

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

**ОТЧЕТ**  
**по лабораторной работе №6**  
**по дисциплине «Построение и Анализ Алгоритмов»**  
**Тема: Ахо-Корасик**

Студент гр. 3384

Рудаков А.Л.

Преподаватель

Шевелева А.М.

Санкт-Петербург

2025

### **Цель работы.**

Написание алгоритма Ахо-Корасика на языке программирования.

### **Задание.**

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

Вход:

Первая строка содержит текст ( $T, 1 \leq |T| \leq 10000$ ).

Вторая - число  $n$  ( $1 \leq n \leq 3000$ ), каждая следующая из  $n$  строк содержит шаблон из набора  $P = \{p_1, \dots, p_n\}$   $1 \leq |p| \leq 75$

Все строки содержат символы из алфавита  $\{A, C, G, T, N\}$

Выход:

Все вхождения образцов из  $P$  в  $T$ .

Каждое вхождение образца в текст представить в виде двух чисел -  $i$   $p$

Где  $i$  - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером  $p$  (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

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  $ab??c?$  с джокером  $?$  встречается дважды в тексте  $xabvccbababcsax$ .

Символ джокер не входит в алфавит, символы которого используются в  $T$ . Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида  $???$  недопустимы.

Все строки содержат символы из алфавита  $\{A,C,G,T,N\}$

Вход:

Текст ( $T, 1 \leq |T| \leq 1000000$ )

Шаблон ( $P, 1 \leq |P| \leq 40$ )

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A\$\$\$A\$

\$

Sample Output:

1

### **Выполнение работы.**

Для задания 1 разработан алгоритм, находящийся в файле `first.cpp`.

В нем реализованы структура вершины бора и структур бора, а также функции для ввода и вывода данных.

Структура *Vertex* содержит в себе информацию о вершине бора. Внутри хранится информация о номерах детей от вершины, о номере родителя, о символе к вершине, о том, является ли вершина концом паттерна, о номере вершины, о номере, куда ведет суффиксная ссылка, о номере и длине паттерна (если вершина конец паттерна).

В структуре *Bohr* есть методы для добавления данных, для добавления суффиксных ссылок, а так же для поиска следующего символа.

Метода `void add_pattern(string pattern, int pattern_idx)` реализует добавления вершин в бор исходя из поданного паттерна. Внутри происходит итерация по символам в паттерне с запоминанием текущего индекса вставки. На каждой итерации происходит проход по всем детям от текущей вершины, если вершина с таким же символом была найдена, то индекс текущей вершины меняется на нее, если же не было найдено такой вершины, то добавляется новая вершина, которая записывается к детям от текущей. Индекс текущей меняется на индекс новой. После итерации по всем символам паттерна, в вершину с текущим индексом устанавливается флаг что она является концом паттерна, а так же записываются значения номера и длины паттерна.

Метод `void add_suff_links()` добавляет суффиксные ссылки всем вершинам бора. Внутри создается очередь, для того, чтобы обрабатывать вершины в правильном порядке (обход в ширину). Далее происходит итерация по очереди, пока очередь не пуста. Из очереди достается первая вершина. Если ее родитель начальная вершина, то суффиксная ссылка будет указывать на нее. Если нет, то суффиксной ссылке присваивается значение, возвращаемое методом `get_suff_link(..)` от значения суффиксной ссылки родителя и символа вершины. После этого в очередь добавляются все дети текущей вершины.

Метод `int get_suff_link(int current_idx, char symb)` возвращает индекс вершины, на которую должна указывать суффиксная ссылка от поданных значений. Внутри происходит итерация по детям поданной вершины, если символ ребенка совпадает с подданным символом, то возвращается индекс данного ребенка. Если текущий индекс вершины равен 0, то возвращается 0.

Если суффиксная ссылка не была найдена, то метод вызывается рекурсивно, до момента обнаружения нужного значения.

Метод `vector<pair<int, int>> find_indexes(string text)` возвращает вектор пар значений найденных в тексте индекс начала паттерна и индекс паттерна. Внутри происходит итерация по тексту, при помощи метода `find_next_idx(..)` находится вершина, содержащая последнюю рассматриваемую часть строки текста, ее индекс запоминается, после чего происходит проход по суффиксным ссылкам от вершин, являющихся суффиксными ссылками текущей вершины до момента, пока проверяемый индекс не равен 0. Если при данном проходе находится вершина, являющаяся концом паттерна, то в возвращаемый вектор записывается пара с индексом паттерна в тексте и его номером.

Метод `int find_next_idx(int current_idx, char symb)` рекурсивно ищет вершину, которая совпадает с подстрокой текста по последнему символу.

Функция `void input(string& text, vector<string>& patterns)` реализует ввод значений из терминала.

Функция `void add_data(Bohr& bohr, vector<string>& patterns)` добавляет вершины в бор а также вызывает метод для добавления суффиксных ссылок.

Функция `void output(vector<pair<int, int>> result)` выводит результат в терминал.

Тестирование приведено в табл. Б1.

Для задания 2 разработан алгоритм, находящийся в файле `second.cpp`.

В нем доработаны структуры бора и вершины бора, а также немного изменены функции ввода, вывода и заполнения бора.

В структуре `Vertex` добавлено поле `vector<int> idx_in_text`, хранящее в себе индексы входа текущего паттерна в общий паттерн. Например если есть паттерн `aa&&f&aa`, то `Vertex`, отвечающий за цепочку `a->a` будет иметь в `idx_in_text` значения 1 и 7.

В структуре *Bohr* в методе *void add\_pattern(string pattern, int idx\_in\_text, int pattern\_idx)* к последней текущей вершине теперь добавляется индекс паттерна в основном паттерне.

Методы *void add\_suff\_links()*, *int get\_suff\_link(int current\_idx, char symb)*, *int find\_next\_idx(int current\_idx, char symb)* остались неизменными. Метод *vector<int> find\_indexes(string text)* теперь возвращает вектор с количеством найденных входов основного паттерна в текст. Внутри происходит такая же итерация по тексту, как в 1 задании, но теперь при проверке вершин, если она является концом паттерна, то происходит проход по индексам вхождения текущего паттерна в основной паттерн, в котором к индексам начала основного паттерна, рассчитанного по формуле текущий индекс – (длина подпаттерна – 1) – индекс паттерна в основном паттерне, добавляется 1. В конце возвращается вектор со значениями в каждом из индексов в тексте.

Функция *int input(string& text, vector<pair<string, int>>& patterns)* теперь не только считывает значения, но и разделяет основной паттерн на подпаттерны.

Функция *void add\_data(Bohr& bohr, vector<pair<string, int>>& patterns)* так же добавляет данные в бор.

Функция *vector<int> preprocessing(vector<int> pre\_result, int pattern\_count, int pattern\_size)* редактирует найденный вектор значений, извлекая из него индексы вхождений основного паттерна в текст. Это происходит путем сравнения количества подпаттернов со значением в индексе поданного вектора, если значения одинаковые, то значит начиная с этого индекса имеется вхождение основного паттерна в текст.

Метод *void output(vector<int> result)* выводит результат в терминал.

Тестирование приведено в табл. Б2.

### **Дополнительное задание.**

Реализовать алгоритм из задания с джокером, но джокер обозначает любой символ + w + любой символ.

### **Выполнение дополнительного задания.**

Для дополнительного задания написан алгоритм, находящийся в файле `add_task.cpp`.

Алгоритм выполняет те же действия, что и в задании 2.

Структуры Бора и вершин бора остались такими же.

Добавлена функция `string transform_input(string pattern, char mask)`, которая преобразует подданный паттерн вместе с символами маски, заменяя каждый символ маски на символ маски + `w` + символ маски. Например `??ab?c` на выходе станет `?w??w?ab?w?c`.

Эта функция вызывается в функции `int input(string& text, vector<pair<string, int>>& patterns)` после ввода паттерна. Далее все идет по сценарию из задания 2: паттерн разделяется на подпаттерны, которые добавляются в бор, ищется вектор вхождений и выводятся индексы входа.

Тестирование приведено в табл. БЗ.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

### **Выводы.**

В ходе работы был реализован алгоритм Ахо-Корасика, созданный для поиска решения задачи точного поиска большого количества заданных паттернов в поданном тексте, работающий на логике построения структуры данных – бор, которая удобно хранит в себе информацию о символах алфавита, которые входят в паттерны, благодаря чему после построения бора и суффиксных ссылок внутри него поиск паттерна в нем становится линейным. Пусть  $n$  – длина текста,  $m$  – общая длина паттернов, тогда сложность алгоритма  $O(m+n)$

Кроме этого был реализован алгоритм для поиска заданного паттерна, включающего в себя символы джокеры, которые обозначают любой символ из алфавита или вне его, который нужно найти в поданном тексте. Написанный

алгоритм построен на основе алгоритма Ахо-Корасика. Он ищет вхождения подпаттернов – целостных частей поданного паттерна, не включающих в себя символов джокеров, увеличивает значения вектора, считающего количество возможных вхождений основного паттерна в текст, после чего сравнивает значения в данном векторе с количеством целостных подпаттернов в основном поданном паттерне, исходя из чего делает вывод, есть ли по данному индексу вхождение основного паттерна или нет.

Алгоритм решения дополнительного задания также основан на алгоритме Ахо-Корасика. Он повторяет алгоритм из задания 2, но перед этим обрабатывает поданный на вход паттерн, изменяя каждый символ джокер на заданную в задании последовательность символов. Пусть  $n$  – длина текста,  $m$  – общая длина паттернов, тогда сложность алгоритма  $O(m+n+n)$

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: first.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

struct Vertex {
    vector <int> childs = {};
    int parent;
    bool is_pattern;
    int suff_link;
    char symb;
    int idx;
    int pattern_idx = -1;
    int pattern_len = -1;

    Vertex(int parent, bool is_pattern, int suff_link, char symb,
int idx) :
        parent(parent), is_pattern(is_pattern),
suff_link(suff_link), symb(symb), idx(idx) {};
};

struct Bohr {
    vector <Vertex*> vertexes;

    Bohr() {
        vertexes.push_back(new Vertex({ 0, true, 0, '\0', 0 }));
    }

    void add_pattern(string pattern, int pattern_idx) {
        int current_idx = 0;
```

```

    for (char symb : pattern) {
        bool is_find = false;
        for (int idx : vertexes[current_idx]->childs) {
            if (vertexes[idx]->symb == symb) {
                current_idx = vertexes[idx]->idx;
                is_find = true;
                break;
            }
        }
        if (!is_find) {
            Vertex* new_vertex = new Vertex( current_idx,
false, -1, symb, vertexes.size());
            vertexes.push_back(new_vertex);
            vertexes[current_idx]-
>childs.push_back(new_vertex->idx);
            current_idx = new_vertex->idx;
        }
    }
    vertexes[current_idx]->is_pattern = true;
    vertexes[current_idx]->pattern_idx = pattern_idx;
    vertexes[current_idx]->pattern_len = pattern.size();
}

void add_suff_links() {
    queue<int> vertex_queue;
    vertex_queue.push(0);

    while (!vertex_queue.empty()) {
        int current_idx = vertex_queue.front();
        vertex_queue.pop();

        if (vertexes[current_idx]->parent == 0) {
            vertexes[current_idx]->suff_link = 0;
        }
        else {
            int parent_suff =
vertexes[vertexes[current_idx]->parent]->suff_link;
            vertexes[current_idx]->suff_link =
get_suff_link(parent_suff, vertexes[current_idx]->symb);

```

```

    }

    for (int child : vertexes[current_idx]->childs) {
        vertex_queue.push(child);
    }
}

int get_suff_link(int current_idx, char symb) {
    for (int child : vertexes[current_idx]->childs) {
        if (vertexes[child]->symb == symb) {
            return child;
        }
    }

    if (current_idx == 0) return 0;

    return    get_suff_link(vertexes[current_idx]->suff_link,
symb);
}

int find_next_idx(int current_idx, char symb) {
    for (int child : vertexes[current_idx]->childs) {
        if (vertexes[child]->symb == symb) {
            return child;
        }
    }

    if (current_idx == 0) return 0;

    return    find_next_idx(vertexes[current_idx]->suff_link,
symb);
}

vector<pair<int, int>> find_indexes(string text) {
    vector<pair<int, int>> result;
    int current_idx = 0;

    for (int i = 0; i < text.size(); i++) {
        current_idx = find_next_idx(current_idx, text[i]);
    }
}

```

```

        int check_idx = current_idx;
        while (check_idx != 0) {
            if (vertexes[check_idx]->is_pattern) {
                result.push_back(make_pair(i -
vertexes[check_idx]->pattern_len + 2, vertexes[check_idx]->pattern_idx));
            }
            check_idx = vertexes[check_idx]->suff_link;
        }
    }

    return result;
}

};

void input(string& text, vector<string>& patterns) {
    cin >> text;
    int n;
    cin >> n;
    string current_pattern;
    for (int i = 0; i < n; i++) {
        cin >> current_pattern;
        patterns.push_back(current_pattern);
    }
}

void add_data(Bohr& bohr, vector<string>& patterns) {
    for (int i = 0; i < patterns.size(); i++) {
        bohr.add_pattern(patterns[i], i + 1);
    }
    bohr.add_suff_links();
}

void output(vector<pair<int, int>> result) {
    sort(result.begin(), result.end());
    for (auto out_res: result) {
        cout << out_res.first << " " << out_res.second << "\n";
    }
}

```

```

int main() {
    Bohr bohr;
    string text;
    vector<string> patterns;

    input(text, patterns);
    add_data(bohr, patterns);

    vector<pair<int, int>> result = bohr.find_indexes(text);
    output(result);

    return 0;
}

```

### Название файла: second.cpp

```

#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

struct Vertex {
    vector <int> childs = {};
    int parent;
    bool is_pattern;
    int suff_link;
    char symb;
    int idx;
    int pattern_idx = -1;
    int pattern_len = -1;
    vector <int> idx_in_text;

    Vertex(int parent, bool is_pattern, int suff_link, char symb,
int idx) :
        parent(parent), is_pattern(is_pattern),
suff_link(suff_link), symb(symb), idx(idx) {};

```

```

};

struct Bohr {
    vector <Vertex*> vertexes;

    Bohr() {
        vertexes.push_back(new Vertex({ 0, true, 0, '\0', 0 }));
    }

    void add_pattern(string pattern, int idx_in_text, int
pattern_idx) {
        int current_idx = 0;
        for (char symb : pattern) {
            bool is_find = false;
            for (int idx : vertexes[current_idx]->childs) {
                if (vertexes[idx]->symb == symb) {
                    current_idx = vertexes[idx]->idx;
                    is_find = true;
                    break;
                }
            }
            if (!is_find) {
                Vertex* new_vertex = new Vertex(current_idx,
false, -1, symb, vertexes.size());
                vertexes.push_back(new_vertex);
                vertexes[current_idx]-
>childs.push_back(new_vertex->idx);
                current_idx = new_vertex->idx;
            }
        }
        vertexes[current_idx]->is_pattern = true;
        vertexes[current_idx]->pattern_idx = pattern_idx;
        vertexes[current_idx]->pattern_len = pattern.size();
        vertexes[current_idx]-
>idx_in_text.push_back(idx_in_text);
    }

    void add_suff_links() {
        queue<int> vertex_queue;

```

```

vertex_queue.push(0);

while (!vertex_queue.empty()) {
    int current_idx = vertex_queue.front();
    vertex_queue.pop();

    if (vertexes[current_idx]->parent == 0) {
        vertexes[current_idx]->suff_link = 0;
    }
    else {
        int parent_suff =
vertexes[vertexes[current_idx]->parent]->suff_link;
        vertexes[current_idx]->suff_link =
get_suff_link(parent_suff, vertexes[current_idx]->symb);
    }

    for (int child : vertexes[current_idx]->childs) {
        vertex_queue.push(child);
    }
}

int get_suff_link(int current_idx, char symb) {
    for (int child : vertexes[current_idx]->childs) {
        if (vertexes[child]->symb == symb) {
            return child;
        }
    }

    if (current_idx == 0) return 0;

    return get_suff_link(vertexes[current_idx]->suff_link,
symb);
}

int find_next_idx(int current_idx, char symb) {
    for (int child : vertexes[current_idx]->childs) {
        if (vertexes[child]->symb == symb) {
            return child;

```

```

        }
    }
    if (current_idx == 0) return 0;
    return find_next_idx(vertexes[current_idx]->suff_link,
symb);
}

```

```

vector<int> find_indexes(string text) {
    int current_idx = 0;
    vector<int> count_patterns(text.size(), 0);

    for (int i = 0; i < text.size(); i++) {
        current_idx = find_next_idx(current_idx, text[i]);

        int check_idx = current_idx;
        while (check_idx != 0) {
            if (vertexes[check_idx]->is_pattern) {
                for (int j = 0; j < vertexes[check_idx]-
>idx_in_text.size(); j++) {
                    int add_idx = i -
(vertexes[check_idx]->pattern_len - 1) - vertexes[check_idx]-
>idx_in_text[j];

                    if (add_idx >= 0) {
                        count_patterns[add_idx] += 1;
                    }
                }
            }
            check_idx = vertexes[check_idx]->suff_link;
        }
    }

    return count_patterns;
}
};

```

```

int input(string& text, vector<pair<string, int>>& patterns) {
    cin >> text;
    string pattern;

```

```

    cin >> pattern;
    char mask;
    cin >> mask;
    string current_pattern = "";
    for (int i = 0; i < pattern.size(); i++) {
        if (pattern[i] == mask) {
            if (current_pattern != "") {
                patterns.push_back(make_pair(current_pattern,
i - current_pattern.size()));
            }
            current_pattern = "";
        }
        else {
            current_pattern += pattern[i];
        }
    }
    if (current_pattern != "") {
        patterns.push_back(make_pair(current_pattern,
pattern.size() - current_pattern.size()));
    }
    return pattern.size();
}

void add_data(Bohr& bohr, vector<pair<string, int>>& patterns) {
    for (int i = 0; i < patterns.size(); i++) {
        bohr.add_pattern(patterns[i].first, patterns[i].second,
i + 1);
    }
    bohr.add_suff_links();
}

vector<int> preprocessing(vector<int> pre_result, int pattern_count,
int pattern_size) {
    vector<int> result;
    for (int i = 0; i < pre_result.size(); i++) {
        if (pre_result[i] == pattern_count && i + pattern_size
<= pre_result.size()) {
            result.push_back(i + 1);
        }
    }
}

```

```

    }
    return result;
}

void output(vector<int> result) {
    for (int out: result) {
        cout << out << '\n';
    }
}

int main() {
    Bohr bohr;
    string text;
    vector<pair<string, int>> patterns;
    int pattern_size = input(text, patterns);

    add_data(bohr, patterns);

    vector<int> pre_result = bohr.find_indexes(text);
    vector<int> result = preprocessing(pre_result, patterns.size(),
pattern_size);

    output(result);

    return 0;
}

```

**Название файла: add\_task.cpp**

```

#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

struct Vertex {
    vector<int> childs = {};

```

```

    int parent;
    bool is_pattern;
    int suff_link;
    char symb;
    int idx;
    int pattern_idx = -1;
    int pattern_len = -1;
    vector<int> idx_in_text;

    Vertex(int parent, bool is_pattern, int suff_link, char symb,
int idx) :
        parent(parent), is_pattern(is_pattern),
suff_link(suff_link), symb(symb), idx(idx) {};
};

struct Bohr {
    vector<Vertex*> vertexes;

    Bohr() {
        vertexes.push_back(new Vertex({ 0, true, 0, '\\0', 0 }));
    }

    void add_pattern(string pattern, int idx_in_text, int
pattern_idx) {
        int current_idx = 0;
        for (char symb : pattern) {
            bool is_find = false;
            for (int idx : vertexes[current_idx]->childs) {
                if (vertexes[idx]->symb == symb) {
                    current_idx = vertexes[idx]->idx;
                    is_find = true;
                    break;
                }
            }
            if (!is_find) {
                Vertex* new_vertex = new Vertex(current_idx,
false, -1, symb, vertexes.size());
                vertexes.push_back(new_vertex);
            }
        }
    }
};

```

```

        vertexes[current_idx]-
>childs.push_back(new_vertex->idx);
        current_idx = new_vertex->idx;
    }
}
vertexes[current_idx]->is_pattern = true;
vertexes[current_idx]->pattern_idx = pattern_idx;
vertexes[current_idx]->pattern_len = pattern.size();
vertexes[current_idx]-
>idx_in_text.push_back(idx_in_text);
}

void add_suff_links() {
    queue<int> vertex_queue;
    vertex_queue.push(0);

    while (!vertex_queue.empty()) {
        int current_idx = vertex_queue.front();
        vertex_queue.pop();

        if (vertexes[current_idx]->parent == 0) {
            vertexes[current_idx]->suff_link = 0;
        }
        else {
            int parent_suff =
vertexes[vertexes[current_idx]->parent]->suff_link;
            vertexes[current_idx]->suff_link =
get_suff_link(parent_suff, vertexes[current_idx]->symb);
        }

        for (int child : vertexes[current_idx]->childs) {
            vertex_queue.push(child);
        }
    }
}

int get_suff_link(int current_idx, char symb) {
    for (int child : vertexes[current_idx]->childs) {
        if (vertexes[child]->symb == symb) {

```

```

        return child;
    }
}

if (current_idx == 0) return 0;

return    get_suff_link(vertexes[current_idx]->suff_link,
symb);
}

int find_next_idx(int current_idx, char symb) {
    for (int child : vertexes[current_idx]->childs) {
        if (vertexes[child]->symb == symb) {
            return child;
        }
    }
    if (current_idx == 0) return 0;
    return    find_next_idx(vertexes[current_idx]->suff_link,
symb);
}

vector<int> find_indexes(string text) {
    int current_idx = 0;
    vector<int> count_patterns(text.size(), 0);

    for (int i = 0; i < text.size(); i++) {
        current_idx = find_next_idx(current_idx, text[i]);

        int check_idx = current_idx;
        while (check_idx != 0) {
            if (vertexes[check_idx]->is_pattern) {
                for (int j = 0; j < vertexes[check_idx]-
>idx_in_text.size(); j++) {
                    int    add_idx    =    i    -
(vertexes[check_idx]->pattern_len    -    1)    -    vertexes[check_idx]-
>idx_in_text[j];

                    if (add_idx >= 0) {
                        count_patterns[add_idx] += 1;
                    }
                }
            }
            check_idx = vertexes[check_idx]->suff_link;
        }
    }
}

```

```

        }
    }
    check_idx = vertexes[check_idx]->suff_link;
}

return count_patterns;
}

};

string transform_input(string pattern, char mask) {
    string new_pattern = "";
    for (char symb : pattern) {
        if (symb == mask) {
            new_pattern = new_pattern + mask + 'w' + mask;
        }
        else {
            new_pattern += symb;
        }
    }
    return new_pattern;
}

int input(string& text, vector<pair<string, int>>& patterns) {
    cin >> text;
    string pattern;
    cin >> pattern;
    char mask;
    cin >> mask;
    pattern = transform_input(pattern, mask);
    string current_pattern = "";
    for (int i = 0; i < pattern.size(); i++) {
        if (pattern[i] == mask) {
            if (current_pattern != "") {
                patterns.push_back(make_pair(current_pattern,
i - current_pattern.size()));
            }
            current_pattern = "";
        }
    }
}

```

```

        else {
            current_pattern += pattern[i];
        }
    }
    if (current_pattern != "") {
        patterns.push_back(make_pair(current_pattern,
pattern.size() - current_pattern.size()));
    }
    return pattern.size();
}

void add_data(Bohr& bohr, vector<pair<string, int>>& patterns) {
    for (int i = 0; i < patterns.size(); i++) {
        bohr.add_pattern(patterns[i].first, patterns[i].second,
i + 1);
    }
    bohr.add_suff_links();
}

vector<int> preprocessing(vector<int> pre_result, int pattern_count,
int pattern_size) {
    vector<int> result;
    for (int i = 0; i < pre_result.size(); i++) {
        if (pre_result[i] == pattern_count && i + pattern_size
<= pre_result.size()) {
            result.push_back(i + 1);
        }
    }
    return result;
}

void output(vector<int> result) {
    for (int out : result) {
        cout << out << '\n';
    }
}

int main() {
    Bohr bohr;

```

```
    string text;
    vector<pair<string, int>> patterns;
    int pattern_size = input(text, patterns);

    add_data(bohr, patterns);

    vector<int> pre_result = bohr.find_indexes(text);
    vector<int> result = preprocessing(pre_result, patterns.size(),
pattern_size);

    output(result);

    return 0;
}
```

## ПРИЛОЖЕНИЕ Б

### ТЕСТИРОВАНИЕ

Тесты для первого задания представлены в табл. Б1.

Таблица Б.1 - Примеры тестовых случаев задания 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	NTAG 3 TAGT TAG T	2 2 2 3	Ok
2.	AAAA 4 A AA AAA AAAA	1 1 1 2 1 3 1 4 2 1 2 2 2 3 3 1 3 2 4 1	Ok
3.	abaraba 3 aba rab bar	1 1 2 3 4 2 5 1	Ok
4.	asdf 2 a f	1 1 4 2	Ok

Тесты для второго задания представлены в табл. Б2.

Таблица Б.2 - Примеры тестовых случаев задания 2.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ACTANCA A\$\$\$ \$	1	Ok
2.	aaa a% % %	1	Ok
3.	aaa %a% %	1	Ok
4.	aaa %%a %	1	Ok
5.	abcdabcdabcdabc *c**b* *	2 6 10	Ok
6.	abcdabcdabcdabc **da*cd *	2 6	Ok

Тесты для дополнительного задания представлены в табл. Б3.

Таблица Б.3 - Примеры тестовых случаев дополнительного задания.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	awcdswq &d& &	1	Ok
2.	wwwwwwwwwwwwww ?w?? ?	1 2 3 4	Ok
3.	qjd1we3ujdw2id d?3u ?	3	Ok
4.	uuuuwuuuu ?uu ?	4	Ok
5.	www ?w ?	1	Ok