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 _____ Кузьмин Д. И.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритм Ахо-Корасик. Освоить навыки разработки программ, реализующих этот алгоритм.

Задание.

1. Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p1,\ldots,pn\}1 \le |pi| \le 75$

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
```

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?a*b*??c? с джокером ?? встречается дважды в тексте xabvccbababcax*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 ($1 \le |T| \le 100000$)
Шаблон ($1 \le |P| \le 40$)
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
```

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).

Основные теоретические положения.

Алгоритм Ахо — Корасик — алгоритм поиска подстроки, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году, реализует поиск множества подстрок из словаря в данной строке.

Описание функций и структур данных.

1) В качестве строк использовался класс std::string
2) Для описания вершин бора и автомата использовался класс Vertex, обладающий полями:

   std::map<char, Vertex*> neighbours - соседи вершины

   Vertex* suffixLink - суффиксная ссылка

   Vertex* cameFrom - родитель вершины

   std::vector<int> reachable - вектор доступных строк

   std::string str - строка, соответствующая вершине

   int end = 0 - номер образца, который соответствует данной вершине

3) В качестве класса, реализующего бор, использовался класс Graph, обладающий полями:

   std::vector<Vertex*> vertexVector - вектор указателей на вершины

   Vertex* root - указатель на корень бора(нулевую вершину)

4) Функции данного класса:

   void buildGraph(std::vector<std::string> templates) - построение бора. templates - набор образцов

   void buildSuffixLinks() - построение суффиксных ссылок

   void findReachableThroughSuffixLinks(std::vector<std::string> patterns) - нахождение образцов, доступных из вершин бора по суффиксным ссылкам, patterns - набор образцов

5) В качестве класса, реализующего автомат использовался класс StateMachine, имеющий поля:

   Vertex* currentState - текущее состояние автомата

   std::vector<char> alphabet - алфавит автомата

std::map<std::pair<Vertex*, char>, Vertex*> statesMap - таблица состояний

6) Функции данного класса:

void buildStateMachine(Graph trie) - построение автомата. trie - бор, в соответствии с которым, строится автомат.

7) Поиск совпадений реализован при помощи функции std::vector<std::pair<int, int>> findMatches(std::string text, std::vector<std::string> patterns, bool skip = false) - поиск образцов в тексте. text - текст, в котором ищутся образцы; patterns - набор образцов, skip - параметр для пропуска пересекающихся строк.

8) Поиск образца с джокером реализован при помощи функции void findWildCard(std::string text, std::string pattern, char sign) - поиск образца с джокером в тексе. pattern - искомый образец, text - текст, в котором ищется образец; sign - символ, являющийся джокером.

Описание алгоритма (Ахо-Корасик)

1) Строится префиксное дерево(бор) в соответствии с набором образцов.
2) Каждому ребру соответствует символ. Каждой вершине помимо начальной соответствует префикс какой-то строки из набора образцов.
3) Построение начинается с создания начальной вершине - корня.
4) Затем обрабатывается символы определенного образца. Если из текущей вершины можно продолжить путь (есть сосед, соединенный соответствующим ребром), то производится переход к этому соседу. Если нельзя - создается новая вершина. Если текущий символ последний в образце, вновь созданная или полученная в результате перехода вершина помечается терминальной.
5) Действия из п. 4 повторяются для каждого образца.
6) Затем создаются суффиксные ссылки для каждой из вершин.

7) Суффиксная ссылка - максимальный суффикс вершины (строки), который также является префиксом какой-то еще строки в боре. Суффиксная ссылка так же является какой-то вершиной бора

8) Вычислить суффиксную ссылку для вершины можно следующим образом:

1. Рассмотреть суффиксную ссылку s_1 для родителя данной вершины;

2. Если среди соседей s_1 есть сосед с символом, таким же, как символ изначальной вершины, то суффиксной ссылкой изначально вершины будет являться этот сосед. Иначе, повторить п.2 для суффиксной ссылки от s_1.

3. Если на каком-то этапе среди соседей не онаружилось подходящего символа и при этом текущая вершина - корень, то тогда суффиксной ссылке изначальной вершины присваивается корень.

"Символ вершины" понимается, как символ на ребре, соединяющий вершину и ее родителя.

9) Создание суффиксных ссылок проводится при помощи обхода графа в ширину.

10) Далее при помощи обхода в ширину каждой вершине сообщается список образцов, доступных из нее по суффиксным ссылкам.

11) Затем строится автомат.

12) Состоянием автомата является какая-то из вершин бора. Начальное состояние - корень бора. Автомат принимает на вход символ из алфавита и производит переход к другому состоянию (вершине) в соответствии с этим символом.

13) Если из данного состояния(вершины) есть прямой путь к соседу, содержащего символ, полученный на вход, то производится переход к этому соседу (в состояние, соответствующее вершине-соседу). Если нет, аналогичная проверка следует уже для суффиксной ссылки текущей вершины. Если на каком-то этапе не удалось найти путь и при этом текущая вершине - корень. То переход осуществляется к корню(начальному состоянию).

14) На каждом состоянии производится проверка, является ли вершина терминальной, а также списка доступных образцов из вершины по суффиксным ссылкам. Эти проверки дадут информацию о вхождении одного или нескольких образцов из заданного набора.

Учитывая то, что таблица состояний автомата хранится в std::map, где доступ к элементу производится за $O(log(n))$, сложность по времени можно оценить $O(nlog(m\sigma) + m\sigma + z)$ по памяти – $O(m \cdot \sigma)$, где n - длина текста, m - сумма длин образцов, z - общая длина совпадений, σ- размер алфавита.

Описание алгоритма (поиск образца с джокером)

1) Дан текст и образец, содержащий какое-то количество символов "джокер" - символ, который может принимать любое значение. Необходимо найти вхождения этого образца в текст.

2) Создается вектор с длиной равной длине текста. Все элементы инициализируются нулями.

3) В образце ищутся подстроки, не содержащие джокер и фиксируются их индексы вхождения в образец.

4) Далее при помощи алгоритма Ахо-Корасик эти подстроки ищутся в исходном тексте.

5) На каждое вхождение таких подстрок значение элемента ранее созданного вектора с индексом равным разности индексов вхождения подстроки в текст и в образец, увеличивается на 1.

6) Если значение элемента вектора равно количеству подстрок (найденных на 2 шаге), то индекс этого элемента - есть вхождение исходного образца в текст.

Учитывая то, что таблица состояний автомата хранится в std::map, сложность по времени можно оценить $O(nlog(m\sigma) + m\sigma + z)$, по памяти –

$O(m\sigma)$, где n -длина текста, m - сумма длин подстрок образца без джокера, σ - размер алфавита, z - общая длина совпадений

Описание алгоритма (поиск непересекающихся подстрок)

1) Алгоритм работает аналогично алгоритму Ахо-Корасик.

2) Однако при нахождение очередного образца состоянию автомата присваивается начальное (корень бора).

Учитывая то, что таблица состояний автомата хранится в std::map, сложность по времени можно оценить $O(nlog(m\sigma) + m\sigma)$, по памяти – $O(\sigma \cdot m)$, где n - длина текста, m - сумма длин образцов, σ- длина алфавита.

Исходный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1., табл. 2 и табл. 3.

Таблица 1 — результаты тестирования алгоритма поиска образцов.

№ п/п	Входные данные	Выходные данные	Комментарий
1	CCNATCCNA 2 NATCC NA	3 1 3 2 8 2	CCNATCCNA CCNATCCNA CCNATCCNA
2	NTAG 3 TAGT TAG T	2 2 2 3	NTAG NTAG
3	AAA 3 A AA AAA	1 1 1 2 1 3 2 1 2 2	AAA AAA AAA AAA AAA

		3 1	AAA
4	ACGTAG 3 CGT A AG	1 2 2 1 5 2 5 3	ACGTAG ACGTAG ACGTAG ACGTAG
5	AAAA -3 A	Некорректный ввод	Отрицательное число образцов

Таблица 2 — результаты тестирования алгоритма поиска образца с джокером

№ п/п	Входные данные	Выходные данные	Комментарий
1	ACTANCA A$$A$ $	1	ACTANCA
2	ACTANCAGG A$ $	1 4 7	ACTANCAGG ACTANCAGG ACTANCAGG
3	ATCATCATCATC ??A?? ?	2 5 8	ATCATCATCATC ATCATCATCATC ATCATCATCATC
4	ATCATC $$$ $	Некорректный ввод	Образец должен содержать хотя бы 1 символ не джокер
5	ATCATC ATC A	Некорректный ввод	Недопустимый символ для джокера.

Таблица 3 — результаты тестирования алгоритма поиска непересекающихся подстрок

№ п/п	Входные данные	Выходные данные	Комментарий
1	CCNATCCNA 2 NATCC NA	3 2 8 2	CCNATCCNA CCNATCCNA
2	AAA 3 A AA AAA	1 1 2 1 3 1	AAA AAA AAA
3	ATCATC 2 ATC TCA	1 1 4 1	ATCATC ATCATC
4	ACGTAG 3 CGT A AG	1 2 2 1 5 2	ACGTAG ACGTAG ACGTAG
5	AAAA -3 A	Некорректный ввод	Отрицательное число образцов.

Выводы.

Был изучен принцип алгоритма Ахо-Корасик. Получены навыки разработки программ, реализующих этот алгоритм.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <string>
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <map>
#include <functional>
#define NEWLINE std::cout<<"\n";
#define INFO
class Vertex {
public:
    std::map<char, Vertex*> neighbours;
    int name;
    Vertex(int  name) :name(name) {
        cameFrom = nullptr;
        suffixLink = nullptr;
    }
    void addNeighbour(char sym, Vertex* v) {
        neighbours[sym] = v;
    }
    bool operator<(Vertex v1) {
        return this->name < v1.name;
    }
    friend std::ostream& operator<<(std::ostream& out, Vertex* v) {
        out << v->name << "(" << v->str << ")";
        return out;
    }
    ~Vertex() {
        delete suffixLink;
        delete cameFrom;
    }
    Vertex* suffixLink;
    Vertex* cameFrom;
    std::vector<int> reachable;
    std::string str;
    char symbol;
    int end = 0; //для обозначениея того, является ли вершина концом
строки

};
template <typename A, typename B>
A findByValue(std::map<A, B> m, B value) {
    for (const auto& it : m)
        if (value == it.second)
```

```
            return it.first;
    return 0;
}
template <typename T>
bool isInVector(T vec, Vertex v) {
    for (auto it : vec)
        if (it->name == v.name)
            return true;
    return false;
}
template <typename T>
void printQueue(T q) {
    while (!q.empty()) {
        std::cout << q.front()->name << " ";
        q.pop();
    }
    NEWLINE
}

//класс для бора
class Graph {
public:
    std::vector<Vertex*> vertexVector;
    Vertex* root;

    Vertex* operator()(char verName) {
        for (auto& it : vertexVector) {
            if (it->name == verName)
                return it;
        }
        return nullptr;
    }


    std::string getStringByVertex(Vertex* v) {
        std::string str;
        Vertex* cf = v->cameFrom;

    }
    //построение бора
    void buildGraph(std::vector<std::string> templates) {
#ifdef INFO
        std::cout << "Построение бора\n";
#endif
        vertexVector.push_back(new Vertex(0));
        int vername = 0;
        int template_num = 1;
```

```
        //просмотр каждого образца
        for (const auto& str : templates) {

            Vertex* curr = vertexVector[0];
#ifdef INFO
            std::cout << "\nОбразец: " << str << "\n";
#endif
            //просмотр каждого символа для данного образца
            for (const auto& sym : str) {

#ifdef INFO
                std::cout << "\nСимвол: " << sym << std::endl;
                std::cout << "Текущая вершина: " << curr << std::endl;


                std::cout << "Соседи текущей вершины: ";
                for (auto& nei : curr->neighbours)
                    std::cout << "(" << nei.first << " " << nei.second-
>name << ") ";
                NEWLINE
#endif
                auto search = curr->neighbours.find(sym);

                //если нельзя перейти по существующей вершине
                if (search == std::end(curr->neighbours)) {
                    curr->neighbours[sym] = new Vertex(++vername);
                    Vertex* cameFrom = curr;
                    curr = curr->neighbours[sym];
                    vertexVector.push_back(curr);
                    curr->cameFrom = cameFrom;
                    curr->str += curr->cameFrom->str + sym;
#ifdef INFO
                    std::cout << "Создание новой вершины: " << curr->name
<< "\n";
#endif
                }
                //если можно перейти по существующей вершине
                else {
                    Vertex* cameFrom = curr;
                    curr = search->second;
                    curr->cameFrom = cameFrom;
#ifdef INFO
                    std::cout << "Переход по существующей вершине: " <<
curr->name << "\n";
#endif
                }
```

```
                curr->symbol = sym;
            }
            curr->end = template_num++;
            curr->reachable.push_back(curr->end);

#ifdef INFO
            std::cout << "Вершина " << curr << " назначена
терминальной\n";
#endif
        }
    }
    void buildSuffixLinks() {
#ifdef INFO
        std::cout << "\nПостроение суффиксных ссылок\n";
#endif
        std::queue< Vertex*> q; //очередь для поиска в ширину
        Vertex* curr = vertexVector[0];
        Vertex* old;
        root = curr;
        curr->suffixLink = curr;
        q.push(curr);
        while (!q.empty()) {
#ifdef INFO
            std::cout << "\nСостояние очереди: ";
            printQueue(q);
#endif
            curr = q.front();
            q.pop();
#ifdef INFO
            std::cout << "Текущая вершина - " << curr << "\nСоседи
текущей вершины ";
#endif
            //добавление соседей в очередь
            for (const auto& it : curr->neighbours) {
#ifdef INFO
                std::cout << it.second << " ";
#endif
                q.push(it.second);
            }
#ifdef INFO
            std::cout << " добавлены в очередь\n";
#endif

#ifdef INFO
            std::cout << "Построение суффиксной ссылки для " << curr <<
"\n";
```

```
#endif
           //изначально в качестве суфф. ссылки присваивется корень бора
            curr->suffixLink = root;
            root->cameFrom = root;
            old = curr;
            char sym;
            if (curr->name != 0) {
                sym = *(curr->str.end() - 1);
            }
            //функция поиска суффиксной ссылки
            std::function<Vertex* (Vertex*)> findSuffixLink = [&](Vertex*
v){
                Vertex* tmpSuffixLink = (v->suffixLink->name == 0) ?
tmpSuffixLink = v->cameFrom->suffixLink : tmpSuffixLink = v->suffixLink;

#ifdef INFO
                std::cout << "Предыдущая вершина - " << v->cameFrom <<
"\n";
                std::cout << "Проверка соседей ее суффиксной ссылки " <<
tmpSuffixLink << ": ";
                for (const auto& nei : tmpSuffixLink->neighbours)
                    std::cout << "(" << nei.first << " " << nei.second-
>name << ") ";
                NEWLINE

#endif
                //поиск подходящего соседа
                auto result = tmpSuffixLink->neighbours.find(sym);
                if (result != std::end(tmpSuffixLink->neighbours) &&
result->second->str.size() < old->str.size())
                    return result->second;

                //рекурсивный вызов функции поиска суффиксной ссылки,
если это возможно
                return (v->name != 0 && tmpSuffixLink->name != 0) ?
findSuffixLink(tmpSuffixLink) : root;
            };

            old->suffixLink = (old->name != 0) ? findSuffixLink(old) :
old;
#ifdef INFO
            std::cout << "Суффиксная ссылка для вершины " << old << " - "
<< old->suffixLink << "\n";
#endif
        }
    }
```

```
    void findReachableThroughSuffixLinks(std::vector<std::string>
patterns) {
#ifdef INFO
        std::cout << "\n\nПоиск строк, достижимых из вершин по суффиксным
ссылкам\n";
#endif

        std::queue<Vertex*> q; //очередь для обхода в ширину
        Vertex* current;
        q.push(root);
        while (!q.empty()) {
            current = q.front();
            q.pop();
            //добавление соседей в очередь
            for (auto& it : current->neighbours){
                q.push(it.second);
            }
            Vertex* tmpSuff = current->suffixLink;

            bool a = tmpSuff->end != 0; //если вершина терминальная
            bool b = !tmpSuff->reachable.empty();//если имеются
достижимые вершины по суффиксным ссылкам

            if (a || b) {
                if (a) {
                    //добавление новой вершины в вектор доступных
                    current->reachable.push_back(tmpSuff->end);
                }
                if (b) {

                    //объединение векторов доступных вершин
                    std::vector<int> v1 = current->reachable;
                    std::vector<int> v2 = tmpSuff->reachable;
                    std::vector<int> dest1;

                    std::sort(v1.begin(), v1.end());
                    std::sort(v2.begin(), v2.end());
                    std::set_union(v1.begin(), v1.end(),
                        v2.begin(), v2.end(),
                        std::back_inserter(dest1));
                    current->reachable = dest1;
                }
            }
        }
#ifdef INFO
        for (const auto& it : vertexVector) {
            for (const auto& it2 : it->reachable)
```

```
                if (it2 != it->end)
                    std::cout << "Образец " << patterns[it2 - 1] << "
достижим из " << it << " по суффиксным ссылкам\n";
        }
#endif
    }

};
template <class T>
void printVector(std::vector<T> vec) {
    for (auto it : vec) {
        std::cout << it << " ";
    }
    std::cout << "\n";
}

//класс, реализующий автомат
class StateMachine {
public:
    Vertex* currentState;
    std::string alphabet = "ACGTN";
    std::map<std::pair<Vertex*, char>, Vertex*> statesMap;
    void buildStateMachine(Graph trie) {
#ifdef INFO
        std::cout << "\nПостроение автомата\n";
#endif
        for (const auto& it : trie.vertexVector) {

#ifdef INFO
            std::cout << "\nВершина " << it->name << "\n";
            std::cout << "Возможные переходы по конечным ссылкам:\n";
            for (const auto& n : it->neighbours) {
                std::cout << "Через " << n.first << " к " << n.second-
>name << "\n";
            }
            if (it->name != 0) {
                std::cout << "Возможные переходы по суффиксным
ссылкам:\n";
                std::function<void(Vertex*)> printSuffixLink =
[&](Vertex* suff) {

                    std::cout << "Суффиксная ссылка " << suff->name <<
":\n";
                    for (const auto& n : suff->neighbours) {
                        std::cout << "Через " << n.first << " к " <<
n.second->name << "\n";
                    }
```

```
                    if (suff->name != 0)
                        printSuffixLink(suff->suffixLink);
                };
                printSuffixLink(it->suffixLink);
            }

#endif
            //вычисляем переход для каждого символа
            for (const auto& symbol : alphabet) {
                statesMap[std::make_pair(it, symbol)] = outState(it,
symbol);
#ifdef INFO
                int s = statesMap[std::make_pair(it, symbol)]->name;
                std::cout << "Символ " << symbol << ". ";
                if (s != 0)
                    std::cout << "Найден переход к вершине " <<
statesMap[std::make_pair(it, symbol)]->name << "\n";
                else
                    std::cout << "Переход не найден. Установлен 0\n";
#endif
            }
            NEWLINE
        }
#ifdef INFO
        std::cout << "Построенный автомат:\n";

        for (const auto& it : trie.vertexVector) {
            std::cout << it->name << ":[";
            for (const auto& it2 : alphabet) {
                std::cout << it2 << "->" << statesMap[std::make_pair(it,
it2)]->name;
                if (it2 != 'N')
                    std::cout << ", ";
            }

            std::cout << "]\nСтрока, соответствующая вершине - " << it-
>str << "\n";
        }
        NEWLINE
#endif
    }
    StateMachine(Vertex* state) :currentState(state) {}

    //функция перехода из состояния в состояние
    Vertex* outState(Vertex* state, char next) {
        for (auto& it : state->neighbours) {
            if (it.first == next) {
```

```
                return it.second;
            }
        }
        if (state->name == 0)
            return state;

        return outState(state->suffixLink, next);
    }
};

std::vector<std::pair<int, int>> findMatches(std::string text,
std::vector<std::string> patterns, bool skip = false) {
    Graph trie;
    std::vector<std::pair<int, int>> indexesAndPatterns;
#ifdef INFO
    std::cout << "Поиск образцов:\n";
    for (const auto& it : patterns)
        std::cout << it << "\n";
    std::cout << "В тексте: ";
    std::cout << text << "\n\n";
#endif
    trie.buildGraph(patterns);
    trie.buildSuffixLinks();
    trie.findReachableThroughSuffixLinks(patterns);
    StateMachine a(trie.root);
    a.buildStateMachine(trie);
#ifdef INFO
    std::cout << "Обработка текста\n\n";
#endif
    for (int i = 0; i < text.size(); i++) {

#ifdef INFO
        std::cout << "Текущее состояние " << a.currentState << "\n";
        std::cout << "Символ на вход " << text[i] << "\n";
#endif
        //обрабатываем очередной символ
        a.currentState = a.statesMap[std::make_pair(a.currentState,
text[i])];

#ifdef INFO
        std::cout << "Переход к состоянию " << a.currentState << "\n";
#endif
        //если встретился конец строки
        if (a.currentState->end > 0) {
#ifdef INFO
```

```
            std::cout << "Найден образец " << a.currentState->end << "("
<< a.currentState->str << ") на позиции " << i - a.currentState-
>str.size() + 2 << "\n";
#endif
            auto tmp_pairss = std::make_pair(i - a.currentState-
>str.size() + 2, a.currentState->end);
            indexesAndPatterns.push_back(tmp_pairss);
            if (skip){
#ifdef INFO
                std::cout << "Переход в начальное состояние\n\n";
#endif
                a.currentState = trie.root;
                continue;
            }
            NEWLINE
        }

        //проверка строк, достижимых по суффиксным ссылкам
        for (const auto& en : a.currentState->reachable){

            if (en != a.currentState->end) {
#ifdef INFO
                std::cout << "Найден образец " << en << "(" <<
patterns[en - 1] << ") на позиции " << i - patterns[en - 1].size() + 2 <<
" через суффиксные ссылки\n";
#endif

                auto tmp_pair = std::make_pair(i - patterns[en -
1].size() + 2, en);
                indexesAndPatterns.push_back(tmp_pair);
                if (skip) {

#ifdef INFO
                    std::cout << "Переход в начальное состояние\n\n";
#endif
                    a.currentState = trie.root;
                    break;
                }
            }
        }

        //-----------------------
#ifdef INFO
        std::cout << "\n";
#endif

    }
```

```
    //сортировка получившегося вектора пар индексов
    auto cmp = [&](const std::pair<int, int>& a, const std::pair<int,
int>& b) {
        return (b.first != a.first) ? b.first > a.first: b.second >
a.second;
    };
    std::sort(indexesAndPatterns.begin(), indexesAndPatterns.end(), cmp);
#ifdef INFO
    std::cout << "Завершение алгоритма\n";
#endif
    return indexesAndPatterns;
}
void findWildCard(std::string text, std::string pattern, char sign) {
#ifdef INFO
    std::cout << "Поиск образца с джокером:\n";
    std::cout << pattern << "\n";
    std::cout << "В тексте: ";
    std::cout << text << "\n\n";
    std::cout << "Создание вектора, заполненного нулями длины " <<
text.size() << "\n";
    std::cout << "Поиск подстрок, не содержащих джокер\n";
#endif
    using namespace std;
    vector<string> noWildCardStrings;
    vector<int> indexes;
    vector<int> indexVector(text.size(), 0);
    string curr;
    int sz = indexVector.size();
    pattern += sign;
    int size = pattern.size();

    //поиск подстрок в образце, не содержащих джокер
    for (int i = 0; i < size; i++) {
        if (pattern[i] == sign) {
            if (!curr.empty()) {
                noWildCardStrings.push_back(curr);
                indexes.push_back(i - curr.size() + 1);
            }
            curr.clear();
            continue;
        }
        curr += pattern[i];
    }

#ifdef INFO
    std::cout << "Полученный вектор подстрок и их индексов вхождений в
образец:\n";
```

```
    for (int i = 0; i < noWildCardStrings.size(); i++) {
         std::cout << noWildCardStrings[i] << " " << indexes[i] << "\n";
     }
    std::cout << "Поиск данных подстрок в тексте с помощью алгоритма Ахо-
Корасик\n\n";
#endif
    //поиск подстрок в тексте алгоритмом Ахо-Корасик
    std::vector<std::pair<int, int>> pVec = findMatches(text,
noWildCardStrings);
#ifdef INFO
    std::cout << "Найденные вхождения:\n";
    for (const auto& it : pVec) {
        std::cout << it.first <<" "<< it.second << "\n";
    }
#endif

    //измененеи соответсвующих элементов вектора индексов
    for (const auto& it : pVec){

        int index = it.first - indexes[it.second - 1];
        if (index >= 0 && index < sz) {
#ifdef INFO
            std::cout << "Увеличение значение элемента нулевого вектора с
индексом " << index + 1 << "\n";
#endif
            indexVector[index]++;
        }
    }

    int nWSize = noWildCardStrings.size();
    int c = 1;

    //просмотр элементов вектора индексов
    for (const auto& it : indexVector) {
        if (it == nWSize && pattern.size() <= text.size() - c + 2) {
#ifdef INFO
            std::cout << "Образец найден на позиции ";
#endif
            std::cout << c << "\n";
        }
        c++;
    }
#ifdef INFO
    std::cout << "Завершение алгоритма\n";
#endif
}
//функция для ввода текста и образцов и реализации алгоритма Ахо-Корасик
```

```
void interfaceAhoCorasick() {

    std::vector<std::string> templates;
    std::string text;
    std::cin >> text;
    int n;
    std::cin >> n;
    if (n < 0) {
        std::cout << "Некорректный ввод\n";
        return;
    }
    for (int i = 0; i < n; i++) {
        std::string t;
        std::cin >> t;
        templates.push_back(t);
    }
    auto pV = findMatches(text, templates);
    for (auto it : pV)
        std::cout << it.first << " " << it.second << "\n";

}

//функция для ввода текста и образца и реализации алгоритма поиска с
джокером
void interfaceWildCard() {
    std::string text;
    std::cin >> text;
    std::string pattern;
    std::cin >> pattern;
    std::string alphabet = "ACGTN";
    bool pass = false;
    char wildCard;
    std::cin >> wildCard;
    for (const auto& it : pattern)
        if (it != wildCard)
            pass = true;

    for (const auto& it : alphabet)
        if (it == wildCard)
            pass = false;

    if (!pass){
        std::cout << "Некорректный ввод\n";
        return;
    }
    findWildCard(text, pattern, wildCard);
}
```

```
int main() {

    setlocale(LC_ALL, "rus");

    //interfaceWildCard();
    interfaceAhoCorasick();
    return 0;
}
```