC DTE ASD TEA EAD	1 1 4 3 6 2 7 4 8 5
ABCBABC 4 ABC BC CBA BAB	1 1 2 2 3 3 4 4 5 1 6 2
DOGNTADOG 3 TA DOG NA	1 2 5 1 7 2
DDDA 1 DD	1 1

Тест с подробным промежуточным выводом:

```
abcaadfab
6
a
ab
bc
bca
c
caa
```

Текущее состояние бора:

Корень:

c:

Суффиксная ссылка: Корень

Родитель: Корень

Потомок: a

b:

Суффиксная ссылка: Корень

Родитель: Корень

Потомок: c

a:

Суффиксная ссылка: Корень

Родитель: Корень

Потомок: b

ca:

Суффиксная ссылка: a

Родитель: c

Потомок: a

bc:

Суффиксная ссылка: c

Родитель: b

Потомок: a

ab:

Суффиксная ссылка: b

Родитель: a

caa:

Суффиксная ссылка: a

Родитель: ca

bca:

Суффиксная ссылка: ca

Родитель: bc

Корень	
с:	Суффиксная цепочка с->Корень
б:	Суффиксная цепочка б->Корень
а:	Суффиксная цепочка а->Корень
са:	Суффиксная цепочка са->а->Корень Цепочка конечных ссылок са->а
бс:	Суффиксная цепочка бс->с->Корень Цепочка конечных ссылок бс->с
аб:	Суффиксная цепочка аб->б->Корень
саа:	Суффиксная цепочка саа->а->Корень Цепочка конечных ссылок саа->а
бса:	Суффиксная цепочка бса->са->а->Корень Цепочка конечных ссылок бса->а
Максимальная длина цепи из суффиксных ссылок - 3	
Максимальная длина цепи из конечных ссылок - 1	

№2:

Ввод	Вывод
ACTANCA A\$\$A \$	1

CATNATCAT \$AT \$	1 4 7
MATFDHYD \$\$D \$	3 6
TDWIK \$D\$LK \$	«»- пустая строка, отсутствие вхождений

**Тест с подробным промежуточным выводом:**

ACTANCA  
A\$A\$A\$  
\$  
Вставляем строку: A  
Текущее состояние бора:  
Корень:  
    Потомок: A  
A:  
    Родитель: Корень

Вставляем строку: A  
Текущее состояние бора:  
Корень:  
    Потомок: A  
A:  
    Родитель: Корень

Строим автомат:  
A:  
    Родитель: Корень  
    Суффиксная ссылка: Корень

Текущее состояние бора:  
Корень:  
    Потомок: A  
A:  
    Суффиксная ссылка: Корень  
    Родитель: Корень

Ищем 'A' из: Корень  
Символ найден!  
Ищем 'C' из: A  
Переходим по суффиксной ссылке: Корень  
Символ не найден!  
Ищем 'T' из: Корень  
Символ не найден!  
Ищем 'A' из: Корень  
Символ найден!  
Ищем 'N' из: A  
Переходим по суффиксной ссылке: Корень  
Символ не найден!  
Ищем 'C' из: Корень  
Символ не найден!  
Ищем 'A' из: Корень  
Символ найден!

1

## **Вывод**

В ходе выполнения лабораторной работы была изучена работа алгоритма Ахо-Корасик. Алгоритм был реализован для поиска вхождений шаблонов из словаря в тексте, а также для поиска шаблона с джокером.

## Приложение А

### Код программы 1

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <queue>
#include <algorithm>
#include <unordered_map>

using namespace std;

class Tree {

    string dbgStr = ""; // Для отладки
    char value; // Значение узла
    size_t numOfPattern = 0; // Номер введенного паттерна
    Tree* parent = nullptr; // Родитель ноды
    Tree* suffixLink = nullptr; // Суффиксная ссылка
    Tree* finishLink = nullptr; // конечная ссылка
    unordered_map <char, Tree*> children; // Потомки узла
public:
    Tree() : value('\0') {}
    Tree(char val) : value(val) {} // Конструктор ноды
    void initialization(vector<string> patterns){
        for(auto &pattern : patterns){
            this->insert(pattern);
        }
    }
    void printInfo(Tree *curr){

        cout << curr->dbgStr << ':' << endl;

        if (curr->suffixLink)
            cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ?
"Корень" : curr->suffixLink->dbgStr) << endl;
        if(curr->finishLink)
            cout << "\tКонечная ссылка: " << (curr->finishLink->dbgStr) <<
endl;

        if(curr -> parent)
            cout << "\tРодитель: " << (curr->parent->value ? curr->parent->dbgStr
: "Корень") << endl;

        if (!curr->children.empty())
            cout << "\tПотомок: ";
        for (auto child : curr->children) {
            cout << child.second->value << ' ';
        }
    }
    // Вставка подстроки в бор
    void insert(const string &str) {
        auto curr = this;
        static size_t countPatterns = 0;

        for (char c : str) { // Идем по строке
            // Если из текущей вершины по текущему символу не было создано
перехода
            if (curr->children.find(c) == curr->children.end()) {
                // Создаем переход
                curr->children[c] = new Tree(c);
            }
        }
    }
};
```

```

        curr->children[c]->parent = curr;
        curr->children[c]->dbgStr += curr->dbgStr + c;
    }
    // Спускаемся по дереву
    curr = curr->children[c];
}

cout << "Вставляем строку: " << str << endl;
printBor();

// Показатель терминальной вершины, значение которого равно порядковому
номера добавления шаблона
curr->numOfPattern = ++countPatterns;
}

//печать бора
void printBor() {
    cout << "Текущее состояние бора:" << endl;

    queue<Tree *> queue;
    queue.push(this);

    while (!queue.empty()) {

        auto curr = queue.front();
        if (!curr->value)
            cout << "Корень:" << endl;
        else
            printInfo(curr);
        for (auto child : curr->children) {
            queue.push(child.second);
        }

        queue.pop();
        cout << endl;
    }
    cout << endl;
}

// Функция для поиска подстроки в строке при помощи автомата
vector<size_t> find(const char c) {
    static const Tree *curr = this; // Вершина, с которой необходимо начать
    следующий вызов
    cout << "Ищем '" << c << "' из: " << (curr->dbgStr.empty() ? "Корень" :
curr->dbgStr) << endl; // Дебаг

    for (; curr != nullptr; curr = curr->suffixLink) {
        // Обходим потомков, если искомый символ среди потомков не найден, то
        // переходим по суффиксной ссылке для дальнейшего поиска
        for (auto child : curr->children)
            if (child.first == c) { // Если символ потомка равен искомому
                curr = child.second; // Значение текущей вершины переносим на
этого потомка
                vector<size_t> visited; // Вектор номеров найденных терм.
вершин
                // Обходим суффиксы, т.к. они тоже могут быть терминальными
вершинами
                for (auto temp = curr; temp->suffixLink; temp = temp-
>suffixLink)
                    if (temp->numOfPattern)
                        visited.push_back(temp->numOfPattern - 1);
                //

```



```

        cout << "Символ найден!" << endl; // Дебаг
        return visited;
    }

    if (curr->suffixLink) {
        cout << "Переходим по суффиксной ссылке: ";
        cout << (curr->suffixLink->dbgStr.empty() ? "Корень" : curr-
>suffixLink->dbgStr) << endl;
    }
    cout << "Символ не найден!" << endl; // Дебаг

    curr = this;
    return {};
}

// Функция для построения недетерминированного автомата
void makeAutomaton() {

    cout << "Строим автомат: " << endl;
    queue<Tree *> queue; // Очередь для обхода в ширину

    for (auto child : children) // Заполняем очередь потомками корня
        queue.push(child.second);

    while (!queue.empty()) {
        auto curr = queue.front(); // Обрабатываем вершину из очереди
        printInfo(curr);
        // Заполняем очередь потомками текущей верхушки
        for (auto child : curr->children) {
            queue.push(child.second);
        }

        if (!curr->children.empty())
            cout << endl;

        queue.pop();
        auto p = curr->parent; // Ссылка на родителя обрабатываемой вершины
        char x = curr->value; // Значение обрабатываемой вершины
        if (p)
            p = p->suffixLink; // Если родитель существует, то переходим по
суффиксной ссылке

        // Пока можно переходить по суффиксной ссылке или пока
        // не будет найден переход в символ обрабатываемой вершины
        while (p && p->children.find(x) == p->children.end())
            p = p->suffixLink; // Переходим по суффиксной ссылке

        // Суффиксная ссылка для текущей вершины равна корню, если не смогли
найти переход
        // в дереве по символу текущей вершины, иначе равна найденной вершине
        curr->suffixLink = p ? p->children[x] : this;

        // Дебаг
        cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ?
"Корень" : curr->suffixLink->dbgStr) << endl << endl;
    }

    // Дебаг
    cout << endl;
    printBor();
}

void makeFinishLink(){

```

```

cout << "Строим конечные ссылки" << endl;

queue<Tree *> queue;
queue.push(this);

while (!queue.empty()) {

    auto curr = queue.front();
    auto next = curr;
    //проходим по суффиксным ссылкам каждой вершины автомата
    while(1){

        if(next->suffixLink && next->suffixLink->value){//есть возможность
перейти по суффиксной ссылке не в корень
            next = next->suffixLink;//переходим
        }
        else break;//цепочка суффиксных ссылок закончилась

        if(next->numOfPattern){//вершина - терминальная
            curr->finishLink = next;//строим конечную ссылку
            break;
        }

    }
    //обход в ширину
    for (auto child : curr->children) {
        queue.push(child.second);
    }

    queue.pop();
}
printBor();
}

void findMaxLinkChain(){//индивидуализация поиск максимальных цепей

    size_t maxSuffixChain = 0;
    size_t maxFinishChain = 0;
    size_t buf = 0;//для хранения длины цепочки из текущей вершины

    queue<Tree *> queue;
    queue.push(this);

    while (!queue.empty()) {

        auto curr = queue.front();
        auto next = curr;

        //проходим по суффиксным ссылкам каждой вершины автомата
        if(curr->value)
            cout << curr->dbgStr << ":" << endl << "\tСуффиксная цепочка ";
        cout << curr->dbgStr;
        buf = 0;
        while(1){
            if(next->suffixLink ){//&& next->suffixLink->value){//есть
возможность перейти по суффиксной ссылке не в корень
                next = next->suffixLink;//переходим
                cout << "->" << next->dbgStr;
                buf++;//увеличиваем длину цепи
            }
            else break;//цепочка суффиксных ссылок закончилась
        }
    }
}

```

```

        cout << "Корень" << endl;
        maxSuffixChain = max(maxSuffixChain, buf);
        //cout << "Текущая максимальная длина цепи суффиксных ссылок: " <<
maxSuffixChain << endl;

        buf = 0;
        next = curr;
        if(curr->finishLink)
            cout << "\tцепочка конечных ссылок " << curr->dbgStr;
        else cout << endl;
        while(1){
            if(next->finishLink ){//есть возможность перейти по конечной
ссылке
                next = next->finishLink;//переходим
                if(next->dbgStr != "")
                    cout << "->" << next->dbgStr;
                buf++;//увеличиваем длину цепи
            }
            else break;//цепочка суффиксных ссылок закончилась
        }
        maxFinishChain = max(maxFinishChain, buf);
        //cout << "Текущая максимальная длина цепи конечных ссылок: " <<
maxFinishChain <<endl;

        //обход в ширину
        for (auto child : curr->children) {
            queue.push(child.second);
        }

        queue.pop();
        cout << endl;
    }
    cout << endl;

    cout << "Максимальная длина цепи из суффиксных ссылок - " <<
maxSuffixChain << endl;
    cout << "Максимальная длина цепи из конечных ссылок - " << maxFinishChain
<< endl;
    cout << endl;

}

~Tree() { // Деструктор ноды
    for (auto child : children) delete child.second;
}

};

auto AhoCorasick(const string &text, const vector <string> &patterns)
{
    Tree bor;
    set <pair<size_t, size_t>> result;

    bor.initialization(patterns);
    bor.makeAutomaton(); // Из полученного бора создаем автомат (путем добавления
суффиксных ссылок)
    bor.makeFinishLink();//добавляем конечные ссылки
    bor.findMaxLinkChain();//поиск максимальных длин цепей ссылок

    {
        size_t j = 0;
        for(auto &el : text){//поиск для каждого символа строки
            for(auto pos : bor.find(el))// Проходим по всем найденным позициям,
записываем в результат

```

```

        result.emplace(j - patterns[pos].size() + 2, pos + 1);
        j++;
    }
}

return result;
}

int main()
{
    string text;
    size_t n;
    cin >> text >> n;
    vector <string> patterns(n); //словарь

    for(auto &pattern : patterns){
        cin >> pattern;
    }

    auto res = AhoCorasick(text, patterns);
    for (auto r : res)
        cout << r.first << ' ' << r.second << endl;

    return 0;
}

```

## Приложение В

### Код программы 2

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <unordered_map>

using namespace std;

class TreeNode {

    string dbgStr = ""; // Для отладки
    char value; // Значение ноды
    TreeNode *parent = nullptr; // Родитель ноды
    TreeNode *suffixLink = nullptr; // Суффиксная ссылка
    TreeNode *finishLink = nullptr; //конечная ссылка
    unordered_map <char, TreeNode*> children; // Потомок ноды
    vector <pair<size_t, size_t>> substringEntries;
    size_t numOfPattern = 0;
public:
    TreeNode(char val) : value(val) {} // Конструктор ноды
    TreeNode() : value('\0') {}

    void printInfo(TreeNode *curr) {

        if (curr->suffixLink)
            cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ?
"Корень" : curr->suffixLink->dbgStr) << endl;
        if(curr->finishLink)
            cout << "\tКонечная ссылка: " << (curr->finishLink->dbgStr) <<
endl;

        if(curr -> parent)
            cout << "\tРодитель: " << (curr->parent->value ? curr->parent->dbgStr
: "Корень") << endl;

        if (!curr->children.empty())
            cout << "\tПотомок: ";
        for (auto child : curr->children) {
            cout << child.second->value << ' ';

        }

    }
    // Отладочная функция для печати бора
    void printBor() {

        cout << "Текущее состояние бора:" << endl;

        queue<TreeNode *> queue;
        queue.push(this);

        while (!queue.empty()) {
            auto curr = queue.front();
            if (!curr->value)
                cout << "Корень:" << endl;
            else
                cout << curr->dbgStr << ':' << endl;

            printInfo(curr);
        }
    }
};
```

```

        for (auto child : curr->children) {
            queue.push(child.second);
        }

        queue.pop();
        cout << endl;
    }
    cout << endl;
}

// Вставка подстроки в бор
void insert(const string &str, size_t pos, size_t size) {
    auto curr = this;
    size_t countPatterns = 0;

    for (char c : str) { // Идем по строке
        // Если из текущей вершины по текущему символу не было создано
        // перехода
        if (curr->children.find(c) == curr->children.end()) {
            // Создаем переход
            curr->children[c] = new TreeNode(c);
            curr->children[c]->parent = curr;
            curr->children[c]->dbgStr += curr->dbgStr + c;
        }
        // Спускаемся по дереву
        curr = curr->children[c];
    }
    cout << "Вставляем строку: " << str << endl;
    printBor();

    curr->substringEntries.emplace_back(pos, size);

    // Показатель терминальной вершины, значение которого равно порядковому
    // номеру добавления шаблона
    curr->numOfPattern = ++countPatterns;
}

vector <pair<size_t, size_t>> find(const char c)
{
    static const TreeNode *curr = this; // Вершина, с которой необходимо
    // начать следующий вызов
    cout << "Ищем '" << c << "' из: " << (curr->dbgStr.empty() ? "Корень" :
    curr->dbgStr) << endl;

    for (; curr != nullptr; curr = curr->suffixLink) {
        // Обходим потомков, если искомый символ среди потомков не найден, то
        // переходим по суффиксной ссылке для дальнейшего поиска
        for (auto child : curr->children)
            if (child.first == c) { // Если символ потомка равен искомому
                curr = child.second; // Значение текущей вершины переносим на
                // вектор пар, состоящих из начала безмасочной подстроки в
                // маске и её длины
                vector <pair<size_t, size_t>> visited;

                // Обходим суффиксы, т.к. они тоже могут быть терминальными
                // вершинами
                for (auto temp = curr; temp->suffixLink; temp = temp-
                >suffixLink)
                    for (auto el : temp->substringEntries)
                        visited.push_back(el);

                cout << "Символ найден!" << endl; // Дебаг
            }
    }
}

```

```

        return visited;
    }

    // Дебаг
    if (curr->suffixLink) {
        cout << "Переходим по суффиксной ссылке: ";
        cout << (curr->suffixLink->dbgStr.empty() ? "Корень" : curr-
>suffixLink->dbgStr) << endl;
    }
    cout << "Символ не найден!" << endl; // Дебаг

    curr = this;
    return {};
}

// Функция для построения недетерминированного автомата
void makeAutomaton() {
    cout << "Строим автомат: " << endl;

    queue<TreeNode *> queue; // Очередь для обхода в ширину

    for (auto child : children) // Заполняем очередь потомками корня
        queue.push(child.second);

    while (!queue.empty()) {
        auto curr = queue.front(); // Обрабатываем верхушку очереди

        // Для дебага
        cout << curr->dbgStr << ':' << endl;
        printInfo(curr);

        // Заполняем очередь потомками текущей верхушки
        for (auto child : curr->children) {
            queue.push(child.second);
        }

        // Дебаг
        if (!curr->children.empty())
            cout << endl;

        queue.pop();
        auto p = curr->parent; // Ссылка на родителя обрабатываемой вершины
        char x = curr->value; // Значение обрабатываемой вершины
        if (p) p = p->suffixLink; // Если родитель существует, то переходим по
суффиксной ссылке

        // Пока можно переходить по суффиксной ссылке или пока
        // не будет найден переход в символ обрабатываемой вершины
        while (p && p->children.find(x) == p->children.end())
            p = p->suffixLink; // Переходим по суффиксной ссылке

        // Суффиксная ссылка для текущей вершины равна корню, если не смогли
        найти переход
        // в дереве по символу текущей вершины, иначе равна найденной вершине
        curr->suffixLink = p ? p->children[x] : this;

        // Дебаг
        cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ?
"Корень" : curr->suffixLink->dbgStr) << endl << endl;
    }

    // Дебаг
    cout << endl;
}

```

```

    printBor();
}

void makeFinishLink(){

    cout << "Строим конечные ссылки" << endl;

    queue<TreeNode *> queue;
    queue.push(this);

    while (!queue.empty()) {

        auto curr = queue.front();
        auto next = curr;
        //проходим по суффиксным ссылкам каждой вершины автомата
        while(1){

            if(next->suffixLink && next->suffixLink->value){//есть возможность
перейти по суффиксной ссылке не в корень
                next = next->suffixLink;//переходим
            }
            else break;//цепочка суффиксных ссылок закончилась

            if(next->numOfPattern){//вершина - терминальная
                curr->finishLink = next;//строим конечную ссылку
                break;
            }

        }
        //обход в ширину
        for (auto child : curr->children) {
            queue.push(child.second);
        }

        queue.pop();
    }
    printBor();
}

void findMaxLinkChain(){//индивидуализация поиск максимальных цепей

    size_t maxSuffixChain = 0;
    size_t maxFinishChain = 0;
    size_t buf = 0;//для хранения длины цепочки из текущей вершины

    queue<TreeNode *> queue;
    queue.push(this);

    while (!queue.empty()) {

        auto curr = queue.front();
        auto next = curr;

        //проходим по суффиксным ссылкам каждой вершины автомата
        if(curr->value)
            cout << curr->dbgStr << ":" << endl << "\tСуффиксная цепочка ";
        cout << curr->dbgStr;
        buf = 0;
        while(1){
            if(next->suffixLink ){//&& next->suffixLink->value){//есть
возможность перейти по суффиксной ссылке не в корень
                next = next->suffixLink;//переходим
                cout << "->" << next->dbgStr;
            }
        }
    }
}

```



```

        buf++; //увеличиваем длину цепи
    }
    else break; //цепочка суффиксных ссылок закончилась
}
cout << "Корень" << endl;
maxSuffixChain = max(maxSuffixChain, buf);

buf = 0;
next = curr;
if(curr->finishLink)
    cout << "\tЦепочка конечных ссылок " << curr->dbgStr;
else cout << endl;
while(1){
    if(next->finishLink) { //есть возможность перейти по конечной
ссылке
        next = next->finishLink; //переходим
        if(next->dbgStr != "")
            cout << "->" << next->dbgStr;
        buf++; //увеличиваем длину цепи
    }
    else break; //цепочка суффиксных ссылок закончилась
}
maxFinishChain = max(maxFinishChain, buf);

//обход в ширину
for (auto child : curr->children) {
    queue.push(child.second);
}

queue.pop();
cout << endl;
}
cout << endl;

cout << "Максимальная длина цепи из суффиксных ссылок - " <<
maxSuffixChain << endl;
cout << "Максимальная длина цепи из конечных ссылок - " << maxFinishChain
<< endl;
cout << endl;

}

~TreeNode()
{
    for (auto child : children)
        delete child.second;
}

};

auto AhoCorasick(const string &text, const string &mask, char joker) {

    TreeNode bor;
    vector <size_t> result;
    vector <size_t> midArr(text.size()); // Массив для хранения кол-ва попаданий
безмасочных подстрок в текст
    string pattern;
    size_t numSubstrs = 0; // Количество безмасочных подстрок

    for (size_t i = 0; i <= mask.size(); i++) { // Заполняем бор безмасочными
подстроками маски
        char c = (i == mask.size()) ? joker : mask[i];
        if (c != joker)

```

```

        pattern += c;
    else if (!pattern.empty()) {
        numSubstrs++;
        bor.insert(pattern, i - pattern.size(), pattern.size());
        pattern.clear();
    }
}

bor.makeAutomaton();
bor.makeFinishLink();
bor.findMaxLinkChain();

for (size_t j = 0; j < text.size(); j++)
    for (auto pos : bor.find(text[j])) {
        // На найденной терминальной вершине вычисляем индекс начала маски в
тексте
        int i = int(j) - int(pos.first) - int(pos.second) + 1;
        if (i >= 0 && i + mask.size() <= text.size())
            midArr[i]++; // Увеличиваем её значение на 1
    }

    for (size_t i = 0; i < midArr.size(); i++) {
        // Индекс, по которым промежуточный массив хранит количество
// попаданий безмасочных подстрок в текст, есть индекс начала вхождения
маски
        // в текст, при условии, что кол-во попаданий равно кол-ву подстрок б/м
        if (midArr[i] == numSubstrs) {
            result.push_back(i + 1);
        }
    }

    return result;
}

int main()
{
    string text, pattern;
    char joker;
    cin >> text >> pattern >> joker;

    for (auto res : AhoCorasick(text, pattern, joker))
        cout << res << endl;

    return 0;
}

```