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. 8303		Сенюшкин Е.В.
Преподаватель		Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Научиться реализовывать алгоритм Ахо-Корасик и реализовать с его помощью программу для поиска вхождений шаблонов в текст и поиска вхождений в текст шаблона с джокером.

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

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

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

Выход:

Все вхождения образцов из *P* в *T*.

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

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

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

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3

Индивидуализация.

Вариант 4.

Реализовать режим поиска, при котором все найденные образцы не пересекаются в строке поиска (т.е. некоторые вхождения не будут найдены; решение задачи неоднозначно).

Описание алгоритма Ахо-Корасик(1 задание).

На вход программа получает текст *T* и массив шаблонов *P*.

Вначале строится бор. Построение начинается с создания корня, корень является начальным состоянием в боре. Теперь программа начинает добавлять в бор шаблоны *P[i]*. Следуем из корня по ребрам отмеченными буквами *P[i][j]* пока это возможно, если перехода по ребру нет, то создается новая вершина, в которой сохраняется родительская вершина, символ *P[i][j]* и длинна прочитанного префикса шаблона. Вершина, в которой шаблон *P[i]* закончился помечается как терминальная, а также в ней сохраняется индекс шаблона в массиве шаблонов. После того как очередной шаблоны был добавлен, берется следующий шаблон и все тоже самое повторяется пока шаблоны не кончатся.

После того как бор был построен в нем нужно построить суффиксные ссылки. Суффиксная ссылки — это ссылка на вершину, которая является самым длинным суффиксом для текущей вершины. Суффиксные ссылки строятся с помощью поиска в глубину. Для построения суффиксных ссылок на текущем уровне достаточно того, чтобы ссылки были посчитаны на уровень ниже. В корне суффиксная ссылка нулится.

Чтобы построить суффиксную ссылку для текущей вершины *v*, получается ссылка на родительскую вершину *p*, производится переход по ее суффиксной ссылки и в этой вершине производится переход по символу, который хранится в вершине *v*, если переход возможен, то суффиксная ссылка в вершине *v* должна указывать на полученную вершину, если переход невозможен, то производится переход по суффиксной ссылки *p.suff* и процесс повторяется пока ссылка не будет указывать на ноль или будет возможен переход по символу, который хранится в вершине *v*, если ссылка указывает на ноль, то суффиксная ссылка вершины *v* должна указывать на корень. Одновременно с этим программа запоминает все вершины, из которых по суффиксным ссылкам можно попасть в терминальные вершины это делается, чтобы случайно не пропустить шаблоны в тексте.

После того как бор и суффиксные ссылки были построены, начинается поиск шаблонов в тексте. Алгоритм считывает очередной символ текста и пытается перейти из текущей вершины по этому символу, если перейти не удается, то происходит переход по суффиксной ссылки, и все повторяется, если алгоритм перешел в терминальную вершину, значит в тексте был найден один из шаблонов, индекс шаблона и индекс начала шаблона в тексте сохраняются.

Описание функций и структур данных.

struct Node - структура вершины бора.

Ее поля:

- Node* parent; - Ссылка на родителя
- Node* suffixLink; - Ссылка на вершину, которая является максимальныым суффиксом текущей строки
  std::map<char, Node*> children - Ссылки на детей
- char value; - Символ, который хранится в вершине
- bool terminal; - Является ли вершина терминальной
- int p; - Если вершина является терминальной, то в p хранится индекс строки в словаре
- std::vector<std::pair<int, int>> terminalSuffixes; - Массив в котором хранятся все терминальные суффиксы, для текущейк строки
- int h; - Высота вершин

Node* buildTrie(std::vector<std::string> P) - функция принимает, массив шаблонов и строит по ним бор. Возвращает ссылку на корень бора.

void createSuffixLink(Node* root) - функция, принимает бор и строит в нем суффиксные ссылки. Функция ничего не возвращает.

vector<std::pair<int, int>> findInText(Node* root, const std::string& T, std::vector<std::string>& P) - Принимает ссылку на бор, и текст, в котором происходит поиск шаблонов, массив строк P передается для вывода промежуточных данных. Возвращает массив пар, в котором хранится индекс шаблона в тексте и индекс шаблона в массиве шаблонов.

Сложность алгоритма.

По скорости.

Построение бора происходит за суммарную длину всех шаблонов O(n), где n -длинна всех шаблонов.

Построение суффиксных ссылок осуществляет с помощью поиска в ширину, но в каждой вершине возможно долго не будет получатеся найти суффиксную ссылку и придется спуститься до самого корня, но так как суффиксные ссылки строятся на основе суффиксных ссылок родительской вершины, то для следующей вершины уже не может случится случай, когда придется пройти весь путь до корня, так что в худшем случае мы поднимемся и спустимся по строке всего один раз. Сложность получается O(2m * log(A)) - где m кол-во вершин в боре,а A - размер алфавита, все дети лежат в ассоциативном массиве, по этому доступ до элемента можно получить за log, в худшем случае количество вершин в боре будет равно общей длине всех шаблонов O(2n)

Поиск шаблонов в тексте осуществляется за O(|T|), где |T| - длинна текста.

Сложность алгоритма получается O(2n * log(A) + |T|)

По памяти.

В худшем случае в боре вершин будет столько же сколько и сумма длин всех строк шаблона, но еще в каждой вершине нужно хранить все терминальные вершины, которые можно достигнуть из нее, тогда O(m * x + n), где m - сумма длинна всех строк шаблона, x - максимальная длинна шаблона, если все его суффиксы являются терминальными вершинами, а n - длинна текста.

Тестирование.

Задание 1.

Тест 1.

Ввод.

ABCASDTEAD

5

ABC CAS ASD TEA EAD

Вывод.

1 1

4 3

7 4

Тест 2.

Ввод.

CATNATCAT

3

ATN NAT CAT

Вывод.

1 3

4 2

7 3

Тест 3.

Ввод.

CCCA

1

CC

Вывод.

1 1

Тест 4.

Ввод.

ABABABABABABAB

1

ABA

Вывод

1 1

5 1

9 1

Тест с промежуточным выводом.

CAA_lb5

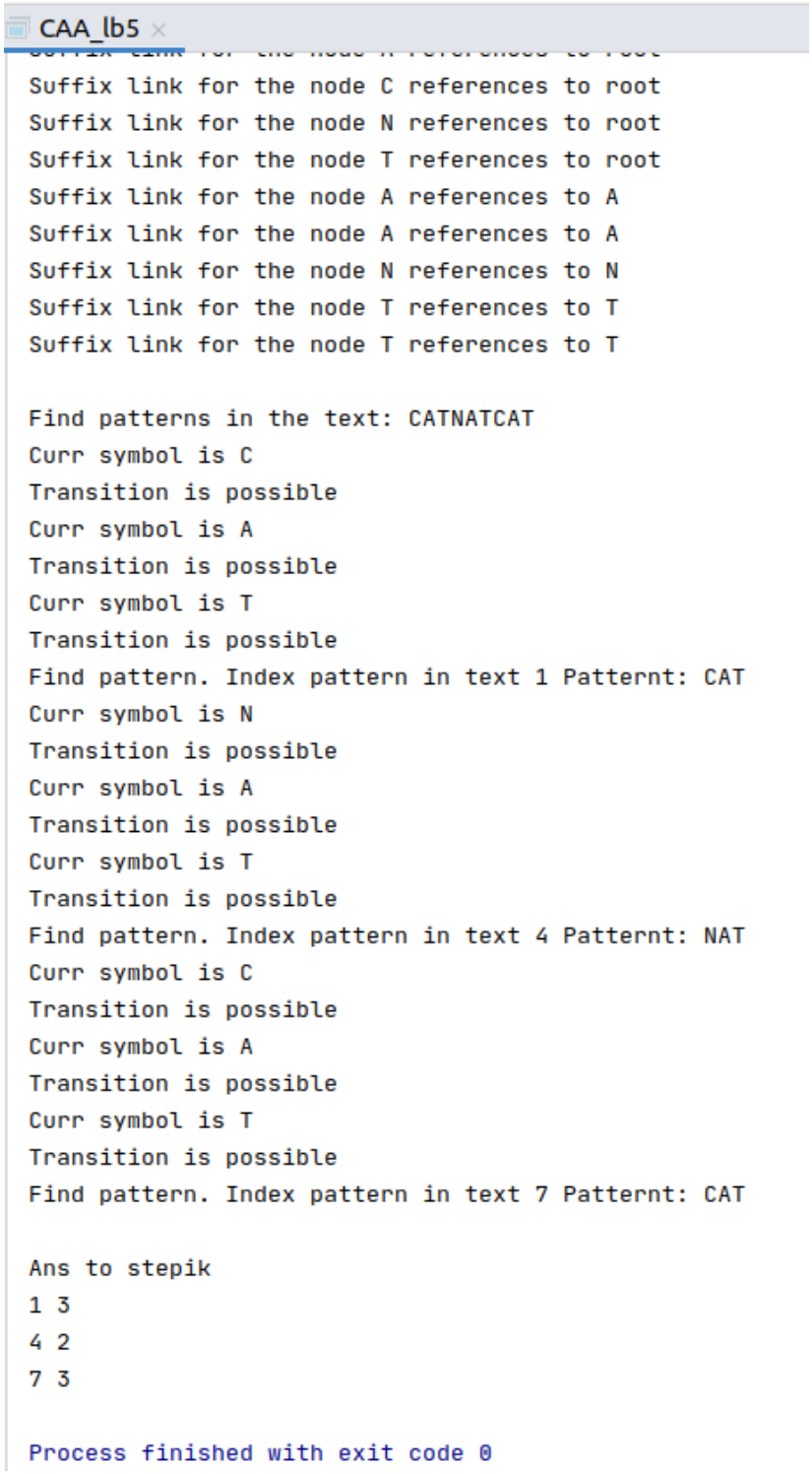
```
/home/egor/CLionProjects/CAA_lb5_Aho-Corasick/cmake-build-debug/CAA_lb5
CATNATCAT

3

ATN NAT CAT
Build trie.
New node: parrent R symbol A word length 1
New node: parrent A symbol T word length 2
New node: parrent T symbol N word length 3
Node N is terminal. Word ATN
New node: parrent R symbol N word length 1
New node: parrent N symbol A word length 2
New node: parrent A symbol T word length 3
Node T is terminal. Word NAT
New node: parrent R symbol C word length 1
New node: parrent C symbol A word length 2
New node: parrent A symbol T word length 3
Node T is terminal. Word CAT

Create suffix link.
Suffix link for the node A references to root
Suffix link for the node C references to root
Suffix link for the node N references to root
Suffix link for the node T references to root
Suffix link for the node A references to A
Suffix link for the node A references to A
Suffix link for the node N references to N
Suffix link for the node T references to T
Suffix link for the node T references to T

Find patterns in the text: CATNATCAT
Curr symbol is C
Transition is possible
Curr symbol is A
Transition is possible
Curr symbol is T
Transition is possible
Find pattern. Index pattern in text 1 Patternt: CAT
```



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

Описание алгоритма 2.

Для того, чтобы найти все вхождение в текст заданного шаблона необходимо обнаружить в тексте все подстроки шаблона, разбитых джокером. Создадим для этого массив строк Q, в котором хранятся все

подстроки, разбитые джокером и массив l, в котором хранятся индекс вхождение этих подстрок в шаблон.

Создадим массив C, в котором на позиции i будет хранится количество встретившихся безмасочных подстрок шаблона, которые начинаются на позиции i. Тогда появление Q[i] в тексте на позиции j будет означать возможное появление шаблона на позиции j – l[i]+ 1.

Построим бор по массиву строк Q. Теперь с помощью алгоритма Ахо-Корасик находим все безмасочные подстроки, когда алгоритм находит подстроку Q[i] он увеличивает на 1 значение C[j – l[i] + 1]

Теперь каждое i, для которого C[i] равно количеству подстрок разбитых джокером является стартовой позицией появления шаблона в тексте.

Описание функций и структур данных.

struct Node - структура вершины бора.

Ее поля:

- Node* parent; - Ссылка на родителя
- Node* suffixLink; - Ссылка на вершину, которая является максимальныым суффиксом текущей строки
  std::map<char, Node*> children - Ссылки на детей
- char value; - Символ, который хранится в вершине
- bool terminal; - Является ли вершина терминальной
- int p; - Если вершина является терминальной, то в p хранится индекс строки в словаре
- std::vector<std::pair<int, std::vector<int> >> terminalSuffixes; - Массив в котором хранятся все терминальные суффиксы, для текущейк строки и индексы по которым хранится подстрока в шаблоне.
- int h; - Высота вершин

Node* buildTrie(std::vector<std::string>& P, std::vector<int>& l) - функция, которая принимает массив подстрок и массив начала подстрок в шаблоне и строит по ним бор. Возвращает ссылку на корень

void createSuffixLink(Node* root) - функция, которая строит в боре суффиксные ссылки и запоминает в каждой вершине все ее терминальные суффиксы. Ничего не возвращает

vector<std::pair<int, int>> findInText(Node* root, const std::string& T, std::vector<std::string>& P) - функция, которая ищет все вхождение шаблонов в тексте . Принимает ссылку на бор, и текст, в котором происходит поиск шаблонов, массив строк P передается для вывода промежуточных данных. Возвращает массив, в котором на позиции i будет хранится количество встретившихся безмасочных подстрок шаблона, которые начинаются на позиции i. хранится

Сложность алгоритма.

По скорости.

Построение бора происходит за суммарную длину всех шаблонов O(n), где n -длинна всех шаблонов.

Построение суффиксных ссылок осуществляет с помощью поиска в ширину, но в каждой вершине возможно долго не будет получатеся найти суффиксную ссылку и придется спуститься до самого корня, но так как суффиксные ссылки строятся на основе суффиксных ссылок родительской вершины, то для следующей вершины уже не может случится случай, когда придется пройти весь путь до корня, так что в худшем случае мы поднимемся и спустимся по строке всего один раз. Сложность получается O(2m * log(A)) - где m кол-во вершин в боре,а A - размер алфавита, все дети лежат в ассоциативном массиве, по этому доступ до элемента можно получить за log

Поиск шаблонов в тексте осуществляется за O(|T|), где |T| - длинна текста.

После построение массива C по нему нужно еще раз пройти и найти все индексы значения, в которых равны количеству подстрок. Длинна массива C не превышает размера текста T. Так что:

Сложность алгоритма получается O(2m * log(A)) + n + 2*|T|)

По памяти.

В худшем случае в боре вершин будет столько же сколько и сумма длин всех строк шаблона, но еще в каждой вершине нужно хранить все терминальные вершины, которые можно достигнуть из нее, тогда O(m * x + n), где m - сумма длинна всех строк шаблона, x - максимальная длинна шаблона, если все его суффиксы являются терминальными вершинами, а n - длинна текста.

Тестирование.

Тест 1.

Ввод.

ACTANCAA

A$$A

$

Вывод.

1

Тест 2.

Ввод.

CATNATCAT

#AT

#

Вывод.

1

4

7

Тест 3.

Ввод.

ABCBABC

B$B

$

Вывод.

2

Тест 4.

Ввод.

ACTANCAACTANCA

A$$A$

$

Вывод.

1

8

Тест с промежуточным выводом.

```
"/home/egor/CLionProjects/CAA_lb5_correct_point search/cmake-build-debug/CAA_lb5_correct_point_search"
CATNATCAT  #AT #
Build trie.
New node: parrent R symbol A word length 1
New node: parrent A symbol T word length 2
Node T is terminal. Word AT

Create suffix link.
Suffix link for the node A references to root
Suffix link for the node T references to root

Find patterns in the text: CATNATCAT
Curr symbol is C
Transition is not possible, use suffix link
Curr symbol is A
Transition is possible
Curr symbol is T
Transition is possible
Find pattern. Index pattern in text 2 Patternt: AT
Curr symbol is N
Transition is not possible, use suffix link
Curr symbol is A
Transition is possible
Curr symbol is T
Transition is possible
Find pattern. Index pattern in text 5 Patternt: AT
Curr symbol is C
Transition is not possible, use suffix link
Curr symbol is A
Transition is possible
Curr symbol is T
Transition is possible
Find pattern. Index pattern in text 8 Patternt: AT

Ans to stepik
1
4
7
```

Вывод

В ходе выполнения лабораторной работы был изучен алгоритм Ахо- Корасик и использован для нахождения вхождений множества строк в тексте, а также для нахождения шаблона с джокером.

Приложение А.

Исходный код.

CAA_lb5_Aho–Corasick.

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


struct Node{
    Node(Node* parent, char value, int h){
        this->parent = parent;
        this->value = value;
        this->h = h;

        suffixLink = nullptr;
        terminal = false;
        p = -1;
    }
```

```
    Node* parent; // Ссылка на родителя
    Node* suffixLink; // Ссылка на вершину, которая является максимальныым суффиксом текущей строки
    std::map<char, Node*> children = {{'A', nullptr}, {'C', nullptr}, {'G', nullptr}, {'T', nullptr}, {'N', nullptr}}; // Ссылки на детей

    char value; // Символ, который хранится в вершине
    bool terminal; // Является ли вершина терминальной
    int p; // Если вершина является терминальной, то в p хранится индекс строки в словаре

    std::vector<std::pair<int, int>> terminalSuffixes; // Массив в котором хранятся все терминальные суффиксы, для текущейк строки

    int h; // Высота вершин
};


Node* buildTrie(std::vector<std::string> P) {
    Node* root = new Node(nullptr, 'R', 0);
    for(int i = 0; i < (int)P.size(); i++) {
        Node* currNode = root;
        for(int j = 0; j < (int)P[i].size(); j++) {
            if(currNode->children[P[i][j]] != nullptr) {
                currNode = currNode->children[P[i][j]];
            } else {
                std::cout << "New node: parrent " << currNode->value << " symbol " << P[i][j] << " word length " << currNode->h + 1 << std::endl;
```

```
            currNode->children[P[i][j]] = new Node(currNode, P[i][j], currNode->h + 1);
            currNode = currNode->children[P[i][j]];
        }
    }
    currNode->terminal = true;
    currNode->p = i + 1;
    std::cout << "Node " << currNode->value << " is terminal. Word " << P[i] << std::endl;
  }

  return root;
}


void createSuffixLink(Node* root) {
  std::queue<Node*> q;
  std::map<char, Node*>::iterator it;

  for(it = root->children.begin(); it != root->children.end(); it++) {
    if (it->second != nullptr)
      q.push(it->second);
  }

  while (!q.empty()) {
    Node* v = q.front();
    q.pop();
    for(it = v->children.begin(); it != v->children.end(); it++) {
      if (it->second != nullptr)
```

```
            q.push(it->second);
        }

        char x = v->value;
        Node* p = v->parent;
        p = p->suffixLink;
        while (p != nullptr && p->children[x] == nullptr)
            p = p->suffixLink;

        if (p == nullptr) {
            v->suffixLink = root;
            std::cout << "Suffix link for the node " << v->value << " references to
root"  << std::endl;
            if (v->terminal) {
                v->terminalSuffixes.emplace_back(v->h, v->p);
            }
        }
        else {
            v->suffixLink = p->children[x];
            v->terminalSuffixes = p->children[x]->terminalSuffixes;
            std::cout << "Suffix link for the node " << v->value << " references to "
<< v->suffixLink->value << std::endl;
            if (v->terminal) {
                v->terminalSuffixes.emplace_back(v->h, v->p);
            }
        }
    }
}
```

```
std::vector<std::pair<int, int>> findInText(Node* root, const std::string& T, std::vector<std::string>& P) {
    std::vector<std::pair<int, int>> foundString;

    Node* currNode = root;
    for(int i = 0; i < (int)T.size(); i++) {
        std::cout << "Curr symbol is " << T[i] << std::endl;
        if (currNode->children[T[i]] != nullptr) {
            std::cout << "Transition is possible" << std::endl;
            currNode = currNode->children[T[i]];
            for(int j = 0; j < (int)currNode->terminalSuffixes.size(); j++) {
                std::cout << "Find pattern. Index pattern in text " << i - currNode->terminalSuffixes[j].first + 2 << " Patternt: " << P[ currNode->terminalSuffixes[j].second - 1] << std::endl;
                foundString.emplace_back(i - currNode->terminalSuffixes[j].first + 2, currNode->terminalSuffixes[j].second);
                currNode = root;
                break;
            }
        } else {
            std::cout << "Transition is not possible, use suffix link" << std::endl;
            while (currNode->children[T[i]] == nullptr && currNode != root) {
                currNode = currNode->suffixLink;
            }
            if (currNode->children[T[i]] != nullptr) {
                i--;
            }
```

```
        }
    }
    return foundString;
}



int main() {
    std::string T; // Текст
    int n; // Количество слов в словаре
    std::vector<std::string> P; // Словарь

    std::cin >> T;
    std::cin >> n;
    for(int i = 0; i < n; i++) {
        std::string tmp;
        std::cin >> tmp;
        P.push_back(tmp);
    }

    // Построение бора
    std::cout << "Build trie." << std::endl;
    Node* root = buildTrie(P);
    std::cout << std::endl;

    std::cout << "Create suffix link." << std::endl;
    // Построение суффиксных ссылок
    createSuffixLink(root);
```

```
    std::cout << std::endl;

    std::cout << "Find patterns in the text: "  << T << std::endl;
    std::vector<std::pair<int, int>> ans = findInText(root, T, P);
    std::cout << std::endl;

    std::cout << "Ans to stepik" << std::endl;
    std::sort(ans.begin(), ans.end());
    for (int i = 0; i < (int)ans.size(); i++) {
        std::cout << ans[i].first << ' ' << ans[i].second << std::endl;
    }

    return 0;
}
```

CAA_lb5_correct_point search.

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


struct Node{
    Node(Node* parent, char value, int h){
        this->parent = parent;
        this->value = value;
        this->h = h;
```

```
        suffixLink = nullptr;
        terminal = false;
        p = -1;
    }

    Node* parent; // Ссылка на родителя
    Node* suffixLink; // Ссылка на вершину, которая является максимальныым суффиксом текущей строки
    std::map<char, Node*> children = {{'A', nullptr}, {'C', nullptr}, {'G', nullptr}, {'T', nullptr}, {'N', nullptr}}; // Ссылки на детей

    char value; // Символ, который хранится в вершине
    bool terminal; // Является ли вершина терминальной
    int p; // Если вершина является терминальной, то в p хранится индекс строки в словаре

    std::vector<std::pair<int, std::vector<int>>> terminalSuffixes; // Массив в котором хранятся все терминальные суффиксы, для текущейк строки
    std::vector<int> arr;

    int h; // Высота вершин
};


Node* buildTrie(std::vector<std::string>& P, std::vector<int>& l) {
    Node* root = new Node(nullptr, 'R', 0);
    for(int i = 0; i < (int)P.size(); i++) {
        Node* currNode = root;
```

```
        for(int j = 0; j < (int)P[i].size(); j++) {
            if(currNode->children[P[i][j]] != nullptr) {
                currNode = currNode->children[P[i][j]];
            } else {
                std::cout << "New node: parrent " << currNode->value << " symbol " << P[i][j] << " word length " << currNode->h + 1 << std::endl;
                currNode->children[P[i][j]] = new Node(currNode, P[i][j], currNode->h + 1);
                currNode = currNode->children[P[i][j]];
            }
        }
        currNode->terminal = true;
        currNode->p = i + 1;
        currNode->arr.emplace_back(l[i]);
        std::cout << "Node " << currNode->value << " is terminal. Word " << P[i] << std::endl;
    }
    return root;
}


void createSuffixLink(Node* root) {
    std::queue<Node*> q;
    std::map<char, Node*>::iterator it;

    for(it = root->children.begin(); it != root->children.end(); it++) {
        if (it->second != nullptr)
            q.push(it->second);
    }
```

```
    while (!q.empty()) {
        Node* v = q.front();
        q.pop();
        for(it = v->children.begin(); it != v->children.end(); it++) {
            if (it->second != nullptr)
                q.push(it->second);
        }

        char x = v->value;
        Node* p = v->parent;
        p = p->suffixLink;
        while (p != nullptr && p->children[x] == nullptr)
            p = p->suffixLink;

        if (p == nullptr) {
            v->suffixLink = root;
            std::cout << "Suffix link for the node " << v->value << " references to
root"  << std::endl;
            if (v->terminal) {
                v->terminalSuffixes.emplace_back(v->h, v->arr);
            }
        }
        else {
            v->suffixLink = p->children[x];
            v->terminalSuffixes = p->children[x]->terminalSuffixes;
            std::cout << "Suffix link for the node " << v->value << " references to "
<< v->suffixLink->value << std::endl;
```

```
            if (v->terminal) {
                v->terminalSuffixes.emplace_back(v->h, v->arr);
            }
        }
    }
}


std::vector<int> findInText(Node* root, const std::string& T, int patternSize, std::vector<std::string>& P) {
    std::vector<int> C(T.size() + 1, 0);

    Node* currNode = root;
    for(int i = 0; i < (int)T.size(); i++) {
        std::cout << "Curr symbol is " << T[i] << std::endl;
        if (currNode->children[T[i]] != nullptr) {
            std::cout << "Transition is possible" << std::endl;
            currNode = currNode->children[T[i]];
            std::vector<std::pair<int, std::vector<int>>> terSuff = currNode->terminalSuffixes;
            for(int j = 0; j < (int)terSuff.size(); j++) {
                std::cout << "Find pattern. Index pattern in text " << i - currNode->terminalSuffixes[j].first + 2 << " Patternt: " << P[currNode->p - 1] << std::endl;
                for (int k = 0; k < (int)terSuff[j].second.size(); k++) {
                    int curr_i = i - (int)terSuff[j].second[k] + 2 - (int)terSuff[j].first;
                    if (curr_i >= 0 && curr_i + patternSize <= T.size())
                    {
                        C[curr_i]++;
```

```
                }
            }

        }
    } else {
        std::cout << "Transition is not possible, use suffix link" << std::endl;
        while (currNode->children[T[i]] == nullptr && currNode != root) {
            currNode = currNode->suffixLink;
        }
        if (currNode->children[T[i]] != nullptr) {
            i--;
        }
    }
  }
  return C;
}



int main() {
  std::string T; // Текст
  std::string pattern; // Шаблон
  char joker;

  std::cin >> T;
  std::cin >> pattern;
  std::cin >> joker;
```

```
std::vector<std::string> Q; // Словарь подстрок разделенных масками
std::vector<int> l; // стартовые позиции подстрок в шаблоне

int j = 0;
bool flag = false;
std::string tmp;
// нахождение всех подстрок и их стартовых позиций в шаблоне
for(int i = 0; i < (int)pattern.size(); i++) {

    if (pattern[i] == joker && flag) {
        flag = false;
        j++;
        Q.emplace_back(tmp);
        tmp.clear();
    }
    if (!flag && pattern[i] != joker) {
        l.push_back(i + 1);
        flag = true;
    }
    if (flag) {
        tmp += pattern[i];
    }
}
if (!tmp.empty())
    Q.emplace_back(tmp);


// Построение бора
```

```
    std::cout << "Build trie." << std::endl;
    Node* root = buildTrie(Q, l);
    std::cout << std::endl;

    // Построение суффиксных ссылок
    std::cout << "Create suffix link." << std::endl;
    createSuffixLink(root);
    std::cout << std::endl;

    std::cout << "Find patterns in the text: "  << T << std::endl;
    std::vector<int> C= findInText(root, T, pattern.size(), Q);
    std::cout << std::endl;

    std::cout << "Ans to stepik" << std::endl;
    int counter = pattern.size();

    for (int i = 0; i < C.size() - l.back() - Q.back().size() + 1; i++) {
        if (C[i] == Q.size() && counter >= pattern.size()){
            std::cout << i + 1 << std::endl;
            counter = 0;
        }
        counter++;
    }

    return 0;
}
```