

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. 9383

Моисейченко К.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить алгоритм Ахо-Корасик. Реализовать данный алгоритм на языке программирования C++.

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 $P = \{p_1, \dots, p_n\}$ $1 \leq |p_i| \leq 75$

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.

Пример входных данных:

NTAG

3

TAGT

TAG

T

Пример выходных данных:

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 шаблон образцу P необходимо найти все вхождения P в текст T .

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

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 , $1 \leq |T| \leq 100000$)

Шаблон (P , $1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Пример входных данных:

ACTANCA

A\$\$\$A\$

\$

Пример выходных данных:

1

Основные теоретические положения.

Алгоритм Ахо-Корасик реализует эффективный поиск всех вхождений всех строк-образцов в заданную строку. На вход алгоритму поступают строка и

несколько строк шаблонов. Задача алгоритма - найти все возможные вхождения строк шаблонов в строку.

Для работы алгоритма необходимо реализовать бор и автомат, основанный на боре.

Бор - это дерево, в котором каждая вершина обозначает какую-то строку (коренная вершина обозначает пустую строку). При создании нового бора в нём находится только корень. Рёбра этого дерева обозначают буквы строки. Для построения автомата на основе бора, потребуются суффиксные ссылки.

Суффиксная ссылка - это наибольший суффикс строки. Для коренной вершины суффиксная строка - это петля. Чтобы построить суффиксную ссылку нужно найти вершину, в которую ведёт ребро с символом из суффиксной ссылки вершины предка.

Конечная суффиксная ссылка - это суффиксная ссылка конечной вершины.

Выполнение работы.

Реализация алгоритма Ахо-Корасик:

1. Рассматривается каждый шаблон в наборе. Строится бор. Проверяется, существует ли вершина, в которую можно перейти по ребру, помеченному символом из шаблона, если такой вершины нет - создаётся такая вершина и ребро. Переходим по бору.
2. Строится автомат на основе бора. Проверяется текущая вершина: если это корень, то суффиксная ссылка тоже корень, иначе суффиксная ссылка - это вершина, в которую ведёт ребро с данным символом из суффиксной ссылки родительской вершины. Если ссылка ищется для вершины, следующей за корнем, то для неё ссылка будет корнем.
3. Если вершина, в которую осуществлён переход, терминальная, то добавляется информация о вхождении в строку соответствующего ей шаблона в список. Если для этой же вершины конечная ссылка не пустая, то переходим по этим ссылкам до тех пор, пока они не пустые.

Для реализации Ахо-Корасик с джокером, бор строится не для шаблона, а для безджокерных подшаблонов, находящихся в нём. На основе бора строится автомат.

Если в строке находится подшаблон, то ячейка изначально нулевого массива по адресу, который образован разностью номера начального символа данного подшаблона в строку и его смещения относительно начала первого подшаблона, инкрементируется. В конечном итоге индексы ячеек массива, значение которых будет равно кол-ву подшаблонов в исходном шаблоне, будут индексами вхождения заданного шаблона в строку.

Сложность.

1. Безмасочные шаблоны. Сложность алгоритма - $O((T+n)\log(s) + k)$, где T - длина строки, в которой ищутся вхождения, s - размер алфавита, k - общее количество вхождений шаблонов в текст.
2. Шаблоны с маской. Сложность - $O((T+n)\log(s)+k*p)$, где p - суммарное количество сдвигов подшаблонов относительно исходного шаблона.
3. Поиск длин самых длинных цепочек из суффиксных и конечных ссылок. Сложность - $O(n*(a+b))$, где количество символов в самом длинном шаблоне, b - количество шаблонов.

Описание функций и структур данных.

struct Node - структура, хранит всю информацию для описания бора, суффиксных ссылок

Node* CreateBor(const std::vector<std::pair<std::string, int>>& patterns) - функция создания бора

std::pair<int, int> LenAllLinks(Node* bor, Node* root, int& depth) - функция вычисления длин наибольших цепочек из суффиксных и конечных ссылок.

void solution(const std::string& t, const std::vector<std::pair<std::string, int>>& patterns, std::vector<std::pair<int, int>>& res, int ptnLength = 0) - функция, реализующая алгоритм Ахо-Корасик в соответствии с заданием.

void preparePatterns(const std::string& p, const char& j, std::vector<std::pair<std::string, int>>& patterns) - разбиение шаблона с джокером на безджокерные подшаблоны.

Примеры работы программы.

```
Test input:
NTAG
3
TAGT
TAG
T

Test output:
2 2
2 3
```

Рисунок 1 - Пример работы программы из задания 1.

```
Test input:
ACTANCA
A$$$A$
$

Test output:
1
```

Рисунок 2 - Пример работы программы из задания 2.

Выводы.

Изучен алгоритм Ахо-Корасик и успешно написана программа, реализующая данный алгоритм.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Node.h

```
#pragma once

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

#define TASK 1

struct Node {
    std::unordered_map<char, Node*> nextEdges;           // рёбра, по
    которым можно перейти
    #if TASK == 2
        std::vector<int> substrShift;                 // сдвиги
    подстроки в шаблоне
    #endif
    Node* Parent;                                       // родительская
    вершина
    Node* suffixLink;
    Node* termSuffixLink;                             // конечная
    ссылка
    char toParent;                                     // ребро, по
    которому пришли из родительской вершины
    bool isTerminal;
    std::vector<int> patternNumber;                   // номера
    шаблонов, в которые входит символ, по которому пришли
    int termPatternNumber;

    Node(Node* parent = nullptr, const char toParent = 0) :
    Parent(parent), toParent(toParent), suffixLink(nullptr),
    termSuffixLink(nullptr), isTerminal(false) {
        if (parent == nullptr || toParent == 0) {
            this->Parent = this;
            this->suffixLink = this;
        }
    }

    // поиск следующей вершины при поиске в строке
    Node* GetLink(const char& c) {
        if (this->nextEdges.find(c) != this->nextEdges.end()) {
            return this->nextEdges.at(c);
        }
        if (this->suffixLink == this) {
            return this;
        }
        return this->suffixLink->GetLink(c);
    }

    ~Node() {}
}
```

```

        for (auto i : this->nextEdges) {
            delete i.second;
        }
    };

```

Название файла: Bor.h

```
#pragma once
```

```
#include "Node.h"
```

```

Node* CreateBor(const std::vector<std::pair<std::string, int>>&
patterns) {
    Node* bor = new Node;
    for (auto& i : patterns) {
        int patternNum = find(patterns.begin(), patterns.end(), i) -
patterns.begin();
        Node* current = bor;
        for (auto& c : i.first) {
            if (current->nextEdges.find(c) ==
current->nextEdges.end()) {
                current->nextEdges.insert({ c, new Node(current,
c) });
            }
            current = current->nextEdges[c];
            current->patternNumber.push_back(patternNum);
        }
        current->termPatternNumber = patternNum;
        current->isTerminal = true;
    #if TASK == 2
        current->substrShift.push_back(i.second);
    #endif
    }
    return bor;
}

void FindSuffixLinks(Node* bor) {
    std::queue<Node*> front({ bor });
    while (!front.empty()) {
        Node* current = front.front();
        front.pop();
        Node* currentLink = current->Parent->suffixLink;
        const char& key = current->toParent;
        bool foundLink = true;

        while (currentLink->nextEdges.find(key) ==
currentLink->nextEdges.end()) {
            if (currentLink == bor) {
                current->suffixLink = bor;
                foundLink = false;
                break;
            }
            currentLink = currentLink->suffixLink;
        }
        if (foundLink) {
            currentLink = currentLink->nextEdges.at(key);
            if (current->Parent == bor) {

```



```

        current->suffixLink = bor;
    }
    else {
        current->suffixLink = currentLink;
        Node* curTlink = current->suffixLink;
        while (curTlink != bor) {
            if (curTlink->isTerminal) {
                current->termSuffixLink = curTlink;
                break;
            }
            curTlink = curTlink->suffixLink;
        }
    }
}

if (!current->nextEdges.empty()) {
    for (auto& nxt : current->nextEdges) {
        front.push(nxt.second);
    }
}

}

std::pair<int, int> LenAllLinks(Node* bor, Node* root, int& depth) {
    std::pair<int, int> longest = { 0, 0 };
    Node* current = bor;
    while (current->suffixLink != root) {
        longest.first++;
        current = current->suffixLink;
    }
    longest.first++;
    current = bor;
    while (current->termSuffixLink != nullptr) {
        longest.second++;
        current = current->termSuffixLink;
    }
    for (auto& n : bor->nextEdges) {
        std::pair<int, int> nextLong = LenAllLinks(n.second, root,
++depth);
        if (nextLong.first > longest.first) {
            longest.first = nextLong.first;
        }
        if (nextLong.second > longest.second) {
            longest.second = nextLong.second;
        }
    }
    depth--;
    return longest;
}

void solution(const std::string& t, const
std::vector<std::pair<std::string, int>>& patterns,
std::vector<std::pair<int, int>>& res, int ptnLength = 0) {
    Node* bor = CreateBor(patterns);
    FindSuffixLinks(bor);
    int depth = 0;

```

```

        std::pair<int, int> longest = LenAllLinks(bor, bor, depth);
        Node* current = bor;
        res.clear();
    #if TASK == 2
        std::vector<int> tInd(t.length(), 0);
    #endif

    for (int i = 0; i < t.length(); i++) {
        current = current->GetLink(t.at(i));
        Node* terminalLink = current->termSuffixLink;
        while (terminalLink != nullptr) {
    #if TASK == 1
            res.push_back({
patterns.at(terminalLink->termPatternNumber).first.length() + 2,
terminalLink->termPatternNumber + 1 });
        #elif TASK == 2
            for (auto& sh : terminalLink->substrShift) {
                int idx = i - sh + 1;
patterns.at(terminalLink->termPatternNumber).first.length() - sh + 1;
                if (!(idx < 0)) {
                    tInd.at(idx)++;
                }
            }
        #endif
        terminalLink = terminalLink->termSuffixLink;
    }
    if (current->isTerminal) {
    #if TASK == 1
        res.push_back({
patterns.at(current->termPatternNumber).first.length() + 2,
current->termPatternNumber + 1 });
        #elif TASK == 2
            for (auto& sh : current->substrShift) {
                int idx = i - sh + 1;
patterns.at(current->termPatternNumber).first.length() - sh + 1;
                if (!(idx < 0)) {
                    tInd.at(idx)++;
                }
            }
        #endif
    }
    }

    #if TASK == 2
        for (int i = 0; i < tInd.size(); i++) {
            if (tInd[i] == patterns.size() && i + ptnLength <= t.length())
        {
                res.push_back({ i + 1, 0 });
            }
        }
    #endif

    delete bor;
}

    #if TASK == 2

```

```

// p -- шаблон с маской, j -- символ разделитель, patterns -- список
шаблонов
void preparePatterns(const std::string& p, const char& j,
std::vector<std::pair<std::string, int>>& patterns) {
    int prev = 0;
    size_t temp;
    do {
        temp = p.find(j, prev);
        if (temp != prev && prev != p.length()) {
            patterns.push_back({ p.substr(prev, temp - prev), prev });
        }
        prev = temp + 1;
    } while (temp != std::string::npos);
}
#endif

```

Название файла: main.cpp

```

#define TASK 1

#include "Node.h"
#include "Bor.h"

int main() {
    std::string t, p;
    char j;
    std::vector<std::pair<std::string, int>> pts;
    std::vector<std::pair<int, int>> res;
    int num = 0;

    std::cin >> t;
    #if TASK == 1
        std::cin >> num;
        for (int i = 0; i < num; i++) {
            std::string s;
            std::cin >> s;
            pts.push_back({ s, 0 });
        }
        solution(t, pts, res);
    #elif TASK == 2
        std::cin >> p;
        std::cin >> j;
        preparePatterns(p, j, pts);
        solution(t, pts, res, p.length());
    #endif

    if (res.empty()) {
        std::cout << "\nNo results.\n";
    }
    else {
        sort(res.begin(), res.end());
        for (auto r : res) {
            std::cout << r.first;
        }
        #if TASK == 1
            std::cout << " " << r.second;
        #endif
        std::cout << "\n";
    }
}

```

```
        }  
    }  
    return 0;  
}
```