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

Студентка гр. 8382 _____ Рочева А.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Разработать программу, решающую задачу точного поиска образцов в строке.

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

3 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? с джокером ? встречается дважды в тексте xabvccbababcax.

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

Индивидуализация.

Вариант 2 – Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Описание первого алгоритма.

На вход алгоритму подается текст, набор образцов и их количество.

Для работы с образцами строится бор (структура данных, представляющая из себя подвешенное дерево с символами на ребрах). Это происходит в методе addPattern класса Bohr. Образец добавляется в бор посимвольно, если переход по следующему символу не существует, то создается новая вершина бора с этим символом на ведущем к нему ребру. Каждые вершины нумеруются, корень имеет номер 0.

Для работы с бором созданы методы, вычисляющие и возвращающие суффиксные ссылки для вершин, сжатые ссуффиксные ссылки и переходы по определенному символу (getSuffLink(), getUpLink() и getAuroLink() соответсвенно).

Поиск образцов в тексте происходит в функции processText(), где по очереди просматриваются символы текста. Начальное состояние находится в вершине 0. Для очередного символа переходим из текущего состояния в состояние, которое вернет метод getAutoLink(), т.е. в качестве нового состояния рассматривается вершина, в которую есть переход в автомате из текущей вершины (если в автомате отсутствует переход из вершины по данной букве, то нужно вычислить суффиксную ссылку этой вершины и искать переход из новой вершины по этой букве). Из нового состояния происходит переход по сжатым суффиксным ссылкам, пока очередная ссылка не окажется ссылкой на корень. При этом проверяется каждое посещенное состояние, и если оно является терминальным для какого-то образца, то номер этого образца и его позиция заносятся в список результатов, который потом и выводится.

Подсчет вершин и нахождение пересекающихся образцов происходит в методе printResVar(). Сразу выводится размер вектора вершин, означающим количество вершин в автомате. Нахождение пересекающихся образцов происходит в циклах, где для каждого индекса вхождения какого-то образца рассматриваются индексы, следующие за ним. Если длина образца, начинающегося в текущем индексе, превосходит или равно расстоянию между текущим индексом и следующими, то образцы этих индексов выводятся (текущий и тот, индекс которого перекрывается).

Описание структур данных для первого алгоритма.

1. *Struct BohrVertex* – структура для представления вершины бора. Содержит поля: vector<int> nextVert – сыновья вершины, vector<int> goArray – массив переходов, int suffixLink – суффиксная ссылка на какую-то вершину (изначально -1), int up – сжатая суффиксная ссылка на какую-то вершину (изначально -1), int parent – индекс вершины-родителя, char toParent – символ на ребре до родителя, int numVert – номер вершины, int numPattern – номер паттерна, для которого эта вершина является терминальной, bool isFinal – флаг, показывающий, является ли вершина терминальной.
2. *map<char, int>bohrAlp* – словарь алфавита, который используется в алгоритме.
3. *map<int, vector<int>> result* – словарь для записи результатов. Ключом является индекс в тексте, значениями – номера образцов, которые начинаются с этого индекса.
4. *vector<BohrVertex*> bohr* – вектор из вершин для представления бора.
5. *map<int, string> patterns* – словарь для хранения образцов. Ключом является номер вершины, в которой оканчивается образец, значением – сам образец.
6. *map<int, string> numPatterns* – словарь, в котором ключом является номер образца, а значением – сам образец.

Описание функций для работы с первым алгоритмом.

1. *BohrVertex *createVertex(int parent, char toParent)* – функция создания вершины. Аргументы: *int parent,* – номер вершины-родителя, *char toParent* – символ на ребре до вершины родителя. Функция возвращает указатель на созданную вершину.
2. *void addPattern(string &pattern)* – функция, добавляющая образец в бор. Аргументы: *string &pattern* – ссылка на образец.
3. *int getAutoLink(int vertex, char path)* – функция, вычисляющая переход в автомате из вершины *vertex* по символу *path.* Аргументы: *int vertex* – номер вершины, из которой вычисляется переход, *char path* – символ, по которому вычисляется переход. Возвращает номер вершины, в которую нужно сделать переход.
4. *int getUpLink(int vertex)* – функция, вычисляющая сжатую суффиксную ссылку для вершины *vertex.* Аргументы: *int vertex* – номер вершины, для которой нужно найти сжатую ссылку. Функция возвращает вершину, являющуюся сжатой ссылкой для заданной вершины.
5. *int getSuffLink(int vertex)* – функция, вычисляющая суффиксную ссылку для вершины *vertex.* Аргументы: *int vertex* – номер вершины, для которой нужно найти суффиксную ссылку ссылку. Функция возвращает вершину, являющуюся суффиксной ссылкой для заданной вершины.
6. *void processText(string &text)* – функция, в которой ведется поиск вхождений образцов в текст. Аргументы: *string &text* – ссылка на текст.
7. *void printResVar(map<int, vector<int>> &result)* – функция, находящая кол-во вершин в автомате и пересекающиеся образцы. Аргументы: *map<int, vector<int>> &result* - словарь результата.
8. void setJoker(char joker) – функция, устанавливающая джокер. Аргументы: char joker – символ джокера.

Сложность первого алгоритма.

Сложность по операциям: построение бора происходит за O(m), где m — суммарная длина всех образцов, т. к. алгоритм последовательно обрабатывает каждый символ образца. Тогда общая сложность O(N+t+m*k), где N — длина текста, в котором происходит поиск (символы рассматриваются последовательно), t — количество всех возможных вхождений образцов в текст, k — размер алфавита.

Сложность по памяти: всего в боре хранится не более m вершин, где m — суммарная длина всех образцов, поэтому сложность будет O(m*k), где k — количество символов алфавита.

Описание второго алгоритма.

Этот алгоритм очень схож с предыдущим. Строка-образец делится на новые образцы по разделителю-джокеру и, как и в первом алгоритме, происходит поиск вхождений этих образцов в строку. При нахождении очередного образца значение arr[index-position(pattern)+1-size(pattern)] увеличивается на 1. После обработки текста просматривается массив с результатом arr, где каждый index, для которого arr[index] равно количеству образцов, является стартовой позицией появления образца с джокером в тексте. В конце выводится только количество вершин в автомате. Пересекающиеся образцы не выводятся, т. к. общий образец по условию только один.

Описание структур данных для второго алгоритма.

1. *Struct BohrVertex* – структура для представления вершины бора. Содержит поля: vector<int> nextVert – сыновья вершины, vector<int> goArray – массив переходов, int suffixLink – суффиксная ссылка на какую-то вершину (изначально -1), int up – сжатая суффиксная ссылка на какую-то вершину (изначально -1), int parent – индекс вершины-

родителя, char toParent – символ на ребре до родителя, int numVert – номер вершины, int numPattern – номер паттерна, для которого эта вершина является терминальной, bool isFinal – флаг, показывающий, является ли вершина терминальной.

2. *map<char, int>bohrAlp* – словарь алфавита, который используется в алгоритме.
3. *vector<int> arr* – вектор результата. Размер равен размеру текста.
4. *vector<BohrVertex*> bohr* – вектор из вершин для представления бора.
5. *map<int, string> patterns* – словарь для хранения образцов. Ключом является номер вершины, в которой оканчивается образец, значением – сам образец.
6. *vector<string> patterns* – вектор с образцами.
7. *vector<int> positions* – вектор с позициями разделенных образцов в общем образце.

Описание функций.

1. *BohrVertex *createVertex(int parent, char toParent)* – функция создания вершины. Аргументы: *int parent,* – номер вершины-родителя, *char toParent* – символ на ребре до вершины родителя. Функция возвращает указатель на созданную вершину.
2. *void addPattern(string &pattern)* – функция, добавляющая образец в бор. Аргументы: *string &pattern* – ссылка на образец.
3. *int getAutoLink(int vertex, int index)* – функция, вычисляющая переход в автомате из вершины *vertex* по символу с номером *index.* Аргументы: *int vertex* – номер вершины, из которой вычисляется переход, *int index* – номер символа, по которому вычисляется переход. Возвращает номер вершины, в которую нужно сделать переход.
4. *int getUpLink(int vertex)* – функция, вычисляющая сжатую суффиксную ссылку для вершины *vertex.* Аргументы: *int vertex* – номер вершины,

для которой нужно найти сжатую ссылку. Функция возвращает вершину, являющуюся сжатой ссылкой для заданной вершины.

5. *int getSuffLink(int vert)* – функция, вычисляющая суффиксную ссылку для вершины *vert.* Аргументы: *int vert* – номер вершины, для которой нужно найти суффиксную ссылку ссылку. Функция возвращает вершину, являющуюся суффиксной ссылкой для заданной вершины.
6. *void processText(string &text)* – функция, в которой ведется поиск вхождений образцов в текст. Аргументы: *string &text* – ссылка на текст.
7. *void setJoker(char joker)* – функция, устанавливающая джокер. Аргументы: *char joker* – символ джокера.

Сложность второго алгоритма.

Сложность по операциям: построение бора происходит за O(m), где m — суммарная длина всех образцов, т. к. алгоритм последовательно обрабатывает каждый символ образца. Тогда сложность без прохода по массиву arr O(N+t+m*k), где N — длина текста, в котором происходит поиск (символы рассматриваются последовательно), t — количество всех возможных вхождений образцов в текст, k — размер алфавита. Массив с результатом arr просматривается за O(N), т. е. общая сложность O(2N+t+m*k)

Сложность по памяти: всего в боре хранится не более m вершин, где m — суммарная длина всех образцов, поэтому сложность будет O(m*k+N), где k — количество символов алфавита, N — размер массива с результатом.

Тестирование

Первый алгоритм (без вывода промежуточных данных):

Входные данные	Выходные данные
ACGACTNCGACGANC 4 AC ACGAC NC	Result: 1 1 1 2 2 4 4 1

CGAC	7 3 8 4 10 1 14 3 Count of vertices: 12 Intersecting patterns: AC & CGAC (1 & 4) ACGAC & CGAC (2 & 4) ACGAC & AC (2 & 1) CGAC & AC (4 & 1) NC & CGAC (3 & 4) CGAC & AC (4 & 1) (не вывод: казалось бы, что некоторые пары повторяются, но нет, у этих пар разные индексы пересечения)
ACACA 2 AC CA	Result: 1 1 2 2 3 1 4 2 Count of vertices: 5 Intersecting patterns: AC & CA (1 & 2) AC & AC (1 & 1) CA & AC (2 & 1) CA & CA (2 & 2) AC & CA (1 & 2)
AACGCNAGGNCCGA 3 AC AG NC	Result: 2 1 7 2 10 3 Count of vertices: 6 Intersecting patterns:

(с выводом промежуточных данных):

Входные данные	Выходные данные
ACNA 1 NA	Add pattern [NA] to bohr: Created vertex 1 for edge N Created vertex 2 for edge A Finding patterns in text [ACNA] current vertex in bohr = root finding autolink for vertex 0 with path A get autolink to root vertex 0 has autolink 0with pathA current vertex in bohr = 0

	Search with this start vertex: finding autolink for vertex 0 with path C get autolink to root vertex 0 has autolink 0with pathC current vertex in bohr = 0 Search with this start vertex: finding autolink for vertex 0 with path N get autolink to vertex 1 vertex 0 has autolink 1with pathN current vertex in bohr = 1 Search with this start vertex: finding compressed link for vertex 1 finding suffix link for vertex 1 get link to root suffix link for 1 = 0 get compressed link to root compressed link for 1 = 0 finding autolink for vertex 1 with path A get autolink to vertex 2 vertex 1 has autolink 2with pathA current vertex in bohr = 2 Search with this start vertex: >>final state for pattern [NA] was found start index in text: 3 finding compressed link for vertex 2 finding suffix link for vertex 2 get link through parent finding suffix link for vertex 1 suffix link for 1 = 0 finding autolink for vertex 0 with path A vertex 0 has autolink 0with pathA suffix link for 2 = 0 get compressed link to root compressed link for 2 = 0 end of text Bohr: From root to vertex 1 with edge N From vertex 1 to vertex 2 with edge A From vertex 2 Used suffix link: From root From vertex 1 to vertex 0

	From vertex 2 to vertex 0 Result: 3 1 Count of vertices: 3 Intersecting patterns:

Второй алгоритм (без вывода промежуточных данных):

Входные данные	Выходные данные
ANAGAGCACGAGN ??AG? ?	Result: 1 3 9 Count of vertices: 3
AGAGAGAGA A*A *	Result: 1 3 5 7 Count of vertices: 2
NCANCNNNNANNN NxNxN x	Result: 4 7 9 Count of vertices: 2
ACGANCGACGNCAGC A**AN *	Result: 1 Count of vertices: 3

(с выводом промежуточных данных):

Входные данные	Выходные данные
ACTAN A* *	Add pattern [A] to bohr: Created vertex 1 for edge A Finding split patterns in text [ACTAN] finding autolink for vertex 0 get autolink to vertex 1 vertex 0 has autolink 1 >>final state for pattern [A] was found start index in text: 1 finding compressed link for vertex 1 finding suffix link for vertex 1 get link to root suffix link for 1 = 0

	get compressed link to root compressed link for 1 = 0 finding autolink for vertex 1 finding suffix link for vertex 1 suffix link for 1 = 0 finding autolink for vertex 0 get autolink to root vertex 0 has autolink 0 vertex 1 has autolink 0 finding autolink for vertex 0 get autolink to root vertex 0 has autolink 0 finding autolink for vertex 0 vertex 0 has autolink 1 >>final state for pattern [A] was found start index in text: 4 finding compressed link for vertex 1 compressed link for 1 = 0 finding autolink for vertex 1 finding suffix link for vertex 1 suffix link for 1 = 0 finding autolink for vertex 0 get autolink to root vertex 0 has autolink 0 vertex 1 has autolink 0 Result: 1 4 Count of vertices: 2

Выводы.

В ходе выполнения работы были разработаны программы, выполняющие поиск вхождений образцов в текст (приложение А) и поиск вхождения образца с джокером в текст (приложение В).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ФАЙЛА MAIN1.CPP

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

using std::vector;
using std::map;
using std::string;
using std::cin;
using std::cout;
using std::endl;
using std::pair;

#define SIZE 5

struct BohrVertex{
    vector<int> nextVert;
    vector<int> goArray;
    int suffLink = -1;
    int up = -1;
    int parent;
    char toParent;
    int numVert; // номер вершины
    int numPattern = 0;
    bool isFinal;
};

class Bohr{
public:
    Bohr(){
        bohr.push_back(createVertex(-1, '0'));
        bohrAlp = {{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} };
    }
    BohrVertex *createVertex(int parent, char toParent){
        auto *vertex = new BohrVertex;
        vertex->nextVert.assign(SIZE, -1);
        vertex->goArray.assign(SIZE, -1);
        vertex->parent = parent;
        vertex->toParent = toParent;
        vertex->numVert = bohr.size();
        vertex->suffLink = -1;
        return vertex;
    }
```

```
    void addPattern(string &pattern){
        cout << "Add pattern [" << pattern << "] to bohr:" << endl;
        int currentVert = 0;
        for (auto & symb : pattern){
            if (bohr[currentVert]->nextVert[bohrAlp[symb]] == -1){
                auto *newVertex = createVertex(currentVert, symb);
                newVertex->isFinal = false;
                bohr.push_back(newVertex);
                bohr[currentVert]->nextVert[bohrAlp[symb]] = bohr.size()-
1;
                cout << "Created vertex " << newVertex->numVert << " for
edge " << symb << endl;
            }
            currentVert = bohr[currentVert]->nextVert[bohrAlp[symb]];
        }
        patterns[currentVert] = pattern;
        bohr[currentVert]->numPattern = patterns.size();
        numPatterns[patterns.size()] = pattern;
        bohr[currentVert]->isFinal = true;
    }

    int getAutoLink(int vertex, char path){
        cout << "finding autolink for vertex " << vertex << " with path "
<< path << endl;
        if (bohr[vertex]->goArray[bohrAlp[path]] == -1){
            if (bohr[vertex]->nextVert[bohrAlp[path]] != -1){
                cout << "get autolink to vertex " << bohr[vertex]-
>nextVert[bohrAlp[path]] << endl;
                bohr[vertex]->goArray[bohrAlp[path]] = bohr[vertex]-
>nextVert[bohrAlp[path]];
            }
            else if (bohr[vertex]->numVert == 0){
                cout << "get autolink to root" << endl;
                bohr[vertex]->goArray[bohrAlp[path]] = 0;
            }
            else {
                bohr[vertex]->goArray[bohrAlp[path]] =
getAutoLink(getSuffLink(vertex), path);
            }
        }
        cout << "vertex " << vertex << " has autolink " << bohr[vertex]-
>goArray[bohrAlp[path]] << "with path" << path << endl;
        return bohr[vertex]->goArray[bohrAlp[path]];
    }

    int getUpLink(int vertex){
        cout << "finding compressed link for vertex " << vertex << endl;
        if (bohr[vertex]->up == -1) {
```

```
            int vert = getSuffLink(vertex);
            if (vert == 0) {
                cout << "get compressed link to root" << endl;
                bohr[vertex]->up = 0;
            }
            else {
                if (bohr[vert]->isFinal){
                    bohr[vertex]->up = vert;
                    cout << "get compressed link to " << vert << endl;
                }
                else {
                    cout << "get compressed link of vertex " << vert <<
endl;
                    bohr[vertex]->up = getUpLink(vert);
                }
            }
        }
        cout << "compressed link for " << vertex << " = " << bohr[vertex]-
>up << endl;
        return bohr[vertex]->up;
    }

    int getSuffLink(int vertex){
        cout << "finding suffix link for vertex " << vertex << endl;
        if (bohr[vertex]->suffLink == -1){
            if (bohr[vertex]->numVert == 0 || bohr[vertex]->parent == 0){
                cout << "get link to root" << endl;
                bohr[vertex]->suffLink = 0;
            }
            else {
                cout << "get link through parent" << endl;
                bohr[vertex]->suffLink =
getAutoLink(getSuffLink(bohr[vertex]->parent), bohr[vertex]->toParent);
            }
        }
        cout << "suffix link for " << vertex << " = " << bohr[vertex]-
>suffLink << endl;
        return bohr[vertex]->suffLink;
    }

    void processText(string &text){
        cout << endl << "Finding patterns in text [" << text << "]" <<
endl;
        map<int, vector<int>> result; // для сортировки еще
        cout << "current vertex in bohr = root" << endl;
        int currentVertex = 0;
        int index = 0;
        for (auto & symb : text){
```

```
            currentVertex = getAutoLink(currentVertex, symb);
            cout << endl << "current vertex in bohr = " << currentVertex
<< endl;
            int i = index;
            cout << "Search with this start vertex:" << endl;
            for (int vert = currentVertex; vert != 0; vert =
getUpLink(vert)){
                if (bohr[vert]->isFinal) {
                    cout << ">>final state for pattern [" <<
numPatterns[bohr[vert]->numPattern] << "] was found" << endl;
                    cout << "start index in text: " << i + 1 -
patterns[vert].size() + 1 << endl << endl;
                    result[i + 1 - patterns[vert].size() +
1].push_back(bohr[vert]->numPattern);
                    i--;
                }
                i++;
            }
            index++;
        }
        cout << "end of text" << endl;

        cout << endl << "Bohr:" << endl;
        for (auto & vert : bohr){
            cout << "From ";
            vert->numVert == 0 ? cout << "root" : cout << "vertex " <<
vert->numVert;
            cout << endl;
            for (auto & next : vert->nextVert){
                if (next == -1)
                    continue;
                cout << "\tto vertex " << next << " with edge " <<
bohr[next]->toParent << endl;
            }
        }
        cout << endl << "Used suffix link:" << endl;
        for (auto & vert : bohr){
            cout << "From ";
            vert->numVert == 0 ? cout << "root" : cout << "vertex " <<
vert->numVert;
            cout << endl;
            if (vert->suffLink == -1)
                continue;
            cout << "\tto vertex " << vert->suffLink << endl;
        }

        cout << endl << "Result:" << endl;
        for (auto & res : result){
```

```
            sort(res.second.begin(), res.second.end());
            for (auto & num : res.second){
                cout << res.first << " " << num << endl;
            }
        }

        printResVar(result);
    }

    void printResVar(map<int, vector<int>> &result){

        cout << "Count of vertices: " << bohr.size() << endl;
        cout << "Intersecting patterns:" << endl;
        for (auto & res : result){
            for (auto & first : res.second) {
                for (auto & newRes : result) {
                    if (res.first >= newRes.first){
                        continue;
                    }
                    if (res.first + numPatterns[first].size() >=
newRes.first){
                        for (auto & second : newRes.second){
                            cout << numPatterns[first] << " & " <<
numPatterns[second] << " (" << first << " & " << second << ")" << endl;
                        }
                    }
                }
            }
        }
    }

private:
    vector<BohrVertex*> bohr;
    map<int, string> patterns; // номер вершины – паттерн
    map<int, string> numPatterns;
    map<char, int> bohrAlp;
};

int main(){
    Bohr *bohr = new Bohr;
    string text;
    cin >> text;
    int num;
    cin >> num;
    vector<string> patterns;
    while (num){
        string pattern;
        cin >> pattern;
```

```
        patterns.push_back(pattern);
        num--;
    }
    for (auto & pattern : patterns){
        bohr->addPattern(pattern);
    }
    bohr->processText(text);
    return 0;
}
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД ФАЙЛА MAIN2.CPP

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

using std::vector;
using std::map;
using std::string;
using std::cin;
using std::cout;
using std::endl;
using std::pair;

#define SIZE 5

// структура вершины
struct BohrVertex{
    vector<int> nextVerts;
    vector<int> goArray;
    int parent;
    vector<int> patternNumber;
    int suffLink;
    int up;
    char toParent;
    bool isFinal;
};


class Bohr{
public:
    Bohr(){
        bohrAlp = {{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} };
        bohr.push_back(createVertex(0, '0'));
        wordLen = 0;
        countPatterns = 0;
    }

    void setJoker(char joker){
        this->joker = joker;
    }

    BohrVertex *createVertex(int parent, char toParent){
        auto *newVertex = new BohrVertex;
        newVertex->nextVerts.assign(SIZE, -1);
        newVertex->goArray.assign(SIZE, -1);
```

```
        newVertex->suffLink = -1;
        newVertex->up = -1;
        newVertex->parent = parent;
        newVertex->toParent = toParent;
        newVertex->isFinal = false;
        return newVertex;
    }

    void addPattern(string &text){

        int index = 0;
        int count = 0;
        wordLen = text.size();

        // split
        for (int i = 0; i < text.length(); i = index){
            string buff = "";
            while (index < text.length() && text[index] == joker)
                index++;
            if (index == text.length())
                break;
            int pos = index;
            while (index < text.length() && text[index] != joker)
                buff += text[index++];
            if (!buff.empty()){
                count++;
                positions.push_back(pos);
                patterns.push_back(buff);
            }
        }
        // create bohr
        for (auto & pattern : patterns){
            cout << "Add pattern [" << pattern << "] to bohr:" << endl;
            int j = 0;
            for (char i : pattern){
                if (bohr[j]->nextVerts[bohrAlp[i]] == -1){
                    auto *newVert = createVertex(j, i);
                    bohr.push_back(newVert);
                    bohr[j]->nextVerts[bohrAlp[i]] = bohr.size() - 1;
                    cout << "Created vertex " << bohr.size()-1 << " for
edge " << i << endl;
                }
                j = bohr[j]->nextVerts[bohrAlp[i]];
            }
            bohr[j]->isFinal = true;
            bohr[j]->patternNumber.push_back(countPatterns);
            countPatterns++;
        }
```

```
    }

    int getSuffLink(int vert){
        cout << "finding suffix link for vertex " << vert << endl;
        if (bohr[vert]->suffLink == -1) {
            if (vert == 0 || bohr[vert]->parent == 0) {
                cout << "get link to root" << endl;
                bohr[vert]->suffLink = 0;
            }
            else {
                cout << "get link through parent" << endl;
                bohr[vert]->suffLink = getAutoLink(getSuffLink(bohr[vert]-
>parent), bohrAlp[bohr[vert]->toParent]);
            }
        }
        cout << "suffix link for " << vert << " = " << bohr[vert]-
>suffLink << endl;
        return bohr[vert]->suffLink;
    }

    int getAutoLink(int vertex, int index){
        cout << "finding autolink for vertex " << vertex << endl;
        if (bohr[vertex]->goArray[index] == -1){
            if (bohr[vertex]->nextVerts[index] != -1) {
                cout << "get autolink to vertex " << bohr[vertex]-
>nextVerts[index] << endl;
                bohr[vertex]->goArray[index] = bohr[vertex]-
>nextVerts[index];
            }
            else {
                if (vertex == 0){
                    cout << "get autolink to root" << endl;
                    bohr[vertex]->goArray[index] = 0;
                }
                else {
                    bohr[vertex]->goArray[index] =
getAutoLink(getSuffLink(vertex), index);
                }
            }
        }
        cout << "vertex " << vertex << " has autolink " << bohr[vertex]-
>goArray[index] << endl;
        return bohr[vertex]->goArray[index];
    }

    int getUpLink(int vertex){
        cout << "finding compressed link for vertex " << vertex << endl;
```

```
        if (bohr[vertex]->up == -1){
            int vert = getSuffLink(vertex);
            if (bohr[vert]->isFinal) {
                cout << "get compressed link to " << vert << endl;
                bohr[vertex]->up = vert;
            }
            else {
                if (vert == 0) {
                    cout << "get compressed link to root" << endl;
                    bohr[vertex]->up = 0;
                }
                else {
                    cout << "get compressed link of vertex " << vert <<
endl;
                    bohr[vertex]->up = getUpLink(vert);
                }
            }
        }
        cout << "compressed link for " << vertex << " = " << bohr[vertex]-
>up << endl;
        return bohr[vertex]->up;
    }

    void processText(string& text){
        for (int i = 0; i < text.size(); i++)
            arr.push_back(0);
        cout << endl << "Finding split patterns in text [" << text << "]"
<< endl;
        int currentVert = 0;
        for (int i = 0; i < text.length(); i++){
            currentVert = getAutoLink(currentVert, bohrAlp[text[i]]);
            int index = i;
            for (int j = currentVert; j != 0; j = getUpLink(j)){
                if (bohr[j]->isFinal){
                    for (auto & in: bohr[j]->patternNumber){
                        cout << ">>final state for pattern [" <<
patterns[in] << "] was found" << endl;
                        cout << "start index in text: " << i + 1 -
patterns[in].size() + 1 << endl << endl;
                        int tmp = index + 1 - patterns[in].size() -
positions[in];
                        if (tmp >= 0 && tmp <= text.length() - wordLen){
                            arr[tmp]++;
                        }
                    }
                }
            }
        }
```

```
        cout << endl <<  "Result:" << endl;
        for (int i = 0; i < text.size(); i++){
            if (arr[i] == patterns.size())
                cout << i + 1 << endl;
        }

        cout << "Count of vertices: " << bohr.size() << endl;
    }


private:
    vector<BohrVertex*> bohr;
    vector<string> patterns;
    int wordLen;
    map<char, int> bohrAlp;
    char joker;
    vector<int> positions;
    int countPatterns;
    vector<int> arr;
};



int main() {
    string text;
    cin >> text;
    string pattern;
    cin >> pattern;
    char joker;
    cin >> joker;

    Bohr bohr;
    bohr.setJoker(joker);
    bohr.addPattern(pattern);
    bohr.processText(text);

    return 0;
}
```