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-Корасика

Студентка гр. 9382 _____ Балаева М.О.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Разработать программу, занимающуюся поиском шаблонов в строке, используя алгоритм Ахо-Корасика. Также разработать программу, которая ищет шаблон в строке, который содержит джокера.

Задание 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,$1 \le |T| \le 100000$).

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{ p_1,\ldots, p_n\} 1 \le |p_i| \le 75$

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:

```
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*P* необходимо найти все вхождения P в текст T.

Например, образец a*b*??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 $\{A,C,G,T,N\}$

Вход:

Текст (T,$1 \le |T| \le 100000$)

Шаблон (P,$1 \le |P| \le 40$)

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
```

Вариант 2. Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Описание алгоритма.

Описание алгоритма для первого задания.

Изначально на вход подаются: исходный текст, количество шаблонов, шаблоны.

По шаблонам строится бор. Бор — дерево, образованное последовательным добавлением всех образцов посимвольно. Построение начинается с корня, далее каждый символ шаблона добавляется в бор, если он встречается первый раз или нет возможности перехода по ребрам с соответствующими символами, если шаблон заканчивается на текущей вершине, она помечается терминальной.

Далее посимвольно считывается исходный текст , переходит по ребру с символом в следующую вершину, если есть прямой переход , если прямой переход отсутствует , выполняется переход по суффиксной ссылке. Суффиксная ссылка из вершины A — ссылка на вершину , соответствующую максимально длинному (под)образцу в автомате , являющемуся несобственным суффиксом (под)образца. Если суффиксная ссылка для вершины не найдена : проверяется является ли данная вершина корнем или его сыном, если является суффиксная ссылка будет равна 0, если не является о выполняется рекурсивный поиск, происходит переход в родителя текущей вершины выполняется переход по его суффиксной ссылке в вершину и из неё осуществляет переход по символу. После перехода в новую вершину инициализируется проверка текущей вершины и её суффиксных ссылок на терминальность, если попадаются терминальные вершины в массив ответов заносится индекс начала вхождения шаблона в исходный текст и его

порядковый номер. Далее считывается очередной символ текста, алгоритм прекратит работу, когда будут обработаны все символы исходного текста.
Запоминаем индекс символа последнего найденного шаблона в тексте, если индекс следующего найденного шаблона в тексте отличается от сохраненного индекса на меньше , чем длина следующего шаблона, следовательно эти два шаблона пересеклись в тексте. Количество вершин в автомате соответствует количеству вершин в боре.

Описание алгоритма для второго задания.

Изначально на вход подаются: исходный текст, шаблон с джокером, символ джокера. Далее шаблон разбивается на два (под)шаблона, которые добавляются в бор, как в первом алгоритме. Выделяется массив индексов, длина которого равна длине рассматриваемой строки, инициализированный нулями. Далее происходит посимвольный анализ исходного текста, как в первом алгоритме, однако(!) при попадании в терминальную вершину , что есть нахождение (под)шаблона в исходном тексте (под)шаблона, инкрементируется значение массива индексов соответствующее началу основного слова относительно найденного (под)шаблона. Получаем, что индексы тех ячеек массива, значение которых будет равно количеству (под)шаблонов в исходном шаблоне, и будут индексами вхождения заданного шаблона в строку. Количество вершин в автомате и пересечение шаблонов находятся аналогично , как в первом алгоритме.

Оценка сложности алгоритма.

Оценка сложности по памяти:

Так как автомат хранится, как индексный массив то O(M*P), где M — размер алфавита , P — суммарная длина всех шаблонов.

Оценка сложности по времени:

O(P*M + T+K), где M — размер алфавита , P — суммарная длина всех шаблонов, T - длина исходного текста, K — общая длина всех вхождений шаблонов в текст.

Тестирование.

Тестирование первой программы.

№	Входные данные	Выходные данные
1	NTAG 3 TAGT TAG T	2 2 2 3 The number of vertices in the automate: 5 List of found patterns that overlap with other found patterns in the search bar: T TAG
2	ALGORITHM 2 ALCO HMM	The search has not given any results. The number of vertices in the automate: 8 List of found patterns that overlap with other found patterns in the search bar, empty
3	ACGTNCTNC TNA 3 TNC NCTN NC	4 1 5 2 5 3 7 1 8 2 8 3 The number of vertices in the automate: 8 List of found patterns that overlap with other found patterns in the search bar: TNC NC NCTN
4.	TTCTAG 3 TT C TAG	1 1

		3 2 4 3 The number of vertices in the automate: 6 List of found patterns that overlap with other found patterns in the search bar, empty

Тестирование второй программы.

№	Входные данные	Выходные данные
1	ACTANCA A$$A$ $	The index of the occurrence of the pattern in the string: 1 The number of vertices in the automate: 2 A list of found patterns that overlap with other found patterns in the search bar , empty
2	ABVGAJHHABN A** *	The index of the occurrence of the pattern in the string: 1 The index of the occurrence of the pattern in the string: 5 The index of the occurrence of the pattern in the string: 9 The number of vertices in the automate: 2 A list of found patterns that overlap with other found patterns in the search bar , empty
3	ACACACACACCCA A$A $	The index of the occurrence of the pattern in the string: 1 The index of the occurrence of the pattern in the string: 3 The index of the occurrence of the pattern in the string: 5 The index of the occurrence of the pattern in the string: 7

		The number of vertices in the automate: 2 A list of found patterns that overlap with other found patterns in the search bar , empty
4	NATNATT NAT% %	The index of the occurrence of the pattern in the string: 1 The index of the occurrence of the pattern in the string: 4 The number of vertices in the automate: 4 A list of found patterns that overlap with other found patterns in the search bar , empty

Выводы.

Была разработана программа, занимающаяся поиском шаблонов в строке и находящая все ее вхождения, и был изучен алгоритм Ахо-Корасика. Также была реализована программа, ищущая шаблон в тексте, в котором содержится джокер.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ ДЛЯ ПЕРВОГО ЗАДАНИЯ.

Название файла: lab5_1.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include <algorithm>

#define N 50


class Bohr {
public:
    int next_vertex[N];
    int pattern_num;
    bool flag;
    int suff_link;
    int auto_move[N];
    int parent;
    char symbol;
    Bohr(int p, char c) : parent(p), symbol(c) {
        memset(next_vertex, 255, sizeof(next_vertex));
        flag = false;
        suff_link = -1;
        memset(auto_move, 255, sizeof(auto_move));
    }
};

std::vector<Bohr> bohr;
std::vector<std::string> pattern;
std::vector<std::pair<int, int>> ans;
std::vector<std::string> cross_Patterns;
int cross_count = 0;
int cross_patt = -1;

void init_bohr() {
    std :: cout << "Start building the bohr \n";
    std::cout<<"The root is denoted as #"<< std::endl;
    Bohr v(0,-30);
    bohr.push_back(v);
}


void add_pattern_to_bohr(const std::string& s) {
    int num = 0;
```

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

        std :: cout << "Consider the symbol : " << i + 1 <<"th: " << s[i] << "\
n";
        char ch = s[i]- 'A';
        if (bohr[num].next_vertex[ch] == -1) {
            Bohr v(num, ch);
            bohr.push_back(v);

            std::cout <<  "A vertex is added to the selection, to which the cur-
rent symbol of the pattern leads (" << char((v.symbol )+'A') << ") \n";
            bohr[num].next_vertex[ch] = bohr.size() - 1;
        }
        else{
            std::cout << "The vertex (" << s[i] << ") already exists for the
current pattern, you are navigating along it \n";}
        num = bohr[num].next_vertex[ch];
    }
    bohr[num].flag = true;
    std::cout << "The vertex to which the transition is made is terminal, the
construction of the boron branch is finished \n";
    pattern.push_back(s);
    bohr[num].pattern_num = pattern.size() - 1;

}

int get_auto_move(int v, char ch);


int get_suff_link(int v) {
    std::cout << "Calculating suffix and end links for vertex: " << char
(bohr[v].symbol + 'A')<<std::endl;
    if (bohr[v].suff_link == -1)
        if (v == 0 || bohr[v].parent == 0){
            bohr[v].suff_link = 0;
            std :: cout << "Suffix reference points to root(#) \n";
        }
    else{

        std :: cout << "Follow the parent's suffix link \n";
        std::cout<<"Parent is "<< bohr[v].parent << std::endl;
        bohr[v].suff_link = get_auto_move(get_suff_link(bohr[v].parent),
bohr[v].symbol);
    }
        return bohr[v].suff_link;
}
```

```
int get_auto_move(int v, char ch) {

    std::cout << "Looking for a path from the vertex " << char (bohr[v].symbol +
'A') << " along the edge with the symbol " << char(ch +'A')  << ". \n";
    if (bohr[v].auto_move[ch] == -1)
        if (bohr[v].next_vertex[ch] != -1){
            std :: cout << "Go to the next node with the corresponding character
\n";
            bohr[v].auto_move[ch] = bohr[v].next_vertex[ch];
        }
        else
            if (v == 0){
                std :: cout << "The next node with the matching character was
not found, go to the root node(#) \n";
                bohr[v].auto_move[ch] = 0;
            }
            else{
                std :: cout << "The next node with the matching character was
not found, let's follow the suffix link \n";
                bohr[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
            }
            return bohr[v].auto_move[ch];
}

void checkout(int v, int i) {

    for (int u = v; u != 0; u = get_suff_link(u)) {
        if (bohr[u].flag){
            std :: cout << "Found pattern in text \n";
            std::cout << "Position: " << i -
pattern[bohr[u].pattern_num].length() + 1 << ", Number of pattern: " <<
bohr[u].pattern_num + 1 << "\n";
            ans.push_back(std::pair<int, int>(i -
pattern[bohr[u].pattern_num].length() + 1, bohr[u].pattern_num + 1));
            if (cross_patt != -1 && i - cross_count <
pattern[bohr[u].pattern_num].length() + pattern[cross_patt].length()) {
                if (std::find(cross_Patterns.begin(), cross_Patterns.end(), pat-
tern[cross_patt]) == cross_Patterns.end())
                    cross_Patterns.push_back(pattern[cross_patt]);
                if (std::find(cross_Patterns.begin(), cross_Patterns.end(), pat-
tern[bohr[u].pattern_num]) == cross_Patterns.end())
                    cross_Patterns.push_back(pattern[bohr[u].pattern_num]);
            }
            cross_patt = bohr[u].pattern_num;
            cross_count = i - pattern[bohr[u].pattern_num].length();

        }
```

```
    }
}

void find_all_pos(const std::string& s) {
    int u = 0;
    for (int i = 0; i<s.length(); i++) {
        std::cout << "Consider the line with the current position: ";
            for (int j = 0; j < s.length(); j++)
                if (j == i)
                    std::cout << "'" << s[j] << "'";
                else
                    std::cout << s[j];
        std::cout<<std::endl;
        u = get_auto_move(u, s[i] - 'A');
        checkout(u, i + 1);
    }

}
void printer(const std ::string& s){
    find_all_pos(s);
    std::sort(ans.begin(),ans.end());
    if(ans.size()==0){
        std::cout<<"The search has not given any results."<<std::endl;
    }
    for (int i = 0; i < ans.size(); ++i) {
        std::cout << ans[i].first << " " << ans[i].second << std::endl;
    }
    std::cout << "The number of vertices in the automate: " << bohr.size() << "\
n";
    std::cout << "List of found patterns that overlap with other found patterns
in the search bar";
    if (cross_Patterns.empty())
        std::cout << ", empty\n";
    else {
        std::cout << ":\n";
        for (auto& i : cross_Patterns) {
            std::cout << i << " ";
        }
    }

}


int main()
{

    std::string text;
    std::string pattern;
```

```
    int n;
    std::cin >> text >> n ;
    std::cout<<std::endl;
    init_bohr();
    for (int i = 0; i<n; i++) {
        std::cin >> pattern;
        std::cout<<"Pattern number: "<< i + 1 <<": "<<pattern<<std::endl;
        add_pattern_to_bohr(pattern);
    }
    printer(text);

    return 0;
}
```

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ ДЛЯ ВТОРОГО ЗАДАНИЯ.

Название файла: lab5_2.cpp

```
#include <iostream>
#include <vector>
#include <sstream>
#include <string>
#include <cstring>
#define N 50

class Bohr {
public:
    int next_vertex[N];
    std::vector<int> pattern_num;
    bool flag;
    int suff_link;
    int auto_move[N];
    int parent;
    char symbol;
    Bohr(int p, char c) : parent(p), symbol(c) {
        memset(next_vertex, 255, sizeof(next_vertex));
        flag = false;
        suff_link = -1;
        memset(auto_move, 255, sizeof(auto_move));
        pattern_num.resize(0);
    }
};

std::vector<Bohr> bohr;
```

```
int cross_count = -1;
bool is_cross = false;
void init_bohr() {
    std :: cout << "Start building the bohr \n";
    std::cout<<"The root is denoted as #"<< std::endl;
    Bohr v(0,-30);
    bohr.push_back(v);
}


void add_pattern_to_bohr(const std::string& s, int& count) {
    int num = 0;
    for (int i = 0; i<s.length(); i++) {
        std :: cout << "Consider the symbol : " << i + 1 <<"th: " << s[i] << "\n";
        char ch = s[i] - 'A';
        if (bohr[num].next_vertex[ch] == -1) {
            Bohr v(num, ch);
            bohr.push_back(v);
            std::cout <<  "A vertex is added to the selection, to which the current sym-
bol of the pattern leads (" << char((v.symbol )+'A') << ") \n";
            bohr[num].next_vertex[ch] = bohr.size() - 1;
        }
        else{
            std::cout << "The vertex (" << s[i] << ") already exists for the current pat-
tern, you are navigating along it \n";}
        num = bohr[num].next_vertex[ch];
    }
    bohr[num].flag = true;
```

```
   std::cout << "The vertex to which the transition is made is terminal, the con-
struction of the boron branch is finished \n";
   bohr[num].pattern_num.push_back(++count);

std::cout<<"-----------------------------------------------------------------------------------
-----------------------"<<std::endl;

std::cout<<"-----------------------------------------------------------------------------------
-----------------------"<<std::endl;
}

int get_auto_move(int v, char ch);

int get_suff_link(int v) {
   std::cout << "Calculating suffix and end links for vertex: " << char
(bohr[v].symbol + 'A')<<std::endl;
   if (bohr[v].suff_link == -1)
      if (v == 0 || bohr[v].parent == 0){
         std :: cout << "Suffix reference points to root(#) \n";
         bohr[v].suff_link = 0;
      }
   else{
      std :: cout << "Follow the parent's suffix link \n";
      std::cout<<"Parent is "<< bohr[v].parent << std::endl;
      bohr[v].suff_link = get_auto_move(get_suff_link(bohr[v].parent),
bohr[v].symbol);
   }
      return bohr[v].suff_link;
}
```

```
int get_auto_move(int v, char ch) {
   std::cout << "Looking for a path from the vertex " << char (bohr[v].symbol +
'A') << " along the edge with the symbol " << char(ch +'A')  << ". \n";
   if (bohr[v].auto_move[ch] == -1)
      if (bohr[v].next_vertex[ch] != -1){
         std :: cout << "Go to the next node with the corresponding character \n";
         bohr[v].auto_move[ch] = bohr[v].next_vertex[ch];
      }
      else
         if (v == 0){
            std :: cout << "The next node with the matching character was not found,
go to the root node(#) \n";
            bohr[v].auto_move[ch] = 0;
         }
         else{
            std :: cout << "The next node with the matching character was not found,
let's follow the suffix link \n";
            bohr[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
         }
         return bohr[v].auto_move[ch];
}

void checkout(int u, int i, const std::string& s, std::vector<int>& count,
std::vector<int>&pattern_offset_mass, int pat_len, std::vector<std::string>&
pat_mass,  std::string pattern){
   for (int tmp = u; tmp != 0; tmp = get_suff_link(tmp)) {
      if (bohr[tmp].flag) {
         for (int j = 0; j < bohr[tmp].pattern_num.size(); j++) {
```

```
            if ((i + 1 - pattern_offset_mass[bohr[tmp].pattern_num[j] - 1] -
pat_mass[bohr[tmp].pattern_num[j] - 1].size() >= 0) && (i + 1 -
pattern_offset_mass[bohr[tmp].pattern_num[j] - 1] -
pat_mass[bohr[tmp].pattern_num[j] - 1].size() <= s.size() - pat_len)) {
                count[i + 1 - pattern_offset_mass[bohr[tmp].pattern_num[j] - 1] -
pat_mass[bohr[tmp].pattern_num[j] - 1].size()]++;
                if (count[i + 1 - pattern_offset_mass[bohr[tmp].pattern_num[j] - 1] -
pat_mass[bohr[tmp].pattern_num[j] - 1].size()] == pattern_offset_mass.size()) {

                }
            }
        }

    }
  }
}

void find_all_pos(const std::string& s, std::vector<int>& count,
std::vector<int>&pattern_offset_mass, int pat_len, std::vector<std::string>&
pat_mass, std::string pattern) {
   int u = 0;
   for (size_t i = 0; i < s.size(); i++) {
      std::cout << "Consider the line with the current position: ";
      for (int j = 0; j < s.length(); j++)
         if (j == i)
            std::cout << "'" << s[j] << "'";
         else
            std::cout << s[j];
       std::cout<<std::endl;
```

```
        u = get_auto_move(u, s[i] - 'A');
        checkout(u, i, s, count, pattern_offset_mass,  pat_len,  pat_mass, pattern );

std::cout<<"-------------------------------------------------------------------------------
-----------------------"<<std::endl;
    }
}

void split_pattern(std::string pattern, char joker, std::vector<std::string>& pat_mas,
std::vector<int>& pattern_offset_mas){
    std::cout << "Break the original pattern by jokers, into (sub)patterns. \n";
    std::string buf = "";
    for (int i = 0; i < pattern.size(); i++) {
        if (pattern[i] == joker){
            if (buf.size() > 0) {
                pat_mas.push_back(buf);
                std::cout << "Found a new subpattern: " << buf << "\n";
                pattern_offset_mas.push_back(i - buf.size());
                buf = "";
            }
        }
        else {
            buf.push_back(pattern[i]);
            if (i == pattern.size() - 1) {

                std::cout << "Found a new subpattern:" << buf << "\n";
                pat_mas.push_back(buf);

                pattern_offset_mas.push_back(i - buf.size() + 1);
```

```
            std::cout << "Starting index in the original pattern:" << i - buf.size () + 1
<< "\n";
          }
       }
   }
}

void out(std::vector<int>& count, int pat_counter, std::string& s, std::string pat-
tern) {
   int k = 0;
   for (int i = 0; i < count.size(); i++) {
      if (count[i] == pat_counter) {
         std::cout<<"The index of the occurrence of the pattern in the
string:"<<std::endl;
         std::cout << i + 1 << "\n";

      } else k++;
   }
   if(k==count.size()){
      std::cout<<"The search has not given any results."<<std::endl;
   }
   std::cout<<"The number of vertices in the automate: " <<bohr.size()<<std::endl;
   std::cout<<"A list of found patterns that overlap with other found patterns in the
search bar";
   if (!is_cross)
      std::cout<<" , empty"<< std::endl;
   else{
      std::cout<<":\n";
      std::cout<<pattern<<std::endl;
```

```
    }
}

int main()
{
    init_bohr();
    std::string text;
    std::string pattern;
    std::vector<int> pattern_offset_mass;
    char joker;
    std::cin >> text >> pattern >> joker;
    std::vector<int> count(text.size(), 0);
    std::vector<std::string> patt_mass;
    split_pattern(pattern, joker, patt_mass, pattern_offset_mass);
    int k = 0;
    for (auto pat : patt_mass) {
        add_pattern_to_bohr(pat, k);
    }
    find_all_pos(text, count, pattern_offset_mass,  pattern.size(),  patt_mass, pat-
tern );

    std::cout<<std::endl;
    out(count, patt_mass.size(), text, pattern);
    return 0;
}
```