

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. 8382

Ершов М.И..

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 4

Реализовать режим поиска, при котором все найденные образцы не пересекаются в строке поиска (т.е. некоторые вхождения не будут найдены; решение задачи неоднозначно).

Описание Бора

Бор – структура данных для хранения набора строк, представляющая из себя дерево с символами на рёбрах. Строки получаются последовательной записью символов между корнем дерева и терминальной вершиной.

Из такой структуры данных возможно построить автомат, для этого необходимо добавить ссылки на максимальные суффиксы строк.

Описание алгоритма задания 1

В программе используется алгоритм Ахо-Корасик.

Для всех строк шаблонов строится автомат по бору. Далее, для каждого символа текста выполняется поиск по автомату.

По возможности переходим либо в потомка, если для текущей вершины он существует, либо 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.

Для того, чтобы найти не пересекающиеся шаблоны в тексте (индивидуализация), был удалён переход по суффиксам и после каждой найденной терминальной вершины - значение текущей позиции в автомате становилось равным корню.

Описание алгоритма задания 2

Здесь шаблонами являются подстроки маски, разделенные символами джокера, обозначим множество таких подстрок как $\{R_1, \dots, R_n\}$. По таким подстрокам также строится автомат по бору. После этого для каждого символа текста выполняется поиск в нём. Появления подстроки в тексте на позиции означает возможное появление маски на позиции $- + 1$, где $-$ индекс начала подстроки в маске. Далее, с помощью вспомогательного массива для таких позиций увеличиваем его значение на 1. Индексы, по которым хранятся значения равному, являются вхождениями маски в текст.

Сложность алгоритма

Построение бора выполняется за $O(m)$, где m – суммарная длина паттернов (для джокеров – $\sum Q_i$). Для построения суффиксных ссылок используется обход в ширину. Его сложность составляет $O(V+E)$, но т.к. кол-во рёбер линейно зависит от кол-ва вершин, то упрощаем до $O(2m) = O(m)$. Прохождение текста по бору составляет $O(n)$, где n – длина текста. В алгоритме поиска маски, в тексте просматривается промежуточный массив, но его размер равен размеру исходного текста, таким образом на сложность это никак не влияет. Итак, сложность по времени составляет $O(m + n)$.

Сложность по памяти для хранения бора составляет $O(m)$ - 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 $O(n*k+m)$.

Описание функций и структур данных

Class `TreeNode` – структура, для хранения данных на вершину бора.

Поля `TreeNode`:

`char value` – символ, по которому был произведён переход;

`TreeNode* parent` – ссылка на родительскую вершину;

`TreeNode* suffixLink` – суффиксная ссылка;

`unordered_map <char, TreeNode*> children` – словарь, ключом которого является символ, по которому можно перейти на потомка;

`size_t numOfPattern` – порядковый номер паттерна (в задании 1)

`vector<pair<size_t, size_t>> substringEntries` – вектор, элементами которого является пара: индекс вхождения в маску и длина подстроки (в задании 2)

Методы `TreeNode`:

`TreeNode(char val)` – конструктор для заполнения поля : значения по которому перешли;

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

`auto find(const char c)` – 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);

`void makeAutomaton()` – делает из бора автомат, путём добавления суффиксных ссылок;

`Class Trie` – обёртка над классом `TreeNode`, состоящая из одного поля и аналогичных методов.

Функции задания 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.

`text` – строка, в которой производится поиск

`patterns` – искомые подстроки

Функции задания 2:

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

`text` – строка, в которой производится поиск

`mask` – маска с джокерами, которая используется для поиска в строке

`joker` – символ-джокер, используемый в маске

Тестирование

Задание 1

Ввод	Вывод
ABCASDTEAD 5 ABC CAS ASD TEA EAD	1 1 4 3 7 4
ABCBABCBA 4 ABC BC CBA BAB	1 1 4 4 7 3
CATNATCAT 3 AT CAT NA	1 2 4 3 7 2
CCCA 1 CC	1 1

Тест с подробным промежуточным выводом:

ABABA

1

ABA

Вставляем строку: ABA	Бор сейчас:
Корень:	Корень:
Потомок: A	Потомок: A
A:	Суффиксная ссылка: Root
Родитель: Корень	Родитель: Корень
Потомок: B	Потомок: B
AB:	Суффиксная ссылка: Root
Родитель: A	Родитель: A
Потомок: A	Потомок: A
ABA:	Суффиксная ссылка: A
Родитель: AB	Родитель: AB
Строим автомат:	
A:	Ищем 'A' из: Корень
Родитель: Корень	Символ 'A' найден!
Потомок: B	Ищем 'B' из: A
Суффиксная ссылка: Корень	Символ 'B' найден!
AB:	Ищем 'A' из: AB
Родитель: A	Символ 'A' найден!
Потомок: A	Ищем 'B' из: ABA
Суффиксная ссылка: Корень	Переходим по суффиксной ссылке: A
ABA:	Символ 'B' найден!
Родитель: AB	Ищем 'A' из: AB
Суффиксная ссылка: A	Символ 'A' найден!
Бор сейчас:	1 1
Корень:	3 1

Задание 2:

Ввод	Вывод
ACTANCA A\$\$A \$	1
CATNATCAT \$AT \$	1 4 7

Тест с подробным промежуточным выводом

ABCBABC

B\$B

\$


```
Вставляем строку: В
Бор сейчас:
Корень:
    Потомок: В
В:
    Родитель: Корень

Вставляем строку: В
Бор сейчас:
Корень:
    Потомок: В
В:
    Родитель: Корень

Строим автомат:
В:
    Родитель: Корень
    Суффиксная ссылка: Корень

Бор сейчас:
Корень:
    Потомок: В
В:
    Суффиксная ссылка: Root
    Родитель: Корень

Ищем 'А' из: Корень
Символ 'А' не найден!
```

```
Ищем 'А' из: Корень
Символ 'А' не найден!
Ищем 'В' из: Корень
Символ 'В' найден!
Ищем 'С' из: В
Переходим по суффиксной ссылке: Корень
Символ 'С' не найден!
Ищем 'В' из: Корень
Символ 'В' найден!
Ищем 'А' из: В
Переходим по суффиксной ссылке: Корень
Символ 'А' не найден!
Ищем 'В' из: Корень
Символ 'В' найден!
Ищем 'С' из: В
Переходим по суффиксной ссылке: Корень
Символ 'С' не найден!
2
```

Вывод

В ходе выполнения лабораторной работы была изучена работа алгоритма Ахо-Корасик. Алгоритм был использован для нахождения вхождений множества строк в тексте, а также для нахождения шаблона с джокером.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ 1

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

using namespace std;

class TreeNode {
public:
    explicit TreeNode(char val) : value(val) {} // Конструктор ноды

    // Отладочная функция для печати бора
    void printTrie() {
        cout << "Бор сейчас:" << 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;

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

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

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

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

    // 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 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;
    printTrie();

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

// Функция для поиска подстроки в строке при помощи автомата
vector<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; // 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> found; // Вектор номеров найденных терм. вершин

                // ИНДИВИДУАЛИЗАЦИЯ
                // if (curr->numOfPattern) { // Для пропуска пересечений, после нахождения терминальной
                // вершины
                //     found.push_back(curr->numOfPattern - 1); // Добавляем к найденным эту вершину
                //     curr = this; // И переходим в корень
                // }
                // НЕ ИНДИВИДУАЛИЗАЦИЯ
                // Обходим суффиксы, т.к. они тоже могут быть терминальными вершинами
                for (auto temp = curr; temp->suffixLink; temp = temp->suffixLink)
                    if (temp->numOfPattern)
                        found.push_back(temp->numOfPattern - 1);
                //

                cout << "Символ '" << c << "' найден!" << endl; // Дебаг
                return found;
            }

        if (curr->suffixLink) {
            cout << "Переходим по суффиксной ссылке: ";
            cout << (curr->suffixLink->dbgStr.empty() ? "Корень" : curr->suffixLink->dbgStr) << endl;
        }
    }
    cout << "Символ '" << c << "' 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;
if (curr->parent && curr->parent->value)
    cout << "\tРодитель: " << curr->parent->dbgStr << endl;
else if (curr->parent)
    cout << "\tРодитель: Корень" << endl;

if (!curr->children.empty())
    cout << "\tПотомок: ";
//

// Заполняем очередь потомками текущей верхушки
for (auto child : curr->children) {
    cout << child.second->value << ' '; // Дебаг
    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;
printTrie();
}

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

private:
string dbgStr = ""; // Для отладки
char value; // Значение ноды
size_t numOfPattern = 0; // Номер введенного паттерна
TreeNode *parent = nullptr; // Родитель ноды
TreeNode *suffixLink = nullptr; // Суффиксная ссылка
unordered_map<char, TreeNode*> children; // Потомок ноды
};

class Trie {
public:
    Trie() : root('\0') {} // Конструктор бора

    void insert(const string &str) { root.insert(str); }
    auto find(const char c) { return root.find(c); }
    void makeAutomaton() { root.makeAutomaton(); }

```

```

private:
    TreeNode root; // Корень бора
};

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

    for (const auto &pattern : patterns) // Заполняем бор введенными паттернами
        bor.insert(pattern);

    bor.makeAutomaton(); // Из полученного бора создаем автомат (путем добавления суффиксных ссылок)

    for (size_t j = 0; j < text.size(); j++) // Проходим циклом по строке, для каждого символа строки запускаем
поиск
        for (auto pos : bor.find(text[j])) // Проходим по всем найденным позициям, записываем в результат
            result.emplace(j - patterns[pos].size() + 2, pos + 1);

    return result;
}

int main()
{
    system("chcp 65001");

    string text;
    size_t n;
    cin >> text >> n;

    vector <string> patterns(n);

    for (size_t i = 0; i < n; i++)
        cin >> patterns[i];

    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 {
public:
    explicit TreeNode(char val) : value(val) {} // Конструктор ноды

    // Отладочная функция для печати бора
    void printTrie() {
        cout << "Бор сейчас:" << 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;

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

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

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

        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;
printTrie();

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

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; // 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
                // вектор пар, состоящих из начала безмасочной подстроки в маске и её длины
                vector <pair<size_t, size_t>> found;

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

                cout << "Символ '" << c << "' найден!" << endl; // Дебаг
                return found;
            }

        // Дебаг
        if (curr->suffixLink) {
            cout << "Переходим по суффиксной ссылке: ";
            cout << (curr->suffixLink->dbgStr.empty() ? "Корень" : curr->suffixLink->dbgStr) << endl;
        }
    }
    cout << "Символ '" << c << "' 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;
        if (curr->parent && curr->parent->value)
            cout << "\tРодитель: " << curr->parent->dbgStr << endl;
        else if (curr->parent)
            cout << "\tРодитель: Корень" << endl;

        if (!curr->children.empty())
            cout << "\tПотомок: ";
    }
}

```

```

//

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

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

// Дебар
cout << endl;
printTrie();
}

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

private:
    string dbgStr = ""; // Для отладки
    char value; // Значение ноды
    TreeNode *parent = nullptr; // Родитель ноды
    TreeNode *suffixLink = nullptr; // Суффиксная ссылка
    unordered_map <char, TreeNode*> children; // Потомок ноды
    vector <pair<size_t, size_t>> substringEntries;
};

class Trie {
public:
    Trie() : root('\0') {}

    void insert(const string &str, size_t pos, size_t size)
    {
        root.insert(str, pos, size);
    }

    auto find(const char c)
    {
        return root.find(c);
    }

    void makeAutomaton()

```



```

{
    root.makeAutomaton();
}

private:
    TreeNode root;
};

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

    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);

            // ИНДИВИДУАЛИЗАЦИЯ
            // для пропуска пересечений, после найденного индекса, увеличиваем его на длину маски
            i += mask.size() - 1;
        }
    }

    return result;
}

int main()
{
    system("chcp 65001");

    string text, mask;
    char joker;
    cin >> text >> mask >> joker;

    for (auto ans : AhoCorasick(text, mask, joker))
        cout << ans << endl;
}

```

```
    return 0;  
}
```