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: Поиск образца в тексте. Алгоритм Рабина-Карпа.

Студентка гр. 1383 Федорова О.В.

Преподаватель Иванов Д.В.

Санкт-Петербург
2023

Цель работы.

Ознакомиться с алгоритмом поиска подстрок в строке с использованием алгоритма Карпа-Рабина. Реализовать программу, в которой используется данный алгоритм.

Задание

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число nn (1≤n≤30001≤n≤3000), каждая следующая из nn строк содержит шаблон из набора P={p1,…,pn}1≤|pi|≤75P={p1,…,pn}1≤|pi|≤75

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

Выход:

Все вхождения образцов из PP в TT.

Каждое вхождение образца в текст представить в виде двух чисел - i i pp

Где ii - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером pp

(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

Вариант 1. На месте джокера может быть любой символ, за исключением заданного.

Ход работы:

Для выполнения работы были созданы следующие структуры и функции:

1. Структура `Vertex` представляет вершину в боре (trie) для алгоритма Ахо-Корасик. Она содержит следующие поля:
    - `next_vertex[k]`: массив, где каждый элемент указывает на следующую вершину по соответствующей букве алфавита.
    - `pat_num`: номер строки-образца, если вершина является терминальной (концом образца).

    - `terminal`: флаг, указывающий, является ли вершина терминальной.
    - `suffix_link`: ссылка на вершину, образующую наибольший суффикс данного образца в боре.
    - `auto_move[k]`: массив, хранящий результаты автоматического перехода из данной вершины.
    - `par`: вершина-отец в дереве.
    - `symb`: символ на ребре от `par` к этой вершине.
    - `suff_flink`: ссылка на вершину, образующую наибольший суффикс, который также является терминальным образцом.
2. `std::vector<Vertex> bohr` - вектор, содержащий все вершины бора.
3. `std::vector<std::string> pattern` - вектор, содержащий образцы (строки), которые нужно найти в тексте.

Функции и процедуры:

1. `make_bohr_vertex` - создает новую вершину бора с заданными параметрами.
2. `bohr_ini` - инициализирует бор, добавляя корневую вершину.
3. `add_str_to_bohr` - добавляет строку в бор.
4. `is_string_in_bohr` - проверяет, содержится ли данная строка в боре.
5. `get_auto_move` - вычисляет результат автоматического перехода из данной вершины по заданному символу.
6. `get_suff_link` - вычисляет ссылку на вершину, образующую наибольший суффикс данной вершины в боре.
7. `get_suff_flink` - вычисляет ссылку на вершину, образующую наибольший суффикс, который также является терминальным образцом.
8. `check` - проверяет наличие терминальных образцов, начиная с данной вершины.
9. `find_all_pos` - находит все вхождения образцов в заданном тексте.
10. `cmp` - функция сравнения для сортировки результатов поиска.

Для демонстрации работы программ были созданы функции `jock_var() и jock_var()` для выполнения 2 и 1 задания на Stepik. В 1 задании в бор последовательно добавляются строки-паттерны в бор, после чего запускается функция find_all_pos.

Для выполнения второго задания и джокером, в момент получения строки-паттерна и символа-джокера, программа в цикле добавляет в бор строки, в которых на месте джокера стоит любой символ из алфавита alpha. Для оптимизации работы программы, массив alpha был урезан до 5, так же создан массив alpha_num для хранения индекса в массиве alpha символа char.

Для выполнения задания по варианту, массив для алфавита был расширен. В качестве символов могут выступать любые символы с кодами от 33 до 127. Вместо того, чтобы записывать номера символов в алфавите в массив, можно теперь просто хранить код стартовой вершины(33) и вычитать ее из кода очередного символа, получив тем самым номер в алфавите.

После считывания строки паттерна и джокера, на место символа-джокера в строку паттерн подставляются все символы алфавита, кроме джокера, после чего запускается поиск.

Выводы:

Изучен алгоритм Ахо-Корасика и реализована программа, демонстрирующая его работу.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: my_lb4.cpp

```
#include <iostream>

#include "vector"

#include "map"
```

```
#include "string"

#include <cstring>

#include "algorithm"

const int k = 5;

std::vector <int> alpha_num =
{0,-1,1,-1,-1,-1,2,-1,-1,2,-1,-1,-1,4,-1,-1,-1,-1,-1,3,-1,-1,-1,-1,-1,-1};

std::vector <char> alpha = {'A', 'C', 'G', 'T', 'N'};

std::map <char, int> gen;

char sym_start = 'A';

struct Vertex {

   int next_vertex[k];

   int pat_num;

   bool terminal = false;

   int suffix_link;

   int auto_move[k];

   int par; //auto_move - запоминание перехода автомата, par - вершина-отец в
дереве

   char symb; //символ на ребре от par к этой вершине

   int suff_flink;

};



std::vector <Vertex> bohr;

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



Vertex make_bohr_vertex(int p, char c){

   Vertex v;

   //(255)=(2^8-1)=(все единицы в каждом байте памяти)=(-1 в дополнительном коде
целого 4-байтного числа int)

   memset(v.next_vertex, 255, sizeof(v.next_vertex));
```

```
    memset(v.auto_move, 255, sizeof(v.auto_move));
    v.terminal = false;
    v.suffix_link = -1;
    v.suff_flink = -1;
    v.par = p;
    v.symb = c;
    return v;
}



void bohr_ini(){
    //добавляем единственную вершину - корень
    bohr.push_back(make_bohr_vertex(0, '$'));
}



void add_str_to_bohr(std::string s){
    int num=0; //начинаем с корня
    for (int i=0; i<s.length(); i++){
        char ch = alpha_num[s[i]- sym_start]; //получаем номер в алфавите
        if (bohr[num].next_vertex[ch]==-1){ //-1 - признак отсутствия ребра
            bohr.push_back(make_bohr_vertex(num, ch));
            bohr[num].next_vertex[ch] = bohr.size()-1;
        }
        num=bohr[num].next_vertex[ch];
    }
    bohr[num].terminal = true;
```

```
    pattern.push_back(s);
    bohr[num].pat_num = pattern.size()-1;
}

bool is_string_in_bohr(std::string s){
    int num=0;
    for (int i=0; i < s.length(); i++){
        char ch = alpha_num[s[i] - sym_start];
        if (bohr[num].next_vertex[ch]==-1){
            return false;
        }
        num = bohr[num].next_vertex[ch];
    }
    return true;
}

int get_auto_move(int v, char ch);

int get_suff_link(int v){
    if (bohr[v].suffix_link==-1) //если еще не считали
        if (v==0 || bohr[v].par==0) //если v - корень или предок v - корень
            bohr[v].suffix_link=0;
        else
            bohr[v].suffix_link = get_auto_move(get_suff_link(bohr[v].par),
bohr[v].symb);
    return bohr[v].suffix_link;
}
```

```
int get_suff_flink(int v){
    if (bohr[v].suff_flink==-1){
        int u = get_suff_link(v);
        if (u==0) //либо v - корень, либо суф. ссылка v указывает на корень
            bohr[v].suff_flink=0;
        else
            bohr[v].suff_flink=(bohr[u].terminal)?u:get_suff_flink(u);
    }
    return bohr[v].suff_flink;
}

int get_auto_move(int v, char ch){
    if (bohr[v].auto_move[ch]==-1)
        if (bohr[v].next_vertex[ch]!=-1)
            bohr[v].auto_move[ch]=bohr[v].next_vertex[ch];
        else
        if (v==0)
            bohr[v].auto_move[ch]=0;
        else
            bohr[v].auto_move[ch]=get_auto_move(get_suff_link(v), ch);
    return bohr[v].auto_move[ch];
}
std::vector <std::pair <int, int>> ans;
void check(int v,int i){
    for(int u=v;u!=0;u=get_suff_flink(u)){
        if (bohr[u].terminal)
            ans.push_back(std::pair<int, int>
(i-pattern[bohr[u].pat_num].length()+1, bohr[u].pat_num + 1));
```

```
        //std::cout<<i-pattern[bohr[u].pat_num].length()+1<<" "<<bohr[u].pat_num
+ 1<<std::endl;

    }

}



void find_all_pos(std::string s){

    int u=0;

    for(int i=0;i<s.length();i++){

        u=get_auto_move(u,alpha_num[s[i]- sym_start]);

        check(u,i+1);

    }

}
bool cmp(std::pair <int, int> & p1, std::pair <int, int> & p2) {

    if(p1.first < p2.first) return true;

    if(p1.first == p2.first && p1.second < p2.second) return true;

    return false;

}



void create_bohr(std::string tmp, int pos, char jock) {

    //std::cout<<"Creating"<<std::endl;

    if(pos == tmp.length()) add_str_to_bohr(tmp);

    else if(tmp[pos] != jock) create_bohr(tmp, pos+1, jock);

    else if(tmp[pos] == jock) {

        for(auto i : gen) {

            tmp[pos] = i.first;

            create_bohr(tmp, pos + 1, jock);

        }

    }
```

```
}


void rec_create(std::string tmp, int pos, char jock) {
   if(pos == tmp.size() - 1) add_str_to_bohr(tmp);
   else if(tmp[pos] != jock) rec_create(tmp, pos + 1, jock);
   else if(tmp[pos] == jock) {
       for (auto sym: alpha) {
           if(sym != jock) {
               tmp[pos] = sym;
               rec_create(tmp, pos + 1, jock);
           }
       }
   }
}
void bohr_var(){
   std::string txt;
   std::cin>>txt;
   int n;
   std::cin>>n;
   bohr_ini();
   for(int i = 0; i < n; i++) {
       std::string tmp;
       std::cin>>tmp;
       add_str_to_bohr(tmp);
   }
   find_all_pos(txt);
   std::sort(ans.begin(), ans.end(), cmp);
```

```
    for(auto i : ans)
        std::cout<<i.first<<" "<<i.second<<std::endl;
}
void jock_var(){
    std::string txt;
    std::cin>>txt;
    for(int i = 0; i < txt.size(); i++) {
        if(gen.find(txt[i]) == gen.end()) {
            gen.insert({txt[i], gen.size()});
        }
    }
    bohr_ini();
    std::string tmp;
    char jock;
    std::cin>>tmp>>jock;
    create_bohr(tmp, 0, jock);
    find_all_pos(txt);
    for(auto i : ans)
        std::cout<<i.first<<std::endl;
}


int main() {
    //bohr_var();
    //jock_var();
    return 0;
}
```