

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

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

Студент гр. 8382

\_\_\_\_\_

Черцницын П.А.

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 алгоритма**

Создаётся корневая вершина бора. В бор добавляются введенные шаблоны.

После добавления в бор шаблонов, для всех его вершин вычисляются суффиксные ссылки. Для корня и его детей ссылка ведёт в корень. Для остальных определяется по следующему правилу: выполняется переход по ссылке родителя. Проверяется, если ли среди потомков данной вершины переход по той же букве, что и в исходную вершину. Если есть, суффиксная ссылка исходной вершины устанавливается на найденную вершину. Если нет, то выполняется переход по суффиксной ссылке данной вершины и процесс повторяется. Если нужная вершина не найдена, суффиксная ссылка устанавливается на корень.

### **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.

Дополнительно было реализовано два режима работы программы: обычный – без вывода промежуточных данных. И режим с выводом промежуточных данных (для запуска прописать флаг `-detail` или `-d`).

## **Сложность алгоритма**

Сложность первого алгоритма:

- Память:  $O(nq)$ , где  $n$  – общая длина слов в словаре,  $q$  – Размер алфавита
- Вычислительная:  $O(nq + N + k)$ , где  $N$  – длина текста,  $k$  – общая длина всех совпадений.

Сложность второго алгоритма:

- Память:  $O(nq)$ , где  $n$  – общая длина слов в словаре,  $q$  – Размер алфавита
- Вычислительная:  $O(nq + N + k)$ , где  $N$  – длина текста,  $k$  – общая длина всех совпадений.

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

Class Data – хранит вводимую пользователем информацию.

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

Запуск программы aho-korasik.cpp. Больше тестов приведено в таблице ниже.

```
Active code page: 65001
NTAG
З
TAGT
TAG
Т
Вставляем строку: TAGT
Бор сейчас:
Корень:
    Потомок: Т
Т:
    Родитель: Корень
    Потомок: А
ТА:
    Родитель: Т
    Потомок: G
TAG:
    Родитель: ТА
    Потомок: Т
TAGT:
    Родитель: TAG

Вставляем строку: TAG
Бор сейчас:
Корень:
    Потомок: Т
Т:
    Родитель: Корень
    Потомок: А
ТА:
    Родитель: Т
    Потомок: G
TAG:
    Родитель: ТА
    Потомок: Т
TAGT:
    Родитель: TAG
```

```
Вставляем строку: Т
Бор сейчас:
Корень:
    Потомок: Т
Т:
    Родитель: Корень
    Потомок: А
ТА:
    Родитель: Т
    Потомок: G
TAG:
    Родитель: ТА
    Потомок: Т
TAGT:
    Родитель: TAG

Строим автомат:
Т:
    Родитель: Корень
    Потомок: А
    Суффиксная ссылка: Корень
ТА:
    Родитель: Т
    Потомок: G
    Суффиксная ссылка: Корень
TAG:
    Родитель: ТА
    Потомок: Т
    Суффиксная ссылка: Корень
TAGT:
    Родитель: TAG
    Суффиксная ссылка: Т
```

```
Бор сейчас:
Корень:
    Потомок: Т
Т:
    Суффиксная ссылка: Root
    Родитель: Корень
    Потомок: А
ТА:
    Суффиксная ссылка: Root
    Родитель: Т
    Потомок: G
TAG:
    Суффиксная ссылка: Root
    Родитель: ТА
    Потомок: Т
TAGT:
    Суффиксная ссылка: Т
    Родитель: TAG

Ищем 'N' из: Корень
Символ 'N' не найден!
Ищем 'Т' из: Корень
Символ 'Т' найден!
Найден шаблон: Т
Ищем 'А' из: Корень
Символ 'А' не найден!
Ищем 'G' из: Корень
Символ 'G' не найден!
2 3
```



Запуск программы joker.cpp. Больше тестов приведено в таблице ниже.

```
АСТАНСА
A$$A$
$
Вставляем строку: A
Бор сейчас:
Корень:
    Потомок: A
A:
    Родитель: Корень

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

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

Бор сейчас:
Корень:
    Потомок: A
A:
    Суффиксная ссылка: Root
    Родитель: Корень
```

```
Ищем 'A' из: Корень
Символ 'A' найден!
Ищем 'C' из: A
Переходим по суффиксной ссылке: Корень
Символ 'C' не найден!
Ищем 'T' из: Корень
Символ 'T' не найден!
Ищем 'A' из: Корень
Символ 'A' найден!
Ищем 'N' из: A
Переходим по суффиксной ссылке: Корень
Символ 'N' не найден!
Ищем 'C' из: Корень
Символ 'C' не найден!
Ищем 'A' из: Корень
Символ 'A' найден!
1
```

Таблица тестирования

aho-korasik input	aho-korasik output
NTAG 3 TAGT TAG T	2 3
hello, world. hehello 3 hello hell world	1 2 8 3 17 2
shelfhers 5 he she her hers	1 2 6 1
qwerty 3 qwe we ret	1 1
Joker input	Joker output
qwerty srty s	3
ACTANCA A\$\$A\$ \$	1
hello, world, helllo hel\$\$ \$	1 15
qweqwrqwt qw\$ \$	1 4 7

## **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 АНО-KORASIK

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

bool d_flag = false; //флаг вывода промежуточных данных

class Data
//здесь храним считанные данные
{
private:
    std::string text;          //текст
    int n;                    //кол-во паттернов
    std::vector<std::string> patterns; //массив паттернов
public:
    Data() = default;
    void init()
    {
        getline(std::cin, text);
        std::cin >> n;
        patterns.resize(n);
        for(int i = 0; i < n; i++)
            std::cin >> patterns[i];
    }
    void printText()
    {
        std::cout << "Text = {" << text << "};" << std::endl;
        std::cout << "N = " << n << "; Patterns = {";
        std::cout << patterns[0];
        for (int i = 1; i < n; i++)
            std::cout << ", " << patterns[i];
        std::cout << "};" << std::endl;
    }
    std::string getText() { return text; }
    std::vector<std::string> getPatterns() { return patterns; }
};

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];
    }

    if (d_flag)
    {
        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
if (d_flag) 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

            if (curr->numOfPattern) { // Для пропуска пересечений, после нахождения терминальной
                found.push_back(curr->numOfPattern - 1); // Добавляем к найденным эту вершину
                curr = this; // И переходим в корень
            }

            if (d_flag) cout << "Символ '" << c << "' найден!" << endl; // Дебаг
            return found;
        }

    if (d_flag && curr->suffixLink) {
        cout << "Переходим по суффиксной ссылке: ";
        cout << (curr->suffixLink->dbgStr.empty() ? "Корень" : curr->suffixLink->dbgStr) << endl;
    }
}

if (d_flag) cout << "Символ '" << c << "' не найден!" << endl; // Дебаг

curr = this;
return {};
}

// Функция для построения недетерминированного автомата
void makeAutomaton() {
    if (d_flag) 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
        if (d_flag) {
            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 вершины
for (auto child : curr->children) {
    if (d_flag) cout << child.second->value << ' '; // Дебаг
    queue.push(child.second);
}

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

// Дебаг
if (d_flag)
{
    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); }
    vector<size_t> find(const char c) { return root.find(c); }

```

```

void makeAutomaton() { root.makeAutomaton(); }

private:
    TreeNode root; // Корень бора
};

set <pair<size_t, size_t>> 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(int argc, char** argv)
{
    //читаем параметры запуска. Если введен -detail или -d, выводятся промежуточные данные
    if (argc == 2 && (!strcmp(argv[1], "-detail\0") || !strcmp(argv[1], "-d\0")))
        d_flag = true;
    if (d_flag) system("chcp 65001");

    Data D;
    D.init();

    auto res = AhoCorasick(D.getText(), D.getPatterns());
    for (auto r : res)
        cout << r.first << ' ' << r.second << endl;

    return 0;
}

```



## ПРИЛОЖЕНИЕ А

### КОД ПРОГРАММЫ АНО-KORASIK-JOKER

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

bool d_flag = false; //флаг вывода промежуточных данных

using namespace std;

class Data
//здесь храним считанные данные
{
private:
    std::string text;    //текст
    std::string pattern; //паттерн
    char joker;         //joke symbol
public:
    Data() = default;
    void init()
    {
        getline(std::cin, text);
        std::cin >> pattern;
        std::cin >> joker;
    }
    void printText()
    {
        std::cout << "Text = {" << text << "};" << std::endl;
        std::cout << "Pattern = {" << pattern << "};" << std::endl;
        std::cout << "Joker is '" << joker << "'" << std::endl;
    }
    std::string getText() { return text; }
    std::string getPattern() { return pattern; }
    char getJoker() { return joker; }
};

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];
    }

    if (d_flag)
    {
        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
    if (d_flag) 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);

        if (d_flag) cout << "Символ '" << c << "' найден!" << endl; // Дебаг
        return found;
    }

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

if (d_flag) cout << "Символ '" << c << "' не найден!" << endl; // Дебаг

curr = this;
return {};
}

// Функция для построения недетерминированного автомата
void makeAutomaton() {
    if (d_flag) 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(); // Обработываем верхушку очереди

        // Для дебага
        if (d_flag) {
            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) {
            if (d_flag) cout << child.second->value << ' '; // Дебаг
            queue.push(child.second);
        }
    }

    // Дебаг

```

```

        if (d_flag && !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
        if (d_flag) cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ? "Корень" : curr->suffixLink-
>dbgStr) << endl << endl;
    }

    // Дебар
    if (d_flag) {
        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); }
    vector<pair<size_t, size_t>> find(const char c) { return root.find(c); }
    void makeAutomaton() { root.makeAutomaton(); }
private:
    TreeNode root;
};

vector<size_t> 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(int argc, char** argv)
{
    //читаем параметры запуска. Если введён -detail или -d, выводятся промежуточные данные
    if (argc == 2 && (!strcmp(argv[1], "-detail\0") || !strcmp(argv[1], "-d\0")))
        d_flag = true;
    if (d_flag) system("chcp 65001");

    Data D;
    D.init();

    for (auto ans : AhoCorasick(D.getText(), D.getPattern(), D.getJoker()))
        cout << ans << endl;

    return 0;
}

```