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. 7304 _____ Шарапенков И.И.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2019

Цель работы.

Изучение алгоритма точного поиска набора образцов в строке – алгоритм Ахо-Корасик.

Задача.

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 $P=\{p1,\ldots,pn\}1 \leq |pi| \leq 75$

   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:

```
CCCA
1
CC
```

Sample Output:

```
1 1
2 1
```

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

   В шаблоне встречается специальный символ, именуемого джокером (wildcard), 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 тексте xabvccbababcax

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределенной длины. В шаблоне входит хотя бы один символ не джокер, те 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.

Sample Input:

```
ACT
A$
$
```

Sample Output:

```
1
```

Описание алгоритма.

1. **Алгоритм Ахо-Корасик**
    1. На основе набора паттернов строим префиксное дерево.
    2. Рассматриваем префиксное дерево как конечный детерминированный автомат. Стартовая позиция в корне.
    3. Считываем первый символ текста.
    4. Переходим в следующее состояние по ребру, обозначающему этот символ. Если такого ребра нет, то идем по суффиксной ссылке. Если суффиксной ссылки нет, то запоминаем символ, по которому пришли в данный узел, берем суффиксную ссылку родителя и пытаемся перейти по этому символу. Данный шаг выполняется рекурсивно, пока не найден такой переход или не достигнут корень.
    5. Выполняем шаг 4 до тех пор, пока не найдем валидный переход по текущему символу текста или пока не достигнем корня.
    6. Проверяем является ли текущее состояние каким-либо паттерном. Для этого переходим по суффиксным ссылкам, проверяя соответствующий флаг. Если это паттерн, выводим его номер и позицию в тексте в консоль.
    7. Если не конец текста переходим к шагу 3.

2. **Алгоритм Ахо-Корасик + паттерн с джокером**
    1. Разбиваем паттерн на части, разделенные джокерами. Запоминаем их позицию в паттерне (индекс).
    2. Используя алгоритм Ахо-Корасик ищем позицию этих паттернов в тексте.
    3. Создаем массив нулей, размер которого совпадает с длиной текста.

4. Для каждого вхождений паттерна инкрементируем *i – j + 1* позицию в массиве, где *i* – индекс вхождения, *j* – позиция данного паттерна в исходном паттерне.
5. Таким образом, в тех позициях массива, где значение совпадает с количеством паттернов, присутствует совпадение с исходным паттерном.

Пример работы.

1. *ATATAT*
   1
   AT

   1 1
   3 1
   5 1

   ATTATNA
   2
   AT
   TN

   1 1
   1 4
   2 5

2. *AT$$$T*
   T$
   $

   2

   ATATNATNT
   T

   1
   3
   6

Вывод

В ходе лабораторной работы был изучен алгоритм Ахо=Карасик для точного поиска набора образцов в строке. Данный алгоритм был реализован на C++. Также, этот алгоритм был использован для поиска в строке паттерна, содержащего символ джокер.

Исходный код программы

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


using Vertex = struct {
    int next_vertex[5];
    int pattern_num[40];
    bool original;
    int suff_link;
    int suff_flink;
    int auto_move[5], par;
    char symb;
};
using Trie = std::vector<Vertex>;

using patternEntry = struct {
    unsigned long long int index;
    int pattern_num;
};

std::map<char, char> Char{
        {'A', 0},
        {'C', 1},
        {'G', 2},
        {'T', 3},
        {'N', 4},
};

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

Vertex make_vertex(int p, char c) {
    Vertex v;
    memset(v.next_vertex, 255, sizeof(v.next_vertex));
    memset(v.auto_move, 255, sizeof(v.auto_move));
    memset(v.pattern_num, 255, sizeof(v.pattern_num));
    v.original = false;
    v.suff_link = -1;
    v.suff_flink = -1;
    v.par = p;
    v.symb = c;
    return v;
}

void trie_init() {
    trie.push_back(make_vertex(-1, -1));
}

void add_string_to_trie(std::string const &s) {
    int num = 0;
    for (char i : s) {
        char ch = Char[i];
        if (trie[num].next_vertex[ch] == -1) {
            trie.push_back(make_vertex(num, ch));
            trie[num].next_vertex[ch] = trie.size() - 1;
        }
        num = trie[num].next_vertex[ch];
    }
    trie[num].original = true;
```

```
    pattern.push_back(s);
    for(int i = 0; i < 40; i++) {
        if(trie[num].pattern_num[i] == -1) {
            trie[num].pattern_num[i] = pattern.size() - 1;
            break;
        }
    }
}

int get_auto_move(int v, char ch);

int get_suff_link(int v) {
    if (trie[v].suff_link == -1)
        if (v == 0 || trie[v].par == 0)
            trie[v].suff_link = 0;
        else
            trie[v].suff_link = get_auto_move(get_suff_link(trie[v].par),
trie[v].symb);
    return trie[v].suff_link;
}

int get_auto_move(int v, char ch) {
    if (trie[v].auto_move[ch] == -1)
        if (trie[v].next_vertex[ch] != -1)
            trie[v].auto_move[ch] = trie[v].next_vertex[ch];
        else if (v == 0)
            trie[v].auto_move[ch] = 0;
        else
            trie[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
    return trie[v].auto_move[ch];
}


int get_suff_flink(int v) {
    if (trie[v].suff_flink == -1) {
        int u = get_suff_link(v);
        if (u == 0)
            trie[v].suff_flink = 0;
        else
            trie[v].suff_flink = (trie[u].original) ? u : get_suff_flink(u);
    }
    return trie[v].suff_flink;
}


void check(int v, int i, std::vector<patternEntry> &p) {
    patternEntry s;
    for (int u = v; u != 0; u = get_suff_flink(u)) {
        if (trie[u].original) {
            for(int j = 0; j < 40; j++) {
                if(trie[u].pattern_num[j] != -1) {
                    s.index = i - pattern[trie[u].pattern_num[j]].length();
                    s.pattern_num = trie[u].pattern_num[j];
                    p.push_back(s);
                } else
                    break;
            }
        }
    }
}


void aho_corasick(const std::string &s, std::vector<patternEntry> &p) {
```

```
    int u = 0;
    for (int i = 0; i < s.length(); i++) {
        u = get_auto_move(u, Char[s[i]]);
        check(u, i + 1, p);
    }
}

int main() {
    std::string text, pattern_str;
    char joker;

    std::cin >> text >> pattern_str >> joker;

    trie_init();

    std::vector<int> c(text.length(), 0);
    std::vector<std::string> patterns;
    std::vector<int> patterns_pos;

    for (int i = 0; i < pattern_str.length(); i++) {
        std::string pat;
        if (pattern_str[i] != joker) {
            patterns_pos.push_back(i + 1);
            for (int j = i; pattern_str[j] != joker && j !=
pattern_str.length(); j++) {
                pat += pattern_str[j];
                i++;
            }
            patterns.push_back(pat);
        }
    }

    for (auto const &el : patterns) {
        add_string_to_trie(el);
    }

    std::vector<patternEntry> p;
    aho_corasick(text, p);

    for (int i = 0; i < p.size(); i++) {
        if(p[i].index < patterns_pos[p[i].pattern_num] - 1) continue;
        c[p[i].index - patterns_pos[p[i].pattern_num] + 1]++;
        if (c[p[i].index - patterns_pos[p[i].pattern_num] + 1] ==
patterns.size() &&
            p[i].index - patterns_pos[p[i].pattern_num] + 1 <= text.length() -
pattern_str.length())
            std::cout << p[i].index - patterns_pos[p[i].pattern_num] + 2 <<
std::endl;
    }

    return 0;
}
```