

**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: Алгоритм Ахо-Корасик**

Студент гр. 8383

\_\_\_\_\_

Мололкин К.А.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

## **Цель работы**

Изучить работу алгоритма Ахо-Корасик, реализующий поиск множества подстрок из словаря в данной строке

## **Постановка задачи**

Вариант 3. Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

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  $P = \{p_1, \dots, p_n\}$   $1 \leq |p_i| \leq 75$

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:**

CCCA

1

CC

### **Sample Output:**

1 1

2 1

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 PP необходимо найти все вхождения PP в текст TT.

Например, образец *ab??c?* с джокером *?* встречается дважды в тексте *xabvcssbababcsax*.

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

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

## **Описание алгоритма Ахо-корасик**

1. На вход алгоритм получает строку текст и набор строк шаблонов.
2. В начале алгоритма строится бор. Бор - это дерево с корнем в некоторой вершине причём каждое ребро дерева подписано некоторой буквой:
  - 2.1. Для построения бора сначала создается корень.
  - 2.2. При добавлении новой строки в бор, перебираются все ребра, исходящие из корня, если нашли ребро с первой буквой новой строки, то переходим к нему. Если не нашли, то создаем новую вершину, связанную с корнем ребром с первым символом входной строки.
  - 2.3. Затем переходим к следующему символу входной строки, просматривая всех потомков текущей вершины.
  - 2.4. Последний символ каждого шаблона помечается терминальной вершиной
3. После построения бора, алгоритм проходится по всем символам текста. Выполняя переходы по бору начиная из вершины по текущему символу текста.
  - 3.1. Если прямого перехода из текущей вершины нет, то переход производится по суффиксальной ссылке. Для потомков корня суффиксальная вершина – корень.
  - 3.2. Для того чтобы найти суффиксальную ссылку для вершины нужно перейти в его родителя, затем из родителя по суффиксальной ссылке, после этого перейти из данной вершины по символу для перехода в текущую вершины, в полученную вершину и будет направлена суффиксальная ссылка, если такой вершины нет, то суффиксальная ссылка направляется в корень.
  - 3.3. После прямого перехода, либо по суффиксальной ссылке, производится проверка, для этого, из текущей вершины алгоритм

переходит по суффиксальной и так далее, пока не найдет терминальную либо корень, если нашли терминальную, значит нашли вхождение.

### **Описание функций и структур данных**

1) class Vertex – класса вершина бора:

Поля:

std::map<char, int> nextVertexes – словарь вершин – потомков.

std::map<char, int> nextNode – словарь переходов из вершины.

std::pair<char, int> parentNode – ребро до вершины.

int suffixLink – суффиксная ссылка.

bool isTerminal – флаг для терминальных вершин.

int patternNumber – номер паттерна(если вершина терминальная).

int level – уровень вершины.

Методы:

Vertex() – конструктор.

Vertex(int prevInd, char prevChar) – конструктор инициализирующий родителя.

2) class Bohr – класс хранящий бор.

Поля:

std::vector<std::pair<int, int>> result – вектор хранящий результат работы алгоритма Ахо-Корасик;

std::vector<Vertex> vertexes – вектор всех вершин бора;

int terminalsNumb – количество терминальных вершин;

int mostDeepLevel – глубина бора;

Методы:

Bohr() – конструктор:

Функция создает корень бора, добавляет его в вектор вершин, и инициализирует остальные поля.

void addStringToBohr(const std::string& str) – добавление строки в бор:

Функция принимает на вход ссылку на строку – str. Создается переменная для хранения индекса текущей вершины. Затем проходится по каждому символу

входной строки. Если среди потомков текущей вершины есть вершина, в которую ведет текущий символ, то переходим в нее, если нет, то создается новая вершина, в которую ведет текущий символ. Последняя вершина обозначается терминальной, записывается ее уровень и номер паттерна.

`int getSuffLink(int i)` – поиск суффиксальной вершины для переданной:

Функция принимает на вход номер вершины – `int i`. Если у данной вершины есть суффиксальная ссылка, то возвращает ее. Если вершина является корнем или его потомком, то суффиксальная ссылка равна 0, если нет, то суффиксальная ссылка равна результату выполнения функции `getNextVertex()`, в которую передается суффиксальная ссылка для родителя переданной вершины и символа ведущего в данную вершину.

`int getNextVertex(int i, char c)` – функция возвращает переход из переданной вершины:

Функция принимает на вход `int I` – номер вершины в векторе вершин, `char c` – символ, по которому совершается переход. Если у переданной вершины нет переход в словаре переходов, то проверяет словарь потомков, если есть потомок в который ведет ребро с переданным символом, то записываем этот путь, в противном случае, если переданная вершина – корень, то записываем в словарь переходов ссылку на сам корень, если не корень, то записываем в словарь переходов результат рекурсивно вызванной функции, с первым аргументом – результат функции `getSuffixLink(i)` и переданным символом. Функция возвращает значение словаря переходов по переданному символу.

`void findAllPatternsOnText(std::string text)` – реализует алгоритм множественного поиска:

Функция принимает на вход `std::string& text` – ссылка на текст в котором будет произведен поиск. Сначала создается переменная для хранения текущей вершины – `int cur`. В цикле проходится по всем элементам текста, к текущей вершине приравнивается значение, возвращаемое функцией `getNextVertex` с аргументами `cur` и `тек` символом в тексте. Затем в цикле для текущей вершины проходим по суффиксальным ссылкам, пока не дойдем для корня либо

терминальной вершины, если нашли терминальную, то добавляем в вектор результатов, пару значений patternNumb, для найденной терминальной и номер начала этого паттерна в строке. После того как пройдем по всем символам в тексте, сортируем вектор результатов и выводим его. В индивидуально варианте программы так же выводятся максимальная длина цепи из суффиксальных ссылок и максимальную длину цепи из прямых ссылок.

static int compare(std::pair<int, int> a, std::pair<int, int> b) – компаратор для сортировки результирующего вектора.

### **Сложность алгоритма по времени**

На построение бора уходит  $O(m)$ , а  $m$  – сумма длин всех символов всех шаблонов. Затем на проход всех символов текста по бору уходит  $O(n)$ , где  $n$  – длина текста, затем производится проверка, на которую уходит  $O(q)$ , где  $q$  – количество вхождений шаблонов в текст. Следовательно общая сложность  $O((n + m + q))$ .

### **Сложность алгоритма по памяти**

По памяти алгоритм занимает  $O(m*k)$ , где  $k$  – размер алфавита, а  $m$  – сумма длин всех символов. В худшем случае бор будет хранить все символы шаблонов на разных ребрах.

### **Описание алгоритма поиска по строке с джокером**

1. Шаблон с джокером разбивается по символу джокеру.
2. Полученные строки ищутся в тексте по описанному выше алгоритму Ахо-Корасик.
3. Для каждого совпадения индекса места начала совпадения храним количество найденных в нем совпадений.
4. Затем ищем место в тексте, в котором число совпадений равно количество подстрок.
5. Если нашли совпадения, то выводим позицию вхождения.

## Описание функция и структур данных

### 1) class Vertex

Класс практически совпадает с классом Vertex для алгоритма Ахо-Корасик. Только добавляется поле – вектор `std::vector<int> posInJokerPattern`, который хранит позиции начала разделенных подстрок в образце.

### 1) class BohrWithJoker

Поля:

`std::vector<Vertex> vertexes` – вектор всех вершин бора.

`char joker` – символ джокера.

`int jokerPatternSize` – длина строки шаблона.

Методы:

`BohrWithJoker(char joker)` – конструктор класса:

Создает корень и добавляет его в вектор вершин.

`void addJokerPattern(std::string jokerPattern)` – добавление шаблона в бор:

Функция принимает на вход ссылку на строку - `std::string& jokerPattern` шаблон. Затем в цикле проходится по всем символам шаблона. Заводим переменные для текущей вершины, счетчик подстроки без джокеров и флаг, которые используются для определения является ли пред. символ джокером. Далее запускается цикл по всем символам шаблона. Если встречается символ джокера, и тек. символ первый в строке, то обнуляются счетчик, если предыдущий символ – джокер, то обнуляются тек. вершина и счетчик, если данные условия не выполняются, то делаем тек. вершину терминальной, в вектор позиций начала разделенных подстрок для тек. вершины добавляется начало найденной подстроки. Если тек. символ не является джокером, то увеличивается счетчик и добавляется тек. символ в бор. После прохода по всем символа шаблона, если последний символ был джокером, то тек. вершина делается терминальной и в вектор позиций начала разделенных подстрок для тек. вершины добавляется начало найденной подстроки.

`int getNextVertex(int i, char c)` – поиск след вершины для автомата:



Функция аналогична описанной выше для множественного поиска.

`int getSuffixLink(int i)` – поиск след. суффиксальной вершины:

Функция аналогична описанной выше для множественного поиска.

`void findAllJokerPatternsOnText(std::string& text)` – поиск для поиска образца с джокером в тексте:

Функция принимает на вход ссылку на строку - `std::string& text` текста для поиска шаблона. Затем создается вектор для хранения числа совпадений, тек. вершина и количество найденных подстрок, которое равно сумме длин всех векторов `posInJokerPattern` для каждой вершины бора. Затем в цикле проходится по бору и всем символам текста совершая переход, используя функцию `getNextVertex`, в которую передается тек. вершина и тек. символ текста, ищем подстроки шаблона при каждом найденном шаблоне вычисляем место в тексте, которому соответствует тек. подстрока и сохраняем в вектор числа совпадений. Затем проходимся по всем элементам вектора числа совпадений ищем элемент, который равен количеству найденных подстрок. Если нашли, то выводим его индекс увеличенный на 1.

### **Сложность алгоритма по времени**

Сложность данного алгоритма, такая же как и у алгоритма Ахо-Корасик, следовательно сложность равна  $O((n + m + q))$ .

### **Сложность алгоритма по памяти**

По памяти алгоритм занимает  $O(m*q+n)$ , так как хранится бор и дополнительно массив, хранящий число совпадений для каждого символа.

## Тестирование

### Алгоритм Ахо-Корасик (индивидуальный вариант)

CCCA

1

CC

Add string: "CC" to bohr

Index of current vertex: 0

No next vertex with node: 'C' create new vertex with node: 'C' it's index in vertexes array: 1

Index of current vertex: 1

No next vertex with node: 'C' create new vertex with node: 'C' it's index in vertexes array: 2

String: "CC" was added to the bohr

Made last vertex terminal

It's level: 2, pattern number: 0 was added to patterns numbers array of ist vertex

This is the most deep vertex of bohr

Created bohr:

Vertex index: 0

Vertex is root

Vertex index: 1

Symbol to vertex: 'C' parent index: 0

Next vertexes: ('C', 2)

Vertex index: 2

Vertex is terminal

Symbol to vertex: 'C' parent index: 1

Vertex level: 2

Pattern numbers: 1

Start searching entry of patterns to text: CCCA

Start from root

Find next move for vertex with index: 0 by symbol: 'C'

No next move by symbol: 'C'

There is forward move to next vertex by symbol: 'C' to vertex with index: 1

Next move vertex with index: 1 by symbol 'C'

Next current is vertex with index: 1

Start searching terminal by suffix link from current vertex

Get suffix link of vertex with index: 1

Vertex doesn't have suffix link

Vertex is root or root child it's suffix link to root

Suffix link to vertex with index: 1 is 0

Find next move for vertex with index: 1 by symbol: 'C'

No next move by symbol: 'C'  
 There is forward move to next vertex by symbol: 'C' to vertex with index: 2  
 Next move vertex with index: 2 by symbol 'C'  
 Next current is vertex with index: 2  
 Start searching terminal by suffix link from current vertex  
 Terminal was founded  
 Pattern index and number: 1 1  
 Get suffix link of vertex with index: 2  
 Vertex doesn't have suffix link  
 To find suffix link go to parent vertex and check it's suffix link  
 Get suffix link of vertex with index: 1  
 Suffix link to vertex with index: 1 is 0  
 Find next move for vertex with index: 0 by symbol: 'C'  
 Next move vertex with index: 1 by symbol 'C'  
 Suffix link to vertex with index: 2 is 1  
 Start searching terminal by suffix link from current vertex  
 Get suffix link of vertex with index: 1  
 Suffix link to vertex with index: 1 is 0  
 Find next move for vertex with index: 2 by symbol: 'C'  
 No next move by symbol: 'C'  
 Next move by suffix link  
 Get suffix link of vertex with index: 2  
 Suffix link to vertex with index: 2 is 1  
 Find next move for vertex with index: 1 by symbol: 'C'  
 Next move vertex with index: 2 by symbol 'C'  
 Next move vertex with index: 2 by symbol 'C'  
 Next current is vertex with index: 2  
 Start searching terminal by suffix link from current vertex  
 Terminal was founded  
 Pattern index and number: 1 2  
 Get suffix link of vertex with index: 2  
 Suffix link to vertex with index: 2 is 1  
 Start searching terminal by suffix link from current vertex  
 Get suffix link of vertex with index: 1  
 Suffix link to vertex with index: 1 is 0  
 Find next move for vertex with index: 2 by symbol: 'A'  
 No next move by symbol: 'A'  
 Next move by suffix link  
 Get suffix link of vertex with index: 2  
 Suffix link to vertex with index: 2 is 1  
 Find next move for vertex with index: 1 by symbol: 'A'

No next move by symbol: 'A'  
 Next move by suffix link  
 Get suffix link of vertex with index: 1  
 Suffix link to vertex with index: 1 is 0  
 Find next move for vertex with index: 0 by symbol: 'A'  
 No next move by symbol: 'A'  
 It is root vertex without child by symbol: 'A' so next move is root  
 Next move vertex with index: 0 by symbol 'A'  
 Next move vertex with index: 0 by symbol 'A'  
 Next move vertex with index: 0 by symbol 'A'  
 Next current is vertex with index: 0  
 All symbols of text was checked, sort result array  
 Search longest suffix link chain  
 Current vertex: 1  
 Suffix link chain for cur vertex: 1  
 Current vertex: 2  
 Get suffix link of vertex with index: 2  
 Suffix link to vertex with index: 2 is 1  
 Go to vertex: 1 by suffix link  
 Suffix link chain for cur vertex: 2  
 Result:  
 Index in text | pattern number  
 1                1  
 2                1  
 Longest link chain: 2  
 Longest suffix link chain: 2

№	Input	Output
1	C C C C A 2 C C C C	1 1 1 2 2 1 2 2 3 1 3 2 Longest link chain: 2 Longest suffix link chain: 2
2	C G T N A N N T T A C C G 5 C G T G T N A N N N A N T T A	1 1 2 2 4 3 4 4 8 5 Longest link chain: 4 Longest suffix link chain: 3
3	G T G T G T	1 1

	4 GTGTGT GTG TGT T	1 2 2 3 2 4 3 2 4 3 4 4 6 4 Longest link chain: 6 Longest suffix link chain: 5
4	AAAAAA 1 DDDDDD	Patterns not founded in text Longest link chain: 6 Longest suffix link chain: 6
5	TTTTTT 2 TTTTTT TTTTTT	1 1 1 2 Longest link chain: 6 Longest suffix link chain: 6

## Алгоритм поиска по строке с джокером

ABA

A\$A

\$

Add pattern with joker: A\$A to bohr

Cur symbol: 'A' is not joker

, no vertexes from current by symbol: A, so add new vertex with index: 1

Go to new vertex new current vertex index: 1

Cur symbol: '\$' is joker, make current vertex terminal

Now current = 0

Cur symbol: 'A' is not joker

Go to new vertex new current vertex index: 1

Make pattern end terminal

Vertex index: 0

Vertex is root

Vertex index: 1

Vertex is terminal

Symbol to vertex: 'A' parent index: 0

Vertex level: 1

Strat to find pattern in text: ABA

Number of founded substrings: 2

Find next move for vertex with index: 0 by symbol: 'A'

No next move by symbol: 'A'

There is forward move to next vertex by symbol: 'A' to vertex with index: 1

Next move vertex with index: 1 by symbol 'A'

Current vertex index: 1

Start searching terminal by suffix link from current vertex

Terminal was founded

Increase element at index: 0 in array of number of matches, it's value: 1

Get suffix link of vertex with index: 1

Vertex doesn't have suffix link

Vertex is root or root child it's suffix link to root

Suffix link to vertex with index: 1 is 0

Find next move for vertex with index: 1 by symbol: 'B'

No next move by symbol: 'B'

Next move by suffix link

Get suffix link of vertex with index: 1

```

Find next move for vertex with index: 0 by symbol: 'B'
No next move by symbol: 'B'
It is root vertex without child by symbol: 'B' so next move is root
Next move vertex with index: 0 by symbol 'B'
Next move vertex with index: 0 by symbol 'B'
Current vertex index: 0
Find next move for vertex with index: 0 by symbol: 'A'
Next move vertex with index: 1 by symbol 'A'
Current vertex index: 1
Start searching terminal by suffix link from current vertex
Terminal was founded
Increase element at index: 2 in array of number of matches, it's value: 1
Increase element at index: 0 in array of number of matches, it's value: 2
Get suffix link of vertex with index: 1
Suffix link to vertex with index: 1 is 0
RESULT:
1

```

№	Input	Output
1	ACTANCA A\$\$\$ \$	1
2	ACCCGAACCCAA A&&&A &	1 6 7
3	xxttxxttx ##tt #	1 5
4	xabvccbababca ab??c? ?	2 8

## Вывод

В результате работы был изучен алгоритм Ахо-Корасик, с его помощью были решены задача множественного поиска и поиска по строке с джокером.

## Приложение А

### Код программы aho-korasik.cpp

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

class Vertex {
public:
    std::map<char, int> nextVertexes; // вектор вершин потомков
    std::pair<char, int> parentNode; //предок
    std::map<char, int> nextNode; //следующее ребро из вершины по
    какому-либо символу
    int suffixLink; //суффиксальная ссылка
    bool isTerminal; //флаг терминальной вершины
    int level; //уровень вершины
    std::vector<int> numbers; //номера паттернов

    Vertex() : isTerminal(false), suffixLink(-1), level(0),
    parentNode(std::pair<char, int>(' ', -1)) {};
    Vertex(int prevInd, char prevChar) : isTerminal(false),
    suffixLink(-1), level(0), parentNode(std::pair<char,
    int>(prevChar, prevInd)) {};

    void printVertex() {
        if(parentNode.second == -1)
            std::cout << "Vertex is root" << std::endl;
        else if(isTerminal){
            std::cout << "Vertex is terminal" << std::endl;
            std::cout << "Symbol to vertex: \' " <<
            parentNode.first << "\' parent index: " << parentNode.second <<
            std::endl;

            std::cout << "Vertex level: " << level << std::endl;
            std::cout << "Pattern numbers: ";
            for(int number : numbers) {
                std::cout << number + 1 << " ";
            }
            std::cout << std::endl;
            if(!nextVertexes.empty()) {
                std::cout << "Next vertexes: ";
                for (auto &nextVertex : nextVertexes) {
                    std::cout << "(" << nextVertex.first << ", "
                    << nextVertex.second << ") ";
                }
                std::cout << std::endl;
            }
        }
        else {
```



```

        std::cout << "Symbol to vertex: \' " <<
parentNode.first << "\' parent index: " << parentNode.second <<
std::endl;
        std::cout << "Next vertexes: ";
        for(auto & nextVertex : nextVertexes) {
            std::cout << "(\' " << nextVertex.first << "\', "
<< nextVertex.second << ") ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
};

class Bohr {
public:
    std::vector<std::pair<int, int>> result; //результатирующие
вектор
    std::vector<Vertex> vertexes; //вершины дерева
    int terminalsNumb; //количество терминальных
    Bohr() {
        Vertex root; //создаем корень и добавляем ввектор, а также
остальные инициализируем поля
        vertexes.push_back(root);
        terminalsNumb = 0;
        mostDeepLevel = 0;
    }

    void addStringToBohr(const std::string& str){ //добавление
строки в бор
        std::cout << "Add string: \' " << str << "\' to bohr" <<
std::endl;
        int cur = 0; //тек. на начало
        for (char i : str) { //проходимся по всем символам
переданной строки
            std::cout << "Index of current vertex: " << cur <<
std::endl;
            if(vertexes[cur].nextVertexes.find(i) ==
vertexes[cur].nextVertexes.end()) { //если в боре нет такой
вершины
                std::cout << "No next vertex with node: \' " << i
<< "\' create new vertex with node: \' " << i << "\' it's index in
vertexes array: " << vertexes.size() << std::endl;
                Vertex curVertex(cur, i); //создаем ее и
добавляем
                vertexes.push_back(curVertex);
                vertexes[cur].nextVertexes[i] = vertexes.size() -
1; //связываем потомка с родителем
            }
            cur = vertexes[cur].nextVertexes[i]; //если в боре уже
есть такая вершина, просто переходим дальше
        }
    }
};

```

```

        std::cout << "String: \"" << str << "\"" was added to the
bohr" << std::endl;
        std::cout << "Made last vertex terminal" << std::endl;
        std::cout << "It's level: " << str.length();
        std::cout << ", pattern number: " << terminalsNumb << "
was added to patterns numbers array of ist vertex" << std::endl;
        vertexes[cur].isTerminal = true; //последняя вершина
терминальная
        vertexes[cur].numbers.push_back(terminalsNumb++);
//добавляем номер шаблона данной вершины
        vertexes[cur].level = str.length();
        if (str.length() > mostDeepLevel) {
            mostDeepLevel = str.length(); //изменяем длину самой
длинной цепочки прямых ссылок
            std::cout << "This is the most deep vertex of bohr" <<
std::endl;
        }
    }

    void printBohr() {
        for (int i = 0; i < vertexes.size(); ++i) {
            std::cout << "Vertex index: " << i << std::endl;
            vertexes[i].printVertex();
        }
    }

    int getSuffixLink(int i) { //получение суффиксальной ссылки
для вершины
        std::cout << "Get suffix link of vertex with index: " << i
<< std::endl;
        if (vertexes[i].suffixLink == -1) { //если еще нет
суффиксальной
            std::cout << "Vertex doesn't have suffix link" <<
std::endl;
            if (i == 0 || vertexes[i].parentNode.second == 0) {
//если корень или его потомок, то ссылка на корень
                std::cout << "Vertex is root or root child it's
suffix link to root" << std::endl;
                vertexes[i].suffixLink = 0;
            }
            else { //если нет то ищем путь из суффиксальной
вершины родителя, по символу от родителя к тек вершине
                std::cout << "To find suffix link go to parent
vertex and check it's suffix link" << std::endl;
                vertexes[i].suffixLink =
getNextVertex(getSuffixLink(vertexes[i].parentNode.second),
vertexes[i].parentNode.first);
            }
        }
        std::cout << "Suffix link to vertex with index: " << i <<
" is " << vertexes[i].suffixLink << std::endl;
        return vertexes[i].suffixLink;
    }
}

```

```

    int getNextVertex(int i, char c) { //следующий шаг автомата
        std::cout << "Find next move for vertex with index: " << i
        << " by symbol: \' " << c << "\' "<< std::endl;
        if (vertexes[i].nextNode.find(c) ==
vertexes[i].nextNode.end()) { //если нет пути в словаре путей
автомата по переданному символу
            std::cout << "No next move by symbol: \' " << c << "\' "
<< std::endl;
            if (vertexes[i].nextVertexes.find(c) !=
vertexes[i].nextVertexes.end()) { //если есть прямая ссылка
                std::cout << "There is forward move to next vertex
by symbol: \' " << c << "\' to vertex with index: " <<
vertexes[i].nextVertexes[c] << std::endl;
                vertexes[i].nextNode[c] =
vertexes[i].nextVertexes[c]; //то добавляем ее в путь
            }
            else {
                if (i == 0) { //если корень и нет потомков с
путем по переданному символу, то ссылка на корень
                    std::cout << "It is root vertex without child
by symbol: \' " << c << "\' so next move is root"<< std::endl;
                    vertexes[i].nextNode[c] = 0;
                }
                else { //в противном случае добавляем в словарь
след вершину из суффиксальной ссылки
                    std::cout << "Next move by suffix link" <<
std::endl;
                    vertexes[i].nextNode[c] =
getNextVertex(getSuffixLink(i), c);
                }
            }
        }
        std::cout << "Next move vertex with index: " <<
vertexes[i].nextNode[c] << " by symbol \' " << c << "\' " <<
std::endl;
        return vertexes[i].nextNode[c];
    }

    void findAllPatternsOnText(std::string& text) { //поиск
        std::cout << "Start searching entry of patterns to text: "
<< text << std::endl;
        int cur = 0; // тек равна корню
        std::cout << "Start from root" << std::endl;
        for (int i = 0; i < text.length(); i++) {
            cur = getNextVertex(cur, text[i]); //получаем путь по
i - ому символу текста
            std::cout << "Next current is vertex with index: " <<
cur << std::endl;
            for (int j = cur; j != 0; j = getSuffixLink(j)) { //
затем проходимся от тек символа до корня по суффиксальным
                std::cout << "Start searching terminal by suffix
link from current vertex" << std::endl;

```

```

        if (vertexes[j].isTerminal) { //если нашли
терминальную
            std::cout << "Terminal was founded" <<
std::endl;
            for(int k = 0; k < vertexes[j].numbers.size();
k++) { //то доавляем в результат найденный паттерн, а если есть
одинаковые паттерны, то добавляем в результат все
                std::pair<int, int>
res(vertexes[j].numbers[k], i + 2 - vertexes[j].level);
                result.push_back(res);
                std::cout <<"Pattern index and number: "
<< vertexes[j].numbers[k] + 1<<" " << i + 2 - vertexes[j].level
<< std::endl;
            }
        }
    }
    std::cout << "All symbols of text was checked, sort result
array" << std::endl;
    sort(result.begin(), result.end(), compare);
//сортировка результат
    if(result.empty()) std::cout << "Patterns not founded in
text" << std::endl;
    else {
        std::cout << "Result: " << std::endl;
        std::cout << "Index in text | pattern number" <<
std::endl;
        for (auto &i : result) { //выводим каждую пару
результата
            std::cout << i.second << "\t\t" << i.first + 1 <<
std::endl;
        }
    }
}

static int compare(std::pair<int, int> a, std::pair<int, int>
b) { //компаратор пар для ответа
    if (a.second == b.second) {
        return a.first < b.first;
    }
    else {
        return a.second < b.second;
    }
}

};

int main() {
    std::string text;
    std::string curPattern;
    int size = 0;
    std::cin >> text;
    std::cin >> size;
    Bohr bohr;

```

```
    for (int i = 0; i < size; ++i) {  
        std::cin >> curPattern;  
        bohr.addStringToBohr(curPattern);  
    }  
    std::cout << "Created bohr:" << std::endl;  
    bohr.printBohr();  
    bohr.findAllPatternsOnText(text);  
}
```

## Приложение Б

### Код программы aho-korasik\_individual.cpp

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

class Vertex {
public:
    std::map<char, int> nextVertexes; // вектор вершин потомков
    std::pair<char, int> parentNode; //предок
    std::map<char, int> nextNode; //следующее ребро из вершины по
    какому-либо символу
    int suffixLink; //суффиксальная ссылка
    bool isTerminal; //флаг терминальной вершины
    int level; //уровень вершины
    std::vector<int> numbers; //номера паттернов

    Vertex() : isTerminal(false), suffixLink(-1), level(0),
    parentNode(std::pair<char, int>(' ', -1)) {};
    Vertex(int prevInd, char prevChar) : isTerminal(false),
    suffixLink(-1), level(0), parentNode(std::pair<char,
    int>(prevChar, prevInd)) {};

    void printVertex() {
        if(parentNode.second == -1)
            std::cout << "Vertex is root" << std::endl;
        else if(isTerminal){
            std::cout << "Vertex is terminal" << std::endl;
            std::cout << "Symbol to vertex: \' " <<
parentNode.first << "\' parent index: " << parentNode.second <<
std::endl;

            std::cout << "Vertex level: " << level << std::endl;
            std::cout << "Pattern numbers: ";
            for(int number : numbers) {
                std::cout << number + 1 << " ";
            }
            std::cout << std::endl;
            if(!nextVertexes.empty()) {
                std::cout << "Next vertexes: ";
                for (auto &nextVertex : nextVertexes) {
                    std::cout << "(" << nextVertex.first << ", "
<< nextVertex.second << ") ";
                }
                std::cout << std::endl;
            }
        }
        else {
```

```

        std::cout << "Symbol to vertex: \' " <<
parentNode.first << "\' parent index: " << parentNode.second <<
std::endl;
        std::cout << "Next vertexes: ";
        for(auto & nextVertex : nextVertexes) {
            std::cout << "(\' " << nextVertex.first << "\', "
<< nextVertex.second << ") ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
};

class Bohr {
public:
    std::vector<std::pair<int, int>> result; //результатирующи
вектор
    std::vector<Vertex> vertexes; //вершины дерева
    int terminalsNumb; //количество терминальных
    int mostDeepLevel; //самая длинная цепочка прямых ссылок
    Bohr() {
        Vertex root; //создаем корень и добавляем ввектор, а также
остальные инициализируем поля
        vertexes.push_back(root);
        terminalsNumb = 0;
        mostDeepLevel = 0;
    }

    void addStringToBohr(const std::string& str){ //добавление
строки в бор
        std::cout << "Add string: \' " << str << "\' to bohr" <<
std::endl;
        int cur = 0; //тек. на начало
        for (char i : str) { //проходимся по всем символам
переданной строки
            std::cout << "Index of current vertex: " << cur <<
std::endl;
            if(vertexes[cur].nextVertexes.find(i) ==
vertexes[cur].nextVertexes.end()) { //если в боре нет такой
вершины
                std::cout << "No next vertex with node: \' " << i
<< "\' create new vertex with node: \' " << i << "\' it's index in
vertexes array: " << vertexes.size() << std::endl;
                Vertex curVertex(cur, i); //создаем ее и
добавляем
                vertexes.push_back(curVertex);
                vertexes[cur].nextVertexes[i] = vertexes.size() -
1; //связываем потомка с родителем
            }
            cur = vertexes[cur].nextVertexes[i]; //если в боре уже
есть такая вершина, просто переходим дальше
        }
    }
}

```

```

        std::cout << "String: \"" << str << "\"" was added to the
bohr" << std::endl;
        std::cout << "Made last vertex terminal" << std::endl;
        std::cout << "It's level: " << str.length();
        std::cout << ", pattern number: " << terminalsNumb << "
was added to patterns numbers array of ist vertex" << std::endl;
        vertexes[cur].isTerminal = true; //последняя вершина
терминальная
        vertexes[cur].numbers.push_back(terminalsNumb++);
//добавляем номер шаблона данной вершины
        vertexes[cur].level = str.length();
        if (str.length() > mostDeepLevel) {
            mostDeepLevel = str.length(); //изменяем длину самой
длинной цепочки прямых ссылок
            std::cout << "This is the most deep vertex of bohr" <<
std::endl;
        }
    }

    void printBohr() {
        for (int i = 0; i < vertexes.size(); ++i) {
            std::cout << "Vertex index: " << i << std::endl;
            vertexes[i].printVertex();
        }
    }

    int getSuffixLink(int i) { //получение суффиксальной ссылки
для вершины
        std::cout << "Get suffix link of vertex with index: " << i
<< std::endl;
        if (vertexes[i].suffixLink == -1) { //если еще нет
суффиксальной
            std::cout << "Vertex doesn't have suffix link" <<
std::endl;
            if (i == 0 || vertexes[i].parentNode.second == 0) {
//если корень или его потомок, то ссылка на корень
                std::cout << "Vertex is root or root child it's
suffix link to root" << std::endl;
                vertexes[i].suffixLink = 0;
            }
            else { //если нет то ищем путь из суффиксальной
вершины родителя, по символу от родителя к тек вершине
                std::cout << "To find suffix link go to parent
vertex and check it's suffix link" << std::endl;
                vertexes[i].suffixLink =
getNextVertex(getSuffixLink(vertexes[i].parentNode.second),
vertexes[i].parentNode.first);
            }
        }
        std::cout << "Suffix link to vertex with index: " << i <<
" is " << vertexes[i].suffixLink << std::endl;
        return vertexes[i].suffixLink;
    }
}

```



```

    int getNextVertex(int i, char c) { //следующий шаг автомата
        std::cout << "Find next move for vertex with index: " << i
        << " by symbol: \' " << c << "\' "<< std::endl;
        if (vertexes[i].nextNode.find(c) ==
vertexes[i].nextNode.end()) { //если нет пути в словаре путей
автомата по переданному символу
            std::cout << "No next move by symbol: \' " << c << "\' "
<< std::endl;
            if (vertexes[i].nextVertexes.find(c) !=
vertexes[i].nextVertexes.end()) { //если есть прямая ссылка
                std::cout << "There is forward move to next vertex
by symbol: \' " << c << "\' to vertex with index: " <<
vertexes[i].nextVertexes[c] << std::endl;
                vertexes[i].nextNode[c] =
vertexes[i].nextVertexes[c]; //то добавляем ее в путь
            }
            else {
                if (i == 0) { //если корень и нет потомков с
путем по переданному символу, то ссылка на корень
                    std::cout << "It is root vertex without child
by symbol: \' " << c << "\' so next move is root"<< std::endl;
                    vertexes[i].nextNode[c] = 0;
                }
                else { //в противном случае добавляем в словарь
след вершину из суффиксальной ссылки
                    std::cout << "Next move by suffix link" <<
std::endl;
                    vertexes[i].nextNode[c] =
getNextVertex(getSuffixLink(i), c);
                }
            }
        }
        std::cout << "Next move vertex with index: " <<
vertexes[i].nextNode[c] << " by symbol \' " << c << "\' " <<
std::endl;
        return vertexes[i].nextNode[c];
    }

    void findAllPatternsOnText(std::string& text) { //поиск
        std::cout << "Start searching entry of patterns to text: "
<< text << std::endl;
        int cur = 0; // тек равна корню
        std::cout << "Start from root" << std::endl;
        for (int i = 0; i < text.length(); i++) {
            cur = getNextVertex(cur, text[i]); //получаем путь по
i - ому символу текста
            std::cout << "Next current is vertex with index: " <<
cur << std::endl;
            for (int j = cur; j != 0; j = getSuffixLink(j)) { //
затем проходимся от тек символа до корня по суффиксальным
                std::cout << "Start searching terminal by suffix
link from current vertex" << std::endl;

```

```

        if (vertexes[j].isTerminal) { //если нашли
терминальную
            std::cout << "Terminal was founded" <<
std::endl;
            for(int k = 0; k < vertexes[j].numbers.size();
k++) { //то доавляем в результат найденный паттерн, а если есть
одинаковые паттерны, то добавляем в результат все
                std::pair<int, int>
res(vertexes[j].numbers[k], i + 2 - vertexes[j].level);
                result.push_back(res);
                std::cout <<"Pattern index and number: "
<< vertexes[j].numbers[k] + 1<<" " << i + 2 - vertexes[j].level
<< std::endl;
            }
        }
    }
    std::cout << "All symbols of text was checked, sort result
array" << std::endl;
    sort(result.begin(), result.end(), compare);
//сортировка результат
    int mostSuffixChain = 1;
    std::cout << "Search longest suffix link chain" <<
std::endl;
    for(int i = 1; i < vertexes.size(); i++){ //проходимся по
каждой вершине бора
        std::cout << "Current vertex: " << i << std::endl;
        int curSuffixChain = 1;
        int curVertex = i;
        while(vertexes[curVertex].suffixLink != 0){ //из
каждой вершины проходимся по суффиксальным и считаем длину суф
цепи
            curSuffixChain++;
            curVertex = getSuffixLink(curVertex);
            std::cout << "Go to vertex: " << curVertex << " by
suffix link";
        }
        std::cout << "Suffix link chain for cur vertex: " <<
curSuffixChain << std::endl;
        if(curSuffixChain > mostSuffixChain) {
            mostSuffixChain = curSuffixChain;
        }
    }
    if(result.empty()) std::cout << "Patterns not founded in
text" << std::endl;
    else {
        std::cout << "Result: " << std::endl;
        std::cout << "Index in text | pattern number" <<
std::endl;
        for (auto &i : result) { //выводим каждую пару
результата
            std::cout << i.second << "\t\t" << i.first + 1 <<
std::endl;

```

```

        }
    }

    std::cout << "Longest link chain: " << mostDeepLevel <<
std::endl; //выводим максимальную цепочку прямых ссылок
    std::cout << "Longest suffix link chain: " <<
mostSuffixChain << std::endl;

}

static int compare(std::pair<int, int> a, std::pair<int, int>
b) { //компаратор пар для ответа
    if (a.second == b.second) {
        return a.first < b.first;
    }
    else {
        return a.second < b.second;
    }
}

};

int main() {
    std::string text;
    std::string curPattern;
    int size = 0;
    std::cin >> text;
    std::cin >> size;
    Bohr bohr;
    for (int i = 0; i < size; ++i) {
        std::cin >> curPattern;
        bohr.addStringToBohr(curPattern);
    }
    std::cout << "Created bohr:" << std::endl;
    bohr.printBohr();
    bohr.findAllPatternsOnText(text);
}

```

## Приложение В

### Код программы joker.cpp

```
#include <iostream>
#include <map>
#include <vector>

class Vertex {
public:
    std::map<char, int> nextVertexes; // вектор вершин потомков
    std::pair<char, int> parentNode; //предок
    std::map<char, int> nextNode; //следующее ребро из вершины по
какому-либо символу
    int suffixLink; //суффиксальная ссылка
    bool isTerminal; //флаг терминальной вершины
    int level; //глубина вершины
    std::vector<int> posInJokerPattern; //позиция в паттерне

    Vertex() : isTerminal(false), suffixLink(-1), level(0),
parentNode(std::pair<char, int>(' ', -1)) {};
    Vertex(int prevInd, char prevChar) : isTerminal(false),
suffixLink(-1), level(0), parentNode(std::pair<char,
int>(prevChar, prevInd)) {};

    void printVertex() {
        if(parentNode.second == -1)
            std::cout << "Vertex is root" << std::endl;
        else if(isTerminal){
            std::cout << "Vertex is terminal" << std::endl;
            std::cout << "Symbol to vertex: \' " <<
parentNode.first << "\' parent index: " << parentNode.second <<
std::endl;

            std::cout << "Vertex level: " << level << std::endl;
            std::cout << std::endl;
            if(!nextVertexes.empty()) {
                std::cout << "Next vertexes: ";
                for (auto &nextVertex : nextVertexes) {
                    std::cout << "(" << nextVertex.first << ", "
<< nextVertex.second << ") ";
                }
                std::cout << std::endl;
            }
        }
        else {
            std::cout << "Symbol to vertex: \' " <<
parentNode.first << "\' parent index: " << parentNode.second <<
std::endl;

            std::cout << "Next vertexes: ";
            for(auto & nextVertex : nextVertexes) {
                std::cout << "(" << nextVertex.first << "\', "
<< nextVertex.second << ") ";
            }
        }
    }
};
```

```

        std::cout << std::endl;
    }
    std::cout << std::endl;
}
};

class BohrWithJoker {
public:
    std::vector<Vertex> vertexes; //все вершины бора
    char joker; //символ джокера
    int jokerPatternSize = 0; //размер шаблона джокера

    explicit BohrWithJoker(char joker) : joker(joker) { //в
конструкторе создаем корень и добавляем в вектор вершин
        Vertex root;
        vertexes.push_back(root);
    }

    void printBohr() {
        for (int i = 0; i < vertexes.size(); ++i) {
            std::cout << "Vertex index: " << i << std::endl;
            vertexes[i].printVertex();
        }
    }

    void addJokerPattern(std::string& jokerPattern){ //добавление
шаблона в бор
        std::cout << "Add pattern with joker: " << jokerPattern <<
" to bohr" << std::endl;
        int cur = 0;
        int counter = 0;
        bool isPrevJoker = false;
        jokerPatternSize = jokerPattern.size();
        for (int j = 0; j < jokerPattern.length(); j++) {
//проходимся по каждому символу шаблона
            std::cout << "Cur symbol: \' " << jokerPattern[j] <<
"\' ";
            if (jokerPattern[j] == joker) { //если встертили
джокер
                std::cout << "is joker ";
                if (j == 0) { //для первого символа
                    std::cout << ", it's first symbol in pattern"
<< std::endl;
                    counter = 0;
                    isPrevJoker = true;
                }
                else if (isPrevJoker) { //если перед тек джокером
был джокер
                    std::cout << ", previous is joker " <<
std::endl;
                    cur = 0;
                    counter = 0;
                }
            }
        }
    }
}

```

```

        else { //делаем последний символ не джокер
терминальной вершиной и добавляем в бор
            std::cout << ", make current vertex terminal"
<< std::endl;

            isPrevJoker = true;
            vertexes[cur].isTerminal = true;
            vertexes[cur].posInJokerPattern.push_back(j -
counter);

            if (vertexes[cur].level == 0) {
                vertexes[cur].level = counter;
            }
            counter = 0;
            cur = 0;
            std::cout << "Now current = 0" << std::endl;
        }
    }
    else {
        isPrevJoker = false;
        counter++; //увеличиваем длину тек подстроки
        std::cout << " is not joker" << std::endl;
        if
(vertexes[cur].nextVertexes.find(jokerPattern[j]) ==
vertexes[cur].nextVertexes.end()) { //если нет потомка по тек
СИМВОЛУ
            std::cout << ", no vertexes from current by
symbol: " << jokerPattern[j] << ", so add new vertex with index: "
<< vertexes.size() << std::endl;
            Vertex vert(cur, jokerPattern[j]);
            vertexes.push_back(vert);
            vertexes[cur].nextVertexes[jokerPattern[j]] =
vertexes.size() - 1;
        }
        cur = vertexes[cur].nextVertexes[jokerPattern[j]];
        std::cout << "Go to new vertex new current vertex
index: " << cur << std::endl;
    }
}
    if (!isPrevJoker) { //если перед последним был джокер
либо последний джокер
        if (vertexes[cur].level == 0) {
            vertexes[cur].level = counter;
        }
        vertexes[cur].isTerminal = true;

vertexes[cur].posInJokerPattern.push_back(jokerPattern.length() -
counter);

        std::cout << "Make pattern end terminal" << std::endl;
    }
}

int getSuffixLink(int i) { //получение суффиксальной ссылки
для вершины

```

```

        std::cout << "Get suffix link of vertex with index: " << i
<< std::endl;
        if (vertexes[i].suffixLink == -1) { //если еще нет
суффиксальной
            std::cout << "Vertex doesn't have suffix link" <<
std::endl;
            if (i == 0 || vertexes[i].parentNode.second == 0) {
//если корень или его потомок, то ссылка на корень
                std::cout << "Vertex is root or root child it's
suffix link to root" << std::endl;
                vertexes[i].suffixLink = 0;
            }
            else { //если нет то ищем путь из суффиксальной
вершины родителя, по символу от родителя к тек вершине
                std::cout << "To find suffix link go to parent
vertex and check it's suffix link" << std::endl;
                vertexes[i].suffixLink =
getNextVertex(getSuffixLink(vertexes[i].parentNode.second),
vertexes[i].parentNode.first);
            }
        }
        std::cout << "Suffix link to vertex with index: " << i <<
" is " << vertexes[i].suffixLink << std::endl;
        return vertexes[i].suffixLink;
    }

    int getNextVertex(int i, char c) { //следующий шаг автомата
        std::cout << "Find next move for vertex with index: " << i
<< " by symbol: \' " << c << "\' "<< std::endl;
        if (vertexes[i].nextNode.find(c) ==
vertexes[i].nextNode.end()) { //если нет пути в словаре путей
автомата по переданному символу
            std::cout << "No next move by symbol: \' " << c << "\' "
<< std::endl;
            if (vertexes[i].nextVertexes.find(c) !=
vertexes[i].nextVertexes.end()) { //если есть прямая ссылка
                std::cout << "There is forward move to next vertex
by symbol: \' " << c << "\' to vertex with index: " <<
vertexes[i].nextVertexes[c] << std::endl;
                vertexes[i].nextNode[c] =
vertexes[i].nextVertexes[c]; //то добавляем ее в путь
            }
            else {
                if (i == 0) { //если корень и нет потомков с
путем по переданному символу, то ссылка на корень
                    std::cout << "It is root vertex without child
by symbol: \' " << c << "\' so next move is root"<< std::endl;
                    vertexes[i].nextNode[c] = 0;
                }
                else { //в противном случае добавляем в словарь
след вершину из суффиксальной ссылки
                    std::cout << "Next move by suffix link" <<
std::endl;

```

```

        vertexes[i].nextNode[c] =
getNextVertex(getSuffixLink(i), c);
    }
}
}
std::cout << "Next move vertex with index: " <<
vertexes[i].nextNode[c] << " by symbol \'' << c << \"\'" <<
std::endl;
return vertexes[i].nextNode[c];
}

void findAllJokerPatternsOnText(std::string& text) { //поиск
    std::cout << "Strat to find pattern in text: " << text <<
std::endl;
    std::vector<int> foundedForSymbols(text.size() );
    int cur = 0;
    int numOfFoundedStr = 0; //количество найденных подстрок
    for (auto & vertex : vertexes) {
        if (vertex.isTerminal) {
            for (int j = 0; j <
vertex.posInJokerPattern.size(); j++) {
                numOfFoundedStr++;
            }
        }
    }
    std::cout << "Number of founded substrings: " <<
numOfFoundedStr << std::endl;
    for (int i = 0; i < text.length(); i++){ //проходимся по
всем символам текста
        cur = getNextVertex(cur, text[i]); //идем по тек
символу по бору
        std::cout << "Current vertex index: " << cur <<
std::endl;
        for (int j = cur; j != 0; j = getSuffixLink(j)) { //
возвращаемся по суффиксальным в корень
            std::cout << "Start searching terminal by suffix
link from current vertex" << std::endl;
            if (vertexes[j].isTerminal) { //если нашли
терминальную
                std::cout << "Terminal was founded" <<
std::endl;
                int indInText = i + 1 - vertexes[j].level;
                for (int k = 0; k <
vertexes[j].posInJokerPattern.size(); k++){ //по вычисленному
месту тек подстроки и добавляем в вектор числа совпадений
                    if (indInText -
vertexes[j].posInJokerPattern[k] >= 0) {
                        foundedForSymbols[indInText -
vertexes[j].posInJokerPattern[k]] ++;
                        std::cout << "Increase element at
index: " << indInText - vertexes[j].posInJokerPattern[k]

```



```

        << " in array of number of matches,
it's value: " << foundedForSymbols[indInText -
vertexes[j].posInJokerPattern[k]] << std::endl;
    }
}
}
}
std::cout << "RESULT:" << std::endl;
for (int i = 0; i < foundedForSymbols.size() -
jokerPatternSize + 1; i++) { //вывод результата
    if (foundedForSymbols[i] == numOfFoundedStr) {
        std::cout << i + 1 << std::endl;
    }
}
}
};

int main() {
    std::string text;
    std::string jokerPattern;
    char joker;
    std::cin >> text;
    std::cin >> jokerPattern;
    std::cin >> joker;
    BohrWithJoker bohrWithJoker(joker);
    bohrWithJoker.addJokerPattern(jokerPattern);
    bohrWithJoker.printBohr();
    bohrWithJoker.findAllJokerPatternsOnText(text);
}

```