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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8383 _____ Костарев К.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с принципами работы алгоритма Ахо-Корасика нахождения вхождений всех образцов в строке и реализовать его, а также реализовать алгоритм для поиска вхождений одного образца с «джокером».

Задача.

1. Разработайте программу, решающую задачу точного поиска набора образцов.

   Вход:

   Первая строка содержит текст (T, $1 \leq |T| \leq 10000$).

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

   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 должна быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

   Sample Input:

   ```
   NTAG
   3
   TAGT
   TAG
   1
   ```

   Sample Output:

   ```
   2 2
   2 3
   ```

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 неопределенной длины. В шаблон входит хотя бы один символ не джокер, т.к. шаблоны вида ??? недопустимы. Все строки содержат символы из алфавита {A, C, G, T, N}.

Вход:

Текст (T, $1 \le |T| \le 10000$).

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 № 2.

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

Описание алгоритма для решения задачи 1.

В начале алгоритма строится бор для образцов, в котором изначально есть лишь одна вершина-корень, которая соответствует пустой строке. Далее поочередно добавляются образцы с последующей обработкой: начиная в корне, происходит хождение по по бору, пока не найдется ребро, соответствующее считанному символу строки. Если такого ребра нет, то оно создается. В случае, если при добавлении строки не было создано ни одного ребра, то есть процесс остановился на внутренней вершине, а не на листе, такая вершина помечается как терминальная (по сути, каждой вершине соответствует своя строка, но помечаются лишь те вершины, которые соответствуют образцам). После построения бора происходит посимвольная обработка текста: для очередного символа либо происходит обычный переход по дереву, если у текущей вершин есть сын с соответствующим ребром, либо переход в корень, если текущая вершина является корнем или сыном корня, либо переход по суффиксной ссылке (функция перехода определена через суффиксную ссылку, а суффиксная ссылка через функцию перехода, поэтому поиск перехода реализован "ленивым" образом через взаимную рекурсию, автомат строится по ходу выполнения, а не заранее). Суффиксная ссылка — это ребро к вершине, соответствующй самому длинному суффиксу, который не заводит бор в тупик. Оказавшись после перехода в новом состоянии, идет проверка по "сжатым" суффиксным ссылкам строки, которые были обнаружены, если новое состояние является терминальным, то соответствующая ему строка тоже отмечается. "Сжатая" суффиксная ссылка либо переходит по суффиксной ссылке, если она ведет к терминальной вершине или в корень, либо у полученной вершины вызывает рекурсивно переход по "сжатой" суффиксной ссылке.

Далее, согласно варианту индивидуального задания, печатается число вершин бора, а после, сравнивая последние позиции вхождения образцов в текст, печатаются пересекающиеся.

Описание алгоритма для решения задачи 2.

Алгоритм отличается лишь тем, что в начале обработки образец разбивается на максимальные подстроки без джокера, позиции таких подстрок запоминаются в массиве. Таким образом получается обычный список образцов $\{q_1, \ldots, q_z\}$ и список стартовых позиций $\{l_1, \ldots, l_z\}$. После этого по тому же алгоритму происходит поиск всех вхождений подстрок в тексте. Когда нашлась очередная подстрока на позиции j, то увеличивается на единицу $c[j - l_i + 1]$. После этого алгоритм последовательно проходится по массиву c, и каждое i, для которого $c[i] = z$, является стартовой позицией появления образца с джокером в тексте.

Описание структур данных и функций.

1. `struct Node` – структура для хранения вершины бора. Имеет следующие поля:
    - `int neighbours[LENGTH]` – массив смежных вершин, размер которого равен количеству символов в алфавите `LENGTH`.
    - `int movePath[LENGTH]` – массив переходов в автомате.
    - `int parent` – вершина-родитель.
    - `int templNumber` – номер строки-образца, соответствующей вершине.
    - `int suffLink` – суффиксная ссылка.
    - `int upSuffLink` – сжатая суффиксная ссылка.
    - `char upToParrent` – символ, ведущий от родителя к вершине.
    - `bool terminal` – является ли вершина терминальной.
2. `Node makeNode (int parent, char tranfer)` – функция, которая создает новую вершину. Суффиксная ссылка изначально приравнивается к -1, так как она еще не вычислена, также для массива переходов. Принимает на вход вершину-родитель `parent` и символ перехода `tranfer`. Возвращает созданную вершину.

3. `void initTrie(Trie& trie)` – функция, которая инициализирует бор `trie`, создавая корень.
4. `void addString(std::string& str, Trie& trie, int count)` – функция, которая добавляет строку в бор, посимвольно проходясь по ней, и если нужно, добавлять новое ребро и начинать проходить от корня. Принимает на вход ссылку на строку `str`, ссылку на бор `trie` и номер строки-образца `count`.
5. `int getSuffLink(int vert, Trie& trie)` – функция, которая вычисляет суффиксную ссылку. Принимает на вход вершину `vert` и бор `trie`. Если вершина `vert` является корнем или его сыном, то функция возвращает 0, т.к. максимальный суффикс у таких вершин нулевой. Иначе функция возвращает результат работы рекурсивного вызова функции вычисления перехода для суффиксной ссылки родителя и символа перехода.
6. `int getLink(int vert, int index, Trie& trie)` – функция вычисления перехода по символу. Принимает на вход вершину `vert`, символ перехода `index`. Если есть обычный переход, функция возвращает сына текущей вершины vert, если вершина vert корень, то функция возвращает корень. Иначе функция возвращает результат работы рекурсивного вызова функции вычисления перехода для суффиксной ссылки вершины vert и символа перехода.
7. `int getUpSuffLink(int vert, Trie& trie)` – функция вычисления сжатой суффиксной ссылки. Принимает на вход вершину `vert` и бор `trie`. Возвращает ближайшую терминальную вершину, до которой можно дойти по суффиксным ссылкам.
8. `void checkUpLink(int vert, int index, Trie& trie, Position& output, Template& templ)` – функция, которая проходит по сжатым суффиксным ссылкам, пока не дойдет до корня. Принимает на вход вершину `vert`, символ перехода `index`, ссылку на

встреченные на строки-образцы `output`, ссылку на массив образцов `templ`. Встреченные строки-образцы по мере хождения добавляются в `output`.

9. `void processText(std::string& text, Trie& trie, Template& templ, Position& output)` - функция обработки текста `text` и поиска в нем образцов `templ`. По очереди проходит по символам текста. Для очередного символа с помощью функции перехода перемещается в новое состояние. Оказавшись в новом состоянии, с помощью функции `checkUpLink()` проходит по "сжатым" суффиксным ссылкам, отмечая встреченные образцы, если текущая вершина тоже терминальная, то соответствующий ему образец тоже отмечается.
10. `void splitString(std::string& str, Template& templ, std::vector<int>& positions, char joker)` – функция вычисления вхождений образца `str` с джокером `joker`, разбивает переданную строку на максимальные подстроки без джокера, которые записывает в `templ`.

Сложность алгоритма по времени.

Сложность построения бора по времени равна

$$O(|P|),$$

где $|P|$ – суммарная длина всех образцов, т.к. в алгоритме просто последовательно обрабатываются символы образцов. Обработка текста происходит за

$$O(|T| + a),$$

где $|T|$ – длина текста, a – суммарное число всех вхождений, т.к. для каждой вершины за $O(1)$ можно найти следующую в суффиксном пути терминальную вершину (т.е. следующее совпадение). Поэтому суммарная сложность алгоритма по времени равна

$$O(|P| + |T| + a).$$

Сложность алгоритма по памяти.

Сложность алгоритма по памяти равна

$$O(|\,|P||A|),$$

где $|A|$ – количество символов в алфавите, т.к. в алгоритме необходимо хранить бор, максимальное число вершин которого равно величине алфавита, в каждой из которых содержится информация переходов по каждому символу алфавита.

В задаче 2, помимо этого, необходимо хранить массив c, следовательно сложность алгоритма по памяти для решения второй задачи равна

$$O(|P||A|+|T|).$$

Тестирование программы решения первой задачи.

При ручном вводе данных с консоли, программа заканчивает считывание при нажатии Enter. Демонстрация работы программы приведена на рис. 1.

Входные данные для демонстрации:

```
AAAGTACNA
2
A
TAC
```

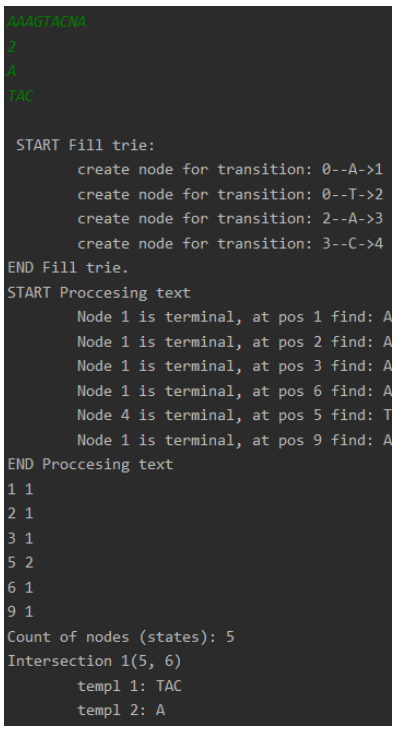

Рисунок 1 – Демонстрация работы первой программы

Тестирование программы приведено в табл. 1. В ходе тестирования все выходные данные оказались корректными.

Таблица 1 – Тестирование первой программы

Входные данные	Выходные данные
A 3 G T A	1 3
ACGTTGTGTGACGTTAC 5 ACGT ACGTT GTG GTGTGACGTTAC GG	1 1 1 2 6 3 6 4 8 3 11 1 11 2
CCCCGCCCG 3 CCC CC CGC	1 1 1 2 2 1 2 2 3 2 4 3 6 1 6 2 7 2
ATATATATAT 1 TATAT	2 1 4 1 6 1
ACGTCGTA 2 CGTC CGTA	2 1 5 2
ACGTCACTAACTCAT 1 N	No.

Тестирование программы решения второй задачи.

При ручном вводе данных с консоли, программа заканчивает считывание при нажатии Enter. Демонстрация работы программы приведена на рис. 2.

Входные данные для демонстрации:

```
CCNTCGCTA
*C*
*
```

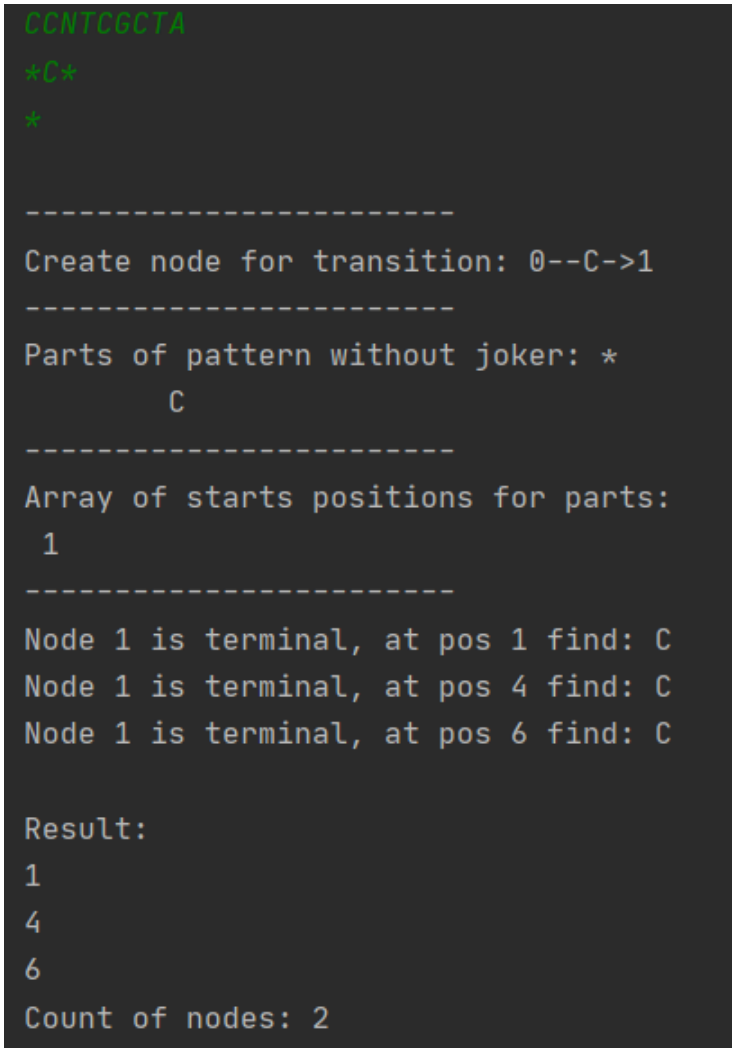

Рисунок 2 – Демонстрация работы второй программы

Тестирование программы приведено в табл. 2. В ходе тестирования все выходные данные оказались корректными.

Таблица 2 – Тестирование второй программы

Входные данные	Выходные данные
CGCCT *C*C* *	No.
AAACCC A**C*C *	1

Окончание табл. 2

ACGTCGTC *CGTC *	1 4
ANGTCNGGA *N**** *	1
NGTNNNGTGT *N* *	3 4 5
CNCNCNCNC C*NC *	No.

Выводы.

В данной лабораторной работе был изучен алгоритм Ахо-Корасика нахождения всех вхождений подстрок в строке с реализацией с линейной сложностью по времени и памяти. Также алгоритм был модернизирован для решения задачи нахождения подстроки, содержащей символы «джокер», что увеличило сложность алгоритма по памяти, но с сохранением линейности.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ РЕШЕНИЯ ПЕРВОЙ ЗАДАЧИ

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

struct Node;
using Template = std::vector<std::string>;              //вектор образцов
using Position = std::vector<std::pair<int,int>>;        //вывод позиция
- образец
using Trie = std::vector<Node>;                         //бор
using Alphabet = std::map<char, int>;                   //алфавит

const int LENGTH = 5;
Alphabet alphabet = { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} };

int getIndex(char symb){
    return alphabet[symb];
}

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

struct Node{
    int neighbours[LENGTH];      // соседние вершины
    int movePath[LENGTH];        // массив переходов
    int parrent;                          // вершина-предок
    int templNumber;                    // номер строки-образца
    int suffLink;                         // суффиксная ссылка
    int upSuffLink;                       // "сжатая" суффиксная ссылка
    char charToParrent;                   // символ ведущий к предку
    bool terminal;                        // является ли терминальной
};                                       // (совпадает со строкой)

Node makeNode(int parrent, char transfer){                    // создание
новой вершины
    Node newNode;
    memset(newNode.neighbours, 255, sizeof(newNode.neighbours));
    memset(newNode.movePath, 255, sizeof(newNode.neighbours));
    newNode.suffLink = newNode.upSuffLink = -1;               //
изначально нет ссылки
    newNode.parrent = parrent;
    newNode.charToParrent = transfer;
    newNode.terminal = false;
    newNode.templNumber = -1;
    return newNode;
}

void initTrie(Trie& trie){
```

```
    // создаем бор, изначально только корень
    trie.push_back(makeNode(0, '#'));
}

void addString(std::string& str, Trie& trie, int count){
    int index = 0;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1){        // если нет ребра
по символу
            trie.push_back(makeNode(index, str[i]));
            trie[index].neighbours[curr] = trie.size() - 1;
            std::cout << "\tcreate node for transition: " << index << "--
" << str[i] << "->" << trie.size()-1 << std::endl;
        }
        index = trie[index].neighbours[curr];
    }
    trie[index].terminal = true;
    trie[index].templNumber = count;
}

bool findString(std::string& str, Trie& trie){       //функция поиска
строки в боре
    int index = 0;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1)
            return false;
        index = trie[index].neighbours[curr];
    }
    return true;
}

int getLink(int vert, int index, Trie& trie);

// Функция для вычисления суффиксной ссылки
int getSuffLink(int vert, Trie& trie){
    if (trie[vert].suffLink == -1)                      // еще не искали
        if (vert == 0 || trie[vert].parrent == 0)       // корень или
родитель корень
            trie[vert].suffLink = 0;                    // для корня
ссылка в корень
        else
            trie[vert].suffLink = getLink(getSuffLink(trie[vert].parrent,
trie), getIndex(trie[vert].charToParrent), trie);
    return trie[vert].suffLink;
}

// Функция для вычисления перехода
int getLink(int vert, int index, Trie& trie){
    if (trie[vert].movePath[index] == -1)
// если переход по данному
        if (trie[vert].neighbours[index] != -1)
// символу ещё не вычислен
            trie[vert].movePath[index] = trie[vert].neighbours[index];
        else if (vert == 0)
            trie[vert].movePath[index] = 0;
```

```
        else
            trie[vert].movePath[index] = getLink(getSuffLink(vert, trie),
index, trie);
    return trie[vert].movePath[index];
}


// Функция для вычисления сжатой суффиксной ссылки
int getUpSuffLink(int vert, Trie& trie){
    if (trie[vert].upSuffLink == -1){             // если сжатая
суффиксная ссылка ещё не вычислена
        int tmp = getSuffLink(vert, trie);
        if (trie[tmp].terminal)                   // если ссылка на
терминальную, то ок
            trie[vert].upSuffLink = tmp;
        else if (tmp == 0)                        // на корень = 0
            trie[vert].upSuffLink = 0;
        else
            trie[vert].upSuffLink = getUpSuffLink(tmp, trie);   //поиск
ближайшей
    }
    return trie[vert].upSuffLink;
}


//проверка сжатых суффиксных ссылок
void checkUpLink(int vert, int index, Trie& trie, Position& output,
Template& templ){
    int n = 0;
    int prev;
    for (int i = vert; i != 0; i = getUpSuffLink(i, trie)){
        if (trie[i].terminal){  // если является терминальной, то нашли
соответствующий образец
            if (n)
                std::cout << "\t" << prev << "--UpLink-->" << i <<
std::endl;
            std::cout << "\tNode " << i << " is terminal, at pos " <<
index-templ[trie[i].templNumber].length()+1 << " find: " <<
templ[trie[i].templNumber] << std::endl;
            output.push_back(std::pair<int,int> ((index-
templ[trie[i].templNumber].length()+1), trie[i].templNumber+1));
            n++;
            prev = i;
        }
    }
}

//Функция для процессинга текста
void processText(std::string& text, Trie& trie, Template& templ,
Position& output){
    int current = 0;
    for (int i = 0; i < text.length(); i++){
        current = getLink(current, getIndex(text[i]), trie);
        checkUpLink(current, i+1, trie, output, templ);
    }
}
```

```
int main(){
    int count;
    Trie trie;
    std::string text;
    Position output;
    std::cin >> text;
    std::cin >> count;
    while (count <= 0 ){
        std::cout << "\nError, try again: ";
        std::cin >> count;
    }
    Template templ(count);
    initTrie(trie);
    for (int i = 0; i < count; i++){
        std::cin >> templ[i];
    }
    std::cout << "\n START Fill trie:" << std::endl;
    for (int i = 0; i < count; i++){
        addString(templ[i], trie, i);
    }
    std::cout << "END Fill trie." << std::endl;
    std::cout << "START Proccesing text" << std::endl;
    processText(text, trie, templ, output);
    std::cout << "END Proccesing text" << std::endl;
    std::sort(output.begin(), output.end(), compare);
    for (auto &curr: output){
        std::cout << curr.first << " " << curr.second << std::endl;
    }
    std::cout << "Count of nodes (states): " << trie.size() << std::endl;
    int pos1, pos2;
    int count_inter = 1;
    if (templ.size() == 1 || text.size() < 2) return 0;
    for (int i = 0; i <= output.size()-2; i++){
        for (int j = i+1; j <= output.size()-1; j++){
            pos1 = templ[output[i].second-1].size() + output[i].first -
1;
            pos2 = output[j].first;
            if (pos1 >= pos2 && output[i].second!=output[j].second){
                std::cout << "Intersection " << count_inter++ << "(" <<
output[i].first << ", " << output[j].first << ")" << std::endl;
                std::cout << "\ttempl 1: " << templ[output[i].second-1]
<< std::endl;
                std::cout << "\ttempl 2: " << templ[output[j].second-1]
<< std::endl;
            }
        }
    }
    return 0;
}
```

э

ПРИЛОЖЕНИЕ В

КОД ПРОГРАММЫ РЕШЕНИЯ ВТОРОЙ ЗАДАЧИ

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

struct Node;
using Template = std::vector<std::string>;              //вектор образцов
using Position = std::vector<std::pair<int,int>>;        //вывод позиция
- образец
using Trie = std::vector<Node>;                         //бор
using Alphabet = std::map<char, int>;                   //алфавит

const int LENGTH = 5;
int wordLength = 0;
Alphabet alphabet = { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} };

int getIndex(char symb){
    return alphabet[symb];
}

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

struct Node{
    int neighbours[LENGTH];      // соседние вершины
    int movePath[LENGTH];        // массив переходов
    int parrent;                          // вершина-предок
    std::vector<int> patternNumber;  // номер строки-образца
    int suffLink;                         // суффиксная ссылка
    int upSuffLink;                       // "сжатая" суффиксная ссылка
    char charToParrent;                   // символ ведущий к предку
    bool terminal;                        // является ли терминальной
};                                       // (совпадает со строкой)

Node makeNode(int parrent, char transfer){                    // создание
новой вершины
    Node newNode;
    memset(newNode.neighbours, 255, sizeof(newNode.neighbours));
    memset(newNode.movePath, 255, sizeof(newNode.neighbours));
    newNode.suffLink = newNode.upSuffLink = -1;               //
изначально нет ссылки
    newNode.parrent = parrent;
    newNode.charToParrent = transfer;
    newNode.terminal = false;
    return newNode;
}

void initTrie(Trie& trie){
```

```
    // создаем бор, изначально только корень
    trie.push_back(makeNode(0, '#'));
}

void addString(std::string& str, Trie& trie, int count){
    int index = 0;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1){        // если нет ребра
по символу
            trie.push_back(makeNode(index, str[i]));
            trie[index].neighbours[curr] = trie.size() - 1;
            std::cout << "\tcreate node for transition: " << index << "--
" << str[i] << "->" << trie.size()-1 << std::endl;
        }
        index = trie[index].neighbours[curr];
    }
    trie[index].terminal = true;
    trie[index].patternNumber.push_back(count);
}

bool findString(std::string& str, Trie& trie){       //функция поиска
строки в боре
    int index = 0;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1)
            return false;
        index = trie[index].neighbours[curr];
    }
    return true;
}

int getLink(int vert, int index, Trie& trie);

// Функция для вычисления суффиксной ссылки
int getSuffLink(int vert, Trie& trie){
    if (trie[vert].suffLink == -1)                      // еще не искали
        if (vert == 0 || trie[vert].parrent == 0)       // корень или
родитель корень
            trie[vert].suffLink = 0;                    // для корня
ссылка в корень
        else
            trie[vert].suffLink = getLink(getSuffLink(trie[vert].parrent,
trie), getIndex(trie[vert].charToParrent), trie);
    return trie[vert].suffLink;
}

// Функция для вычисления перехода
int getLink(int vert, int index, Trie& trie){
    if (trie[vert].movePath[index] == -1)
// если переход по данному
        if (trie[vert].neighbours[index] != -1)
// символу ещё не вычислен
            trie[vert].movePath[index] = trie[vert].neighbours[index];
        else if (vert == 0)
            trie[vert].movePath[index] = 0;
```

```
        else
            trie[vert].movePath[index] = getLink(getSuffLink(vert, trie),
index, trie);
    return trie[vert].movePath[index];
}


// Функция для вычисления сжатой суффиксной ссылки
int getUpSuffLink(int vert, Trie& trie){
    if (trie[vert].upSuffLink == -1){             // если сжатая
суффиксная ссылка ещё не вычислена
        int tmp = getSuffLink(vert, trie);
        if (trie[tmp].terminal)                   // если ссылка на
терминальную, то ок
            trie[vert].upSuffLink = tmp;
        else if (tmp == 0)                        // на корень = 0
            trie[vert].upSuffLink = 0;
        else
            trie[vert].upSuffLink = getUpSuffLink(tmp, trie);   //поиск
ближайшей
    }
    return trie[vert].upSuffLink;
}

//проверка сжатых суффиксных ссылок
void checkUpLink(int c[], int vert, int index, Trie& trie, std::string&
text, Template & templ, std::vector<int>& positions){
    int n = 0;
    int prev;
    for (int i = vert; i != 0; i = getUpSuffLink(i, trie)){
        if (trie[i].terminal){
            if (n)
                std::cout << "\t" << prev << "--UpLink-->" << i <<
std::endl;
            n++;
            prev = i;
            for (auto& in: trie[i].patternNumber){
                int tmp = index + 1 - templ[in].size() - positions[in];
                if (tmp >= 0 && tmp <= text.length() - wordLength){
                    c[tmp]++;
                    std::cout << "\tNode " << i << " is terminal, at pos
" << index-templ[in].length()+1 << " find: " << templ[in] << std::endl;
                }
            }
        }
    }
}

//Функция для процессинга текста
void processText(int c[], std::string& text, Trie& trie, Template& templ,
Position& outPositions, std::vector<int>& positions){
    int current = 0;
    for (int i = 0; i < text.length(); i++){
        current = getLink(current, getIndex(text[i]), trie);
        checkUpLink(c, current, i, trie, text, templ, positions);
    }
    std::cout << "\nResult: " << std::endl;
```

```
    int num = 0;
    for (int i = 0; i < text.size(); i++){
        if (c[i] == templ.size()){
            std::cout << i+1 << std::endl;
            num++;
        }
    }
}

// Функция разбиение образца на максимальные подстроки без джокера
void splitString(std::string& str, Template & templ, std::vector<int>&
positions, char joker){
    int index = 0;
    int position = 0;
    int count = 0;
    for (int i = 0; i < str.length(); i = index){
        std::string buff = "";
        while (index < str.length() && str[index] == joker)   //
пропускаем джокера
            index++;
        if (index == str.length()) return;
        position = index;
        while(index < str.length() && str[index] != joker)    // новоя
подстрока
            buff += str[index++];
        if(!buff.empty()){
            count++;
            positions.push_back(position);
//запоминаем позиции подстрок
            templ.push_back(buff);
        }
    }

}

int main(){
    int count;
    std::vector<int> positions;
    char joker;
    Position outPositions;
    Trie trie;
    std::string text;
    std::string word;
    std::cin >> text;
    std::cin >> word;
    std::cin >> joker;
    auto it = alphabet.find(joker);
    while (it != alphabet.end()){
        std::cout << "\nError, joker can`t be ALphabet symbol: ";
        std::cin >> joker;
        it = alphabet.find(joker);
    }
    wordLength = word.length();
    Template templ;
    initTrie(trie);
    splitString(word, templ, positions, joker);
    int i = 0;
```

```
    std::cout << "\n START Fill trie:" << std::endl;
    for (auto& buff: templ){
        addString(buff, trie, i++);
    }
    std::cout << "END Fill trie." << std::endl;
    int c[text.size()] = {0};
    std::cout << "Count of nodes: " << trie.size() << std::endl;
    std::cout << "Parts of pattern without joker " << joker << std::endl;
    for (auto& buff: templ){
        std::cout << "\t" << buff << std::endl;
    }
    std::cout << "Array of starts positions for parts" << std::endl;
    for (auto& buff: positions){
        std::cout << " " << buff;
    }
    std::cout << "\nSTART Proccesing text" << std::endl;
    processText(c,text, trie, templ, outPositions, positions);
    std::cout << "END Proccesing text" << std::endl;
    return 0;
}
```