**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**

1. **Временная и пространственная сложность:**

Осуществляется поиск множества заданных шаблонов, временная сложность равна O(P * Σ + N + M). Это объясняется тем, что сначала строится бор, а затем автомат (суффиксные и терминальные ссылки) за время

пропорциональное количеству вершин и алфавиту, затем выполняется один линейный проход по тексту, и в процессе обработки возможен вывод всех найденных вхождений. Пространственная сложность в этом случае - O(P * Σ), поскольку каждая вершина может содержать до Σ переходов, и общее количество вершин не превосходит P.

**2.** Шаблон с символами ? разбивается на s непустых подстрок, которые добавляются в автомат как отдельные шаблоны. Временная сложность в этом случае - O(P * Σ + N * s). Первая часть - это построение автомата, аналогично первой версии. Вторая часть возникает из-за того, что на каждой позиции текста может произойти проверка для каждой из s подстрок, и если она совпадает, увеличивается счётчик возможных полных совпадений шаблона. Пространственная сложность остаётся аналогичной - O(P * Σ), плюс дополнительно требуется массив размера N для хранения счётчиков совпадений и маски совпадений, что добавляет ещё O(N).

**Описание способа хранения частичных решений.**

1. Частичные решения хранятся в самом Trie:
    1. Вершины Trie представляют все префиксы шаблонов. Каждая вершина кодирует состояние автомата, в котором может находиться алгоритм после обработки некоторого префикса текста.
    2. Переходы (next) по символам указывают, куда двигаться при чтении очередной буквы.
    3. Суффиксные ссылки (link): если по текущему символу нет перехода, используется ссылка к "наибольшему возможному суффиксу", чтобы продолжить обработку без возврата к началу.
    4. Поле output в вершине содержит индексы шаблонов, которые заканчиваются в этой вершине - это и есть зафиксированные частичные совпадения, которые завершились на текущем шаге.
    5. При обработке текста, накапливаются позиции всех окончаний совпадений, используя эти output.

Таким образом, частичные результаты поиска шаблонов хранятся в Trie, а завершённые - в списке result.

2. В дополнение к предыдущему заданию:

1. Шаблон с ? разбивается на подстроки без ?. Каждая из них добавляется в Trie как отдельный шаблон, но с учётом смещения в оригинальном шаблоне.
2. При обработке текста:

- Ведётся массив match[i], где match[i] хранит количество совпавших подстрок шаблона, начинающихся в позиции i.
- Если в какой-то позиции i все s подстрок (без ?) совпадают, значит, в этой позиции начинается полное совпадение шаблона.

3. Таким образом, массив match хранит частичные решения на уровне текстовой строки — сколько фрагментов шаблона совпало в данной позиции.

Trie по-прежнему хранит состояние автомата (префиксы, суффиксные связи, выходы), но дополнительно фиксируются частичные совпадения шаблона в match — это реализация «пересечения» частичных решений.

**Использованные оптимизации.**

1. Уплотнённое хранение переходов

Переходы (next) хранятся как массив фиксированной длины ALPHABET вместо хеш-таблицы или словаря. Обеспечивает постоянное время доступа и минимальные накладные расходы на память.

2. Терминальные ссылки (вторая программа)

Если по суффиксной ссылке попали в вершину без совпадений, переход совершается дальше - по цепочке терминальных ссылок, ускоряет перебор всех шаблонов, совпавших в текущем состоянии, без лишней логики.

3. Использование массива match (вторая программа)

Хранится количество совпавших подстрок шаблона на каждой позиции текста, что позволяет за один проход текста определить позиции, где совпали все подстроки, т.е. весь шаблон.

**Тестирование. Показ граничных случаев алгоритма.**

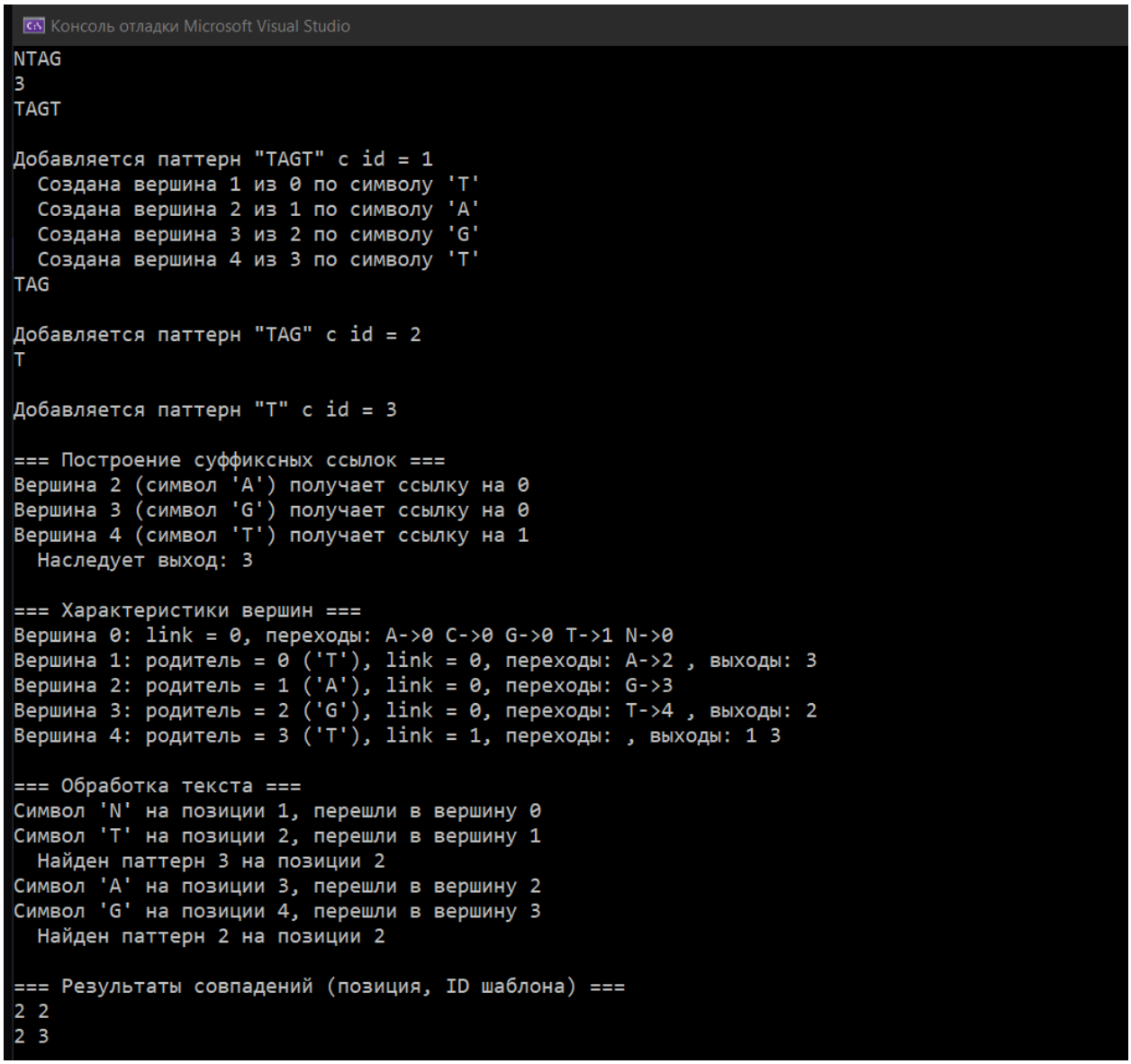


Рисунок 1 – программа 1.

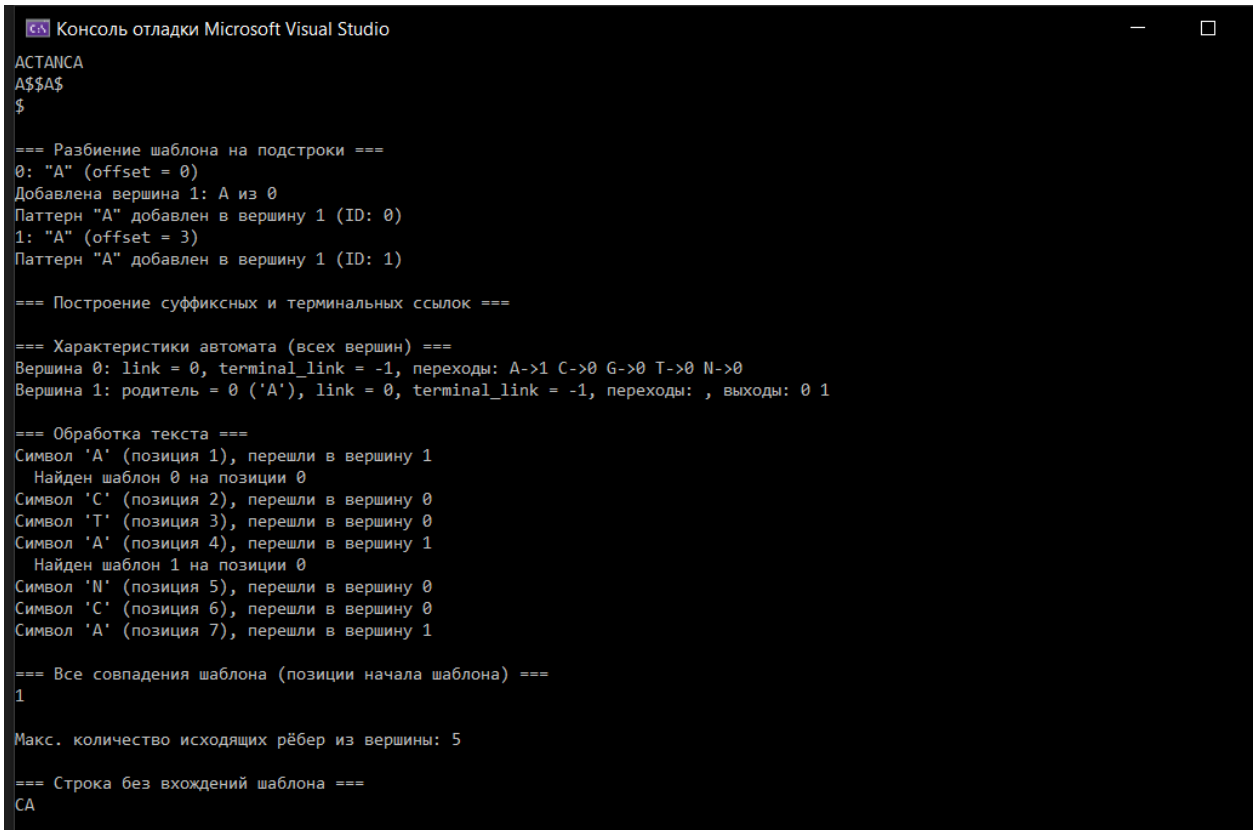


Рисунок 2 – программа 2.

## 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;
    vector<int> output;
    int parent;
    char parent_char;
    Node() {
        memset(next, -1, sizeof(next));
        link = -1;
        parent = -1;
        parent_char = 0;
    }
};

Node trie[MAXNODES];
int trie_size = 1;

void add_pattern(const string& pattern, int id) {
    cout << "\nДобавляется паттерн \"" << pattern << "\" с id =
" << id << endl;
    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;
            cout << "  Создана вершина " << trie_size << " из "
<< node
                 << " по символу '" << pattern[i] << "'\n";
            trie_size++;
        }
        node = trie[node].next[c];
    }
    trie[node].output.push_back(id);
}
```

```
void build_links() {
    cout << "\n=== Построение суффиксных 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];
                cout << "Вершина " << u << " (символ '" <<
trie[u].parent_char << "') получает ссылку на "
                    << trie[u].link << endl;

                for (int j = 0; j <
trie[trie[u].link].output.size(); j++) {
                    int out = trie[trie[u].link].output[j];
                    trie[u].output.push_back(out);
                    cout << "  Наследует выход: " << out <<
endl;
                }

                q.push(u);
            }
        }
    }

    cout << "\n=== Характеристики вершин ===\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 << ", 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 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 << "\n=== Обработка текста ===\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) {
            node = trie[node].link;
        }
        node = trie[node].next[c];

        cout << "Символ '" << T[i] << "' на позиции " << i + 1
<< ", перешли в вершину " << node << endl;

        for (int j = 0; j < trie[node].output.size(); j++) {
            int pattern_id = trie[node].output[j];
            int pos = i - pattern_lengths[pattern_id - 1] + 2;
            cout << "  Найден паттерн " << pattern_id << " на
позиции " << pos << endl;
            result.push_back({ pos, pattern_id });
        }
    }

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

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

    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];
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;

            cout << "Добавлена вершина " << trie_size << ": " <<
pattern[i] << " из " << 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 << "\n=== Построение 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 << "\n=== Характеристики автомата (всех вершин) ===\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 << ", ";
        cout << "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 id : trie[i].output) cout << id << " ";
        }
        cout << 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 << "\n=== Разбиение шаблона на подстроки ===\n";
    for (int i = 0; i < subpatterns.size(); i++) {
        cout << i << ": \"" << subpatterns[i].s << "\" (offset =
" << subpatterns[i].offset << ")\n";
        add_pattern(subpatterns[i].s, i);
    }

    build_links();

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

    cout << "\n=== Обработка 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)
            node = trie[node].link;
        node = trie[node].next[c];

        cout << "Символ '" << T[i] << "' (позиция " << i + 1 <<
"), перешли в вершину " << node << endl;

        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 << " на
позиции " << pos << endl;
                }
            }
            temp = trie[temp].terminal_link;
        }
    }
```

```
    cout << "\n=== Все совпадения шаблона (позиции начала
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 = 0;
    for (int i = 0; i < trie_size; i++) {
        int cnt = 0;
        for (int j = 0; j < ALPHABET; j++)
            if (trie[i].next[j] != -1) cnt++;
        max_out = max(max_out, cnt);
    }
    cout << "\nМакс. количество исходящих рёбер из вершины: " <<
max_out << endl;

    cout << "\n=== Строка 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;
}
```