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

Студентка гр. 9383 _____ Чебесова И. Д.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с алгоритмом Ахо-Корасик, реализовать алгоритм на одном из языков программирования.

Вариант 0. Работа без варианта. Решение двух задач на платформе степик.

Задание.

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

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

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

Задание 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?ab??c? с джокером ?? встречается дважды в тексте xabvccbababcaxxabvccbababcax.

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, $1 \le |T| \le 100000$)

Шаблон (P, $1 \le |P| \le 40$)

Символ джокера

Выход:

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

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Основные теоретические положения.

Чтобы говорить об алгоритме необходимо ввести ряд понятий:

Бор — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

Бор для набора образцов {he,she,his,hers}:

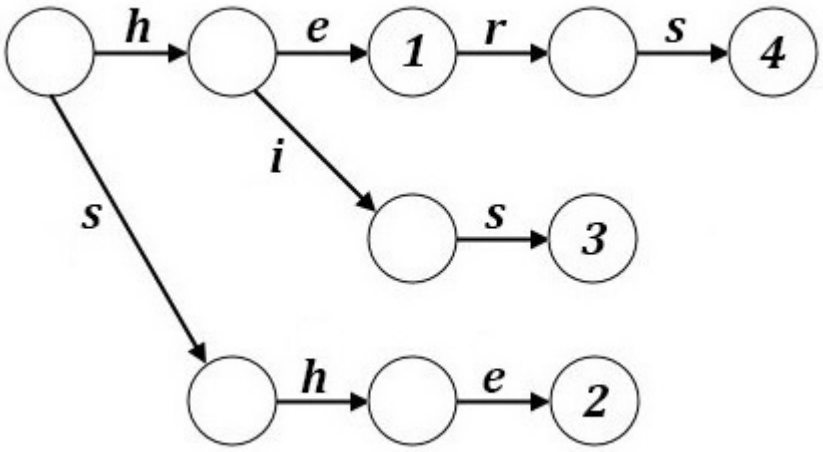

Рисунок 1 – пример бора для набора образцов

Корень обозначается пустым символом (в моей реализации — это специальный символ, который не входит в алфавит - @).

Видно, что некоторые вершины имеют число – это обозначает, что вершина является *терминальной* (конечной) для определенного образца.

Назовем *суффиксной ссылкой* вершины *v* указатель на вершину *u*, такую что строка *u* — наибольший собственный суффикс строки *v*, или, если

такой вершины нет в боре, то указатель на корень. В частности, ссылка из корня ведет в него же.

Например, для бора:

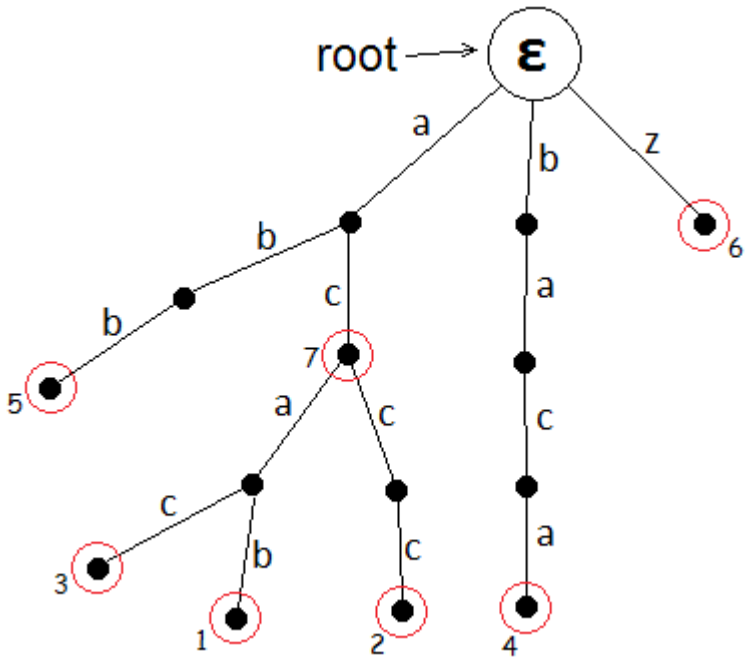

Рисунок 2 – пример бора

Суффиксные ссылки будут выглядеть следующим образом:

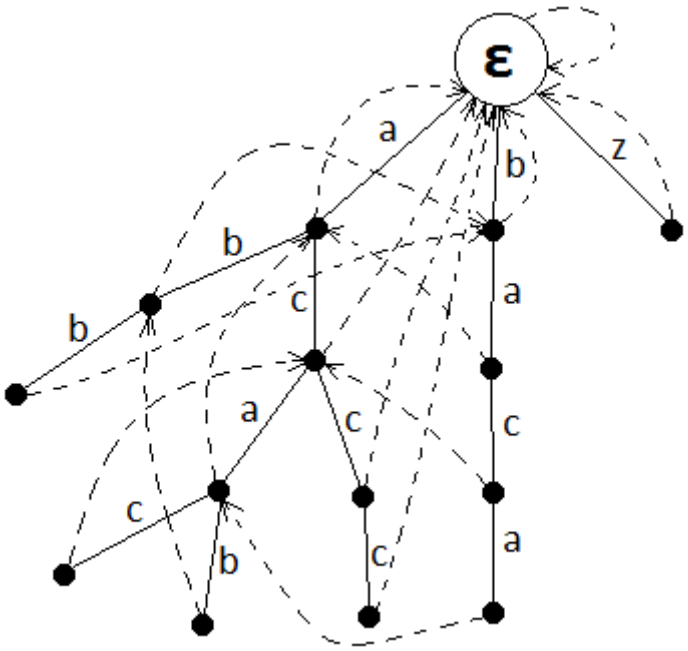

Рисунок 3 – пример суффиксных ссылок в боре

Наша задача — построить *конечный детерминированный автомат*. Состояние автомата — это какая-то вершина бора. Переход из состояний

осуществляется по двум параметрам — текущей вершине *v* и символу *ch*. по которому нам надо сдвинуться из этой вершины. Необходимо найти вершину *u*, которая обозначает наидлиннейшую строку, состоящую из суффикса строки *v* (возможно нулевого) + символа *ch*. Если такого в боре нет, то идем в корень.

Описание алгоритма.

Алгоритм состоит из трех этапов: построение бора, добавление суффиксных ссылок и самого прохода с помощью автомата.

1. Построение бора.

Бор строится из подстрок, переданных для поиска. Рассматривая каждый образец последовательно добавляем вершины в бор.

- если по текущему символу подстроки можно перейти в следующую вершину – переходим
- если ребра с текущим символом нет – создаем ребро, добавляя вершину и переходим в нее

2. Преобразование бора, добавлением суффиксных ссылок для каждой вершины.

При построении следует помнить:

- ссылка из корня идет в корень

- ссылка из потомка корня идет в корень

- ссылка из любой другой вершины с ребром e ищется следующим образом:

- смотрим на родителя
- пытаемся пройти по ребру e
- если не получается поднимаемся выше и повторяем
- если уперлись в корень и так и не нашли ребро – что ж, ссылка пойдет в корень

- иначе запоминаем ссылку на ту вершину, в которую попали, переходя по ребрам

Также еще одним этапом является построение конечной ссылки. Для этого:

- ходим по ссылкам, пока не упремся в терминальную – конечную вершину
- нашли – отлично, запоминаем
- не нашли и уперлись в корень – что ж, ссылка пуста

3. Сам алгоритм.

- пытаемся перейти в автомате по ребру с текущим рассматриваемым символом
- если такое ребро есть – переходим
- если такого ребра нет – переходим по суффиксной ссылке (если были в корне, то перейдем в него же) и повторяем с первой точки
- если попали в терминальную вершину – то ура, нашли вхождение – запоминаем
- проходим цепочку конечных ссылок, запоминая результаты

Для решения задания с джокером используется тот же алгоритм с некоторыми изменениями:

- выделяем максимальные подстроки, не содержащие джокера из заданного шаблона
- для каждого такого образца запоминаем смещение
- внутри алгоритма создаем дополнительный массив, заполненный 0, длинна которого равна длине текста
- запускаем поиск по тексту
- если нашли подстроку, то увеличиваем на 1 значение ячейки, которое соответствует разности между номером начального символа образца в тексте и смещением образца (если их несколько – то проделать со всеми)

Таким образом получится, что шаблонная подстрока будет начинаться в тех местах текста, для которых соответствующая ячейка массива содержит количество образцов с учетом кратности.

Сложность алгоритма Ахо-Корасика: O(M*k+N+t), где M – размер бора, k – размер алфавита, t - количество всех возможных вхождений всех строк-образцов, а N – длина исходной строки.

Описание функций и структур данных.

struct Bor – для хранения бора и работы с ним.

Его поля:

char edge; - хранит ребро для перехода в следующую вершину (вес ребра – символ char)

int final_vertex_number; - хранит номер образца соответствующий конечной (терминальной вершине)

std::vector<Bor> children;* - хранит ссылки на детей вершины

Bor parent;* - хранит ссылку на родителя

Bor suffix_link;* - хранит суффиксную ссылку

Bor end_suffix_link;* - хранит конечную ссылку

Методы:

Bor go_to_child_by_edge (char child_edge)* – для перехода в потопка по заданному ребру

void add_suffix_link () – для добавления суффиксной ссылки в вершину

struct DFA – структура, которая предоставляет возможность работать с автоматом и содержит сам алгоритм.

Поля:

Bor root;* - для хранения корня

std::vector<std::string> patterns; - для хранения массива шаблонов

Методы:

void bild_bor (std::vector<std::string>& new_patterns) – функция для построения бора по переданному массиву шаблонов

void add_all_suffix_links () – для добавления всех суффиксных ссылок

void aho_corasick (std::string& text) - сам алгоритм

void print_result (std::vector<std::pair<int, int>>& result) – для печати результата на экран

Дополнительные функции:

bool cmp_for_vec_pair(std::pair<int, int> elem1, std::pair<int, int> elem2) – для сортировки вектора пар по возрастанию обоих параметров

Тестирование.

Задание 1.

Входные данные:

NTAG

3

TAGT

TAG

T

Выходные данные:

```
Введите текст T:
Введите количество подстрок:
Введите все подстроки:

----------Начинаем построение бора----------
Строка TAGT добавлена в бор
Строка TAG добавлена в бор
Строка T добавлена в бор
----------Построение бора закончено----------

----------Начинаем создание суффиксных ссылок----------
Текущая вершина - корень, ссылается на себя.
Текущая вершина с ребром T потомок корня, ссылается на корень.
Текущая вершина с ребром A далека от корня, начато построение ссылки. Построение ссылки завершено.
Текущая вершина с ребром G далека от корня, начато построение ссылки. Построение ссылки завершено.
Текущая вершина с ребром T далека от корня, начато построение ссылки. Построение ссылки завершено.
----------Создание суффиксных ссылок закончено----------

----------Начинаем алгоритм Ахо-Корасика----------
Найдено вхождение образца: T
Найдено вхождение образца: TAG
----------Алгоритм Ахо-Корасика закончен----------

----------Результат работы алгоритма----------
Позиция в тексте: 2 Номер образца: 2
Позиция в тексте: 2 Номер образца: 3
```

Рисунок 4 – тестирование программы 1

Входные данные:

ABCDEF

5

A

B

CD

CDEF

DC

Выходные данные:

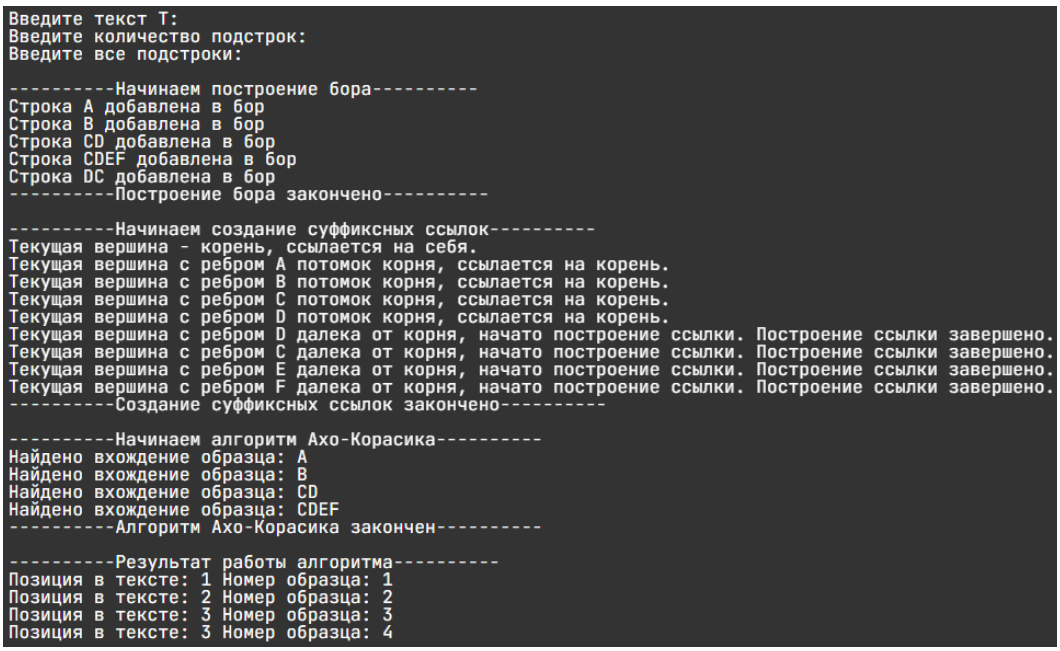

Рисунок 5 – тестирование программы 2

Задание 2.

Входные данные:

ACTANCA

A$$A$

$

Выходные данные:

Введите текст T:

Введите шаблон:

Введите символ джокера:

Результат работы алгоритма:

1

Входные данные:

ACTANCAACTAN

A$$A$

$

Выходные данные:

Введите текст T:

Введите шаблон:

Введите символ джокера:

Результат работы алгоритма:

1

4

8

Выводы.

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Ахо-Корасик, который находит все вхождения подстрок шаблонов в заданный текст. Также был реализован поиск подстроки с джокером с использованием данного алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл task1.cpp:

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


//компаратор для сортировки контейнера std::vector<std::pair<int, int>> по возрастанию обоих элементов пары
bool cmp_for_vec_pair(std::pair<int, int> elem1, std::pair<int, int> elem2)
{
    if (elem1.first < elem2.first)
    {
        return true;
    }
    else if (elem1.first == elem2.first && elem1.second < elem2.second)
    {
        return true;
    }
    else
    {
        return false;
    }
}


//структура для хранени и работы с бором, в частности для добавления суффиксной ссылки для текущей вершины
struct Bor
{
    char edge;
    int final_vertex_number;
    std::vector<Bor*> children;
```

```
        Bor* parent;
        Bor* suffix_link;
        Bor* end_suffix_link;

        Bor (char new_edge, Bor* new_parent)
        {
            edge = new_edge;
            final_vertex_number = 0;
            parent = new_parent;
            suffix_link = nullptr;
            end_suffix_link = nullptr;
        }

        //функция для спуска по бору по ребру к потомку
        Bor* go_to_child_by_edge (char child_edge)
        {
            for (auto &child : children)
            {
                if (child->edge == child_edge)
                {
                    return child;
                }
            }
            return nullptr;
        }

        //функция создания ссылки для одной вершины
        void add_suffix_link ()
        {
            if (edge == '@') //если это корень, то ссылка в себя
            {
                std::cout << "Текущая вершина - корень, ссылается на себя.\
n";
                suffix_link = this;
            }
            else if (parent->edge == '@') //если предок корень, то ссылка в
предка, т.е. в корень
```

```
            {
                std::cout << "Текущая вершина с ребром " << edge << "
потомок корня, ссылается на корень.\n";
                suffix_link = parent;
            }
            else //если же корень не близко - нужно смотреть
            {
                std::cout << "Текущая вершина с ребром " << edge << " далека
от корня, начато построение ссылки. ";
                Bor* current_state = parent->suffix_link; //суффиксная
ссылка родителя
                Bor* next_state = current_state-
>go_to_child_by_edge(edge); //пытаемся от родителя пройти по текущему символу
вниз
                while (next_state == nullptr) //если пройти не получилось -
нет такого ребра
                {
                    if (current_state->edge == '@') //если уперлись в корень
                    {
                        next_state = current_state; //то сохраняем для
ссылки корень и выходим
                        break;
                    }   //если нет
                    current_state = current_state->suffix_link; //проходим
по ссылке дальше
                    next_state = current_state-
>go_to_child_by_edge(edge); //пытаемся снова спуститься
                }
                suffix_link = next_state;  //сохраняем нашу получившуюся
ссылку
                //далее проверим вершину на конечность
                while (next_state->final_vertex_number == 0) //если вершина
не конечная для какой-то подстроки
                {
                    if (next_state->edge == '@')  //если пришли в корень, то
дальше некуда, заканчиваем
                    {
                        break;
                    }
                    next_state = next_state->suffix_link;  //иначе идем по
суффиксной ссылке дальше
```

```
                }
                if (next_state->final_vertex_number > 0)  //если так
получилось, что мы нашли вершину и она конечная
                {
                    end_suffix_link = next_state; //запоминаем ее в конечную
суффиксную ссылку
                }
                std::cout << "Построение ссылки завершено.\n";
            }
        }
    };

    //структура для конечного детерминированного автомата
    struct DFA
    {
        Bor* root;
        std::vector<std::string> patterns;

        DFA (std::vector<std::string>& new_patterns)
        {
            root = new Bor('@', nullptr);
            patterns = new_patterns;
            bild_bor(new_patterns);
            add_all_suffix_links();
        }

        //создание бора путем добавления в него строк-паттернов
        void bild_bor (std::vector<std::string>& new_patterns)
        {
            std::cout << "----------Начинаем построение бора----------\n";
            int string_number = 1;
            for (auto &current_pattern : new_patterns)
            {
                Bor* current_state = root;  //текущее состояние в корне
                for (int i = 0; i < current_pattern.size(); i++)
                {
```

```
                    Bor* next_state = current_state->go_to_child_by_edge(current_pattern[i]); //пробуем перейти к ребенку
                    if (next_state == nullptr) //если вершины при переходе по ребру еще нет
                    {
                        next_state = new Bor(current_pattern[i], current_state); //создаем вершину
                        current_state->children.push_back(next_state);  //добавляем ее в число детей
                    }
                    if (i == current_pattern.size()-1)  //если строка закончилась - то текущая вершина терминальная для данной подстроки
                    {
                        next_state->final_vertex_number = string_number; //добавляем ее номер
                        std::cout << "Строка " << current_pattern << " добавлена в бор\n";
                    }
                    else
                    {
                        current_state = next_state; //иначе просто переходим дальше, т.к. вершина уже есть
                    }
                }
                string_number++;
            }
            std::cout << "----------Построение бора закончено----------\n\n";
        }

        //создаем суффиксные ссылки для каждой вершины
        void add_all_suffix_links ()
        {
            std::cout << "----------Начинаем создание суффиксных ссылок----------\n";
            std::queue<Bor*> queue; //используем очередь, чтобы ходить по вершинам
            Bor* current_state = root;  //начинаем с корня
            queue.push(root);
            while (!queue.empty())  //пока есть по чему ходить
            {
```

```
                current_state->add_suffix_link(); //создаем ссылку для
текущей вершины

                queue.pop(); //удаляем ее из очереди, т.к. обработали

                for (auto &current_child : current_state-
>children)  //добавляем всех ее детей в очередь на обработку

                {

                    queue.push(current_child);

                }

                current_state = queue.front(); //переходим к следующей в
очереди вершине для обработки

            }

            std::cout << "----------Создание суффиксных ссылок
закончено----------\n\n";

        }


        //сам алгоритм Ахо-Корасик

        void aho_corasick (std::string& text)

        {

            std::cout << "----------Начинаем алгоритм Ахо-
Корасика----------\n";

            Bor* current_state = root; //начинаем хождение из корня

            std::vector<std::pair<int, int>> result;  //вектор записи
результата

            for (int i = 0; i < text.size(); i++) //проходимся по символам
текста

            {

                Bor* next_state = current_state-
>go_to_child_by_edge(text[i]);   //спускаемся из текущей вершине по ребру из
текста к потомку

                while (next_state == nullptr) //если такого ребра у нас нет

                {

                    if (current_state == root)  //если мы сейчас находимся в
корне переходим в него же и прерываем

                    {

                        next_state = current_state;

                        break;

                    }

                    current_state = current_state->suffix_link; //если нет,
то переходим по суффиксной ссылке

                    next_state = current_state-
>go_to_child_by_edge(text[i]); //и снова пытаемся спустится
```

```
                }
                if (next_state->final_vertex_number > 0) //если мы достигли
конечной вершины какого-то слова-запоминаем
                {
                    std::cout << "Найдено вхождение образца: " <<
patterns[next_state->final_vertex_number-1] << '\n';
                    result.emplace_back(i-patterns[next_state-
>final_vertex_number-1].size()+2, next_state->final_vertex_number);
                }
                if (next_state->end_suffix_link != nullptr) //если для
следующего состояния есть ссылка на конец
                {
                    Bor* tmp_state = next_state->end_suffix_link;
                    while (tmp_state != nullptr)  //то пока она указывает на
конец - добавляем в результат
                    {
                        result.emplace_back(i-patterns[tmp_state-
>final_vertex_number-1].size()+2, tmp_state->final_vertex_number);
                        tmp_state = tmp_state->end_suffix_link;
                    }
                }
                current_state = next_state; //переходим к следующей
            }
            std::cout << "----------Алгоритм Ахо-Корасика
закончен----------\n\n";
            std::sort(result.begin(), result.end(),
cmp_for_vec_pair); //сортируем согласно заданию
            print_result(result); //печатаем на экран
        }

        //функция печати на экран результата
        void print_result (std::vector<std::pair<int, int>>& result)
        {
            std::cout << "----------Результат работы алгоритма----------\n";
            for (auto &answer : result)
            {
            std::cout << "Позиция в тексте: " << answer.first << " Номер
образца: " << answer.second << '\n';
            }
        }
```

```
};

int main ()
{
    std::cout << "Введите текст T:\n";
    std::string text;
    std::cin >> text;
    std::cout << "Введите количество подстрок:\n";
    int number_pattenrs;
    std::cin >> number_pattenrs;
    std::cout << "Введите все подстроки:\n\n";
    std::vector<std::string> patterns;
    std::string new_pattern;
    for (int i = 0; i < number_pattenrs; i++)
    {
        std::cin >> new_pattern;
        patterns.push_back(new_pattern);
    }
    DFA new_DFA(patterns);
    new_DFA.aho_corasick(text);
    return 0;
}
```

Файл task2.cpp:

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

bool cmp_for_vec_pair(std::pair<int, int> elem1, std::pair<int, int>
elem2)
{
    if (elem1.first < elem2.first)
    {
        return true;
```

```
    }
    else if (elem1.first == elem2.first && elem1.second < elem2.second)
    {
        return true;
    }
    else
    {
        return false;
    }
}

struct Bor
{
    char edge;
    std::vector<int> final_vertex_number;
    std::vector<Bor*> children;
    Bor* parent;
    Bor* suffix_link;
    Bor* end_suffix_link;

    Bor (char new_edge, Bor* new_parent)
    {
        edge = new_edge;
        parent = new_parent;
        suffix_link = nullptr;
        end_suffix_link = nullptr;
    }

    Bor* go_to_child_by_edge (char child_edge)
    {
        for (auto &child : children)
        {
            if (child->edge == child_edge)
            {
                return child;
            }
```

```
    }
    return nullptr;
}

void add_suffix_link ()
{
    if (edge == '@')
    {
        suffix_link = this;
    }
    else if (parent->edge == '@')
    {
        suffix_link = parent;
    }
    else
    {
        Bor* current_state = parent->suffix_link;
        Bor* next_state = current_state->go_to_child_by_edge(edge);
        while (next_state == nullptr)
        {
            if (current_state->edge == '@')
            {
                next_state = current_state;
                break;
            }
            current_state = current_state->suffix_link;
            next_state = current_state->go_to_child_by_edge(edge);
        }
        suffix_link = next_state;
        Bor* tmp_state = next_state;
        while (tmp_state->final_vertex_number.empty())
        {
            if (tmp_state->edge == '@')
            {
                break;
            }
```

```
                    tmp_state = tmp_state->suffix_link;
                }
                if (!tmp_state->final_vertex_number.empty())
                {
                    end_suffix_link = tmp_state;
                }
            }
        }
    };

    struct DFA
    {
        Bor* root;
        std::vector<std::pair<std::string, int>> patterns;
        int last_jokers;

        DFA (std::vector<std::pair<std::string, int>>& new_patterns, int
new_last_jokers)
        {
            root = new Bor('@', nullptr);
            last_jokers = new_last_jokers;
            patterns = new_patterns;
            add_patterns(new_patterns);
            add_all_suffix_links();
        }

        void add_patterns (std::vector<std::pair<std::string, int>>&
new_patterns)
        {
            for (auto &current_pattern : new_patterns)
            {
                Bor* current_state = root;
                for (int i = 0; i < current_pattern.first.size(); i++)
                {
                    Bor* next_state = current_state-
>go_to_child_by_edge(current_pattern.first[i]);
                    if (next_state == nullptr)
```

```
                    {
                        next_state = new Bor(current_pattern.first[i],
current_state);
                        current_state->children.push_back(next_state);
                    }
                    if (i == current_pattern.first.size()-1)
                    {
                        next_state-
>final_vertex_number.push_back(current_pattern.second);
                    }
                    else
                    {
                        current_state = next_state;
                    }
                }
            }
        }

        void add_all_suffix_links ()
        {
            std::queue<Bor*> queue;
            Bor* current_state = root;
            queue.push(root);
            while (!queue.empty())
            {
                current_state->add_suffix_link();
                queue.pop();
                for (auto &current_child : current_state->children)
                {
                    queue.push(current_child);
                }
                current_state = queue.front();
            }
        }

        void aho_corasick (std::string& text)
        {
```

```
            Bor* current_state = root;
            std::vector<int> length_patterns_current(text.size(), 0);
            for (int i = 0; i < text.size()-last_jokers; i++)
            {
                Bor* next_state = current_state->go_to_child_by_edge(text[i]);
                while (next_state == nullptr)
                {
                    if (current_state == root)
                    {
                        next_state = current_state;
                        break;
                    }
                    current_state = current_state->suffix_link;
                    next_state = current_state->go_to_child_by_edge(text[i]);
                }
                for (auto &current_terminal : next_state->final_vertex_number)
                {
                    if (i-current_terminal+1 >= 0)
                    {
                        length_patterns_current[i-current_terminal+1]++;
                    }
                }
                Bor* tmp_state = next_state->end_suffix_link;
                while (tmp_state != nullptr)
                {
                    for (auto &current_terminal : tmp_state->final_vertex_number)
                    {
                        if (i-current_terminal+1 >= 0)
                        {
                            length_patterns_current[i-current_terminal+1]++;
                        }
                    }
                    tmp_state = tmp_state->end_suffix_link;
                }
```

```
            current_state = next_state;
        }
        std::vector<int> result;
        for (int i = 0; i < length_patterns_current.size(); i++)
        {
            if (length_patterns_current[i] == patterns.size())
            {
                result.push_back(i+1);
            }
        }
        print_result(result);
    }

    void print_result(std::vector<int>& result)
    {
        std::cout << "Результат работы алгоритма: \n";
        sort(result.begin(), result.end());
        for (auto &answer : result)
        {
            std::cout << answer << '\n';
        }
    }
};

int main ()
{
    std::cout << "Введите текст T:\n";
    std::string text;
    std::cin >> text;
    std::cout << "Введите шаблон:\n";
    std::string pattern;
    std::cin >> pattern;
    char joker;
    std::cout << "Введите символ джокера:\n";
    std::cin >> joker;
    std::vector<std::pair<std::string, int>> all_patterns;
```

```
    std::string pattern_separation;
    int last_jokers;
    for (int i = 0; i < pattern.size(); i++)
    {
        if (pattern[i] != joker)
        {
            pattern_separation += pattern[i];
            last_jokers = 0;
        }
        else
        {
            if (!pattern_separation.empty())
            {
                all_patterns.emplace_back(pattern_separation, i);
            }
            pattern_separation = "";
            last_jokers++;
        }
    }
    if (!pattern_separation.empty())
    {
        all_patterns.emplace_back(pattern_separation, pattern.size());
    }
    DFA new_dfa(all_patterns, last_jokers);
    new_dfa.aho_corasick(text);
    return 0;
}
```