МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8382 _____ Щеглов А.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Разработать программу для точного поиска вхождения набора образцов в тесте с использованием алгоритма Ахо-Корасик.

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n ($1 \leq n \leq 3\ 000$), каждая следующая из n строк содержит шаблон из набора $P = \{p1, \ldots, pn\}$, $1 \leq |pi| \leq 75$. 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 PP необходимо найти все вхождения PP в текст TT.

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

Символ джокер не входит в алфавит, символы которого используются в TT. 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

Вариант дополнительного задания.

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

Описание алгоритма

Задание 1.

Алгоритм Ахо-Корасик позволяет за линейное время найти все позиции вхождений набора образцов в тексте. Из образцов строится бор. После построения бора происходит итерация по символам текста и, если из текущего состояния автомата можно перейти дальше по текущему символу текста, то происходит переход по ребру, иначе – по суффиксной ссылке. Суффиксная ссылка указывает на узел, путь до которого представляет наибольший суффикс полученного пути. Если на очередной итерации по тексту было достигнуто терминальное состояние автомата – данная позиция означает окончание вхождение данного образца.

Задание 2.

Алгоритм Ахо-Корасик может быть использован для поиска по тексту вхождений шаблона со специальным символом (джокером), означающим любой символ алфавита. Для этого необходимо разбить шаблон по джокеру на подстроки, запомнить для каждой подстроки её смещение в исходном шаблоне. Затем найти индексы вхождения всех подстрок в тексте с учётом полученного смещения. Индекс, в котором число найденных подстрок равно их количеству, является индексом вхождения шаблона в тексте. Исходный код программы для поиска шаблона с джокером расположен в Приложении Б.

Пояснение к алгоритму и индивидуальному заданию:

Бор (англ. trie, луч, нагруженное дерево) — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах.

Самая длинная цепочка из конечных ссылок в автомате – это самая длинная цепочка сжатых суффиксных ссылок.

Сжатая суффиксная ссылка

parent(u) — возвращает родителя вершины u;

$\pi(u)=\delta(\pi(parent(u)),c)$ — суффиксная ссылка, и существует переход из parent(u) в u по символу c;

$\delta(u,c)=v$, если в v можно перейти по символу c . — функция перехода.

$\delta(u,c)=root$, если u корень и у него нет потомка с переходом по символу c.

$\delta(u,c)=\delta(\pi(u),c)$, иначе. — функция перехода.

$up(u)=\pi(u)$, если $\pi(u)$ терминальная.

$up(u)=\emptyset$, если $\pi(u)$ корень.

$up(u)= up(\pi(u))$, иначе.

где up — сжатая суффиксная ссылка, т.е. ближайшее допускающее состояние (терминал) перехода по суффиксным ссылкам.

Реализация алгоритма из задания 1.

Для реализации узла бора были написаны классы *Vertex* и *Bohr*. Vertex представляет из себя одну из вершин, имеющую предка *parentNode* , вектор вершин потомков *nextVertexes,* суффиксальную ссылку *suffixLink,* уровень вершины *level,* а также флаг терминальной вершины *isTerminal* с вектором паттернов *numbers* для которых она таковой является. У классы *Bohr* есть вектор из *Vertex* , результирующий вектор и переменные со значениями самой длинной цепочки прямых ссылок и количеством терминальных вершин. Так же у этого класса есть методы:

int getSuffixLink(int i) – получение суффиксной ссылки вершины с индексом i другую на вершину(возвращает индекс вершины)

int getNextVertex(int i, char c) – выполняет шаг автомата из вершины с индексом i по символу c

void findAllPatternsOnText(std::string& text) – находит все вхождения всех паттернов этого бора в text и сразу их выводит (так же находит максимальную цепочку прямых ссылок и суффиксальных)

void printBohr() – выводит бор в доступном для понимания виде

Реализация алгоритма из задания 2.

Для реализации узла бора были так же написаны классы *Vertex* и *BohrWithJoker*. Vertex представляет из себя одну из вершин, имеющую предка *parentNode* , вектор вершин потомков *nextVertexes*, суффиксальную ссылку *suffixLink*, уровень вершины *level*, а также флаг терминальной вершины *isTerminal* с вектором паттернов *numbers* для которых она таковой является. У классы *BohrWithJoker* есть вектор из *Vertex* , значение символа джокера, размер шаблона с джокером. Так же у этого класса есть методы:

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

int getSuffixLink(int i) – получение суффиксной ссылки вершины с индексом i другую на вершину(возвращает индекс вершины)

int getNextVertex(int i, char c) – выполняет шаг автомата из вершины с индексом i по символу c

void findAllJokerPatternsOnText(std::string& text) – находит все вхождения всех паттерна этого бора в text. Так же выводит индексы вхождений.

void printBohr() – выводит бор в доступном для понимания виде

Оценка сложности программы.

Сложность для первого алгоритма.

Память: O(n*q), где n – общая длина слов в словаре, q – Размер алфавита

Вычислительная: O(nq + H + k), где H – длина текста, k – общая длина всех совпадений.

Сложность для второго алгоритма.

Память: O(n*q), где n – общая длина слов в словаре, q – Размер алфавита

Вычислительная: O(nq + H + k), где H – длина текста, k – общая длина всех совпадений.

Тестирование с промежуточными выводами

Для программы без джокера:

```
tgtggtg
4
tgt
tg
gt
t
Vertex index: 0
Vertex is root
Next vertexes: ('g', 4) ('t', 1)

Vertex index: 1
This vertex is terminal
Next vertexes: (g, 2)

Vertex index: 2
This vertex is terminal
Next vertexes: (t, 3)

Vertex index: 3
This vertex is terminal


Vertex index: 4
Next vertexes: (t, 5)

Vertex index: 5
This vertex is terminal


Start from root

Find next move for vertex with index: 0 by symbol: 't'
No next move by symbol: 't'
There is forward move to next vertex by symbol: 't' to vertex with index: 1
Next move vertex with index: 1 by symbol 't'
Start finding terminal vertexes by suffix links:
[
]
Next current is vertex with index: 1
```

```
Find next move for vertex with index: 1 by symbol: 'g'
No next move by symbol: 'g'
There is forward move to next vertex by symbol: 'g' to vertex with index: 2
Next move vertex with index: 2 by symbol 'g'
Start finding terminal vertexes by suffix links:
[

Find next move for vertex with index: 0 by symbol: 'g'
No next move by symbol: 'g'
There is forward move to next vertex by symbol: 'g' to vertex with index: 4
Next move vertex with index: 4 by symbol 'g'
]
Next current is vertex with index: 2

Find next move for vertex with index: 2 by symbol: 't'
No next move by symbol: 't'
There is forward move to next vertex by symbol: 't' to vertex with index: 3
Next move vertex with index: 3 by symbol 't'
Start finding terminal vertexes by suffix links:
[

Find next move for vertex with index: 4 by symbol: 't'
No next move by symbol: 't'
There is forward move to next vertex by symbol: 't' to vertex with index: 5
Next move vertex with index: 5 by symbol 't'

Find next move for vertex with index: 0 by symbol: 't'
Next move vertex with index: 1 by symbol 't'
]
Next current is vertex with index: 3
```

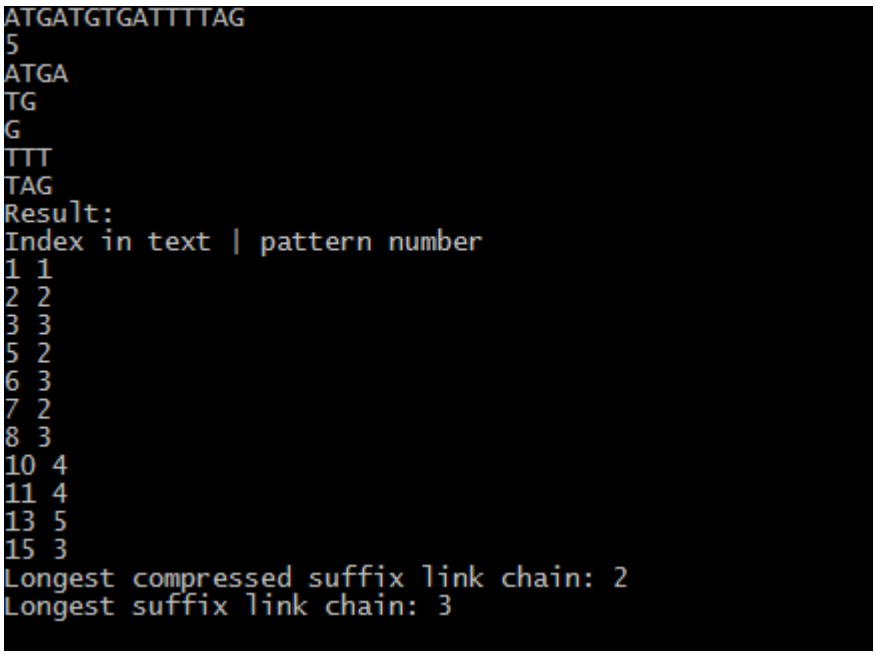
```
Find next move for vertex with index: 3 by symbol: 'g'
No next move by symbol: 'g'
Next move by suffix link
{

Find next move for vertex with index: 5 by symbol: 'g'
No next move by symbol: 'g'
Next move by suffix link
{

Find next move for vertex with index: 1 by symbol: 'g'
Next move vertex with index: 2 by symbol 'g'
}
Next move vertex with index: 2 by symbol 'g'
}
Next move vertex with index: 2 by symbol 'g'
Start finding terminal vertexes by suffix links:
[
]
Next current is vertex with index: 2

Find next move for vertex with index: 2 by symbol: 'g'
No next move by symbol: 'g'
Next move by suffix link
{

Find next move for vertex with index: 4 by symbol: 'g'
No next move by symbol: 'g'
Next move by suffix link
{

Find next move for vertex with index: 0 by symbol: 'g'
Next move vertex with index: 4 by symbol 'g'
}
Next move vertex with index: 4 by symbol 'g'
}
Next move vertex with index: 4 by symbol 'g'
Start finding terminal vertexes by suffix links:
[
]
Next current is vertex with index: 4
```

```
Find next move for vertex with index: 4 by symbol: 't'
Next move vertex with index: 5 by symbol 't'
Start finding terminal vertexes by suffix links:
[
]
Next current is vertex with index: 5

Find next move for vertex with index: 5 by symbol: 'g'
Next move vertex with index: 2 by symbol 'g'
Start finding terminal vertexes by suffix links:
[
]
Next current is vertex with index: 2
Search longest compressed and suffix link chain
Start from index = 1
Start from index = 2
Now in index = 4
Start from index = 3
Now in index = 5
Now in index = 1
Start from index = 4
Start from index = 5
Now in index = 1
Result:
Index in text | pattern number
1 1
1 2
1 4
2 3
3 2
3 4
5 3
6 2
6 4
Longest compressed suffix link chain: 2
Longest suffix link chain: 3
```

Тестирование без промежуточных данных

```
NTAG
3
TAGT
TAG
T
Result:
Index in text | pattern number
2 2
2 3
Longest compressed suffix link chain: 1
Longest suffix link chain: 2
```

```
TAGTTTT
3
TAGT
AGT
GT
Result:
Index in text | pattern number
1 1
2 2
3 3
Longest compressed suffix link chain: 2
Longest suffix link chain: 4
```



Тестирование с промежуточными выводами

Для программы с джокером:

```
ACTANCAACTANCA
AXXTANXA
X
Vertex index: 0
Vertex is root
Next vertexes: ('A', 1) ('T', 2)

Vertex index: 1
This vertex is terminal


Vertex index: 2
Next vertexes: (A, 3)

Vertex index: 3
Next vertexes: (N, 4)

Vertex index: 4
This vertex is terminal



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 finding terminal vertexes by suffix links:
[

]

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


Find next move for vertex with index: 1 by symbol: 'C'
No next move by symbol: 'C'
Next move by suffix link
{

Find next move for vertex with index: 0 by symbol: 'C'
No next move by symbol: 'C'
It is root vertex without child by symbol: 'C' so next move is root
Next move vertex with index: 0 by symbol 'C'
}
Next move vertex with index: 0 by symbol 'C'
```

```
CURRENT VERTEX INDEX: 0
Start finding terminal vertexes by suffix links:


Find next move for vertex with index: 0 by symbol: 'T'
No next move by symbol: 'T'
There is forward move to next vertex by symbol: 'T' to vertex with index: 2
Next move vertex with index: 2 by symbol 'T'

CURRENT VERTEX INDEX: 2
Start finding terminal vertexes by symbol: 'A'
[


Find next move for vertex with index: 2 by symbol: 'A'
No next move by symbol: 'A'
There is forward move to next vertex by symbol: 'A' to vertex with index: 3
Next move vertex with index: 3 by symbol 'A'

CURRENT VERTEX INDEX: 3
Start finding terminal vertexes by suffix links:
[

Find next move for vertex with index: 0 by symbol: 'A'
Next move vertex with index: 1 by symbol 'A'

]

Increase element at index: 3 in array of number of matches, it's value: 1


Find next move for vertex with index: 3 by symbol: 'N'
No next move by symbol: 'N'
There is forward move to next vertex by symbol: 'N' to vertex with index: 4
Next move vertex with index: 4 by symbol 'N'

CURRENT VERTEX INDEX: 4
Start finding terminal vertexes by suffix links:
[

]
```

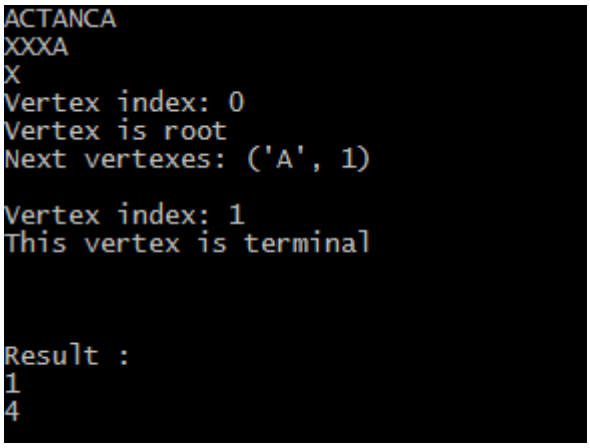
```
Find next move for vertex with index: 1 by symbol: 'N'
No next move by symbol: 'N'
Next move by suffix link
{

Find next move for vertex with index: 0 by symbol: 'N'
No next move by symbol: 'N'
It is root vertex without child by symbol: 'N' so next move is root
Next move vertex with index: 0 by symbol 'N'
}
Next move vertex with index: 0 by symbol 'N'


Find next move for vertex with index: 4 by symbol: 'C'
No next move by symbol: 'C'
Next move by suffix link
{

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

CURRENT VERTEX INDEX: 0
Start finding terminal vertexes by suffix links:


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 finding terminal vertexes by suffix links:
[

]

Increase element at index: 6 in array of number of matches, it's value: 1
```

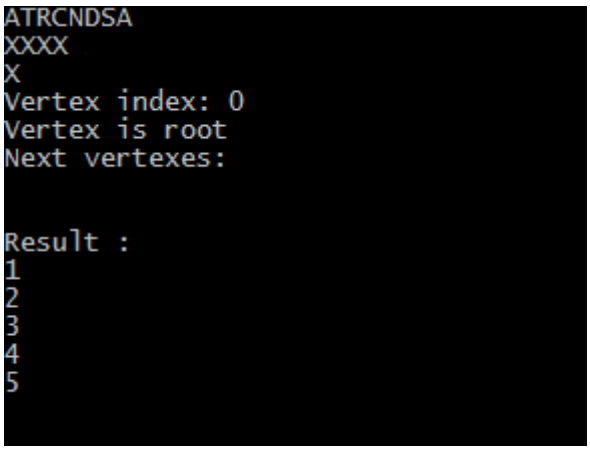
```
Find next move for vertex with index: 1 by symbol: 'A'
No next move by symbol: 'A'
Next move by suffix link
{

Find next move for vertex with index: 0 by symbol: 'A'
Next move vertex with index: 1 by symbol 'A'
}
Next move vertex with index: 1 by symbol 'A'

CURRENT VERTEX INDEX: 1
Start finding terminal vertexes by suffix links:
[

]

Increase element at index: 7 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


Find next move for vertex with index: 1 by symbol: 'C'
Next move vertex with index: 0 by symbol 'C'

CURRENT VERTEX INDEX: 0
Start finding terminal vertexes by suffix links:


Find next move for vertex with index: 0 by symbol: 'T'
Next move vertex with index: 2 by symbol 'T'

CURRENT VERTEX INDEX: 2
Start finding terminal vertexes by suffix links:
[


Find next move for vertex with index: 2 by symbol: 'A'
Next move vertex with index: 3 by symbol 'A'

CURRENT VERTEX INDEX: 3
Start finding terminal vertexes by suffix links:
[

]
```

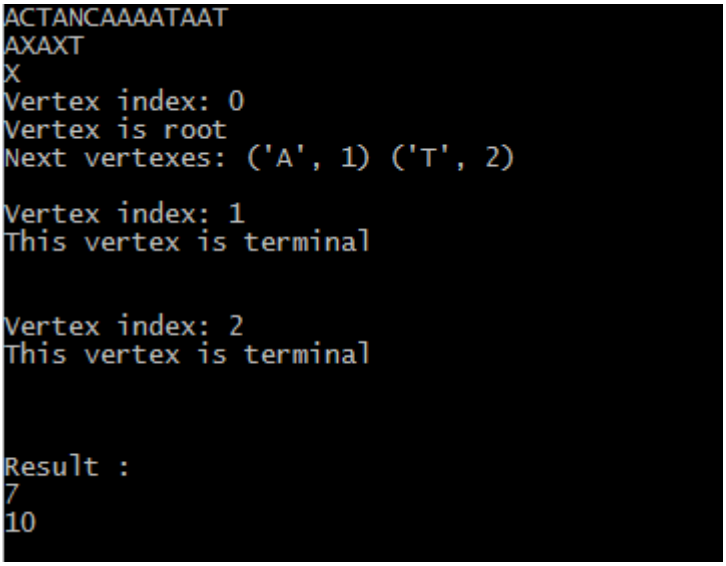
```
Increase element at index: 10 in array of number of matches, it's value: 1

Increase element at index: 3 in array of number of matches, it's value: 2


Find next move for vertex with index: 3 by symbol: 'N'
Next move vertex with index: 4 by symbol 'N'

CURRENT VERTEX INDEX: 4
Start finding terminal vertexes by suffix links:
[

]

Increase element at index: 6 in array of number of matches, it's value: 2


Find next move for vertex with index: 4 by symbol: 'C'
Next move vertex with index: 0 by symbol 'C'

CURRENT VERTEX INDEX: 0
Start finding terminal vertexes by suffix links:


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 finding terminal vertexes by suffix links:
[

]

Increase element at index: 13 in array of number of matches, it's value: 1

Increase element at index: 6 in array of number of matches, it's value: 3


Result :
7
```

Тестирование без промежуточных выводов.

```
TANANACTANA
ANXXXA
X
Vertex index: 0
Vertex is root
Next vertexes: ('A', 1)

Vertex index: 1
This vertex is terminal
Next vertexes: (N, 2)

Vertex index: 2
This vertex is terminal



Result :
4
```







Выводы.

В ходе выполнения лабораторной работы была реализована программа для поиска всех вхождения образцов в текст с использованием алгоритма АхоКорасик. Также была реализована программа для нахождения вхождений шаблона с джокером. Была получена асимптотическая оценка времени работы алгоритма.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД.

```
#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)
      {
         if (isTerminal) {
            std::cout <<"This vertex is terminal "<< 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 << "Vertex is root" << 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;  //результирующий 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
            int cur = 0;   //тек. на начало
            for (char i : str) {  //проходимся по всем символам переданной строки
                if (vertexes[cur].nextVertexes.find(i) == vertexes[cur].nextVertexes.end()) {  //если в боре нет такой
вершины
                    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
            }
            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
            }
        }

        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
            if (vertexes[i].suffixLink == -1) { //если еще нет суффиксальной
                if (i == 0 || vertexes[i].parentNode.second == 0) {  //если корень или его потомок, то ссылка на
корень
                    vertexes[i].suffixLink = 0;
                }
                else { //если нет то ищем путь из суффиксальной вершины родителя, по символу от родителя к
текущей вершине
                    vertexes[i].suffixLink = getNextVertex(getSuffixLink(vertexes[i].parentNode.second),
vertexes[i].parentNode.first);
                }
            }
            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 сиволу
                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 суффикасальной 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
        int cur = 0; // текущая 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 текущего символа до корня по
суффиксальным
                if (vertexes[j].isTerminal) { //если нашли терминальную
                    for (int k = 0; k < vertexes[j].numbers.size(); k++) {  //то добавляем в результат найденный
паттерн, а если есть одинаковые паттерны, то добаляем в результат все
                        std::pair<int, int> res(vertexes[j].numbers[k], i + 2 - vertexes[j].level);
                        result.push_back(res);
                    }
                }
            }
        }
        sort(result.begin(), result.end(), compare);   //сортировка результата
        int mostSuffixChain = 1;
        int mostCompSuffixChain = 0;
        std::cout << "Search longest compressed and suffix link chain" << std::endl;
        for (int i = 1; i < vertexes.size(); i++) {  //проходимся по каждой вершине бора
            int curSuffixChain = 1;
            int curCompSuffixChain = 0;
            int curVertex = i;
            int flag = 1;
            std::cout << "Start from index = "<<curVertex << std::endl;
            while (vertexes[curVertex].suffixLink !=0) {  //из каждой вершины проходимся по суффиксальным и
считаем длину суф цепи
                curSuffixChain++;
                curVertex = getSuffixLink(curVertex);
                std::cout << "Now in index = " << curVertex << std::endl;
                if (vertexes[curVertex].isTerminal && flag == 1)
                {
                    curCompSuffixChain++;
                }
                else
                {
                    flag = 0;
                }
            }
            if (curCompSuffixChain > mostCompSuffixChain) {
                mostCompSuffixChain = curCompSuffixChain;
            }
            if (curSuffixChain > mostSuffixChain) {
                mostSuffixChain = curSuffixChain;
            }
        }
```

```
            if (!result.empty())
            {
               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 << " " << i.first + 1 << std::endl;
               }
            }
            else
            {
               std::cout << "Patterns not founded in text" << std::endl;
            }
            std::cout << "Longest compressed suffix link chain: " << mostCompSuffixChain << 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);
         }
         bohr.printBohr();
         bohr.findAllPatternsOnText(text);
      }
```

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД.

```
      #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)


            {
                if (isTerminal) {
                    std::cout << "This vertex is terminal " << 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 << "Vertex is root" << 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
            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
                if (jokerPattern[j] == joker) {  //если встретили джокер
                    if (j == 0) {  //для первого символа
```

```
                counter = 0;
                isPrevJoker = true;
            }
            else if (isPrevJoker) { //если перед тек джокером был джокер
                cur = 0;
                counter = 0;
            }
            else { //делаем последний символ не джокер терминальной вершиной и добавляем в бор
                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;
            }
        }
        else {
            isPrevJoker = false;
            counter++;  //увеличиваем длину текущей подстроки
            if (vertexes[cur].nextVertexes.find(jokerPattern[j]) == vertexes[cur].nextVertexes.end()) {   //если нет
потомка по текущему символу
                Vertex vert(cur, jokerPattern[j]);
                vertexes.push_back(vert);
                vertexes[cur].nextVertexes[jokerPattern[j]] = vertexes.size() - 1;
            }
            cur = vertexes[cur].nextVertexes[jokerPattern[j]];
        }
    }
    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);
      }
}

int getSuffixLink(int i) {   //получение суффиксальной ссылки для вершины
    if (vertexes[i].suffixLink == -1) { //если еще нет суффиксальной
        if (i == 0 || vertexes[i].parentNode.second == 0) {  //если корень или его потомок, то ссылка на
корень
            vertexes[i].suffixLink = 0;
        }
        else { //если нет то ищем путь из суффиксальной вершины родителя, по символу от родителя к тек
вершине
            vertexes[i].suffixLink = getNextVertex(getSuffixLink(vertexes[i].parentNode.second),
vertexes[i].parentNode.first);
        }
    }
    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 сиволу
        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 суффикасальной 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::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++;
                    }
                }
            }
            for (int i = 0; i < text.length(); i++) {  //проходимся по всем симвоалм такста
                cur = getNextVertex(cur, text[i]);  //идем по текущему символу по бору
                std::cout << std::endl<< "Current vertex index: " << cur << std::endl;
                for (int j = cur; j != 0; j = getSuffixLink(j)) {  // возвращаемся по суффиксальным в корень
                    if (vertexes[j].isTerminal) {  //если нашли терминальную
                        int indInText = i + 1 - vertexes[j].level;
                        for (int k = 0; k < vertexes[j].posInJokerPattern.size(); k++) {   //по вычисленному месту текущей
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;
                            }
                        }
                    }
                }
            }
            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);
}
```