**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: Ахо-Корасик

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9382 |  | Голубева В.П. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2021

## Цель работы.

Ознакомиться с алгоритмом Ахо-Корасик, научиться применять его для поиска набора образцов в тексте, а также для поиска шаблонной подстроки.

## Задание 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (*T*, 1≤∣*T*∣≤100000).

Вторая - число n (1≤n≤3000), каждая следующая из n строк содержит шаблон из набора *P* = {*p1, …, pn*} 1≤∣pi∣≤75

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

Выход:

Все вхождения образцов из P

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

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

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

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3

**Задание 2**

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу *P* необходимо найти все вхождения Р в текст Т.

Например, образец *аb*??*с*? 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≤∣*T*∣≤100000 )

Шаблон (*P*, 1≤∣*P*∣≤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**

Бор - дерево, образованное последовательным добавлением всех образцов посимвольно. При добавлении символа создаётся вершина, если соответствующий подобразец ещё не добавлялся; иначе просто осуществляется переход к ранее созданной вершине. Изначально бор состоит из корня.

Автомат - это бор, дополненный суффиксными и конечными ссылками. Это дополнение может происходить заранее или во время обработки текста.

Суффиксная ссылка из вершины А - это ссылка на вершину, соответствующую максимально длинному (под)образцу в автомате, являющемуся несобственным суффиксом (под)образца А. Для корня и его сыновей суффиксная ссылка указывает на корень.

Конечная ссылка из вершины А - это ссылка на вершину, соответствующую максимально длинному образцу, который может быть получен при выполнении нескольких переходов по суфф. ссылкам, начиная с А. Если образцов получить нельзя, то конечнаяссылка пуста.

Для построения **суффиксных ссылок** следует применять правило:

* Суффиксная ссылка из корня или из сына корня ведёт в корень.
* Для вычисления суффиксной ссылки вершины х нужно:

1. Перейти к вершине-родителю.

2. Пройти по суффиксным ссылкам минимальное число раз — но не менее 1 раза — чтобы появился путь по ребру х\*, или до попадания в корень.

3. Пройти по ребру х. Если мы в корне и ребрах из него нет, то остаться в корне.

Полученная вершина и есть искомая суффиксная ссылка.

\* Имя вершины совпадает с именем ребра бора, ведущего в эту

вершину.

Для построения **конечной ссылки** (сжатой суффиксной ссылки) нужно переходить по суффиксным ссылкам до тех, пока не попадём в терминальную вершину (т. е. соответствующую образцу). Полученная вершина будет искомой конечной ссылкой. Если попали в корень, то конечная ссылка пуста.

**Алгоритм Ахо-Корасик**

1. Построить бор из образцов.

2. Построить автомат из бора (можно выполнять по ходу обработки текста).

3. Перейти в корень бора.

4. Посимвольная обработка текста. Для каждого символа:

4.1. Совершить переход в автомате из текущей вершины по

рассматриваемому символу:

4.1а. Если есть соответствующее ребро, то перейти по нему;

4.16. Если нет, то:

4.1.6.а. Если находимся в корне, то ничего не делать.

4.1.6.6. Если находимся не в корне, то перейти по суффиксной ссылке и перейти кп. 4.1.

4.2. Добавить в результат вхождение образца, если попали в конечную вершину.

4.3. Обойти цепочку конечных ссылок до конца, сохраняя результаты.

**Алгоритм поиска шаблонной подстроки**

1. Построить автомат Ахо-Корасик из образцов, полученных выделением максимальных безджокерных подстрок из шаблонной подстроки.

2. Для каждого образца записать смещение (смещения), по которому (по которым) образец находится в шаблонной строке.

3. Инициализировать массив, заполненный нулями, длиной, совпадающей с текстом.

4. Выполнить поиск по тексту с использованием автомата. При обнаружении образца инкрементировать ячейку массива по адресу, образованному разностью номера начального символа образца в тексте и смещения образца. Если у образца несколько смещений, то инкрементировать все соответствующие ячейки массива.

В результате шаблонная подстрока будет начинаться в тех местах текста, для которых соответствующая ячейка массива содержит количество образцов с учётом кратности.

**Оценка сложности по памяти и времени**

Пусть Т - длина текста, в которой выполняется поиск, n - суммарная длина всех образцов, s - размер алфавита, k - общая длина всех вхождений образцов в текст.

Автомат хранится как индексный массив, сложность по времени - О(ns+T+k), по памяти - О(ns).

**Тестирование**

Результаты тестирования программ можно посмотреть в приложениях А и Б.

## Выводы.

Был изучен алгоритм Ахо-Корасик, реализована программа, осуществляющая поиск набора образцов в указанном тексте.

**ПРИЛОЖЕНИЕ А**

**ТЕСТИРОВАНИЕ ПРОГРАММЫ 1**

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| NTAG  3  TAGT  TAG  TG | 2 2 |
| NTCACGCAC  3  TAG  TC  AC | 2 2  4 3  8 3 |
| NTCACGCAC  2  NTCA  C | 1 1  3 2  5 2  7 2  9 2 |
| NTCACGCAC  1  A | 4 1  8 1 |
| ACACACACN  1  N | 9 1 |

**ПРИЛОЖЕНИЕ Б**

**ТЕСТИРОВАНИЕ ПРОГРАММЫ 2**

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| ACTANKA  A$$A$  $ | 1 |
| ANANACAACA  #NACA#  # | 3 |
| GTA  @T@  @ | 1 |
| TACA  AC  % | 2 |
| NAGAT  ^  ^ | 1  2  3  4  5 |

**ПРИЛОЖЕНИЕ В**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: lab5\_1.cpp

#include <iostream>

#include <vector>

#include <cstring>

#include <map>

#include <algorithm>

#define ALPHABET\_LEN 5

std::vector <std::pair<int, int>> output;

class BohrVertex{

public:

int parent;

char value;

int next\_ver[ALPHABET\_LEN];

int numPattern = 0;

bool flag = false; // является ли шаблоном

int suffLink = -1; // суффиксная ссылка от этой вершины

int move[ALPHABET\_LEN];

BohrVertex(int, char);

BohrVertex() = default;

~BohrVertex() = default;

};

class Bohr{

std::vector <BohrVertex> bohr;

int get\_move(int v, int edge);

int get\_suffix\_link(int v);

void res(std::vector <std::string>& patterns, int v, int i);

public:

Bohr();

~Bohr() = default;

void add\_in\_bohr(std::map<char, int> m, int index, std::string s);

void find\_matches(std::vector <std::string> patterns, std::string s, std::map<char, int> m);

};

Bohr::Bohr(){

bohr.push\_back(BohrVertex(0,0));

}

BohrVertex::BohrVertex(int parent, char symbol){

for (int i = 0; i < ALPHABET\_LEN; i++){

next\_ver[i] = -1;

move[i] = -1;

}

this->parent = parent;

this->value = symbol;

}

// добавляем образец в бор

void Bohr::add\_in\_bohr(std::map<char, int> m, int index, std::string s){

int n = 0;

int edge = 0;

int len = s.length();

for (int i = 0; i < len; i++){

char symb = s[i];

edge = m[symb];//вычисляем номер символа

if (bohr[n].next\_ver[edge] == -1){ // нет ребра

bohr.push\_back(BohrVertex(n, edge));//добавляем ребро

bohr[n].next\_ver[edge] = bohr.size() - 1;

}

n = bohr[n].next\_ver[edge];

}

bohr[n].flag = true;

bohr[n].numPattern = index;

}

// получаем суффиксную ссылку

int Bohr::get\_suffix\_link(int v){

if (bohr[v].suffLink == -1)

if (v == 0 || bohr[v].parent == 0) // корень указывает на себя

bohr[v].suffLink = 0;

else

bohr[v].suffLink = get\_move(get\_suffix\_link(bohr[v].parent), bohr[v].value);

return bohr[v].suffLink;

}

// перемещаемся по бору по ребру edge

int Bohr::get\_move(int v, int edge){

if (bohr[v].move[edge] == -1)

if (bohr[v].next\_ver[edge] != -1)

bohr[v].move[edge] = bohr[v].next\_ver[edge];

else

if (v == 0)

bohr[v].move[edge] = 0;

else

bohr[v].move[edge] = get\_move(get\_suffix\_link(v), edge);

return bohr[v].move[edge];

}

void Bohr::res(std::vector <std::string>& patterns, int v, int i){

for(int u = v; u != 0; u = get\_suffix\_link(u)){

if (bohr[u].flag)

output.push\_back(std::make\_pair(i - patterns[bohr[u].numPattern].length() + 1, bohr[u].numPattern + 1));

}

}

//находим шаблоны в тексте

void Bohr::find\_matches(std::vector <std::string> patterns, std::string s, std::map<char, int> m){

int u = 0;

int edge;

int len = s.length();

for (int i = 0; i < len; i++){

char symb = s[i];

edge = m[symb];

u = get\_move(u, edge);

res(patterns, u, i + 1);

}

}

int main(){

std::map<char, int> m { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4}}; // буква и её значение

Bohr bohr;

std::vector <std::string> patterns;

std::string text;

int number = 0;

std::string pattern;

std::cin >> text >> number;

for (int i = 0; i < number; i++){

std::cin >> pattern;

bohr.add\_in\_bohr(m, i, pattern);

patterns.push\_back(pattern);

}

bohr.find\_matches(patterns, text, m);

sort(output.begin(), output.end());

for (std::vector <std::pair<int, int>>::iterator it = output.begin(); it!=output.end(); ++it)

std::cout<< (\*it).first << ' ' << (\*it).second << "\n";

return 0;

}

Название файла: lab5\_2.cpp

#include <iostream>

#include <vector>

#include <map>

#include <cstring>

#include <sstream>

#include <algorithm>

#define ALPHABET\_LEN 5

class BohrVertex{

public:

int parent;

char value;

int next\_ver[ALPHABET\_LEN];

std::vector <int> num\_pattern;

bool flag = false; // является ли шаблоном

int suffix\_link = -1; // суффиксная ссылка от этой вершины

int move[ALPHABET\_LEN];

BohrVertex(int, char);

BohrVertex() = default;

~BohrVertex() = default;

};

class Bohr{

std::vector <BohrVertex> bohr;

std::vector < std::string > pattern;

int get\_move(int v, int edge);

int get\_suffix\_link(int v);

public:

Bohr();

~Bohr() = default;

void add\_in\_bohr(std::map<char, int> m, std::string s);

void find\_matches(std::map<char, int> m, std::string &s, std::vector <int> & array, const std::vector <int> & len);

void res(int v, int i, std::vector <int> &array, std::vector <int> len);

std::vector <int> patterns(std::map<char, int> m, std::stringstream& string\_pattern, char joker);

void print\_res(const std::vector<int>& array, int t\_size, int length);

};

Bohr::Bohr(){

bohr.push\_back(BohrVertex(0,0));

}

BohrVertex::BohrVertex(int parent, char symbol){

for (int i = 0; i < ALPHABET\_LEN; i++){

next\_ver[i] = -1;

move[i] = -1;

}

this->parent = parent;

this->value = symbol;

num\_pattern.resize(0);

}

// Добавление образца в бор

void Bohr::add\_in\_bohr(std::map<char, int> m, std::string s){

int n = 0;

int edge = 0;

int len = s.length();

for (int i = 0; i < len; i++){

char symb = s[i];

edge = m[symb];//вычисляем номер символа

if (bohr[n].next\_ver[edge] == -1){ // нет ребра

bohr.push\_back(BohrVertex(n, edge));//добавляем ребро

bohr[n].next\_ver[edge] = bohr.size() - 1;

}

n = bohr[n].next\_ver[edge];

}

bohr[n].flag = true;

pattern.push\_back(s);

bohr[n].num\_pattern.push\_back(pattern.size() - 1);

}

// Получаем суффиксную ссылку

int Bohr::get\_suffix\_link(int v){

if (bohr[v].suffix\_link == -1)

if (v == 0 || bohr[v].parent == 0) // корень указывает на себя

bohr[v].suffix\_link = 0;

else

bohr[v].suffix\_link = get\_move(get\_suffix\_link(bohr[v].parent), bohr[v].value);

return bohr[v].suffix\_link;

}

// перемещаемся по бору по ребру edge

int Bohr::get\_move(int v, int edge){

if (bohr[v].move[edge] == -1)

if (bohr[v].next\_ver[edge] != -1)

bohr[v].move[edge] = bohr[v].next\_ver[edge];

else

if (v == 0)

bohr[v].move[edge] = 0;

else

bohr[v].move[edge] = get\_move(get\_suffix\_link(v), edge);

return bohr[v].move[edge];

}

void Bohr::res(int v, int i, std::vector <int>& array, std::vector <int> len){

for(int u = v;u != 0;u = get\_suffix\_link(u))

if (bohr[u].flag){

for (const auto &j : bohr[u].num\_pattern)

if ((i - len[j] < array.size()))

array[i - len[j]]++;

}

}

std::vector <int> Bohr::patterns(std::map<char, int> m, std::stringstream& string\_pattern, char joker){

std::vector < int > len;

int length = 0;

std::string buffer;

while (getline(string\_pattern, buffer, joker)){

if (buffer.size() > 0){

length += buffer.size();

len.push\_back(length);

add\_in\_bohr(m, buffer);

}

length++;

}

return len;

}

void Bohr::print\_res(const std::vector<int>& array, int t\_size, int length){

for (int i = 0; i < t\_size; i++)

if ((array[i] == pattern.size()) && (i + length <= t\_size)){

std::cout << i + 1 << "\n";

}

}

void Bohr::find\_matches(std::map<char, int> m, std::string &s, std::vector <int> & array, const std::vector <int> & len){

int edge;

int u = 0;

int lenght = s.length();

for (int i = 0; i < lenght; i++){

char symb = s[i];

edge = m[symb];

u = get\_move(u, edge);

res(u, i + 1, array, len);

}

}

int main(){

std::map<char, int> m { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4}}; // буква и её значение

Bohr bohr;

std::string text, pattern\_text;

char joker;

std::cin >> text >> pattern\_text >> joker;

std::stringstream str\_stream(pattern\_text);

std::vector <int> len = bohr.patterns(m, str\_stream, joker);

std::vector <int> array(text.length(), 0);

bohr.find\_matches(m, text, array, len);

bohr.print\_res(array, text.size(), pattern\_text.length());

}