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 _____ Орлов Д.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритм Ахо-Корасик поиска набора образцов в строке, применить его в решении поставленной задачи на языке программирования C++. Реализовать тестирование программы.

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

```
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 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≤|*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

Основные теоретические положения.

Пусть дан набор строк в алфавите размера k суммарной длины m. Алгоритм Ахо-Корасик строит для этого набора строк структуру данных "бор", а затем по этому бору строит автомат, всё за O(m) времени и O(m*k) памяти. Полученный автомат уже может использоваться в различных задачах, простейшая из которых — это нахождение всех вхождений каждой строки из данного набора в некоторый текст за линейное время.

Сложности алгоритма по операциям O(n*A + T + k), где n – общая длина всех слов в словаре, A – размер алфавита, T – длина текста, в котором проводится поиск, k – общая длина всех совпадений

Выполнение работы.

Для представления вершины бора был реализован класс *Vertex*. Данный класс содержит в себе следующие поля:

- char symb, которое хранит имя узла;
- vertex* suffLink – суффиксная ссылка;
- Vertex* parent – указатель на предка вершины;
- std::vector<Vertex*> nextVertices, представляющее из себя вектор указателей на узлы-потомки;
- bool isWord – показывает, является ли вершина концом слова.
- int pattern_num – номер паттерна для данной вершины.
- std::vector<std::pair<char, Vertex*>> autoMove – для запоминания перехода автомата.

В программе реализованы следующие функции:

- *void bohrIni(std::vector<Vertex*> &bohr)* – создает начальную вершину бора.
- *Vertex* findNextVertex(Vertex* &v, char c)* – возвращает указатель на предка данной вершины с определенным именем.
- *void addNextVertex(Vertex* &v1, Vertex* v2)* – добавляет ребро между двумя вершинами.
- *int findInBohr(std::vector<Vertex*> &bohr, char c)* – ищет вершину с определенным именем в бору.
- *void addStringToBohr(std::string &inputStr, std::vector<Vertex*> &bohr, std::vector<std::string> &pattern)* – добавляет строку в бор.
- *bool isStringInBohr(std::string &potentialStr, std::vector<Vertex*>*

&bohr) – проверяет наличие строки в бору.

- *Vertex* getSuffLink(Vertex* v)* – находит суффиксную ссылку.
- *Vertex* findInAutoMove(Vertex* &v, char c)* – возвращает вершину, куда перешел автомат.
- *void addToAutoMove(Vertex* v, char c, Vertex* nextVertex)* – добавляет переход в массив.
- *Vertex* getAutoMove(Vertex* v, char c)* – реализует переход автомата.
- *void addToAnswer(std::vector <std::pair<int, int>> &output, int a, int b)* – формирует выходной массив
- *void check(Vertex* v, int i, std::vector<std::string> &pattern, std::vector <std::pair<int, int>> &output)* и *std::vector <std::pair<int, int>> Aho_Corasick(std::string &inputStr, std::vector<Vertex*> &bohr, std::vector<std::string> &pattern)* – реализуют поиск по автомату.
- *void printBohr(std::vector<Vertex*> &bohr)* – выводит бор.
- *int cmp(const void* a, const void* b)* – компаратор для сортировки выходного массива.

Разработанный программный код см. в приложении А.

Тестирование.

Задание1.

1) Входные данные:

NTAG
3
TAGT
TAG
T

Выходные данные:

2 2
2 3

2) Входные данные:

ACGT
3
CGTA
GT
A

Выходные данные:

1 3
3 2

3) Входные данные:

GGTA
3
GTA
AG
T

Выходные данные:

2 1

3 3

Задание2.

1) Входные данные:

ACTANCA
A$$A$
$

Выходные данные:

1

2) Входные данные:

CGTTTTNCGAS
CG**
*

Выходные данные:

1
8

3) Входные данные:

TNAAAGCAAGAAG
AA&
&

Выходные данные:

3
4
8
11

Выводы.

В ходе работы был изучен алгоритм Ахо-Корасик поиска набора образцов в строке, применён в решении поставленной задачи на языке программирования C++.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Задание 1:

Название файла: lab5.cpp

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

struct Vertex
{
    char symb;
    Vertex* suffLink = nullptr;
    Vertex* goodSuffLink = nullptr;
    Vertex* parent;
    std::vector<Vertex*> nextVertices;
    std::vector<std::pair<char, Vertex*>> autoMove;
    bool isWord = false;
    int pattern_num;

    Vertex(char symbV = '@', Vertex* parentV = nullptr)
    {
        this->symb = symbV;
        this->parent = parentV;
    }
};

int cmp(const void* a, const void* b)
{
    int a1 = (*(std::pair<int, int>*)a).first;
    int a2 = (*(std::pair<int, int>*)a).second;

    int b1 = (*(std::pair<int, int>*)b).first;
    int b2 = (*(std::pair<int, int>*)b).second;

    if(a1 == b1)
    {
        return a2 - b2;
    }
    else
    {
        return a1 - b1;
    }
}

void bohrIni(std::vector<Vertex*> &bohr)
{
    Vertex* v = new Vertex;
    bohr.push_back(v);
}

Vertex* findNextVertex(Vertex* &v, char c)
{
    for(int i = 0; i < v->nextVertices.size(); i++)
    {
        if(v->nextVertices[i]->symb == c)
```

```
        {
            return v->nextVertices[i];
        }
    }
    return nullptr;
}

void addNextVertex(Vertex* &v1, Vertex* v2)
{
    v1->nextVertices.push_back(v2);
}

int findInBohr(std::vector<Vertex*> &bohr, char c)
{
    for(int i = bohr.size() - 1; i >= 0; i--)
    {
        if(bohr[i]->symb == c)
        {
            return i;
        }
    }
    return -1;
}

void addStringToBohr(std::string &inputStr, std::vector<Vertex*>
&bohr,  std::vector<std::string> &pattern)
{
    Vertex* currVertex = bohr[0];
    char ch;
    for(int i = 0; i < inputStr.length(); i++)
    {
        ch = inputStr[i];
        Vertex* nextVertex = findNextVertex(currVertex, ch);
        if(nextVertex == nullptr)
        {
            Vertex* v = new Vertex(ch, currVertex);
            bohr.push_back(v);
            addNextVertex(currVertex, v);
            currVertex = v;
        }
        else
        {
            currVertex = nextVertex;
        }
    }
    currVertex->isWord = true;
    pattern.push_back(inputStr);
    currVertex->pattern_num = pattern.size() - 1;
}

bool isStringInBohr(std::string &potentialStr, std::vector<Vertex*>
&bohr)
{
    Vertex* currVertex = bohr[0];
    char ch;
    for(int i = 0; i < potentialStr.length(); i++)
    {
        ch = potentialStr[i];
        Vertex* nextVertex = findNextVertex(currVertex, ch);
        if(nextVertex == nullptr)
```

```
        {
            return false;
        }
        currVertex = nextVertex;
    }
    return true;
}

Vertex* getAutoMove(Vertex* v, char c);

Vertex* getSuffLink(Vertex* v)
{
    if(v->suffLink == nullptr)
    {
        if(v->parent == nullptr)
        {
            v->suffLink = v;
        }
        else if(v->parent->parent == nullptr)
        {
            v->suffLink = v->parent;
        }
        else
        {
            v->suffLink = getAutoMove(getSuffLink(v->parent), v-
>symb);
        }
    }
    return v->suffLink;
}

Vertex* findInAutoMove(Vertex* &v, char c)
{
    for(int i = 0; i < v->autoMove.size(); i++)
    {
        if(v->autoMove[i].first == c)
        {
            return v->autoMove[i].second;
        }
    }
    return nullptr;
}

void addToAutoMove(Vertex* v, char c, Vertex* nextVertex)
{
    v->autoMove.push_back(std::make_pair(c, nextVertex));
}

Vertex* getAutoMove(Vertex* v, char c)
{
    Vertex* nextAuto = findInAutoMove(v,c);
    if(nextAuto == nullptr)
    {
        Vertex* nextVertex = findNextVertex(v, c);
        if(nextVertex != nullptr)
        {
            addToAutoMove(v, c, nextVertex);
        }
        else
        {
```

```
            if(v->parent == nullptr)
            {
                addToAutoMove(v, c, v);
            }
            else
            {
                addToAutoMove(v, c, getAutoMove(getSuffLink(v), c));
            }
        }
    }
    return findInAutoMove(v, c);
}

Vertex* getGoodSuffLink(Vertex* v)
{
    if(v->goodSuffLink == nullptr)
    {
        Vertex* u = getSuffLink(v);
        if(u->parent == nullptr)
        {
            v->goodSuffLink = nullptr;
        }
        else if(u->isWord == true)
        {
            v->goodSuffLink = getSuffLink(u);
        }
    }
    return v->goodSuffLink;
}

void addToAnswer(std::vector <std::pair<int, int>> &output, int a,
int b)
{
    output.push_back(std::make_pair(a,b));
}

void check(Vertex* v, int i, std::vector<std::string> &pattern,
std::vector <std::pair<int, int>> &output)
{

    for(Vertex* u = v; u->parent != nullptr; u = getSuffLink(u))
    {
        if(u->isWord == true)
        {
            addToAnswer(output, i - pattern[u->pattern_num].length()
+ 1, u->pattern_num + 1);
        }
    }
}

std::vector <std::pair<int, int>> Aho_Corasick(std::string
&inputStr, std::vector<Vertex*> &bohr, std::vector<std::string>
&pattern)
{
    Vertex* u = bohr[0];
    std::vector <std::pair<int, int>> output;
    for(int i = 0; i < inputStr.length(); i++)
    {
        u = getAutoMove(u, inputStr[i]);
        check(u, i+1, pattern, output);
```

```
    }
    return output;
}

void printBohr(std::vector<Vertex*> &bohr)
{
    for(auto i : bohr)
    {
        std::cout<<i->symb<<": \n";
        for(int j = 0; j < i->nextVertices.size(); j++)
        {
            std::cout<<"    "<<i->nextVertices[j]->symb<<"("<<i-
>nextVertices[j]->parent->symb<<")\n";
        }
    }
}

int main()
{
    int n;
    std::vector <Vertex*> bohr;
    std::vector <std::string> pattern;
    std::vector <std::pair<int, int>> answer;
    std::string inputStr, s;
    bohrIni(bohr);

    std::cin>>inputStr>>n;

    for(int i = 0; i < n; i++)
    {
        std::cin>>s;
        addStringToBohr(s, bohr, pattern);
    }
    answer = Aho_Corasick(inputStr,bohr,pattern);

    std::qsort(&answer[0], answer.size(), sizeof(std::pair<int,
int>), cmp);

    for(auto i : answer)
    {
        std::cout<<i.first<<" "<<i.second<<"\n";
    }



    for(int i = 0; i < bohr.size(); i++)
    {
        delete bohr[i];
    }

    return 0;
}
```

Задание 2:

Название файла: lab5_joker.cpp

```
#include <iostream>
#include <vector>
```

```
#include <algorithm>

struct Vertex
{
    char symb;
    Vertex* suffLink = nullptr;
    Vertex* goodSuffLink = nullptr;
    Vertex* parent;
    std::vector<Vertex*> nextVertices;
    std::vector<std::pair<char, Vertex*>> autoMove;
    bool isWord = false;
    int pattern_num;

    Vertex(char symbV = '@', Vertex* parentV = nullptr)
    {
        this->symb = symbV;
        this->parent = parentV;
    }
};

void bohrIni(std::vector<Vertex*> &bohr)
{
    Vertex* v = new Vertex;
    bohr.push_back(v);
}

Vertex* findNextVertex(Vertex* &v, char c)
{
    for(int i = 0; i < v->nextVertices.size(); i++)
    {
        if(v->nextVertices[i]->symb == c)
        {
            return v->nextVertices[i];
        }
    }
    return nullptr;
}

void addNextVertex(Vertex* &v1, Vertex* v2)
{
    v1->nextVertices.push_back(v2);
}

int findInBohr(std::vector<Vertex*> &bohr, char c)
{
    for(int i = bohr.size() - 1; i >= 0; i--)
    {
        if(bohr[i]->symb == c)
        {
            return i;
        }
    }
    return -1;
}

void addStringToBohr(std::string &inputStr, std::vector<Vertex*>
&bohr,  std::vector<std::pair<std::string, int>> &pattern, int
index)
{
    Vertex* currVertex = bohr[0];
```

```
    char ch;
    for(int i = 0; i < inputStr.length(); i++)
    {
        ch = inputStr[i];
        Vertex* nextVertex = findNextVertex(currVertex, ch);
        if(nextVertex == nullptr || i == (inputStr.length() - 1))
        {
            Vertex* v = new Vertex(ch, currVertex);
            bohr.push_back(v);
            addNextVertex(currVertex, v);
            currVertex = v;
        }
        else
        {
            currVertex = nextVertex;
        }
    }
    currVertex->isWord = true;
    pattern.push_back(std::make_pair(inputStr, index));
    currVertex->pattern_num = pattern.size() - 1;
}

bool isStringInBohr(std::string &potentialStr, std::vector<Vertex*>
&bohr)
{
    Vertex* currVertex = bohr[0];
    char ch;
    for(int i = 0; i < potentialStr.length(); i++)
    {
        ch = potentialStr[i];
        Vertex* nextVertex = findNextVertex(currVertex, ch);
        if(nextVertex == nullptr)
        {
            return false;
        }
        currVertex = nextVertex;
    }
    return true;
}

Vertex* getAutoMove(Vertex* v, char c);

Vertex* getSuffLink(Vertex* v)
{
    if(v->suffLink == nullptr)
    {
        if(v->parent == nullptr)
        {
            v->suffLink = v;
        }
        else if(v->parent->parent == nullptr)
        {
            v->suffLink = v->parent;
        }
        else
        {
            v->suffLink = getAutoMove(getSuffLink(v->parent), v-
>symb);
        }
    }
```

```
    return v->suffLink;
}

Vertex* findInAutoMove(Vertex* &v, char c)
{
    for(int i = 0; i < v->autoMove.size(); i++)
    {
        if(v->autoMove[i].first == c)
        {
            return v->autoMove[i].second;
        }
    }
    return nullptr;
}

void addToAutoMove(Vertex* v, char c, Vertex* nextVertex)
{
    v->autoMove.push_back(std::make_pair(c, nextVertex));
}

Vertex* getAutoMove(Vertex* v, char c)
{
    Vertex* nextAuto = findInAutoMove(v,c);
    if(nextAuto == nullptr)
    {
        Vertex* nextVertex = findNextVertex(v, c);
        if(nextVertex != nullptr)
        {
            addToAutoMove(v, c, nextVertex);
        }
        else
        {
            if(v->parent == nullptr)
            {
                addToAutoMove(v, c, v);
            }
            else
            {
                addToAutoMove(v, c, getAutoMove(getSuffLink(v), c));
            }
        }
    }
    return findInAutoMove(v, c);
}

Vertex* getGoodSuffLink(Vertex* v)
{
    if(v->goodSuffLink == nullptr)
    {
        Vertex* u = getSuffLink(v);
        if(u->parent == nullptr)
        {
            v->goodSuffLink = nullptr;
        }
        else if(u->isWord == true)
        {
            v->goodSuffLink = getSuffLink(u);
        }
    }
    return v->goodSuffLink;
```

```
}


void check(Vertex* v, int i, std::vector<std::pair<std::string,
int>> &pattern, std::vector<int> &counter)
{
    for(Vertex* u = v; u->parent != nullptr; u = getSuffLink(u))
    {
        if(u->isWord == true)
        {
            int index = i - pattern[u->pattern_num].first.length() -
pattern[u->pattern_num].second + 1;
            if(index >= 0)
            {
                counter[index] ++;
            }
        }
    }
}

void Aho_Corasick(std::string &inputStr, std::vector<Vertex*> &bohr,
std::vector<std::pair<std::string, int>> &pattern, std::vector<int>
&counter)
{
    Vertex* u = bohr[0];
    for(int i = 0; i < inputStr.length(); i++)
    {
        u = getAutoMove(u, inputStr[i]);
        check(u, i, pattern, counter);
    }
}

void printBohr(std::vector<Vertex*> &bohr)
{
    for(auto i : bohr)
    {
        std::cout<<i->symb<<"("<<i->isWord<<")\n";
        for(int j = 0; j < i->nextVertices.size(); j++)
        {
            std::cout<<"    "<<i->nextVertices[j]->symb<<"("<<i-
>nextVertices[j]->parent->symb<<")\n";
        }
    }
}

int main()
{
    char c;
    std::vector <Vertex*> bohr;
    std::vector <std::pair<std::string,int>> pattern;
    std::string inputStr, s, str = "";
    bohrIni(bohr);

    std::cin>>inputStr;
    std::cin>>s>>c;
    std::vector<int> counter(inputStr.length(), 0);
    int start = 0, amount = 0;

    s+= c;
```

```
    for(int i = 0; i < s.length(); i++)
    {
        if(s[i] != c)
        {
            str += s[i];
        }
        else
        {
            if(str != "")
            {
                amount++;
                addStringToBohr(str, bohr, pattern, start);
                Aho_Corasick(inputStr, bohr, pattern, counter);
                for(int i = 0; i < bohr.size(); i++)
                {
                    delete bohr[i];
                }
                bohr.clear();
                bohrIni(bohr);
                str = "";
            }
            start = i+1;
        }
    }


    for(int i = 0; i < counter.size(); i++)
    {
        if(counter[i] == amount && ((i + s.length() - 1) <=
inputStr.length()))
        {
            std::cout<<i+1<<"\n";
        }
    }


    for(int i = 0; i < bohr.size(); i++)
    {
        delete bohr[i];
    }

    return 0;
}
```