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

ОТЧЕТ

по практической работе №5
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Алгоритм Ахо-Корасик

Студент гр. 8382 _____ Вербин К.М.

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

Входные данные

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}

Выходные данные

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.

Пример входных данных

NTAG

3

TAGT

TAG

T

Соответствующие выходные данные

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? 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}

Входные данные

Текст (T, $1 \leq |T| \leq 100000$)

Шаблон (P, $1 \leq |P| \leq 40$)

Символ джокера

Выходные данные

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер). Номера должны выводиться в порядке возрастания.

Пример входных данных

ACTANCA

A$$A$

$

Соответствующие выходные данные

1

Вариант 3

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

Описание алгоритма

алгоритма Ахо-Корасика

В программе реализован алгоритм Ахо-Корасик. По словарю шаблонов строится бор. Затем для каждого символа текста выполняется поиск по автомату. Переход в автомате осуществляется следующим образом. Если возможно,то выполняется переход в потомка, в другом случае - по суффиксной ссылке. После перехода выполняется проверка на то, является ли вершина и всевозможные её суффиксы – терминальными. Если да, то возвращаем все такие найденные номера паттернов. Если символа в автомате не оказалось, то текущая вершина принимает значение корня – вхождение не найдено.

Для корневого узла суффиксная ссылка — петля. Для остальных правило таково: если последний распознанный символ — x, то осуществляется обход по суффиксной ссылке родителя, если оттуда есть дуга, нагруженная символом x, суффиксная ссылка направляется в тот узел, куда эта дуга ведёт. Иначе — алгоритм проходит по суффиксной ссылке ещё раз, пока либо не придет в корень, либо не найдёт дугу, нагруженную символом x.

Конечные ссылки связаны с суффиксными: конечная ссылка ведёт на ближайшую по суффиксным ссылкам конечную вершину; обход по конечным ссылкам даёт все совпавшие строки.

Описание структур данных

Class TreeNode – структура, для хранения данных о корневой вершине бора.

Поля класса:

string dbgStr; - строка, которую можно получить при переходе в текущую вершину по ребрам

char value – Значение ноды, символ, по которому был произведён переход;

TreeNode* parent – ссылка на родительскую вершину;

TreeNode* suffixLink – суффиксная ссылка;

unordered_map <char, TreeNode*> children – асссоциативный неупорядочный контейнер потомков, ключом которого является символ, по которому можно перейти на потомка;

size_t numOfPattern – порядковый номер паттерна

vector<pair<size_t, size_t>> substringEntries – вектор, элементами которого является пара: индекс вхождения в маску и длина подстроки

1) TreeNode(char val) – конструктор для заполнения поля значения значением символа, по которому перешли; Принимает на вход значение val, по которому осуществлен переход

TreeNode() : value(\0)— конструктор для создания корневой вершины value — поле класса, содержащее информацию о том, по какому ребру произошел переход, для корня — это нулевой символ

2) void insert(const string &str) – метод для вставки строки в бор;

Метод принимает на вход строку (const string &str), которую необходимо вставить в бор. Результат работы метода — модифицированный бор.

Метод проверяет, был ли создан переход по текущему символу строки с помощью функции find в контейнере потомков узла и либо создает переход по этому символу (добавляет нового потомка), если он не существует на текущей позиции, либо спускается вниз по бору. После вставки строки увеличивается счетчик количества паттернов и изменяется поле класса numOfPattern,

являющееся индификатором паттерна в словаре.

3) auto find(const char c) – метод для поиска подстроки в строке при помощи автомата, выполняет поиск, по заданному символу, в боре, в случае найденной терминальной вершины, возвращает либо вектор size_t (задание 1), либо вектор пар size_t (задание 2);

Метод принимает на вход символ, который необходимо рассмотреть и возвращает вектор номеров найденных терминальных вершин в 1 задании и вектор пар, состоящих из начала безмасочной подстроки в маске и ее длины, во 2 задании. Обходим всех потомков текущей позиции и переходим по суффиксной ссылке. Если среди потомков не было искомого символа, то переходи по суффиксной ссылке для дальнейшего поиска. Если символ потомка равен искомому, спускаемся в эту вершину и обходим его суффиксы, так как они тоже могут быть терминальными вершинами, то есть вхождениями, заполняем вектор этими терминальными вхождениями в 1 задании и парами во 2 задании и возвращаем вектор.

4) void makeAutomaton() – функция, которая модифицирует бор в автомат путём добавления суффиксных ссылок;

Метод возвращает void. Описание работы данного метода находится в описании построения бора и автомата в начале отчета. Результат работы — модифицированный бор с суффиксными ссылками, то есть автомат.

5) void makeFinishLink() - функция построения конечных ссылок на основе суффиксных

Метод возвращает void и не имеет аргументов, он модифицирует бор путем добавления конечных ссылок. С помощью обхода в ширину рассматривается каждая позиция бора и выполняется переход по цепи суффиксных ссылок до ближайшей терминальной вершины. В конце получим значение конечной ссылки для каждой вершины, оно будет равняться либо ближайшей терминальной при переходе по суффиксным ссылкам, либо нулевому указателю.

6) void findMaxLinkChain() - функция поиска максимальной длины ссылочных цепей

Метод не принимает на вход никаких аргументов, не возвращает никакое значение и не модифицирует бор, в нем осущетсвляется обход автомата в ширину и переход по цепям сначала суффиксных ссылок , затем конечных из каждой вершины и рассчитывается длина каждой цепи. Результатом работы является вывод информации о максимальных длинах таких цепей.

№1:

set<pair<size_t, size_t>> AhoCorasick(const string &text, const vector<string> &patterns) – функция, возвращающая множество, состоящее из пары индекса вхождения в текст и номера паттерна, который был найден в нём. Принимает на вход строку текста(const string &text) и вектор шаблонов(const vector <string> &patterns), точнее — их ссылки. В начале создается и инициализируется бор, затем из него строится автомат, автомат заполняется конечными ссылками, находятся максимальные длины цепей суффиксных и конечных ссылок и выполняется поиск вхождений. Результат работы функции — вектор пар номеров вхождений и их индексов в тексте.

№2:

vector <size_t> AhoCorasick(const string &text, const string &mask, const char joker)– функция, возвращающая вектор индексов вхождения маски в текст. Принимает на вход строку-текст, строку-маску и симовл джокера.

Результат работы - строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер). Функция работает аналогично №1, за исключением инициализации бора паттернами и специфичного вывода.

Сложность алгоритма по памяти

Каждый символ шаблона представляет собой вершину бора, поэтому сложность по памяти составляет O(n). Т.к. на каждой позиции в тексте могут встретиться все p шаблонов, полная сложность по памяти составляет O(n + m * p).

Временная сложность алгоритма

Построение бора имеет линейную сложность O(n), где n - сумма длин паттернов. Построение суффиксных ссылок реализуется через обход в ширину со сложностью O(V + E), где E – кол-во ребер, V - кол-во вершин, эту сложность можно адаптировать для бора: O(n + n) = O(2n) = O(n). Перебор символов текста в боре занимает O(m), где m - длина текста. В конечном итоге сложность алгоритма составляет O(2n + m) = O(n + m).

Алгоритм поиска подстрок с джокером

В качестве словаря шаблонов используются подстроки маски, разделенные символами джокера. Аналогично заданию №1 сначала строится бор из этих подстрок и выполняется поиск по автомату для каждого символа текста. Появления подстроки в тексте на позиции j означает возможное появление маски на позиции j-l+1, где l – индекс начала подстроки в маске. Далее, с помощью вспомогательного массива для таких позиций увеличиваем его значение на 1. Индексы, по которым хранятся значения равному n (количеству подстрок), являются вхождениями маски в текст.

Сложность алгоритма по памяти

Сложность по памяти, как и в первом алгоритме, составляет O(n+m*p), где m – количество вершин n – длинна текста, p – число шаблонов.

Временная сложность алгоритма

В случае с поиском шаблона с джокером построение бора будет иметь сложность O(h), где h - сумма длин подстрок. Построение суффиксных ссылок так же, как и в первом алгоритме имеет сложность O(h).

Прохождение текста по бору также составляет O(n), где n - длина текста. Не учитывается прохождение дополнительно вектора, т.к. его длина равна длине исходного текста. В конечном итоге сложность алгоритма составляет O(2h + n)

Тестирование

Input	Output
Ахо Корасик	
ABCASDTEAD 5 ABC DTE ASD TEA EAD	1 1 4 3 6 2 7 4 8 5
ABCBABC 4 ABC BC CBA BAB	1 1 2 2 3 3 4 4 5 1 6 2
DOGNTADOG 3 TA DOG NA	1 2 5 1 7 2
DDDA 1 DD	1 1 1 2
Алгоритм с джокером	
MATFDHYD $$D $	3 6

CATNATCAT $AT $	1 4 7
TDWIK DLK $	
ACTANCA A$$A $	1 4

Вывод

Было получено теоретическое представление об алгоритме Кнута-МоррисаПратта и на основе него были реализованы программы для поиска вхождений подстроки и поиска циклического сдвига строки.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

алгоритма Ахо-Корасика

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <queue>
#include <algorithm>
#include <unordered_map>

using namespace std;

class Tree {

    string dbgStr = ""; // Для отладки
    char value; // Значение узла
    size_t numOfPattern = 0; // Номер введенного паттерна
    Tree* parent = nullptr; // Родитель ноды
    Tree* suffixLink = nullptr; // Суффиксная ссылка
    Tree* finishLink = nullptr; // конечная ссылка
    unordered_map <char, Tree*> children; // Потомки узла
public:
    Tree() : value('\0') {}
    Tree(char val) : value(val) {} // Конструктор ноды
    void initialization(vector<string> patterns) {
        for (auto& pattern : patterns) {
            this->insert(pattern);
        }
    }
```

```
        void printInfo(Tree* curr) {

            cout << curr->dbgStr << ':' << endl;

            if (curr->suffixLink)
                cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ? "Корень" : curr->suffixLink->dbgStr) << endl;
            if (curr->finishLink)
                cout << "\tКонечная ссылка: " << (curr->finishLink->dbgStr) << endl;

            if (curr->parent)
                cout << "\tРодитель: " << (curr->parent->value ? curr->parent->dbgStr : "Корень") << endl;

            if (!curr->children.empty())
                cout << "\tПотомок: ";
            for (auto child : curr->children) {
                cout << child.second->value << ' ';

            }
        }
        // Вставка подстроки в бор
        void insert(const string& str) {
            auto curr = this;
            static size_t countPatterns = 0;

            for (char c : str) { // Идем по строке
                // Если из текущей вершины по текущему символу не было создано перехода
```

```
            if (curr->children.find(c) == curr->children.end()) {
                // Создаем переход
                curr->children[c] = new Tree(c);
                curr->children[c]->parent = curr;
                curr->children[c]->dbgStr += curr->dbgStr + c;
            }
            // Спускаемся по дереву
            curr = curr->children[c];
        }

        cout << "Вставляем строку: " << str << endl;
        printBor();

        // Показатель терминальной вершины, значение которого равно порядковому номеру добавления шаблона
        curr->numOfPattern = ++countPatterns;
    }

    //печать бора
    void printBor() {
        cout << "Текущее состояние бора:" << endl;

        queue<Tree*> queue;
        queue.push(this);

        while (!queue.empty()) {

            auto curr = queue.front();
            if (!curr->value)
                cout << "Корень:" << endl;
```

```
            else
                printInfo(curr);
            for (auto child : curr->children) {
                queue.push(child.second);
            }

            queue.pop();
            cout << endl;
        }
        cout << endl;

    }


    // Функция для поиска подстроки в строке при помощи автомата
    vector<size_t> find(const char c) {
        static const Tree* curr = this; // Вершина, с которой необходимо начать следующий вызов
        cout << "Ищем '" << c << "' из: " << (curr->dbgStr.empty() ? "Корень" : curr->dbgStr) << endl; // Дебаг

        for (; curr != nullptr; curr = curr->suffixLink) {
            // Обходим потомков, если искомый символ среди потомков не найден, то
            // переходим по суффиксной ссылке для дальнейшего поиска
            for (auto child : curr->children)
                if (child.first == c) { // Если символ потомка равен искомому
                    curr = child.second; // Значение текущей вершины переносим на этого потомка
```

```
                vector<size_t> visited; // Вектор номеров найденных терм. вершин
                // Обходим суффиксы, т.к. они тоже могут быть терминальными вершинами
                for (auto temp = curr; temp->suffixLink; temp = temp->suffixLink)
                    if (temp->numOfPattern)
                        visited.push_back(temp->numOfPattern - 1);
                //

                cout << "Символ найден!" << endl; // Дебаг
                return visited;
            }

            if (curr->suffixLink) {
                cout << "Переходим по суффиксной ссылке: ";
                cout << (curr->suffixLink->dbgStr.empty() ? "Корень" : curr->suffixLink->dbgStr) << endl;
            }
        }
        cout << "Символ не найден!" << endl; // Дебаг

        curr = this;
        return {};
    }

    // Функция для построения недетерминированного автомата
    void makeAutomaton() {

        cout << "Строим автомат: " << endl;
```

```
queue<Tree*> queue; // Очередь для обхода в ширину

for (auto child : children) // Заполняем очередь потомками корня
   queue.push(child.second);

while (!queue.empty()) {
   auto curr = queue.front(); // Обрабатываем вершину из очереди
   printInfo(curr);
   // Заполняем очередь потомками текущей верхушки
   for (auto child : curr->children) {
      queue.push(child.second);
   }

   if (!curr->children.empty())
      cout << endl;

   queue.pop();
   auto p = curr->parent; // Ссылка на родителя обрабатываемой вершины
   char x = curr->value; // Значение обрабатываемой вершины
   if (p)
      p = p->suffixLink; // Если родитель существует, то переходим по суффиксной ссылке
   if (p) cout << "\t\tПоиск суффиксной ссылки символа '" << x << "' в '" << ((p->dbgStr.length() != 0) ? p->dbgStr : "Корень") << "' потомках" << endl;
   else cout << "\t\tПоиск суффиксной ссылки символа '" << x << "' в " <<  "'Корень'" << " потомках" << endl;
   // Пока можно переходить по суффиксной ссылке или пока
   // не будет найден переход в символ обрабатываемой вершины
   while (p && p->children.find(x) == p->children.end()) {
```

```
                p = p->suffixLink; // Переходим по суффиксной ссылке
                if (p) cout << "\t\tПоиск суффиксной ссылки символа '" << x << "' в '" <<(( p->dbgStr.length() != 0) ? p->dbgStr : "Корень") << "' потомках" << endl;
            }

            // Суффиксная ссылка для текущей вершины равна корню, если не смогли найти переход
            // в дереве по символу текущей вершины, иначе равна найденной вершине
            curr->suffixLink = p ? p->children[x] : this;

            // Дебаг
            cout << "\tСуффиксная ссылка: " << (curr->suffixLink == this ? "Корень" : curr->suffixLink->dbgStr) << endl << endl;
        }

        // Дебаг
        cout << endl;
        printBor();
    }


    void makeFinishLink() {

        cout << "Строим конечные ссылки" << endl;

        queue<Tree*> queue;
        queue.push(this);
```

```
        while (!queue.empty()) {

            auto curr = queue.front();
            auto next = curr;
            //проходим по суффиксным ссылкам каждой вершины автомата
            while (1) {

                if (next->suffixLink && next->suffixLink->value) {//есть возможность перейти по суффиксной ссылке не в корень
                    next = next->suffixLink;//переходим
                }
                else break;//цепочка суффиксных ссылок закончилась

                if (next->numOfPattern) {//вершина - терминальная
                    curr->finishLink = next;//строим конечную ссылку
                    break;
                }


            }
            //обход в ширину
            for (auto child : curr->children) {
                queue.push(child.second);
            }

            queue.pop();
        }
        printBor();
    }
```

```
void findMaxLinkChain() {//индивидуализация поиск максимальных цепей

    size_t maxSuffixChain = 0;
    size_t maxFinishChain = 0;
    size_t buf = 0;//для хранения длины цепочки из текущей вершины

    queue<Tree*> queue;
    queue.push(this);

    while (!queue.empty()) {

        auto curr = queue.front();
        auto next = curr;

        //проходим по суффиксным ссылкам каждой вершины автомата
        if (curr->value)
            cout << curr->dbgStr << ":" << endl << "\tСуффиксная цепочка ";
        cout << curr->dbgStr;
        buf = 0;
        while (1) {
            if (next->suffixLink) {//&& next->suffixLink->value){//есть возможность перейти по суффиксной ссылке не в корень
                next = next->suffixLink;//переходим
                cout << "->" << next->dbgStr;
                buf++;//увеличиваем длину цепи

            }
            else break;//цепочка суффиксных ссылок закончилась
```

```
            }
            cout << "Корень" << endl;
            maxSuffixChain = max(maxSuffixChain, buf);
            //cout << "Текущая максимальная длина цепи суффиксных ссылок: " << maxSuffixChain << endl;

            buf = 0;
            next = curr;
            if (curr->finishLink)
               cout << "\tЦепочка конечных ссылок " << curr->dbgStr;
            else cout << endl;
            while (1) {
               if (next->finishLink) {//есть возможность перейти по конечной ссылке
                  next = next->finishLink;//переходим
                  if (next->dbgStr != "")
                     cout << "->" << next->dbgStr;
                  buf++;//увеличиваем длину цепи
               }
               else break;//цепочка суффиксных ссылок закончилась
            }
            maxFinishChain = max(maxFinishChain, buf);
            //cout << "Текущая максимальная длина цепи конечных ссылок: " << maxFinishChain <<endl;

            //обход в ширину
            for (auto child : curr->children) {
               queue.push(child.second);
            }
```

```
            queue.pop();
            cout << endl;
        }
        cout << endl;

        cout << "Максимальная длина цепи из суффиксных ссылок - " << maxSuffixChain << endl;
        cout << "Максимальная длина цепи из конечных ссылок - " << maxFinishChain << endl;
        cout << endl;

    }

    ~Tree() { // Деструктор ноды
        for (auto child : children) delete child.second;
    }


};

auto AhoCorasick(const string& text, const vector <string>& patterns)
{
    Tree bor;
    set <pair<size_t, size_t>> result;

    bor.initialization(patterns);
    bor.makeAutomaton(); // Из полученного бора создаем автомат (путем добавления суффиксных ссылок)
    bor.makeFinishLink();//добавляем конечные ссылки
    bor.findMaxLinkChain();//поиск максимальных длин цепей ссылок
```

```
        {
            size_t j = 0;
            for (auto& el : text) {//поиск для каждого символа строки
                for (auto pos : bor.find(el))// Проходим по всем найденным позициям, записываем в результат
                    result.emplace(j - patterns[pos].size() + 2, pos + 1);
                j++;
            }
        }

        return result;
    }

    int main()
    {
        setlocale(LC_ALL, "Russian");
        string text;
        size_t n;
        cin >> text >> n;
        vector <string> patterns(n);//словарь

        for (auto& pattern : patterns) {
            cin >> pattern;
        }

        auto res = AhoCorasick(text, patterns);
        for (auto r : res)
            cout << r.first << ' ' << r.second << endl;
```

```
    return 0;
}
```

Алгоритм с Джокером

```
#include <iostream>
#include <vector>
#include <fstream>
#include <map>
#include <unordered_map>
#include <queue>
#include <algorithm>

using namespace std;
char inputConsole(string* str, string* templates);

class TNode {
private:
    char symbol;
    unordered_map<char, TNode*> sons;
    TNode* parent = nullptr;
    TNode* suffLink = nullptr;
    string str = "";
    int terminated;
    vector<pair<int, int>> arrayPatterns;

public:
    explicit TNode(char c) : symbol(c), terminated(0) {}

    void insert(const string& temp, int pos, int size) {
```

```
            TNode* curr = this;
            for (char symbol : temp) {
                if (curr->sons[symbol] == nullptr) {
                    curr->sons[symbol] = new TNode(symbol);  // создаем нового
ребенка
                    curr->sons[symbol]->parent = curr;
                    curr->sons[symbol]->str = curr->str + symbol;
                }
                curr = curr->sons[symbol];
            }
            cout << "Insert substring: " << temp << endl;
            curr->terminated = 1;
            curr->arrayPatterns.emplace_back(pos, size); // сохраняем все
вставленные маски в вектор
        }                                       // pos - позиция начала в шаблоне
                                               // size - длина маски


        //  поиск символа префикса в боре и всех вхождений шаблонов в его
путь
        vector<pair<int, int>> getChain(char c, int* maxSuffLen, int*
maxEndLen) {
            vector <pair<int, int>> templatesInside;
            int currSuffLen = 0;
            int currEndLen = 0;
            static const TNode* curr = this;

            for (; curr != nullptr; curr = curr->suffLink) {
                for (auto son : curr->sons) {
                    cout << "Child: " << son.first << endl;
```

```
                if (son.first == c) {
                    cout << "This is the child they were looking for! \n";
                    curr = son.second;
                    for (auto node = curr; node->suffLink != nullptr; node = node->suffLink, currSuffLen++) {
                        if (node->terminated > 0)
                            currEndLen++;
                        for (auto it : node->arrayPatterns) {
                          //  cout << "\tpass " << it.second << " symbols" << endl;
                            templatesInside.push_back(it);
                        }
                    }
                    *maxSuffLen = (*maxSuffLen < currSuffLen) ? currSuffLen : *maxSuffLen;
                    *maxEndLen = (*maxEndLen < currEndLen) ? currEndLen : *maxEndLen;
                    cout << "Current max suffix link chain lenght: " << *maxSuffLen << endl;
                    cout << "Current max end link chain lenght: " << *maxEndLen << endl;
                    return templatesInside;
                }
            }
        }
        curr = this;
        return {};
    }


    // функция для построения суффиксных ссылок
```

```
        void makeSuffixLinks() {

            queue<TNode*> q;
            for (auto son : sons) {     // можно внести это в цикл
               q.push(son.second);
            }
            while (!q.empty()) {
               TNode* curr = q.front();  //  берем вершину из очереди для обработки
               cout << "Considered vertex: " << curr->symbol << " Substring: " << curr->str << endl;
               for (pair<const char, TNode*> son : curr->sons)
                  q.push(son.second);
               q.pop();

               TNode* par = curr->parent;
               if (par != nullptr)   // переходим по суфф. ссылке предыдущей вершины
                  par = par->suffLink;


               while (par && par->sons.find(curr->symbol) == par->sons.end()) //проверка, есть ли нужный символ
                  par = par->suffLink;                        // в потомках рассматриваемой вершины,
                                                              // если нет, то переходим по суфф ссылке
               if (par) { curr->suffLink = par->sons[curr->symbol]; cout << " Suffix link: " << curr->suffLink->symbol << " Substring: " << curr->suffLink->str << endl; }
```

```
            //  присваиваем суффиксную ссылку, если она найдена
            else curr->suffLink = this;      //  иначе присваиваем ссылку в себя
        }

    }

    void printTrie(TNode* root) {
        TNode* curr = root;
        cout << "\nString:" << curr->str << endl;
        if (curr->terminated > 0)
            cout << "--->Terminated!" << "\n";

        if (curr->parent && curr->parent->symbol != '\0') {
            cout << "    Symbol:" << curr->symbol << "\n";
            cout << "    Parent:" << curr->parent->symbol << endl;
        }
        else if (curr->parent && curr->parent->symbol == '\0')
            cout << "    Parent: root" << endl;
        else
            cout << "    Root" << endl;

        if (curr->suffLink)
            cout << "    Suffix link: " << curr->suffLink->str << endl;

        cout << "    Children:";
        if (curr->sons.size() > 0) {
            for (auto c : curr->sons) {
                cout << c.first << " ";
            }
            cout << endl;
```

```
            }
            else cout << " none \n";
            for (auto tmp : curr->sons) {
                if (tmp.second) {
                    printTrie(tmp.second);
                }
            }
        }
    };

    class Trie {
    private:
        TNode node;
        int maxSuffLen;
        int maxEndLen;
    public:
        Trie() : node('\0'), maxSuffLen(0), maxEndLen(0) {}

        void printMaxLenghts() {
            cout << "\nMax suffix link chain lenght: " << maxSuffLen << endl;
            cout << "Max end link chain lenght: " << maxEndLen << endl;
        }

        TNode* getRoot() { return &node; }

        vector<pair<int, int>> getChain(char c) { return node.getChain(c,
&maxSuffLen, &maxEndLen); }

        void makeSuffixLinks() { node.makeSuffixLinks(); }
```

```
        void insert(const string& temp, int pos, int size) { node.insert(temp, pos,
size); }
    };

    int main() {
        string str;
        char joker;
        string pattern;
        //    joker = inputConsole(&str, &pattern);
        //----------------------------------------
        cin >> str >> pattern >> joker;
        //----------------------------------------
        int freePart = 0;  // счетчик подстрок в шаблоне без джокера
        Trie root;
        string mask;
        vector <size_t> maskEnter(str.size());  // массив флагов попадания
подстроки в текст
        char c;
        cout << "\nStarted the construction of the Trie... \n";
        // построение бора
        for (size_t i = 0; i <= pattern.size(); i++)
        {
            if (i == pattern.size())  // если встречен конец строки, присваиваем с
значение джокера,
                                      // чтобы вставить оставшуюся маску в бор
                c = joker;

            else c = pattern[i];

            if (c != joker) {       // накапливаем маску без джокеров
```

```
                mask += c;
                cout << "flag\t\t" <<mask << endl;
            }
            else if (!mask.empty()) { // если встречен джокер и подстрока
непустая, добавляем маску в бор
                freePart++;
                root.insert(mask, i - mask.size(), mask.size());
                cout << i - mask.size() << endl;
                mask.clear();
            }
        }
        cout << "------------------------- Suffix links ----------------------------\n";
        cout << "\nThe process of creating suffix links...\n";
        root.makeSuffixLinks();

        cout << "------------------------Built the trie ----------------------------\n";
        root.getRoot()->printTrie(root.getRoot());
        cout << "----------------------- Substring search ----------------------------
\n";
        for (int j = 0; j < str.size(); j++) {
            cout << "\nSymbol: " << str[j] << "  Index: " << j << endl;
            vector<pair<int, int>> tmp = root.getChain(str[j]);
            for (auto pos : tmp) {                    // добавление всех вхождений
масок
                int i = j - pos.first - pos.second + 1;
                if (i >= 0 && i + pattern.size() <= str.size())  // отмечаем
вхождение, если границы шаблона
                    if(i >= 0) cout << "Pattern position: " << pos.first  << endl;
                    maskEnter[i]++;                          // входят в границы текста
            }
```

```
    }

    root.printMaxLenghts();

    cout << "Index Pattern\n";
    for (int i = 0; i < maskEnter.size(); i++)
        if (maskEnter[i] == freePart) {
            cout << i + 1 << endl;     // печать индексов вхождения шаблона
        }
    return 0;
}

char inputConsole(string* str, string* templates) {
    ifstream file;
    char joker;
    file.open("input.txt");
    if (file.is_open()) {
        file >> *str >> templates[0] >> joker;
        file.close();
    }
    else {
        cout << "File isn't open!";
    }
    return joker;
}
```