МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

ТЕМА: «ПОИСК НАБОРА ПОДСТРОК В СТРОКЕ(АХО-КОРАСИК)»

ВАРИАНТ 5

Гусакова К.А.

Студент гр. 3388 _____

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

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

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.
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}.

Задание на вариант:

Вычислить максимальное количество дуг, исходящих из одной вершины в боре; вырезать из строки поиска все найденные образцы и вывести остаток строки поиска.

Описание алгоритма для решения задачи

Алгоритм Ахо-Корасика — это метод эффективного поиска нескольких образцов (подстрок) одновременно в одном тексте. Он строит автомат по множеству шаблонов и обходит текст за линейное время.

Алгоритм работает в 2 этапа:

1. Построение бора (trie) — дерево всех шаблонов.
2. Построение автомата Ахо-Корасика:
   - добавление суффиксных ссылок;
   - добавление терминальных ссылок;
   - создание переходов «по несуществующим ребрам» через ссылки.

Затем запускается фаза поиска, где по каждому символу текста происходит переход по автомату, и на каждом шаге ищутся вхождения шаблонов.

1. Шаги:
    1. Пользователь вводит набор строк-шаблонов (например, AT, CGT, и т.д.).
    2. По этим строкам строится бор (add_pattern()).
    3. Далее по бору строятся суффиксные и терминальные ссылки (build_links()).
    4. Автомат готов, и запускается поиск по тексту: на каждом символе текста происходит переход по автомату, в случае успеха — извлекаются все шаблоны, которые заканчиваются на текущей вершине или терминальных ссылках.
    5. Выводятся позиции всех вхождений.

2. Шаблон P разбивается на подстроки без подстановок. Пример: A?T?G -> ["A", "T", "G"] с сохранением их смещений в шаблоне.

Каждая такая подстрока добавляется в бор с сохранением ID (add_pattern()).

По бору строится автомат Ахо-Корасика (build_links()).

Далее запускается сканирование текста, как и в классическом Ахо-Корасике.

Если шаблон P состоит из 3 подстрок (например, A, T, G), то чтобы считать, что вся маска совпала, нужно чтобы все 3 подстроки совпали в тексте с нужными смещениями относительно текущей позиции.

Это реализовано через вектор match[pos], где pos - предполагаемая начальная позиция шаблона P в тексте. Каждый найденный фрагмент увеличивает счетчик. Если счетчик достиг количества подстрок, значит, полное совпадение найдено.

Оценка сложности алгоритма по времени и памяти

Временная и пространственная сложность:

1. Сначала строится префиксное дерево (trie), где каждая вершина соответствует уникальному префиксу из шаблонов. Этот этап требует обработки всех символов всех шаблонов. Затем на основе дерева строятся

суффиксные и терминальные ссылки, которые позволяют быстро переходить к другим возможным шаблонам при несовпадении символов во время обхода текста. Эти ссылки также используются для нахождения всех возможных совпадений в каждой позиции текста, включая перекрывающиеся.

Во время обработки текста алгоритм последовательно читает каждый символ и с помощью построенных переходов и ссылок быстро находит все шаблоны, которые заканчиваются на текущей позиции. При этом не происходит возвратов в тексте, и каждый символ обрабатывается эффективно. После нахождения всех совпадений шаблоны удаляются из текста, при этом для каждого символа лишь один раз проверяется, нужно ли его оставить.

Таким образом, затраты времени зависят от общей длины всех шаблонов, длины текста и количества найденных совпадений. Память используется на хранение trie, ссылок, шаблонов, текста и вспомогательных структур, и она пропорциональна их размеру.

Временная O(m·k + T + k), где *m* - суммарная длина всех шаблонов, *k* - размер алфавита, *T* - длина текста, *k* - общее количество найденных совпадений.

Пространственная O(m·k + T·n), где n - число шаблонов.

2. Построение автомата занимает O(S), где S - суммарная длина всех подстрок шаблона (без джокеров). Построение суффиксных ссылок также O(S).

Обработка текста длины n - O(n + K), где K - общее количество срабатываний подстрок в тексте

Хранение автомата требует O(S * A), где A = 5 - размер алфавита (A, C, G, T, N). Плюс O(S) для выходов и ссылок.

Временная: O(S + n + K)

Пространственная: O(S * A)

Описание способа хранения частичных решений.

1. Частичные решения хранятся в вершинах через:

префиксы (переходы next), выходы из шаблонов (списки output) и переходы к другим завершённым шаблонам (терминальные ссылки)

2. В дополнение к предыдущему заданию:

1. Частичные решения хранятся в массиве match[], где для каждой позиции текста фиксируется, сколько подстрок из шаблона (без джокеров) совпали с соответствующим смещением. Если на позиции совпали все подстроки, то это полное вхождение шаблона. Такой способ позволяет эффективно объединять результаты совпадений подстрок в одно итоговое решение. - сколько фрагментов шаблона совпало в данной позиции.

Граф автомата хранится неявно в массиве структур trie[], где каждая вершина содержит массив переходов next[] размером по алфавиту (в данном случае 5), суффиксную ссылку link, терминальную ссылку terminal_link, список выходов output, а также данные о родителе и символе перехода. Переходы между вершинами задаются через массив next[], где по индексу символа хранится номер следующей вершины. Это реализует ориентированный граф, в котором вершины - состояния автомата, а дуги - переходы по символам.

По заданию в программе для каждой вершины перебираются все возможные символы алфавита (A, C, G, T, N, то есть 5 штук). Если из вершины по какому-то символу есть реальное добавленное ребро (а не "переход по умолчанию", добавленный в корне), то счётчик увеличивается. Максимум среди таких счётчиков по всем вершинам и является искомым значением.

Использованные оптимизации.

1. Граф не хранится как список смежности. Вместо этого каждая вершина содержит фиксированный массив next[ALPHABET], что даёт быстрый доступ к переходу по символу.

2. real_edge Позволяет отличать добавленные переходы от автоматических(например, в корне все неопределённые переходы указывают на сам корень).

Тестирование. Показ граничных случаев алгоритма.

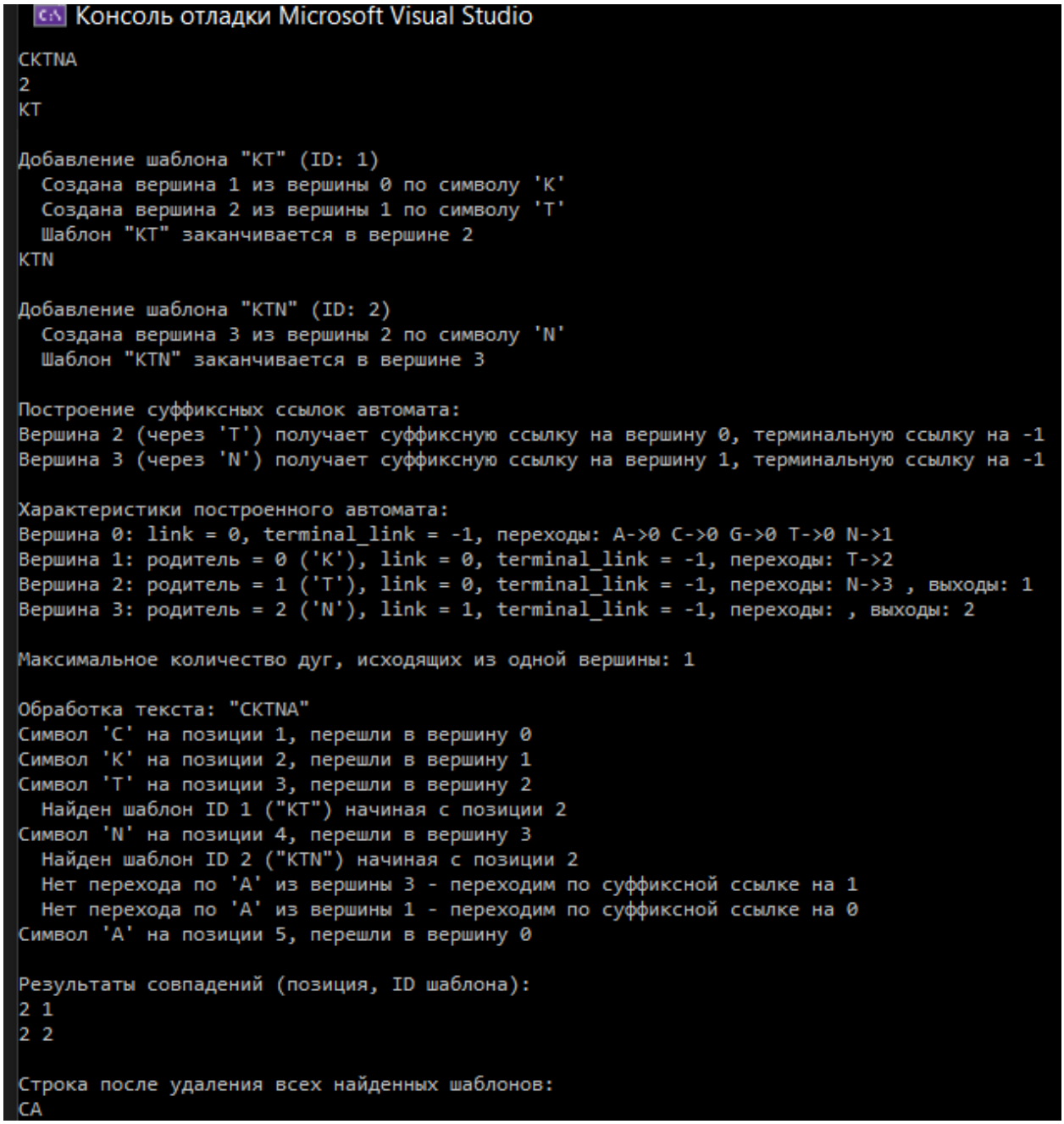

Рисунок 1 – программа 1.

```
ACTATCNTCTA
CT*
*

Разбиение шаблона на подстроки:
0: "CT" (смещение = 0)
Добавлена вершина 1: 'C' из вершины 0
Добавлена вершина 2: 'T' из вершины 1
Паттерн "CT" добавлен в вершину 2 (ID: 0)

Построение суффиксных и терминальных ссылок
Вершина 2: суффиксная ссылка -> 0, терминальная ссылка -> -1

Характеристики автомата:
Вершина 0:
  Суффиксная ссылка: 0
  Терминальная ссылка: -1
  Переходы: C->1
  Выходы:
--------------------------
Вершина 1:
  Родитель = 0 (по символу 'C')
  Суффиксная ссылка: 0
  Терминальная ссылка: -1
  Переходы: T->2
  Выходы:
--------------------------
Вершина 2:
  Родитель = 1 (по символу 'T')
  Суффиксная ссылка: 0
  Терминальная ссылка: -1
  Переходы:
  Выходы: (ID: 0)
--------------------------

Граф автомата (переходы):
  0 --C--> 1
  1 --T--> 2

Обработка текста:
Символ 'A' (позиция 1), перешли в вершину 0
Символ 'C' (позиция 2), перешли в вершину 1
Символ 'T' (позиция 3), перешли в вершину 2
  Найден шаблон 0 ("CT") на позиции 1
  Перехода по 'A' нет из вершины 2, следуем по ссылке: 0
Символ 'A' (позиция 4), перешли в вершину 0
Символ 'T' (позиция 5), перешли в вершину 0
Символ 'C' (позиция 6), перешли в вершину 1
  Перехода по 'N' нет из вершины 1, следуем по ссылке: 0
Символ 'N' (позиция 7), перешли в вершину 0
Символ 'T' (позиция 8), перешли в вершину 0
Символ 'C' (позиция 9), перешли в вершину 1
Символ 'T' (позиция 10), перешли в вершину 2
  Найден шаблон 0 ("CT") на позиции 8
  Перехода по 'A' нет из вершины 2, следуем по ссылке: 0
Символ 'A' (позиция 11), перешли в вершину 0

Позиции начала полного совпадения шаблона:
2
9

Максимальное количество дуг, исходящих из одной вершины: 1

Строка без вхождений шаблона:
ATCNT
```

Рисунок 2 – программа 2.

ВЫВОД

Обе программы реализуют эффективные методы множественного поиска шаблонов в строке с использованием автомата Ахо-Корасика и построением суффиксных и терминальных ссылок. Несмотря на различия в постановке задачи (первая программа - классический множественный поиск, вторая - поиск с шаблоном, содержащим подстановочные символы), обе решения демонстрируют высокую эффективность за счёт:

- линейной временной сложности по отношению к длине текста и суммарной длине всех шаблонов,
- компактного хранения переходов и ссылок в боре,
- повторного использования общих префиксов,
- минимального количества дополнительных структур.

В первой программе реализован стандартный Ахо-Корасик, что позволяет быстро находить все вхождения набора шаблонов в строку. Во второй программе этот алгоритм расширен для работы с шаблонами, содержащими подстановочные символы, путём разбиения шаблона на подстроки и последующей сборки полного совпадения по массиву частичных результатов.

ПРИЛОЖЕНИЕ. ИСХОДНЫЙ КОД ПРОГРАММЫ

1.

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <cstring>
using namespace std;

const int ALPHABET = 5;
const int MAXNODES = 300000;

int to_index(char c) {
    if (c == 'A') return 0;
    if (c == 'C') return 1;
    if (c == 'G') return 2;
    if (c == 'T') return 3;
    return 4; // N
}

char from_index(int i) {
    return "ACGTN"[i];
}

struct Node {
    int next[ALPHABET];
    int link;
    int terminal_link;
    vector<int> output;
    int parent;
    char parent_char;
    Node() {
        memset(next, -1, sizeof(next));
        link = -1;
```

```
        terminal_link = -1;
        parent = -1;
        parent_char = 0;
    }
};

Node trie[MAXNODES];
bool real_edge[MAXNODES][ALPHABET] = {};
int trie_size = 1;

void add_pattern(const string& pattern, int id) {
    cout << "\nДобавление шаблона \"" << pattern << "\" (ID: "
<< id << ")\n";
    int node = 0;
    for (int i = 0; i < pattern.size(); i++) {
        int c = to_index(pattern[i]);
        if (trie[node].next[c] == -1) {
            trie[trie_size] = Node();
            trie[trie_size].parent = node;
            trie[trie_size].parent_char = pattern[i];
            trie[node].next[c] = trie_size;
            real_edge[node][c] = true;
            cout << "  Создана вершина " << trie_size << " из
вершины " << node
                 << " по символу '" << pattern[i] << "'\n";
            trie_size++;
        }
        node = trie[node].next[c];
    }
    trie[node].output.push_back(id);
    cout << "  Шаблон \"" << pattern << "\" заканчивается в
вершине " << node << "\n";
}

void build_links() {
    cout << "\nПостроение суффиксных ссылок автомата:\n";
    queue<int> q;
    trie[0].link = 0;

    for (int c = 0; c < ALPHABET; c++) {
        if (trie[0].next[c] != -1) {
            trie[trie[0].next[c]].link = 0;
            q.push(trie[0].next[c]);
        }
        else {
            trie[0].next[c] = 0;
        }
    }

    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (int c = 0; c < ALPHABET; c++) {
            int u = trie[v].next[c];
```

```
            if (u != -1) {
                int f = trie[v].link;
                while (trie[f].next[c] == -1)
                    f = trie[f].link;
                trie[u].link = trie[f].next[c];

                // Построение терминальных ссылок
                trie[u].terminal_link =
(!trie[trie[u].link].output.empty())
                    ? trie[u].link
                    : trie[trie[u].link].terminal_link;

                cout << "Вершина " << u << " (через '" <<
trie[u].parent_char
                    << "') получает суффиксную ссылку на вершину
" << trie[u].link
                    << ", терминальную ссылку на " <<
trie[u].terminal_link << "\n";

                for (int j : trie[trie[u].link].output) {
                    trie[u].output.push_back(j);
                    cout << "  Наследован выход шаблона ID: " <<
j << "\n";
                }

                q.push(u);
            }
        }
    }

    cout << "\nХарактеристики построенного автомата:\n";
    for (int i = 0; i < trie_size; i++) {
        cout << "Вершина " << i << ": ";
        if (i != 0) {
            cout << "родитель = " << trie[i].parent
                << " ('" << trie[i].parent_char << "'), ";
        }
        cout << "link = " << trie[i].link
            << ", terminal_link = " << trie[i].terminal_link
            << ", переходы: ";
        for (int j = 0; j < ALPHABET; j++) {
            if (trie[i].next[j] != -1) {
                cout << from_index(j) << "->" << trie[i].next[j]
<< " ";
            }
        }
        if (!trie[i].output.empty()) {
            cout << ", выходы: ";
            for (int x : trie[i].output) cout << x << " ";
        }
        cout << endl;
    }
```

```
    int max_out_degree = 0;
    for (int i = 0; i < trie_size; i++) {
        int count = 0;
        for (int j = 0; j < ALPHABET; j++) {
            if (real_edge[i][j]) count++;
        }
        max_out_degree = max(max_out_degree, count);
    }
    cout << "\nМаксимальное количество дуг, исходящих из одной
вершины: "
        << max_out_degree << "\n";
}

int main() {
    setlocale(LC_ALL, "Russian");
    string T;
    int n;
    cin >> T >> n;

    vector<string> patterns(n);
    vector<int> pattern_lengths(n);

    for (int i = 0; i < n; i++) {
        cin >> patterns[i];
        pattern_lengths[i] = patterns[i].size();
        add_pattern(patterns[i], i + 1);
    }

    build_links();

    cout << "\nОбработка текста: \"" << T << "\"\n";
    vector<pair<int, int>> result;
    int node = 0;
    for (int i = 0; i < T.size(); i++) {
        int c = to_index(T[i]);
        while (trie[node].next[c] == -1) {
            cout << "  Нет перехода по '" << T[i] << "' из
вершины " << node
                << " — переходим по суффиксной ссылке на " <<
trie[node].link << endl;
            node = trie[node].link;
        }
        node = trie[node].next[c];
        cout << "Символ '" << T[i] << "' на позиции " << i + 1
            << ", перешли в вершину " << node << endl;

        // Обработка терминальных ссылок
        int temp = node;
        while (temp != -1 && temp != 0) {
            for (int j : trie[temp].output) {
                int pos = i - pattern_lengths[j - 1] + 2;
                cout << "  Найден шаблон ID " << j << " (\"" <<
patterns[j - 1]
```

```
                    << "\") начиная с позиции " << pos << "\n";
                result.push_back({ pos, j });
            }
            temp = trie[temp].terminal_link;
        }
    }

    sort(result.begin(), result.end());

    cout << "\nРезультаты совпадений (позиция, ID шаблона):\n";
    for (const auto& p : result) {
        cout << p.first << " " << p.second << "\n";
    }

    vector<bool> keep(T.size(), true);
    for (const auto& p : result) {
        int start = p.first - 1;
        int len = pattern_lengths[p.second - 1];
        for (int i = start; i < start + len && i < T.size();
i++) {
            keep[i] = false;
        }
    }

    string remaining;
    for (int i = 0; i < T.size(); i++) {
        if (keep[i]) remaining += T[i];
    }

    cout << "\nСтрока после удаления всех найденных шаблонов:\n"
<< remaining << endl;

    return 0;
}
```

2.

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
#include <algorithm>
using namespace std;

const int ALPHABET = 5;
const int MAXNODES = 300000;

int to_index(char c) {
    if (c == 'A') return 0;
    if (c == 'C') return 1;
    if (c == 'G') return 2;
    if (c == 'T') return 3;
    return 4; // 'N'
}
```

```
char from_index(int i) {
    return "ACGTN"[i];
}

struct Node {
    int next[ALPHABET];
    int link;
    int terminal_link;
    vector<int> output;
    int parent;
    char parent_char;
    Node() {
        memset(next, -1, sizeof(next));
        link = -1;
        terminal_link = -1;
        parent = -1;
        parent_char = 0;
    }
};

Node trie[MAXNODES];
bool real_edge[MAXNODES][ALPHABET] = {};
int trie_size = 1;

struct SubPattern {
    string s;
    int offset;
};

void add_pattern(const string& pattern, int id) {
    int node = 0;
    for (int i = 0; i < pattern.size(); i++) {
        int c = to_index(pattern[i]);
        if (trie[node].next[c] == -1) {
            trie[trie_size] = Node();
            trie[trie_size].parent = node;
            trie[trie_size].parent_char = pattern[i];
            trie[node].next[c] = trie_size;
            real_edge[node][c] = true;
            cout << "Добавлена вершина " << trie_size << ": '"
<< pattern[i] << "' из вершины " << node << endl;
            trie_size++;
        }
        node = trie[node].next[c];
    }
    trie[node].output.push_back(id);
    cout << "Паттерн \"" << pattern << "\" добавлен в вершину "
<< node << " (ID: " << id << ")\n";
}

void build_links() {
    cout << "\nПостроение суффиксных и терминальных ссылок\n";
    queue<int> q;
```

```
    trie[0].link = 0;

    for (int c = 0; c < ALPHABET; c++) {
        if (trie[0].next[c] != -1) {
            trie[trie[0].next[c]].link = 0;
            q.push(trie[0].next[c]);
        }
        else {
            trie[0].next[c] = 0;
        }
    }

    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (int c = 0; c < ALPHABET; c++) {
            int u = trie[v].next[c];
            if (u != -1) {
                int f = trie[v].link;
                while (trie[f].next[c] == -1) {
                    f = trie[f].link;
                }
                trie[u].link = trie[f].next[c];
                trie[u].terminal_link =
(!trie[trie[u].link].output.empty())
                    ? trie[u].link
                    : trie[trie[u].link].terminal_link;

                cout << "Вершина " << u << ": суффиксная ссылка
-> " << trie[u].link
                    << ", терминальная ссылка -> " <<
trie[u].terminal_link << endl;

                q.push(u);
            }
        }
    }

    cout << "\nХарактеристики автомата:\n";
    for (int i = 0; i < trie_size; i++) {
        cout << "Вершина " << i << ":\n";
        if (i != 0) {
            cout << "  Родитель = " << trie[i].parent << " (по
символу '" << trie[i].parent_char << "')\n";
        }
        cout << "  Суффиксная ссылка: " << trie[i].link << "\n";
        cout << "  Терминальная ссылка: " <<
trie[i].terminal_link << "\n";
        cout << "  Переходы: ";
        for (int j = 0; j < ALPHABET; j++) {
            if (real_edge[i][j]) {
                cout << from_index(j) << "->" << trie[i].next[j]
<< " ";
            }
```

```
        }
        cout << "\n  Выходы: ";
        for (int id : trie[i].output) {
            cout << "(ID: " << id << ") ";
        }
        cout << "\n-------------------------\n";
    }

    cout << "\nГраф автомата (переходы):\n";
    for (int i = 0; i < trie_size; i++) {
        for (int j = 0; j < ALPHABET; j++) {
            if (real_edge[i][j]) {
                cout << "  " << i << " --" << from_index(j) <<
"--> " << trie[i].next[j] << endl;
            }
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    string T, P;
    char wildcard;
    cin >> T >> P >> wildcard;

    vector<SubPattern> subpatterns;
    string current = "";
    int offset = 0;

    for (int i = 0; i < P.size(); i++) {
        if (P[i] == wildcard) {
            if (!current.empty()) {
                subpatterns.push_back({ current, offset });
                current = "";
            }
            offset = i + 1;
        }
        else {
            if (current.empty()) offset = i;
            current += P[i];
        }
    }
    if (!current.empty()) {
        subpatterns.push_back({ current, offset });
    }

    cout << "\nРазбиение шаблона на подстроки:\n";
    for (int i = 0; i < subpatterns.size(); i++) {
        cout << i << ": \"" << subpatterns[i].s << "\" (смещение
= " << subpatterns[i].offset << ")\n";
        add_pattern(subpatterns[i].s, i);
    }
```

```
    build_links();

    int m = P.size();
    vector<int> match(T.size(), 0);

    cout << "\nОбработка текста:\n";
    int node = 0;
    for (int i = 0; i < T.size(); i++) {
        int c = to_index(T[i]);
        while (trie[node].next[c] == -1) {
            cout << "  Перехода по '" << T[i] << "' нет из
вершины " << node << ", следуем по ссылке: " << trie[node].link
<< endl;
            node = trie[node].link;
        }
        node = trie[node].next[c];
        cout << "Символ '" << T[i] << "' (позиция " << i + 1 <<
"), перешли в вершину " << node << endl;
        //тут находится шаблон
        int temp = node;
        while (temp != -1 && temp != 0) {
            for (int j = 0; j < trie[temp].output.size(); j++) {
                int pat_id = trie[temp].output[j];
                int pos = i - subpatterns[pat_id].s.size() + 1 -
subpatterns[pat_id].offset;
                if (0 <= pos && pos + m <= T.size()) {
                    match[pos]++;
                    cout << "  Найден шаблон " << pat_id << "
(\"" << subpatterns[pat_id].s << "\") на позиции " << pos <<
endl;
                }
            }
            temp = trie[temp].terminal_link;
        }
    }

    cout << "\nПозиции начала полного совпадения шаблона:\n";
    vector<int> found_positions;
    for (int i = 0; i < T.size(); i++) {
        if (match[i] == subpatterns.size()) {
            cout << (i + 1) << "\n";
            found_positions.push_back(i);
        }
    }

    int max_out_degree = 0;
    for (int i = 0; i < trie_size; i++) {
        int count = 0;
        for (int j = 0; j < ALPHABET; j++) {
            if (real_edge[i][j]) count++;
        }
        max_out_degree = max(max_out_degree, count);
    }
```

```
    cout << "\nМаксимальное количество дуг, исходящих из одной
вершины: " << max_out_degree << "\n";

    cout << "\nСтрока без вхождений шаблона:\n";
    vector<bool> masked(T.size(), false);
    for (int start : found_positions) {
        for (int i = start; i < start + m; i++) {
            if (i < T.size()) masked[i] = true;
        }
    }

    for (int i = 0; i < T.size(); i++) {
        if (!masked[i]) cout << T[i];
    }
    cout << endl;

    return 0;
}
```