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 _____ Бородкин Ю.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритма Ахо-Корасик для решения задач точного поиска набора образцов и поиска образца с джокером.

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 $P = \{p1, \ldots, pn\}\ \ 1 \leq\ |p_i|\ \leq 75$

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:

CCCA

1

CC

Sample Output:

1 1

2 1

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$b??$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, $1 \leq| T |\leq 100000$)

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 СД бор

Бор – структура данных для хранения набора строк в виде дерева, корнем которого является «пустой элемент». Создание такого дерева происходит следующим образом: берётся первый символ строки, для него создаём указатель от корня к новой вершине, далее выполняется переход в созданную вершину, для

которой выполняем ту же самую процедуру, только уже не относительно корня, а относительно созданной вершины и со следующим символом строки. Так, дойдя до конца строки, придём к вершине, объявляющей, что строка, вставленная в дерево – кончилась, назовём такую вершину терминальной.

Из такой СД можно построить автомат, для этого необходимо добавить ссылки на максимальные суффиксы строк.

Описание алгоритма задания 1

В программе используется алгоритм Ахо-Корасик. Он заключается в том, что для всех строк шаблонов строится автомат по бору. Далее, для каждого символа текста выполняется поиск в нём. Если для текущей вершины мы нашли потомка, то переходим в него, иначе переходим по суффиксным ссылкам, в поисках такой вершины, для которого существует переход по обрабатываемому символу. После перехода выполняется проверка на то, является ли вершина и всевозможные её суффиксы (вершины, по которым от данной можно перейти через суффиксные ссылки) – терминальными. Если да, то возвращаем все такие найденные номера паттернов. Если символа в автомате не оказалось, то текущая вершина принимает значение корня.

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

Описание алгоритма задания 2

В задании 2 шаблонами являются подстроки маски, разделенные символами джокера, обозначим множество таких подстрок как $\{Q_1, \dots, Q_n\}$. По таким подстрокам также строится автомат по бору. После этого для каждого символа текста выполняется поиск в нём. Появления подстроки Q_i в тексте на позиции j означает возможное появление маски на позиции $j - l_i + 1$, где l_i – индекс начала подстроки Q_i в маске. Далее, с помощью вспомогательного массива для таких позиций увеличиваем его значение на 1. Индексы, по которым хранятся значения равному n, 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 можно считать сложность как

$O(2m) = O(m)$. 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 $O(m)$, т.к. каждый символ представляет собой вершину бора, также на каждой позиции текста могут встретиться все k шаблонов, 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 (для задания 1);
- `vector<pair<size_t, size_t>> substringEntries` – вектор, элементом которого является пара: индекс вхождения в маску и длина подстроки (для задания 2);

Методы TreeNode:

- `TreeNode(char val)` – конструктор для заполнения поля $value$: 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 же 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 $TreeNode\ root$ 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.

Функции задания 2:

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

Тестирование

Задание 1:

Тест 1:

```
ABCASDTEAD
5
ABC
CAS
ASD
TEA
EAD
```

Вывод:

```
1 1
4 3
7 4
```

Тест 2:

```
ABCBABCBA
4
ABC
BCB
CBA
BAB
```

Вывод:

```
1 1
4 4
7 3
```

Тест 3:

```
CATNATCAT
3
```

```
AT
CAT
NA
```

Вывод:

```
1 2
4 3
7 2
```

Тест 4:

```
CCCA
1
CC
```

Вывод:

```
1 1
```

Тест с подробным промежуточным выводом:

```
ABABA
1
ABA
```

```
Inserting string: ABA
Current state of trie:
Root:
        Children: A
A:
        Parent: Root
        Children: B
AB:
        Parent: A
        Children: A
ABA:
        Parent: AB


Automaton building:
A:
        Parent: Root
        Children: B
        Suffix Link: Root

AB:
        Parent: A
        Children: A
        Suffix Link: Root

ABA:
        Parent: AB
        Suffix Link: A
```

```
Current state of trie:
Root:
        Children: A
A:
        Suffix Link: Root
        Parent: Root
        Children: B
AB:
        Suffix Link: Root
        Parent: A
        Children: A
ABA:
        Suffix Link: A
        Parent: AB


Find 'A' from: Root
Symbol 'A' found
Find 'B' from: A
Symbol 'B' found
Find 'A' from: AB
Symbol 'A' found
Find 'B' from: Root
Symbol 'B' not found
Find 'A' from: Root
Symbol 'A' found
1 1
```

Задание 2:

Тест 1:

```
ACTANCA
A$$A
$
```

Вывод:

```
1
```

Тест 2:

```
CATNATCAT
#AT
#
```

Вывод:

```
1
4
7
```

Тест с подробным промежуточным выводом:

```
ABCBABC
B$B
$
Current state of trie:
Root:
        Children: B
B:
        Parent: Root


Inserting string: B
Current state of trie:
Root:
        Children: B
B:
        Parent: Root


Automaton building:
B:
        Parent: Root
        Suffix Link: Root


Current state of trie:
Root:
        Children: B
B:
        Suffix Link: Root
        Parent: Root


Find 'A' from: Root
Symbol 'A' not found
Find 'B' from: Root
Symbol 'B' found
Find 'C' from: B
Go to suffix link: Root
Symbol 'C' not found
Find 'B' from: Root
Symbol 'B' found
Find 'A' from: B
Go to suffix link: Root
Symbol 'A' not found
Find 'B' from: Root
Symbol 'B' found
Find 'C' from: B

Go to suffix link: Root
Symbol 'C' not found
2
```

Вывод

В ходе выполнения лабораторной работы был изучен алгоритм Ахо-Корасик и использован для нахождения вхождений множества строк в тексте, а также для нахождения шаблона с джокером.

Приложение А

Исходный код задания 1

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <queue>
#include <unordered_map>
#define DEBUG
// для индивидуализации необходимо объявить макрос SKIP_INTERSECTIONS
#define SKIP_INTERSECTIONS

using namespace std;

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

#ifdef DEBUG
    void printTrie()
    {
        cout << "Current state of trie:" << endl;

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

        while (!queue.empty())
        {
            auto curr = queue.front();
            if (!curr->value)
                cout << "Root:" << endl;
            else
                cout << curr->dbgStr << ':' << endl;
            if (curr->suffixLink)
                cout << "\tSuffix Link: " << (curr->suffixLink == this ?
"Root" : curr->suffixLink->dbgStr) << endl;
            if (curr->parent && curr->parent->value)
                cout << "\tParent: " << curr->parent->dbgStr << endl;
            else if (curr->parent)
                cout << "\tParent: Root"<< endl;

            if (!curr->children.empty())
```

```
                    cout << "\tChildren: ";
                for (auto child : curr->children) {
                    cout << child.second->value << ' ';
                    queue.push(child.second);
                }
                queue.pop();
                cout << endl;
            }
            cout << endl;

        }
    #endif

        void insert(const string &str)
        {
            auto curr = this;
            static size_t countPatterns = 0;
            //пробегаемся по строке
            for (char c : str)
            {
                //если из текущей вершины по текущему символу не было создано
перехода
                if (curr->children.find(c) == curr->children.end())
                {
                    //создаём переход по символу
                    curr->children[c] = new TreeNode(c);
                    curr->children[c]->parent = curr;
    #ifdef DEBUG
                    curr->children[c]->dbgStr += curr->dbgStr + c;
    #endif
                }
                //двигаемся "вниз" по дереву
                curr = curr->children[c];
            }
    #ifdef DEBUG
            cout << "Inserting string: " << str << endl;
            printTrie();
    #endif
            // маркер терминальной вершины, значение которого равно порядковому
номеру добавления шаблона
            curr->numOfPattern = ++countPatterns;
        }
```

```
        vector <size_t> find(const char c)
        {
            // статическая переменная для хранения вершины, с которой
необходимо начать следующий вызов
            static const TreeNode* curr = this;
    #ifdef DEBUG
            cout << "Find '" << c << "' from: " << (curr->dbgStr.empty() ?
"Root" : curr->dbgStr) << endl;
    #endif

            for (; curr != nullptr; curr = curr->suffixLink) {
                // обходим потомков, если искомого символа среди потомков
найдено не будет, то переходим по суффиксной ссылке, для дальнейшего поиска
                for (auto child : curr->children)
                    // если символ потомка равен искомому
                    if (child.first == c) {
                        // значение текущей вершины переносим на этого потомка
                        curr = child.second;
                        // вектор номеров найденных терминальных вершин
                        vector<size_t> found;
    #ifdef SKIP_INTERSECTIONS
                        // для пропуска пересечений, после нахождения
терминальной вершины
                        if (curr->numOfPattern) {
                            // добавляем к найденным эту вершину
                            found.push_back(curr->numOfPattern - 1);
                            // и переходим в корень
                            curr = this;
                        }
    #else
                        // 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);
    #endif
    #ifdef DEBUG
                        cout << "Symbol '" << c << "' found" << endl;
    #endif
                        return found;
                    }
    #ifdef DEBUG
```

```
                if (curr->suffixLink)
                    cout << "Go to suffix link: " << (curr->suffixLink->dbgStr.empty() ? "Root" : curr->suffixLink->dbgStr) << endl;
    #endif
            }
    #ifdef DEBUG
            cout << "Symbol '" << c << "' not found" << endl;
    #endif
            curr = this;
            return {};
        }

        void makeAutomaton()
        {
    #ifdef DEBUG
            cout << "Automaton building: " << endl;
    #endif
            // очередь для обхода в ширину
            queue <TreeNode*> queue;
            // закидываем потомков корня
            for (auto child : children)
                queue.push(child.second);

            while (!queue.empty())
            {
                // обрабатываем верхушку очереди
                auto curr = queue.front();
    #ifdef DEBUG
                cout << curr->dbgStr << ':' << endl;
                if (curr->parent && curr->parent->value) {
                    cout << "\tParent: " << curr->parent->dbgStr << endl;
                }
                else if (curr->parent) {
                    cout << "\tParent: Root"<< endl;
                }

                if (!curr->children.empty()) {
                    cout << "\tChildren: ";
                }
    #endif
                // закидываем потомков текущей верхушки
                for (auto child : curr->children) {
    #ifdef DEBUG
```

```
                cout << child.second->value << ' ';
    #endif
                queue.push(child.second);
            }

    #ifdef DEBUG
            if (!curr->children.empty())
                cout << endl;
    #endif

            queue.pop();
            // ссылка на родителя обрабатываемой вершины
            auto p = curr->parent;
            // значение обрабатываемой вершины
            char x = curr->value;
            // если родитель не nullptr, то переходим по суффиксной ссылке
            if (p) p = p->suffixLink;

            // пока можно переходить по суфф ссылке или же пока не найдем
переход в символ обрабатываемой вершины
            while (p && p->children.find(x) == p->children.end()) {
                p = p->suffixLink;
            }

            // суффиксная ссылка для текущей вершины равна корню, если не
смогли найти переход в дереве по символу тек вершины
            // иначе найденной вершине
            curr->suffixLink = p ? p->children[x] : this;
    #ifdef DEBUG
            cout << "\tSuffix Link: " << (curr->suffixLink == this ? "Root"
: curr->suffixLink->dbgStr) << endl << endl;
    #endif
        }
    #ifdef DEBUG
        cout << endl;
        printTrie();
    #endif
    }


    private:
    #ifdef DEBUG
        string dbgStr = "";
```

```
#endif
    char value;
    size_t numOfPattern = 0;
    TreeNode *parent = nullptr;
    TreeNode *suffixLink = nullptr;
    unordered_map <char, TreeNode*> children;
};

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

    void insert(const string &str)
    {
        root.insert(str);
    }

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

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

private:
    TreeNode root;
};

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

    // закидываем паттерны в бор
    for (const auto &pattern : patterns)
        bor.insert(pattern);
    //делаем автомат из полученного дерева, путём добавления суффиксных
ссылок
    bor.makeAutomaton();
```

```
        for (size_t j = 0; j < text.size(); j++)
            for (auto pos : bor.find(text[j]))
                result.emplace(j - patterns[pos].size() + 2, pos + 1);

        return result;
    }

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

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

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

        return 0;
}
```

Исходный код задания 2

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <unordered_map>
#define DEBUG
// для индивидуализации необходимо объявить макрос SKIP_INTERSECTIONS
#define SKIP_INTERSECTIONS


using namespace std;

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

#ifdef DEBUG
    void printTrie()
    {
        cout << "Current state of trie:" << endl;

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

        while (!queue.empty())
        {
            auto curr = queue.front();
            if (!curr->value)
                cout << "Root:" << endl;
            else
                cout << curr->dbgStr << ':' << endl;
            if (curr->suffixLink)
                cout << "\tSuffix Link: " << (curr->suffixLink == this ?
"Root" : curr->suffixLink->dbgStr) << endl;
            if (curr->parent && curr->parent->value)
                cout << "\tParent: " << curr->parent->dbgStr << endl;
            else if (curr->parent)
                cout << "\tParent: Root"<< endl;

            if (!curr->children.empty())
                cout << "\tChildren: ";
            for (auto child : curr->children) {
```

```
                    cout << child.second->value << ' ';
                    queue.push(child.second);
                }
                queue.pop();
                cout << endl;
            }
            cout << endl;

        }
    #endif
        void insert(const string &str, size_t pos, size_t size)
        {
            auto curr = this;
            for (char c : str)
            {
                //если из текущей вершины по текущему символу не было создано
перехода
                if (curr->children.find(c) == curr->children.end())
                {
                    //создаём переход по символу
                    curr->children[c] = new TreeNode(c);
                    curr->children[c]->parent = curr;
    #ifdef DEBUG
                    curr->children[c]->dbgStr += curr->dbgStr + c;
    #endif
                }
                //двигаемся "вниз" по дереву
                curr = curr->children[c];
            }
    #ifdef DEBUG
            cout << "Inserting string: " << str << endl;
            printTrie();
    #endif

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

        vector <pair<size_t, size_t>> find(const char c)
        {
            // статическая переменная для хранения вершины, с которой
необходимо начать следующий вызов
            static const TreeNode* curr = this;
    #ifdef DEBUG
```

```
            cout << "Find '" << c << "' from: " << (curr->dbgStr.empty() ?
"Root" : curr->dbgStr) << endl;
    #endif

            for (; curr != nullptr; curr = curr->suffixLink) {
                // обходим потомков, если искомого символа среди потомков
найдено не будет, то переходим по суффиксной ссылке, для дальнейшего поиска
                for (auto child : curr->children)
                    // если символ потомка равен искомому
                    if (child.first == c) {
                        // значение текущей вершины переносим на этого потомка
                        curr = child.second;
                        // 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);
    #ifdef DEBUG
                        cout << "Symbol '" << c << "' found" << endl;
    #endif
                        return found;
                    }

    #ifdef DEBUG
                if (curr->suffixLink)
                    cout << "Go to suffix link: " << (curr->suffixLink-
>dbgStr.empty() ? "Root" : curr->suffixLink->dbgStr) << endl;
    #endif
            }
    #ifdef DEBUG
            cout << "Symbol '" << c << "' not found" << endl;
    #endif
            curr = this;
            return {};
        }

        void makeAutomaton()
        {
    #ifdef DEBUG
```

```
        cout << "Automaton building: " << endl;
#endif
        // очередь для обхода в ширину
        queue <TreeNode*> queue;
        // закидываем потомков корня
        for (auto child : children)
            queue.push(child.second);

        while (!queue.empty())
        {
            // обрабатываем верхушку очереди
            auto curr = queue.front();
#ifdef DEBUG
            cout << curr->dbgStr << ':' << endl;
            if (curr->parent && curr->parent->value) {
                cout << "\tParent: " << curr->parent->dbgStr << endl;
            }
            else if (curr->parent) {
                cout << "\tParent: Root"<< endl;
            }

            if (!curr->children.empty()) {
                cout << "\tChildren: ";
            }
#endif
            // закидываем потомков текущей верхушки
            for (auto child : curr->children) {
#ifdef DEBUG
                cout << child.second->value << ' ';
#endif
                queue.push(child.second);
            }

#ifdef DEBUG
            if (!curr->children.empty())
                cout << endl;
#endif

            queue.pop();
            // ссылка на родителя обрабатываемой вершины
            auto p = curr->parent;
            // значение обрабатываемой вершины
            char x = curr->value;
```

```
                // если родитель не nullptr, то переходим по суффиксной ссылке
                if (p) p = p->suffixLink;

                // пока можно переходить по суфф ссылке или же пока не найдем переход в символ обрабатываемой вершины
                while (p && p->children.find(x) == p->children.end()) {
                    p = p->suffixLink;
                }

                // суффиксная ссылка для текущей вершины равна корню, если не смогли найти переход в дереве по символу тек вершины
                // иначе найденной вершине
                curr->suffixLink = p ? p->children[x] : this;
    #ifdef DEBUG
                cout << "\tSuffix Link: " << (curr->suffixLink == this ? "Root" : curr->suffixLink->dbgStr) << endl << endl;
    #endif
            }
    #ifdef DEBUG
            cout << endl;
            printTrie();
    #endif
        }

    private:
    #ifdef DEBUG
        string dbgStr = "";
    #endif
        char value;
        TreeNode *parent = nullptr;
        TreeNode *suffixLink = nullptr;
        vector <pair<size_t, size_t>> substringEntries;
        unordered_map <char, TreeNode*> children;
    };

    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;
    // массив для хранения кол-ва попаданий безмасочных подстрок в текст
    vector <size_t> midArr(text.size());
    string pattern;
    // кол-во безмасочных подстрок
    size_t numSubstrs = 0;

    // закидываем в бор все безмасочные подстроки маски
    for (size_t i = 0; i <= mask.size(); i++)
    {
        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())
                    // и увеличиваем её значение на 1
                    midArr[i]++;
            }

        for (size_t i = 0; i < midArr.size(); i++)
            // индекс, по которым промежуточный массив хранит кол-во
            // 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);
    #ifdef SKIP_INTERSECTIONS
                // для пропуска пересечений, после найденного индекса,
                // увеличиваем его на длину маски
                i += mask.size() - 1;
    #endif
            }

        return result;
    }

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

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

        return 0;
    }
```