

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. 9382

Павлов Р.В.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

Цель работы

Изучение и реализация алгоритма Ахо-Корасик для поиска вхождений нескольких шаблонов или вхождения шаблона с маской в текст.

Задание 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

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

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 $ab??c?$ 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 $\{A, C, G, T, N\}$

Вход:

Текст ($T, 1 \leq |T| \leq 1000000$)

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

Задание 3 (индивидуализация)

Вариант 3. Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

Описание алгоритма

1. Алгоритм Ахо-Корасик (поиск вхождений безмасочных шаблонов).

1) Рассматривается каждый шаблон в данном наборе. По шаблонам строится структура данных бор. Создаётся начальная вершина – корень – из которой выходят рёбра, помеченные символами начал шаблонов.

а) Если для текущего рассматриваемого символа шаблона не существует вершины, в которую можно перейти по ребру, помеченному данным символом, то создаётся такая вершина и соответствующее ребро.

б) Выполняется переход по ребру, помеченному данным символом, в следующую вершину.

с) Если в шаблоне ещё есть символы, вернуться к пункту а), иначе пометить вершину терминальной (конечной) и перейти к следующему шаблону, продолжить построение, начиная с корня бора.

2) На основе бора строится автомат, содержащий для каждой вершины суффиксные и конечные ссылки. Суффиксные ссылки ведут в максимальный префикс, являющийся максимальным суффиксом для строки, построенной на основе пути до данной вершины, причём могут вести в ветви бора, построенные для других шаблонов. Конечные ссылки ведут в концы других шаблонов.

а) Если текущая вершина – корень, то суффиксная ссылка – корень.

Иначе суффиксная ссылка – это вершина, в которую ведёт ребро с данным символом из суффиксной ссылки родительской вершины. При этом, если ссылка ищется для вершины, следующей за корнем, то для неё ссылка будет корнем. Пока такого ребра нет, перейти к пункту а).

б) Пока текущая суффиксная ссылка – не корень:

Если текущая суффиксная ссылка – терминальная вершина, конечная ссылка найдена.

Иначе перейти к суффиксной ссылке текущей ссылки, перейти к пункту б).

3) Посимвольно рассматривается строка, в которой ищутся вхождения. Начальная вершина – корень.

Если из текущей вершины выходит ребро, помеченное текущим рассматриваемым символом, перейти по нему в следующую вершину, иначе переходить по суффиксным ссылкам, пока такое ребро не будет найдено или не будет встречен корень и при этом из него также не будет выходить такое ребро (в последнем случае осуществляется переход в корень). Если вершина, в которую осуществлён переход, терминальная, добавить информацию о вхождении в строку соответствующего ей

шаблона в список. Если для этой вершины конечная ссылка не пуста, переходить по конечным ссылкам, пока они не пусты, и для каждой также добавлять информацию о вхождении.

2. Алгоритм Ахо-Корасик (поиск вхождений шаблона с маской).

Для данного алгоритма также строится бор, но не для шаблона, а для безмасочных подшаблонов, находящихся в нём. Выделяется массив индексов, длина которого равна длине рассматриваемой строки, инициализированный нулями. На основе бора строится автомат, и дальше выполняется посимвольное рассмотрение строки.

Если в строке нашёлся какой-либо подшаблон, то ячейка массива по адресу, образованному разностью номера начального символа данного вхождения подшаблона в строке и его смещения относительно начала исходного шаблона (если этот адрес не меньше 0), инкрементируется. Если у подшаблона несколько смещений, то данная операция выполняется для каждого из них.

В итоге индексы тех ячеек массива, значение которых будет равно количеству подшаблонов в исходном шаблоне, и будут индексами вхождения заданного шаблона в строку.

3. Поиск длин самых длинных цепочек из суффиксных и конечных ссылок.

Рекурсивно рассматриваются вершины бора. Для каждой из них выполняется переход по суффиксным ссылкам, пока не встречен корень, подсчитывается длина цепочки из суффиксных ссылок. Аналогичным образом подсчитывается и длина цепочки конечных ссылок, но переход и увеличение длины осуществляется только при наличии конечной ссылки.

Сложность алгоритма

1. Алгоритм Ахо-Корасик (поиск вхождений безмасочных шаблонов).

Сложность по памяти.

Поскольку автомат хранится как красно-чёрное дерево, его сложность по памяти – $O(n)$, где n – суммарная длина шаблонов.

Сложность по времени.

Сложность алгоритма по времени в данном случае – $O((T + n)\log(s) + k)$, где T – длина строки, в которой ищутся вхождения, s – размер алфавита, k – общее количество вхождений шаблонов в текст, так как время на построение бора (и автомата) сокращается по сравнению с другими реализациями (лишние символы не добавляются), что влияет и на время обработки строки.

2. Алгоритм Ахо-Корасик (поиск вхождений шаблона с маской).

Сложность по памяти.

Поскольку ищется один шаблон, то сложность по памяти составит $O(n + T)$, где n – суммарная длина всех подшаблонов в шаблоне с маской, а T – длина строки (на сложность влияет добавление массива индексов).

Сложность по времени.

Поскольку время тратится также и на заполнение массива индексов, сложность по памяти составляет $O((T + n)\log(s) + k * p)$, где s – размер алфавита, где k – общее количество вхождений шаблонов в текст, p – суммарное количество сдвигов подшаблонов относительно исходного шаблона.

3. Поиск длин самых длинных цепочек из суффиксных и конечных ссылок.

Сложность по памяти.

Сложность по памяти данного алгоритма – $O(n)$, где n – суммарная длина всех шаблонов, поскольку рассматриваются вершины бора, и для каждой хранится пара значений с наибольшими длинами.

Сложность по времени.

Для каждой вершины рассматриваются цепочки из суффиксных и конечных ссылок (первых в боре не больше, чем количество символов в самом длинном шаблоне – a , а последних – не больше, чем шаблонов – b), поэтому сложность по времени данного алгоритма составляет $O(n \cdot (a + b))$.

Описание функций и СД

- `struct Node {`
 `unordered_map<char, Node*> next;` // рёбра, по которым можем перейти
 `vector<int> shifts;` // сдвиги подстрок в шаблоне
 `Node* parent;` // родительская вершина (откуда пришли)
 `Node* link;` // суффиксная ссылка
 `Node* tLink;` // конечная ссылка
 `char toParent;` // ребро, по которому пришли из родительской вершины
 `bool terminal;` // является ли терминальной вершиной (признак конца шаблона)
 `vector<int> ptnNum;` // номера шаблонов, в которые входит символ, по которому пришли
 `int termPtnNum;` // номер шаблона конечной вершины

 • `Node(Node* pr = nullptr, const char toPr = 0)` – конструктор; `pr` – указатель на родительскую вершину, `toPr` – символ, ведущий в неё
 • `Node* getLink(const char& c)` – получение следующей вершины для перехода; `c` – символ, по которому требуется перейти
 • `~Node()` – деструктор
};
- `void indent(int n)` – отступ для вывода промежуточной информации в рекурсивном алгоритме; `n` – глубина рекурсии

- **Node* createBohr(const vector<pair<string, int>>& patterns)** – построение бора; **patterns** – набор шаблонов для поиска
- **void charInfo(Node* n)** – вывод информации о вершине; **n** – указатель на вершину
- **void writeLinks(Node* bohr)** – вычисление ссылок в боре (построение автомата); **bohr** – указатель на корень бора
- **pair<int, int> longestLinks(Node* bohr, Node* root, int& depth)** – вычисление длин наибольших цепочек из суффиксных и конечных ссылок; **bohr** – указатель на текущую вершину, **root** – указатель на корень бора, **depth** – глубина рекурсии
- **void ahoCor(const string& t, const vector<pair<string, int>>& patterns, vector<pair<int, int>>& res)** – алгоритм поиска вхождений в строку; **t** – строка, в которой ищутся вхождения шаблонов, **patterns** – набор шаблонов, **res** – список результатов.
- **void preparePts(const string& p, const char& j, vector<pair<string, int>>& patterns)** – разбиение шаблона с маской на безмасочные подшаблоны; **p** – шаблон с маской, **j** – символ-разделитель, **patterns** – список пар вида <шаблон : смещение>, куда записываются безмасочные подшаблоны с соответствующими смещениями их относительно начала строки-шаблона с маской.
- **int main()** – ввод исходных данных, вывод результатов

Тестирование

Результаты тестирования для задания 1 представлены в таблице 1.

Таблица 1

Входные данные	Выходные данные
NTAG 3 TAGT TAG T	2 2 2 3
kkkaratotarakan 10 ot tak rak rab ara arka tara otara karat rabota	3 9 4 5 8 1 8 8 9 7 10 5 11 3
algorithm 2 alco hmm	Поиск не дал результатов.

Результаты тестирования для задания 2 представлены в таблице 2.

Таблица 2

Входные данные	Выходные данные
what_the_hell_is_going_on_on_the_hill?! the_#### #	6 30
ссabxabxxсхасabсababсхассabxabaccсхаb **ab*ab**с*ac *	1 12

Результаты тестирования для задания 3 представлены в таблице 3.

Таблица 3

Входные данные	Выходные данные
oabababaoaboababobaaoaabbabboabaoab 3 aba aboba abb	<p>Длина наибольшей цепочки из суффиксных ссылок: 2</p> <p>Длина наибольшей цепочки из конечных ссылок: 0</p> <p>2 1</p> <p>4 1</p> <p>6 1</p> <p>13 1</p> <p>15 2</p> <p>22 3</p> <p>25 3</p> <p>29 1</p>
gavgaxgongignugulag 3 g ag lag	<p>Длина наибольшей цепочки из суффиксных ссылок: 3</p> <p>Длина наибольшей цепочки из конечных ссылок: 2</p> <p>1 1</p> <p>4 1</p> <p>7 1</p> <p>10 1</p> <p>12 1</p> <p>15 1</p> <p>17 3</p> <p>18 2</p> <p>19 1</p>
testing_costs_300_dollars l**sl *	<p>Длина наибольшей цепочки из суффиксных ссылок: 2</p> <p>Длина наибольшей цепочки из конечных ссылок: 1</p> <p>Поиск не дал результатов</p>

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

Имя файла: alg_lab5.cpp

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

#define TASK 2
#define DEBUG

using namespace std;

struct Node {
    unordered_map<char, Node*> next; // рёбра, по которым можем перейти
#if TASK == 2
    vector<int> shifts; // сдвиги подстрок в шаблоне
#endif
    Node* parent; // родительская вершина (откуда пришли)
    Node* link; // суффиксная ссылка
    Node* tLink; // конечная ссылка
    char toParent; // ребро, по которому пришли из родительской вершины
    bool terminal; // является ли терминальной вершиной (признак конца шаблона)
    vector<int> ptnNum; // номера шаблонов, в которые входит символ, по которому
    пришли
    int termPtnNum;

    Node(Node* pr = nullptr, const char toPr = 0) : parent(pr), toParent(toPr),
    link(nullptr), tLink(nullptr), terminal(false) {
        if (pr == nullptr || toPr == 0) { // если создаём корень (он ссылается на себя
            же)
            this->parent = this;
            this->link = this;
        }
    }

    // поиск следующей (для перехода) вершины при поиске в строке
    Node* getLink(const char& c) {
        if (this->next.find(c) != this->next.end()) { // если нашли путь по заданному
            символу из текущей вершины
                return this->next.at(c);
            }
        if (this->link == this) { // если дошли до корня, не найдя пути
            return this;
        }
        return this->link->getLink(c); // если не нашли путь, но ещё не в корне
    }

    ~Node() {
        for (auto l : this->next) {
            delete l.second;
        }
    }
};

#ifdef DEBUG
void indent(int n) {
    while (n > 0) {
        cout << "\t";
    }
}
```

```

        n--;
    }
}
#endif

// создание бора
Node* createBohr(const vector<pair<string, int>>& patterns) {
#ifdef DEBUG
    cout << "\nПостроение бора\n";
#endif
    Node* bohr = new Node;    // корень бора
    for (auto& pt : patterns) {
        int ptnNum = find(patterns.begin(), patterns.end(), pt) - patterns.begin();
#ifdef DEBUG
        cout << "\n-> Рассматривается " << ptnNum + 1 << "-й шаблон: " << pt.first <<
"\n";
#endif
        Node* cur = bohr;    // ищем путь, начиная с корня
        for (auto& c : pt.first) {
            if (cur->next.find(c) == cur->next.end()) { // если такого ребра ещё
нет в cur->next, добавляем
                cur->next.insert({ c, new Node(cur, c) });
#ifdef DEBUG
                cout << "\t-> В бор добавлена вершина, в которую ведёт текущий
символ шаблона (" << c << ")\n";
#endif
            }
            else {
                cout << "\t-> Ребро (" << c << ") для текущего шаблона уже
существует, выполняется переход по нему\n";
            }
            cur = cur->next[c]; // переходим по данному ребру
            cur->ptnNum.push_back(ptnNum);
        }
        // для конечной вершины
        cur->termPtnNum = ptnNum;
        cur->terminal = true;
#ifdef DEBUG
        cout << "\t-> Вершина, в которую выполнен переход, является терминальной,
закончено построение ветви бора\n";
#endif
    }
    if TASK == 2
        cur->shifts.push_back(pt.second);
#ifdef DEBUG
        cout << "-> Данный шаблон имеет сдвиг " << pt.second << " относительно начала
шаблона с маской\n";
#endif
    }
#ifdef DEBUG
    cout << "\n";
#endif
    return bohr;
}

#ifdef DEBUG
void charInfo(Node* n) {
    cout << n->toParent << ") из ";
    if (n->ptnNum.size() == 1) cout << "шаблона № " << n->ptnNum.at(0) + 1 << "\n";
    else if (n->ptnNum.size() > 1) {
        cout << "шаблонов № " << n->ptnNum.at(0) + 1;
        for (int i = 1; i < n->ptnNum.size(); i++) {
            cout << ", " << n->ptnNum.at(i) + 1;

```

```

    }
    cout << "\n";
}
#endif

// нахождение суффиксных и конечных ссылок
void writeLinks(Node* bohr) {
#ifdef DEBUG
    cout << "\nВычисление суффиксных и конечных ссылок\n";
#endif
    queue<Node*> front({ bohr });    // вершины одного уровня в боре
    while (!front.empty()) {
        Node* cur = front.front();
        front.pop();
        Node* curLink = cur->parent->link; // взяли родительскую ссылку в качестве
текущей
        const char& key = cur->toParent; // запомнили символ, для которого ищем
ссылку
        bool foundLink = true;    // по умолчанию ссылка находится
#ifdef DEBUG
        cout << "\n-> Текущая вершина ";
        if (cur->link == cur) cout << "- корень\n";
        else {
            cout << "- (";
            charInfo(cur);
        }
#endif
        //-----
        while (curLink->next.find(key) == curLink->next.end()) { // пока из суффиксной
ссылки не найден переход по key
            if (curLink == bohr) {
#ifdef DEBUG
                cout << "\t-> Суффиксные ссылки не найдены, ссылка установлена
на корень\n";
#endif
                cur->link = bohr;    // если и из корня нет пути, то ссылку
устанавливаем в корень
                foundLink = false; // и ссылка, не равная корню, не была
найдена
                break;
            }
            curLink = curLink->link;
        }
        if (foundLink) {
            curLink = curLink->next.at(key); // это ссылка для key
            if (cur->parent == bohr) {
#ifdef DEBUG
                cout << "\t-> Текущая вершина является началом слова, ссылка
установлена на корень\n";
#endif
                cur->link = bohr;    // так или иначе, для вершин первого уровня
ссылки ведут в корень
                // (если это не учесть, эти
                // вершины будут ссылаться на себя)
            }
            else {
#ifdef DEBUG
                cout << "\t-> Для текущей вершины найдена суффиксная ссылка (";
                charInfo(curLink);
#endif
                cur->link = curLink;
                Node* curTlink = cur->link;
                while (curTlink != bohr) { // поиск конечной ссылки, если дошли
до корня - её нет

```

```

        if (curTlink->terminal) {
#ifdef  DEBUG
            cout << "\t-> Для текущей вершины найдена конечная
(сжатая) ссылка (";
            charInfo(curTlink);
#endif
            cur->tLink = curTlink;
            break;
        }
        curTlink = curTlink->link;
    }
}

//-----

if (!cur->next.empty()) { // добавляем новые вершины в очередь
    for (auto& nxt : cur->next) {
        front.push(nxt.second);
    }
}

}

pair<int, int> longestLinks(Node* bohr, Node* root, int& depth) {
    pair<int, int> longest = { 0, 0 };
#ifdef  DEBUG
    cout << "\n";
    indent(depth);
    cout << "-> Рассматривается ";
    if (bohr == root) {
        cout << "корень\n";
    }
    else {
        cout << "вершина (";
        charInfo(bohr);
    }
#endif
    Node* cur = bohr;
    while (cur->link != root) {
#ifdef  DEBUG
        indent(depth);
        cout << "\t-> Текущая суффиксная ссылка: (";
        charInfo(cur->link);
#endif
        longest.first++;
        cur = cur->link;
    }
    longest.first++;
#ifdef  DEBUG
    indent(depth);
    cout << "\t-> Текущая суффиксная ссылка - корень, поиск закончен\n";
#endif
    cur = bohr;
    while (cur->tLink != nullptr) {
#ifdef  DEBUG
        indent(depth);
        cout << "\t-> Текущая конечная (сжатая) ссылка: (";
        charInfo(cur->tLink);
#endif
        longest.second++;
        cur = cur->tLink;
    }
#ifdef  DEBUG
    indent(depth);
    cout << "\t-> Конечных ссылок нет, поиск закончен\n";

```

```

        indent(depth);
        cout << "-> Длина цепочки из суффиксных ссылок для данной вершины : " <<
longest.first << "\n";
        indent(depth);
        cout << "-> Длина цепочки из конечных ссылок для данной вершины: " << longest.second
<< "\n";
    #endif
    for (auto& n : bohr->next) {
        pair<int, int> nextLon = longestLinks(n.second, root, ++depth);
        if (nextLon.first > longest.first) {
            longest.first = nextLon.first;
        }
        if (nextLon.second > longest.second) {
            longest.second = nextLon.second;
        }
    }
    depth--;
    return longest;
}

void ahoCor(const string& t, const vector<pair<string, int>>& patterns, vector<pair<int,
int>>& res, int ptnLength = 0) {
    Node* bohr = createBohr(patterns);
    writelinks(bohr);
#ifdef DEBUG
    cout << "\n\nПоиск самых длинных цепочек из суффиксных и конечных (сжатых) ссылок\n";
#endif
    int depth = 0;
    pair<int, int> longest = longestLinks(bohr, bohr, depth);
    cout << "\nДлина наибольшей цепочки из суффиксных ссылок: " << longest.first <<
"\nДлина наибольшей цепочки из конечных ссылок: " << longest.second << "\n";
#ifdef DEBUG
    cout << "\n\nПоиск вхождений шаблонов в строке\n";
#endif
    Node* cur = bohr;
    res.clear();
    if TASK == 2
        vector<int> tInd(t.length(), 0);
    #endif

    for (int i = 0; i < t.length(); i++) {
        cur = cur->getLink(t.at(i)); // получили ссылку для перехода (для
текущего символа строки)
#ifdef DEBUG
        cout << "\n-> Текущий символ строки: " << t[i] << ", текущая вершина: ";
        if (cur == bohr) {
            cout << "корень\n";
        }
        else {
            cout << "(";
            charInfo(cur);
        }
    }
#endif
    Node* tLink = cur->tLink;
    while (tLink != nullptr) { // если у этой вершины есть конечная ссылка,
записываем, что нашли соответствующий ей шаблон
#ifdef DEBUG
        cout << "\t-> Найдена конечная ссылка (";
        charInfo(tLink);
    }
#endif
    if TASK == 1
        res.push_back({ i - patterns.at(tLink->termPtnNum).first.length() + 2,
tLink->termPtnNum + 1 });
#ifdef DEBUG

```

```

        cout << "\t-> Вхождение " << patterns.at(tLink->termPtnNum).first << "
добавлено в список результатов\n";
#endif
#elif TASK == 2
    for (auto& sh : tLink->shifts) {
        int idx = i - patterns.at(tLink->termPtnNum).first.length() - sh
+ 1;
        if (!(idx < 0)) {
            tInd.at(idx)++;
        }
    }
#ifdef DEBUG
    cout << "\t-> Текущее состояние массива, в котором записано количество
наложений шаблонов:\n\n\t-> ";
    for (auto& e : tInd) {
        cout << e << " ";
    }
    cout << "\n\n";
#endif
#endif
    tLink = tLink->tLink;        // и так, пока цепочка из конечных ссылок не
прервётся
    }
    if (cur->terminal) { // если терминальная - шаблон найден
#ifdef DEBUG
        cout << "\t-> Текущая вершина - терминальная\n";
#endif
        if TASK == 1
            res.push_back({ i - patterns.at(cur->termPtnNum).first.length() + 2,
cur->termPtnNum + 1 });
#ifdef DEBUG
        cout << "\t-> Вхождение " << patterns.at(cur->termPtnNum).first << "
добавлено в список результатов\n";
#endif
        elif TASK == 2
            for (auto& sh : cur->shifts) {
                int idx = i - patterns.at(cur->termPtnNum).first.length() - sh +
1;
                if (!(idx < 0)) {
                    tInd.at(idx)++;
                }
            }
#ifdef DEBUG
            cout << "\t-> Текущее состояние массива, в котором записано количество
наложений шаблонов:\n\n\t-> ";
            for (auto& e : tInd) {
                cout << e << " ";
            }
            cout << "\n\n";
#endif
        endif
    }
}

}

#if TASK == 2
    for (int i = 0; i < tInd.size(); i++) {
        if (tInd[i] == patterns.size() && i + ptnLength <= t.length()) {
            res.push_back({ i + 1, 0 });
        }
    }
#endif
    delete bohr;
}

```



```

#if TASK == 2
void preparePts(const string& p, const char& j, vector<pair<string, int>>& patterns) {
    int prev = 0;
    size_t delim;
    do {
        delim = p.find(j, prev);
        if (delim != prev && prev != p.length()) {
#ifdef DEBUG
            cout << "Найден безмасочный шаблон: " << p.substr(prev, delim - prev)
<< ", смещение относительно начала исходного шаблона: " << prev << "\n";
#endif
            patterns.push_back({ p.substr(prev, delim - prev), prev });
        }
        prev = delim + 1;
    } while (delim != string::npos);
}
#endif

int main() {
    setlocale(LC_ALL, "rus");

    string t, p;
    char j;
    vector<pair<string, int>> pts;
    vector<pair<int, int>> res;
    int num = 0;

    cin >> t;
    #if TASK == 1
        std::cin >> num;
        for (int i = 0; i < num; i++) {
            string s;
            std::cin >> s;
            pts.push_back({ s, 0 });
        }
        ahoCor(t, pts, res);
    #elif TASK == 2
        cin >> p;
        cin >> j;
        preparePts(p, j, pts);
        ahoCor(t, pts, res, p.length());
    #endif

    if (res.empty()) {
        cout << "\nПоиск не дал результатов\n";
    }
    else {
        sort(res.begin(), res.end());
        cout << "\nРезультаты поиска (индекс вхождения ";
    #if TASK == 1
        cout << "в строке, порядковый номер шаблона):\n";
    #elif TASK == 2
        cout << "шаблона в строку):\n";
    #endif
        for (auto r : res) {
            cout << r.first;
            #if TASK == 1
                cout << " " << r.second;
            #ifdef DEBUG
                cout << " - " << pts.at(r.second - 1).first;
            #endif
            #endif
            cout << "\n";
        }
    }
}

```

```
    }  
    system("pause");  
    return 0;  
}
```