**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: Ахо-Корасик. Вариант 2 Подсчитать количество вершин в автомате; вывести список найденных

образцов, имеющих пересечения с другими найденными образцами в строке поиска.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Синицкая Д.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

## Цель работы.

Целью лабораторной работы является изучение алгоритма Ахо-Корасика для эффективного поиска множественных образцов в тексте.

## Задание.

5.1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,1≤∣T∣≤100000 ).

Вторая - число n (1≤n≤3000), каждая следующая из n строк содержит шаблон из набора P = {p1, … , pn} 1≤∣pi∣≤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

5.2

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 Р в текст Т.

Например, образец аb??с? 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≤∣T∣≤100000 )

Шаблон (P,1≤∣P∣≤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

## Выполнение работы.

5.1

Основные шаги алгоритма:

1. Построение Trie (бора): каждый шаблон добавляется в префиксное дерево, где каждая вершина соответствует символу. На концах шаблонов создаются выходные метки.

2. Построение суффиксных и терминальных ссылок: для каждой вершины создается суффиксная ссылка, указывающая на наиболее длинный собственный суффикс, совпадающий с другим путем в дереве. Терминальные ссылки указывают на ближайший узел, содержащий шаблон.

3. Поиск по тексту: текст сканируется символ за символом с использованием автомата. При отсутствии перехода выполняется переход по суффиксной ссылке. При достижении узла с выходной меткой (или по терминальной ссылке) фиксируются найденные шаблоны.

4. Обработка найденных вхождений: сохраняются позиции найденных шаблонов, проверяется наличие пересечений между шаблонами в тексте.

Описание функций и методов:

1. *int charToIndex(char c)* — функция преобразует символ последовательности (A, C, G, T, N) в числовой индекс от 0 до 4.

Параметры:

*char c —* входной символ.

Возвращает:

Целое число от 0 до 4 — индекс, соответствующий символу.

2. *void printTrie(TrieNode\* node, const vector<string>& patterns, const string& prefix = "", const string& indent = "", bool isLast = true)* — мтеод рекурсивно визуализирует структуру построенного Trie (автомата Ахо-Корасика).

Параметры:

*TrieNode\* node* — текущий узел Trie.

*const vector<string>& patterns* — список всех шаблонов.

*const string& prefix* — текущий префикс (путь к узлу).

*const string& indent* — отступ для форматирования дерева.

*bool isLast* — флаг, указывает, является ли узел последним в списке дочерних.

3. *void buildTrie(const vector<string>& patterns, TrieNode\* root)* — метод создаёт Trie по входным шаблонам и настраивает суффиксные и терминальные ссылки для автомата Ахо-Корасика.

Параметры:

*const vector<string>& patterns* — список шаблонов.

*TrieNode\* root* — корневой узел Trie.

4. *void searchPatterns(const string& text, TrieNode\* root, vector<string>& patterns)* — метод ищет вхождения всех шаблонов в заданном тексте, используя автомат Ахо-Корасика.

Параметры:

*const string& text —* строка для поиска.

*TrieNode\* root* — корневой узел Trie.

*vector<string>& patterns* — список шаблонов.

5. *int main()* — точка входа в программу. Считывает строку и шаблоны. Инициализирует корень Trie. Вызывает buildTrie() и searchPatterns(). Выводит количество узлов в построенном Trie.

Структуры:

1. *struct TrieNode* — описывает узел в Trie-дереве автомата Ахо-Корасика.

Поля:

*map<char, TrieNode\*> children* — отображение символов в дочерние узлы.

*TrieNode\* fail* — суффиксная ссылка на другой узел.

*TrieNode\* terminalLink* — терминальная ссылка на ближайший терминальный узел.

*vector<int> output* — список индексов шаблонов, заканчивающихся в узле.

Конструктор:

*TrieNode() —* инициализирует fail и terminalLink как nullptr.

Глобальные переменные:

*const int ALPHABET\_SIZE* = 5 — размер алфавита (A, C, G, T, N).

*const char CHAR\_TO\_INDEX[]* = {'A', 'C', 'G', 'T', 'N'} — массив символов алфавита.

*int nodeCount = 0* — счётчик количества созданных узлов.

Оценка сложности алгоритма:

Временная сложность:

1. Построение Trie: O(∑|P|), где |P| — суммарная длина всех шаблонов.

2. Построение суффиксных ссылок: O(∑|P|) — каждый узел обрабатывается один раз.

3. Поиск в тексте: O(n + k), где n — длина текста, k — общее количество найденных вхождений.

Итоговая сложность: O(∑∣P∣+n+k)

Сложность по памяти:

1. Trie: O(∑|P|) — каждый уникальный префикс создает новую вершину.

2. Дополнительная память: для хранения ссылок, очереди, выходных списков — также O(∑|P| + m), где m — количество различных символов в алфавите.

Итоговая сложность: O(∑∣P∣+n+k)

5.2

Основные шаги алгоритма:

1. Разбиение шаблона на подстроки между символами-джокерами.

2. Построение Trie (боров) на основе полученных подстрок (частей шаблона).

3. Построение суффиксных и терминальных ссылок в автомате Ахо-Корасика для эффективного перехода при поиске.

4. Поиск вхождений шаблонов в тексте с помощью автомата: Переход по Trie. Использование выходных списков и терминальных ссылок для фиксации вхождений.

5. Фиксация и анализ пересечений найденных шаблонов, выявление пересекающихся образцов.

Описание функций и методов:

1. *void buildTrie(const vector<string>& pieces, TrieNode root)\** — метод строит Trie (бор) и автомат Ахо-Корасика по частям шаблона (вырезки между джокерами).

Аргументы:

*pieces* — куски шаблона между символами-джокерами.

*Root* — указатель на корневой узел Trie.

2. *void printAutomaton(TrieNode root)\** — метод выводит в консоль визуальное представление построенного автомата для каждой вершины показывает: суффиксную ссылку, терминальную ссылку, переходы по символам, индексы шаблонов, которые заканчиваются в этой вершине

Аргументы:

*root —* Корень Trie.

3. *void search(const string& text, TrieNode root, const vector<pair<int, int>>& pieceInfo, int patternSize, const vector<string>& pieces)\** — метод выполняет поиск шаблонов в заданном тексте с учётом символов-джокеров.

Аргументы:

*text* — исходный текст, в котором нужно искать шаблоны.

*Root* — корень автомата.

*PieceInfo* — вектор пар {длина куска, позиция в шаблоне}, необходимый для корректного вычисления позиции полного шаблона.

*PatternSize* — полная длина шаблона (включая джокеры).

*Pieces* — куски шаблона без джокеров.

4. *int main()* — Главная функция. Выполняет следующие шаги: считывает входные данные: *text, pattern, wildcard*, разбивает *pattern* на части (куски между *wildcard*), вызывает *buildTrie()* для построения автомата, запускает *search()* для поиска шаблона в тексте, выводит найденные позиции и пересекающиеся шаблоны, выводит общее количество вершин в автомате.

Структуры:

1. s*truct TrieNode* — структура представляет узел в Trie (боре), который используется для построения автомата Ахо-Корасика.

Поля:

*map<char, TrieNode\*> children* — ассоциативный массив, в котором ключ — символ, а значение — указатель на дочерний узел, по которому осуществляется переход.

*TrieNode\* fail* — суффиксная ссылка: указывает на узел, соответствующий самой длинной возможной суффиксной подстроке текущей строки.

*TrieNode\* terminalLink* — терминальная ссылка: указывает на следующий узел, содержащий выходной шаблон, если такой существует.

*vector<pair<int, int>> output* — массив пар (id, значение), в котором id — индекс шаблона, который заканчивается в этом узле. Используется для хранения шаблонов, заканчивающихся в данной вершине.

*int id* — уникальный идентификатор узла.

*static int counter* — счётчик, используемый для назначения уникального id каждому новому узлу.

Конструктор:

*TrieNode() : fail(nullptr), terminalLink(nullptr), id(counter++) {}* — инициализирует узел с nullptr в ссылках и присваивает уникальный id.

Глобальные переменные:

*totalNodes* — счётчик, отражающий общее количество созданных узлов в Trie. Обновляется при создании новых узлов.

Оценка сложности алгоритма:

Временная сложность:

1. Построение Trie: O(∑|P|), где |P| — суммарная длина всех шаблонов.

2. Построение суффиксных ссылок: O(∑|P|) — каждый узел обрабатывается один раз.

3. Поиск в тексте: O(n + k), где n — длина текста, k — общее количество найденных вхождений.

Итоговая сложность: O(∑∣P∣+n+k)

Сложность по памяти:

1. Trie: O(∑|P|) — каждый уникальный префикс создает новую вершину.

2. Дополнительная память: для хранения ссылок, очереди, выходных списков — также O(∑|P| + m), где m — количество различных символов в алфавите.

Итоговая сложность: O(∑∣P∣+n+k)

## Тестирование.

Результаты тестирования представлены в таблицах 1-2.

Таблица 1 – Результаты тестирования задания 5.1

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | ACGTACGT  1  CGT | Найденные вхождения:  2 1  6 1  Образцы с пересечениями:  Количество вершин в автомате: 4 | Один шаблон присутствует в тексте один раз. Результат соответствует ожидаемому. |
| 2. | ACGTACGT  2  AAA  TTT | Найденные вхождения:  Образцы с пересечениями:  Количество вершин в автомате: 7 | Ни один из шаблонов не встречается в тексте.. Результат соответствует ожидаемому. |
| 3. | ACGACGACG  2  ACG  CGA | 1 1  2 2  4 1  5 2  7 1  8 2Найденные вхождения:  1 1  2 2  4 1  5 2  7 1  Образцы с пересечениями:  ACG  CGA  Количество вершин в автомате: 7 | Проверка обработки перекрывающихся вхождений. Результат соответствует ожидаемому. |
| 4. | AAAAAA  2  AA  AAA | Найденные вхождения:  1 1  1 2  2 1  2 2  3 1  3 2  4 1  4 2  5 1  Образцы с пересечениями:  AA  AAA  Количество вершин в автомате: 4 | Проверка повторяющихся символов и шаблонов, состоящих из одинаковых букв. Результат соответствует ожидаемому. |
| 5. | NTAG  3  TAGT  TAG  T | Найденные вхождения:  2 2  2 3  Образцы с пересечениями:  TAG  T  Количество вершин в автомате: 5 | Пример со степика. Результат соответствует ожидаемому. |

Таблица 2 – Результаты тестирования задания 5.2

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | ACGTACGT  AC?T  ? | Позиции вхождения шаблона:  1  5  Образцы, участвующие в пересечениях:  Количество вершин в автомате: 4 | Базовый случай (один символ-подстановщик). Результат соответствует ожидаемому. |
| 2. | ABABA  A?A  ? | Позиции вхождения шаблона:  1  3  Образцы, участвующие в пересечениях:  A?A  Количество вершин в автомате: 2 | Наложение шаблона. Результат соответствует ожидаемому. |
| 3. | AACGTGAA  ?CGT?  ? | Позиции вхождения шаблона:  2  Образцы, участвующие в пересечениях:  Количество вершин в автомате: 4 | Подстановочный символ находится в начале и в конце шаблона. Результат соответствует ожидаемому. |
| 4. | ACGTACGT  TT??GG  ? | Позиции вхождения шаблона:  Образцы, участвующие в пересечениях:  Количество вершин в автомате: 5 | Текст и шаблон не имеют общих совпадений. Результат соответствует ожидаемому. |
| 5. | ACTANCA  A$$A$  $ | Позиции вхождения шаблона:  1  Образцы, участвующие в пересечениях:  Количество вершин в автомате: 2 | Пример со степика. Результат соответствует ожидаемому. |

## Выводы.

В результате выполнения лабораторной работы была реализована структура данных и алгоритм Ахо-Корасика, подтверждена его эффективность в задачах множественного поиска, а также получены практические навыки работы с автоматами и строковыми алгоритмами.

# Приложение А Исходный код программы

Название файла: lr\_5\_1.cpp

#include <iostream>

#include <vector>

#include <queue>

#include <string>

#include <map>

#include <algorithm>

#include <set>

using namespace std;

const int ALPHABET\_SIZE = 5; // размер алфавита

const char CHAR\_TO\_INDEX[] = {'A', 'C', 'G', 'T', 'N'}; // массив для преобразования символов в индексы

// структура для узла в Trie

struct TrieNode {

map<char, TrieNode\*> children; // ассоциативный массив для хранения дочерних узлов

TrieNode\* fail; // суффиксная ссылка

TrieNode\* terminalLink; // терминальная ссылка

vector<int> output; // массив, который хранит номера шаблонов, заканчивающихся в этом узле

TrieNode() : fail(nullptr), terminalLink(nullptr) {} // инициализация узла

};

// функция преобразования символа в индекс

int charToIndex(char c) {

switch (c) {

case 'A': return 0;

case 'C': return 1;

case 'G': return 2;

case 'T': return 3;

default: return 4;

}

}

// метод визуализации структуры Trie

void printTrie(TrieNode\* node, const vector<string>& patterns, const string& prefix = "", const string& indent = "", bool isLast = true) {

static bool isRoot = true; // переменная для отслеживания корня

if (isRoot) {

cout << "Корень\n";

isRoot = false;

}

// определение ветки для визуализации узла

string branch = indent + (isLast ? "└── " : "├── ");

if (!prefix.empty()) {

cout << branch << prefix.back(); // вывод последнего символа префикса

// если выходные ссылки существуют, вывести их

if (!node->output.empty()) {

cout << " (конец шаблона ";

for (size\_t i = 0; i < node->output.size(); ++i) {

int patternIndex = node->output[i];

cout << "[" << patternIndex << "] - \"" << patterns[patternIndex] << "\"";

if (i + 1 < node->output.size()) cout << ", ";

}

cout << ")";

}

cout << "\n";

}

// рекурсивный вызов для всех детей узла

auto it = node->children.begin(); // итератор для перебора детей

while (it != node->children.end()) {

string nextIndent = indent + (isLast ? " " : "│ ");

bool childIsLast = next(it) == node->children.end();

printTrie(it->second, patterns, prefix + it->first, nextIndent, childIsLast);

++it;

}

}

int nodeCount = 0; // счетчик узлов в Trie

// метод построения автомата Ахо-Корасика

void buildTrie(const vector<string>& patterns, TrieNode\* root) {

cout << "Построение trie:\n";

// перебор всех шаблонов

for (int i = 0; i < patterns.size(); ++i) {

TrieNode\* node = root; // начанием с корня

cout << " Шаблон " << i + 1 << ": " << patterns[i] << endl;

// перебор всех символов в шаблоне

for (char c : patterns[i]) {

int index = charToIndex(c); // индекс текущего символа

// если символ не найден среди детей текущего узла

if (node->children.find(CHAR\_TO\_INDEX[index]) == node->children.end()) {

node->children[CHAR\_TO\_INDEX[index]] = new TrieNode(); // создаем новый узел для этого символа

nodeCount++; // увеличиваем счетчик узлов

cout << " Добавлена вершина для символа '" << c << "'\n";

} else {

// если узел уже существует, переходим к нему

cout << " Переход по уже существующему символу '" << c << "'\n";

}

node = node->children[CHAR\_TO\_INDEX[index]]; // переход к дочернему узлу

}

node->output.push\_back(i); // номер шаблона в выходных ссылках узла

cout << " Отметка конца шаблона в вершине\n";

}

// построение суффиксных и терминальных ссылок

cout << "\n Построение суффиксных и терминальных ссылок:\n";

queue<TrieNode\*> q; // очередь для работы с узлами

// перебор дочерних узлов корня

for (auto& pair : root->children) {

pair.second->fail = root; // установка суффиксной ссылки на корень

q.push(pair.second); // добавление узела в очередь для дальнейшей обработки

cout << " Корневой потомок '" << pair.first << "' получает ссылку на корень\n";

}

// поиск в ширину для создания суффиксных ссылок

while (!q.empty()) {

TrieNode\* node = q.front(); // получение текущего узела из очереди

q.pop(); // удаление его из очереди

// перебор детей текущего узла

for (auto& pair : node->children) {

char c = pair.first; // текущий символ

TrieNode\* child = pair.second; // дочерний узел

TrieNode\* failNode = node->fail; // переменная для поиска суффиксной ссылки

// поиск подходящей суффиксной ссылки

while (failNode != nullptr && failNode->children.find(c) == failNode->children.end()) {

failNode = failNode->fail; // переход по суффиксной ссылке

}

// установка суффиксной ссылки

if (failNode == nullptr) {

child->fail = root; // если подходящий узел не найден, указываем на корень

} else {

child->fail = failNode->children[c]; // устанавливаем ссылку на найденный узел

}

cout << " Вершина по '" << c << "' получает суффиксную ссылку на ";

if (child->fail == root) cout << "корень\n";

else cout << "другую вершину по '" << c << "'\n";

// установка терминальных ссылок

if (!child->fail->output.empty()) {

child->terminalLink = child->fail; // если родительский узел имеет выходные ссылки, устанавливаем терминальную ссылку

} else {

child->terminalLink = child->fail->terminalLink; // унаследуем ссылку

}

cout << " Вершина по '" << c << "' получает терминальную ссылку на ";

if (child->terminalLink == nullptr) {

cout << "ничего (nullptr)\n";

} else if (!child->terminalLink->output.empty()) {

cout << "вершину с шаблонами: ";

for (size\_t i = 0; i < child->terminalLink->output.size(); ++i) {

int patIndex = child->terminalLink->output[i];

cout << "\"" << patterns[patIndex] << "\"";

if (i + 1 < child->terminalLink->output.size()) cout << ", ";

}

cout << endl;

} else {

cout << "вершину (унаследованную), но без собственного шаблона\n";

}

cout<<endl;

q.push(child); // добавляем дочерний узел в очередь

}

}

cout << "\nСтруктура построенного trie:\n";

printTrie(root, patterns); // визуализация структуры Trie

}

// метод поиска всех вхождений шаблонов в текст

void searchPatterns(const string& text, TrieNode\* root, vector<string>& patterns) {

TrieNode\* node = root; // начинаем с корня

vector<pair<int, int>> results; // массив для хранения результатов поиска

vector<pair<int, int>> intervals; // массив для хранения интервалов найденных шаблонов

map<int, string> idToPattern; // ассоциативный массив для сопоставления идентификаторов шаблонов с их текстом

// заполнение ассоциативного массива для сопоставления идентификаторов шаблонов с их текстом

for (int i = 0; i < patterns.size(); ++i)

idToPattern[i + 1] = patterns[i]; // сохраняем каждый шаблон с его индексом

cout << "\n Поиск шаблонов в тексте:\n";

// проход по каждому символу текста

for (int i = 0; i < text.size(); ++i) {

char currentChar = text[i]; // текущий символ текста

int index = charToIndex(currentChar); // индекс текущего символа

cout << " Позиция " << i << ": символ '" << currentChar << "' — ";

// переход по суффиксным ссылкам, если нет перехода к дочернему узлу

while (node != nullptr && node->children.find(CHAR\_TO\_INDEX[index]) == node->children.end()) {

cout << "нет перехода к дочернему узлу, переходим по суффиксной ссылке ";

node = node->fail; // переход по суффиксной ссылке

if (node == nullptr) cout << "к корню\n"; // если достигли корня

}

// если достигли корня, перезапускаем поиск

if (node == nullptr) {

node = root; // сброс узела на корень

cout << "перешли к корню\n";

} else {

node = node->children[CHAR\_TO\_INDEX[index]]; // переход к дочернему узлу

cout << "переход выполнен\n";

}

TrieNode\* temp = node; // указатель на текущий узел

// проверка всех выходов в текущем узле

while (temp != nullptr) {

// перебор выходных индексов

for (int patternIndex : temp->output) {

int startPos = i - patterns[patternIndex].size() + 2; // начальная позиция вхождения

results.push\_back({startPos, patternIndex + 1}); // сохранение результатов

intervals.push\_back({startPos, i + 1}); // сохранение интервала вхождения

cout << " Найден шаблон " << patterns[patternIndex] << " на позиции " << startPos << endl;

}

temp = temp->terminalLink; // переход к терминальной ссылке

}

}

sort(results.begin(), results.end()); // сортируем результаты по позициям

cout << "\nНайденные вхождения:\n";

for (const auto& res : results) {

cout << res.first << " " << res.second << "\n";

}

// анализ пересечений

set<int> overlappingPatterns; // множество для хранения пересекающихся шаблонов

sort(intervals.begin(), intervals.end()); // сортируем интервалы

// поиск пересекающихся шаблонов

// перебор всех интервалов

for (int i = 0; i < intervals.size(); ++i) {

// сравнение с последующими интервалами

for (int j = i + 1; j < intervals.size(); ++j) {

// проверка на пересечение

if (intervals[i].second >= intervals[j].first) {

// сохранение идентификаторов пересекающихся шаблонов

overlappingPatterns.insert(results[i].second);

overlappingPatterns.insert(results[j].second);

} else break;

}

}

cout << "\nОбразцы с пересечениями:\n";

for (int id : overlappingPatterns) {

cout << idToPattern[id] << "\n";

}

}

int main() {

string text; // стока для хранения текста для поиска

cin >> text;

int n; // количество шаблонов

cin >> n;

vector<string> patterns(n); // массив для хранения шаблонов

for (int i = 0; i < n; ++i) {

cin >> patterns[i];

}

TrieNode\* root = new TrieNode(); // создание корневого узла Trie

nodeCount = 1; // начальное значение счетчика узлов

buildTrie(patterns, root); // построение Trie на основе шаблонов

searchPatterns(text, root, patterns); // поиск шаблонов в тексте

cout << "\nКоличество вершин в автомате: " << nodeCount << endl;

return 0;

}

Название файла: lr\_5\_2.cpp

#include <iostream>

#include <vector>

#include <queue>

#include <map>

#include <set>

#include <algorithm>

using namespace std;

// структура для узла Trie

struct TrieNode {

map<char, TrieNode\*> children; // ассоциативный массив для хранения дочерних узлов

TrieNode\* fail; // суффиксная ссылка

TrieNode\* terminalLink; // терминальная ссылка

vector<pair<int, int>> output; // массив, который хранит номера шаблонов, заканчивающихся в этом узле

int id; // уникальный идентификатор узла

static int counter; // счетчик для уникальных идентификаторов узлов

TrieNode() : fail(nullptr), terminalLink(nullptr), id(counter++) {} // конструктор узла

};

// метод визуализации структуры Trie

void printAutomaton(TrieNode\* root) {

cout << "\n Структура построенного trie:\n";

queue<TrieNode\*> q;

set<TrieNode\*> visited;

q.push(root);

visited.insert(root);

while (!q.empty()) {

TrieNode\* node = q.front();

q.pop();

cout << "Вершина " << node->id << ":\n";

if (node->fail)

cout << " Суффиксная ссылка -> " << node->fail->id << "\n";

else

cout << " Суффиксная ссылка -> nullptr\n";

if (node->terminalLink)

cout << " Терминальная ссылка -> " << node->terminalLink->id << "\n";

if (!node->output.empty()) {

cout << " Выходы: ";

for (auto& [pid, \_] : node->output)

cout << pid << " ";

cout << "\n";

}

for (auto& [c, child] : node->children) {

cout << " Переход по '" << c << "' -> вершина " << child->id << "\n";

if (visited.find(child) == visited.end()) {

q.push(child);

visited.insert(child);

}

}

}

}

int TrieNode::counter = 0; // счетчик узлов

int totalNodes = 0; // общее количество узлов в Trie

// метод построения автомата Ахо-Корасика

void buildTrie(const vector<string>& pieces, TrieNode\* root) {

totalNodes = 1; // корень уже существует

cout << " Построение trie\n";

// перебор всех шаблонов

for (int id = 0; id < pieces.size(); ++id) {

TrieNode\* node = root; // добавление с корня

cout << "Добавляем кусок шаблона: \"" << pieces[id] << "\" (id=" << id << ")\n";

// перебор символов в текущем шаблоне

for (char c : pieces[id]) {

// если символ еще не существует в дочерних узлах

if (node->children.find(c) == node->children.end()) {

node->children[c] = new TrieNode(); // создаем новый узел для символа

++totalNodes; // увеличиваем счетчик узлов

cout << " Создана вершина " << node->children[c]->id << " по символу '" << c << "'\n";

}

node = node->children[c]; // переход к дочернему узлу

}

node->output.push\_back({id, 0}); // запоминаем конец шаблона в узле

cout << " Отметка конца шаблона в вершине " << node->id << "\n";

}

queue<TrieNode\*> q; // очередь для хранения узлов

// инициализация суффиксных ссылок для детей корня

for (auto& [c, child] : root->children) {

child->fail = root; // установка ссылок на корень

q.push(child); // добавление узла в очередь

}

cout << "\n Построение суффиксных и терминальных ссылок\n";

// поиск в ширину для построения суффиксных ссылок

while (!q.empty()) {

TrieNode\* node = q.front(); // получение текущего узла

q.pop(); // удаление узла из очереди

// перебор детей текущего узла

for (auto& [c, child] : node->children) {

TrieNode\* failNode = node->fail; // инициализация узла для поиска по суффиксной ссылке

// поиск подходящей суффиксной ссылки

while (failNode && failNode->children.find(c) == failNode->children.end()) {

failNode = failNode->fail; // переход по суффиксной ссылке

}

if (failNode) {

child->fail = failNode->children[c]; // если найдена подходящая ссылка, назначение ссылки на узел

cout << "Вершина " << child->id << " получает суффиксную ссылку на " << child->fail->id << "\n";

} else {

child->fail = root; // если не найдено, указываем на корень

cout << "Вершина " << child->id << " получает суффиксную ссылку на корень\n";

}

// установка терминальной ссылки

if (!child->fail->output.empty()) {

child->terminalLink = child->fail; // если узел-предок имеет выходные ссылки, устанавливаем терминальную ссылку

cout << " Установлена терминальная ссылка на вершину " << child->terminalLink->id << "\n";

} else {

child->terminalLink = child->fail->terminalLink; // унаследуем ссылку

if (child->terminalLink)

cout << " Унаследована терминальная ссылка на вершину " << child->terminalLink->id << "\n";

}

q.push(child); // добавляем дочерний узел в очередь

}

}

printAutomaton(root); // визуализация структуры Trie

}

// метод поиска шаблонов в тексте

void search(const string& text, TrieNode\* root, const vector<pair<int, int>>& pieceInfo, int patternSize, const vector<string>& pieces) {

int n = text.size(); // размер текста

vector<int> count(n, 0); // массив для подсчета вхождений шаблонов

vector<vector<string>> matches(n); // массив для хранения найденных совпадений в текстовых позициях

TrieNode\* node = root; // начинаем с корня

cout << "\n Поиск в тексте\n";

// перебор каждого символа в тексте

for (int i = 0; i < n; ++i) {

char c = text[i]; // текущий обрабатываемый символ

cout << "Обработка символа '" << c << "' (позиция " << i << "):\n";

// переход по суффиксным ссылкам, если перехода к символу нет

while (node && node->children.find(c) == node->children.end()) {

cout << " Переход по суффиксной ссылке от вершины " << node->id << "\n";

node = node->fail; // переход к родительскому узлу

}

// если достигли корня, возвращаемся

if (!node) {

node = root;

cout << " Вернулись в корень\n";

} else {

node = node->children[c]; // переход к дочернему узлу

cout << " Перешли в вершину " << node->id << "\n";

}

TrieNode\* temp = node; // указатель для проверки узлов

// проверка выходов из текущего узла

while (temp) {

// перебор выходов

for (auto& [pieceId, \_] : temp->output) {

int pieceLen = pieceInfo[pieceId].first; // длина образца

int piecePos = pieceInfo[pieceId].second; // позиция образца в шаблоне

int start = i - pieceLen + 1 - piecePos; // вычисляем стартовую позицию

// проверка, что стартовая позиция действительна

if (start >= 0 && start + patternSize <= n) {

count[start]++; // увеличиваем счетчик для найденного вхождения

matches[start].push\_back(pieces[pieceId]); // добавление совпадающего образца в позицию

cout << " Найден кусок \"" << pieces[pieceId] << "\" => возможное вхождение шаблона на позиции " << start + 1 << "\n";

}

}

temp = temp->terminalLink; // переход к терминальной ссылке

}

}

vector<int> resultPositions; // массив для хранения фактических позиций вхождений шаблона

// перебор позиций в тексте

for (int i = 0; i <= n - patternSize; ++i) {

if (count[i] == pieceInfo.size()) {

resultPositions.push\_back(i); // если количество вхождений соответствует количеству образцов, сохраняем позицию

}

}

cout << "Позиции вхождения шаблона:\n";

for (int pos : resultPositions) {

cout << pos + 1 << endl; //

}

// поиск пересекающихся шаблонов

set<string> overlappingPatterns; // множество для хранения пересекающихся образцов

sort(resultPositions.begin(), resultPositions.end()); // сортируем позиции

// проверка пересечений между образцами

for (int i = 0; i < resultPositions.size(); ++i) {

int startA = resultPositions[i]; // начало первого вхождения

int endA = startA + patternSize - 1; // конец первого вхождения

for (int j = i + 1; j < resultPositions.size(); ++j) {

int startB = resultPositions[j]; // начало второго вхождения

// проверка на пересечение

if (startB <= endA) {

for (const string& p : matches[startA]) overlappingPatterns.insert(p); // сохранение образцов из первого вхождения

for (const string& p : matches[startB]) overlappingPatterns.insert(p); // сохранение образцов из второго вхождения

} else break;

}

}

cout << "\nОбразцы, участвующие в пересечениях:\n";

// если найден только один шаблон

if (resultPositions.size() < 2) {

for (const string& p : pieces) { // вывод всех образцов

cout << p << endl;

}

} else { // если есть несколько шаблонов

for (const string& pat : overlappingPatterns) { // перебор пересекающихся образцов

cout << pat << endl;

}

}

cout << "\nКоличество вершин в автомате: " << totalNodes << endl;

}

int main() {

string text, pattern; // переменные для хранения текста и шаблона

char wildcard; // символ джокера

cin >> text >> pattern >> wildcard;

TrieNode\* root = new TrieNode(); // создание корневого узла Trie

vector<string> pieces; // массив для хранения частей шаблона

vector<pair<int, int>> pieceInfo; // массив для хранения информации о частях шаблона

int m = pattern.size(); // размер шаблона

// перебор символов в шаблоне

for (int i = 0; i < m;) {

if (pattern[i] == wildcard) { //

++i; // если символ - джокер, пропускаем этот символ

continue; // переход к следующему

}

int j = i; // начинаем с текущего символа

string piece; // строка для хранения текущей части шаблона

// считываем всю часть шаблона до джокера

while (j < m && pattern[j] != wildcard) {

piece += pattern[j]; // добавляем символ к части

++j; // переход к следующему символу

}

pieces.push\_back(piece); // сохраняем часть шаблона в вектор

pieceInfo.push\_back({(int)piece.size(), i}); // сохраняем информацию о части шаблона

i = j; // обновляем индекс

}

cout << "\nНайденные куски шаблона:\n";

for (int i = 0; i < pieces.size(); ++i) {

cout << " \"" << pieces[i] << "\" (позиция в шаблоне: " << pieceInfo[i].second << ")\n";

}

cout << "\n";

buildTrie(pieces, root); // построение Trie на основе найденных частей

search(text, root, pieceInfo, pattern.size(), pieces); // поиск шаблонов в тексте

return 0;

}