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 алгоримов»

Тема: Алгоритм Ахо-Корасика

Студент гр. 8304 _____ Матросов Д.В.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Изучить алгоритм Ахо-Корасик для оптимального поиска всех вхождений данных подстрок в строку.

Задание 1.

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 $P = \{p_1, \ldots, p_n\} 1 \leq |p_i| \leq 75$.

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.

Задание 2.

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 T.

Например, образец $ab??c?$ 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 ($T, 1 \leq |T| \leq 100000$).

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.

Описание алгоритма.

Бор — это дерево, в котором каждая вершина обозначает какую-то строку (корень обозначает нулевую строку — ε). На ребрах между вершинами написана 1 буква (в этом его принципиальное различие с суффиксными деревьями и др.), таким образом, добираясь по ребрам из корня в какую-нибудь вершину и контангенируя буквы из ребер в порядке обхода, мы получим строку, соответствующую этой вершине. Строить бор будем последовательным добавление исходных строк. Изначально у нас есть 1 вершина, корень (root) — пустая строка. Добавление строки происходит так: начиная в корне, двигаемся по нашему дереву, выбирая каждый раз ребро, соответствующее очередной букве строки. Если такого ребра нет, то мы создаем его вместе с вершиной. Бор играет роль автомата, который будет использоваться при обработке текста. Обрабатываться он будет посимвольно, из корня мы стараемся перейти в состояние автомата, соответствующее этому символу, если оно не рассчитано возвращаемся по суффиксной ссылке. После получения вершины, отвечающей за следующее состояние, проходим до корня бора по ссылкам. Если на пути мы нашли вершину-лист, то значение в векторе шаблонов увеличивается. При условии, что оно равно количеству подстрок в инзначальном шаблоне, в этой позиции и начинается искомый шаблон.

Сложность алгоритма O(m+n+a), где m — длина текста, n — общая длина подстрок, a — количество вхождений.

Тестирование 1:

Ввод	Вывод
NTAG	2 2 2 3

3 TAGT TAG T	
asdfad 2 a adf	5 1 1 1
afafgj 4 gh sd qw b	-

Тестирование 2:

Ввод	Вывод
ACTANCA A$$A$ $	1
asfgsgds g$ $	4 6

Выводы.

В результате лабораторной работы были изучены алгоритм Ахо-Корасик для поиска вхождений нескольких подстрок в строку.

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


//struct Top
//{
//    std::map<char, int> next;
//    std::map<char, int> go;
//    int prev = 0;
//    char prevChar = 0;
//    int suffix = -1;
//    int number = 0;
//    int deep = 0;
//    bool isLeaf = false;
//};
//
//class AhoCorasick
//{
//public:
//
//    AhoCorasick() {
//        Top root;
//        root.prev = -1;
//        tops.push_back(root);
//        countTop = 0;
//    }
//
```

```
//    void addPattern(const std::string& str)
//    {
//        currentTop = 0;
//        for (char i : str) {
//                  if (tops[currentTop].next.find(i) ==
tops[currentTop].next.end()) {
//                Top vertex;
//                vertex.prev = currentTop;
//                vertex.prevChar = i;
//                tops.push_back(vertex);
//                 tops[currentTop].next[i] = tops.size()
- 1;
//            }
//            currentTop = tops[currentTop].next[i];
//        }
//
//        countTop++;
//        patternsArr.push_back(str);
//        tops[currentTop].number = countTop;
//        tops[currentTop].isLeaf = true;
//        tops[currentTop].deep = str.size();
//    }
//
//    void search(const std::string& str)
//    {
//        int curr = 0;
//        bool terminalVertexFound;
//        for (size_t i = 0; i < str.size(); i++) {
//            curr = getGo(curr, str[i]);
//            terminalVertexFound = false;
```

```
//                 for (int tmp = curr; tmp != 0; tmp =
getSuffix(tmp)) {
//                 if (tops[tmp].isLeaf) {
//                     result.push_back(i - tops[tmp].deep
+ 2);
//                     result.push_back(tops[tmp].number);
//                     terminalVertexFound = true;
//                     break;
//                 }
//             }
//         }
//     }
//
//     void printResult(const std::string& text) const {
//         std::vector<bool> cutStr(text.size());
//         std::string textRest;
//         for (int i = result.size()-1; i >= 1; i -= 2) {
//           std::cout << result[i - 1] << " " << result[i]
<< std::endl;
//             for (int j = 0; j < patternsArr[result[i] -
1].size(); j++)
//                 cutStr[result[i - 1] - 1 + j] = true;
//         }
//
//         for (int i = 0; i < cutStr.size(); i++) {
//             if (!cutStr[i])
//                 textRest.push_back(text[i]);
//         }
//     }
//
//private:
```

```
//
//    int getSuffix(int index)
//    {
//        if (tops[index].suffix == -1) {
//            if (index == 0 || tops[index].prev == 0) {
//                tops[index].suffix = 0;
//            }
//            else {
//                                    tops[index].suffix  =
getGo(getSuffix(tops[index].prev), tops[index].prevChar);
// иначе переходим ищем суффикс через суффикс родителя
//            }
//        }
//
//
//        return tops[index].suffix;
//    }
//
//    int getGo(int index, char ch)
//    {
//                        if   (tops[index].go.find(ch)   ==
tops[index].go.end()) {
//                        if  (tops[index].next.find(ch)  !=
tops[index].next.end()) {
//              tops[index].go[ch] = tops[index].next[ch];
//
//            }
//            else {
//                if (index == 0) {
//                    tops[index].go[ch] = 0;
//
```

```
//                }
//                else {
//
//                                        tops[index].go[ch]  =
getGo(getSuffix(index), ch);
//                }
//            }
//        }
//        return tops[index].go[ch];
//    }
//
//private:
//    std::vector<Top> tops;
//    std::vector<int> result;
//    int countTop;
//    int currentTop;
//    std::vector<std::string> patternsArr;
//};
//
//
//int main() {
//    std::string str;
//    int count = 0;
//    std::cin >> str;
//    std::cin >> count;
//
//    std::string pattern;
//    std::vector<std::string> patterns;
//    for (int i = 0; i < count; i++) {
//        std::cin >> pattern;
```

```
//        patterns.push_back(pattern);
//    }
//    auto* ahoCorasick = new AhoCorasick();
//    for (int i = 0; i < count; i++) {
//        ahoCorasick->addPattern(patterns[i]);
//    }
//    ahoCorasick->search(str);
//    ahoCorasick->printResult(str);
//
//    return 0;
//}
//
struct Top
{
    std::map<char, int> next;
    std::map<char, int> go;
    std::vector<int> number;
    int prev = 0;
    int deep = 0;
    int suffix = -1;
    bool isLeaf = false;
    char prevChar = 0;
};

class AhoCorasick
{
public:
    explicit        AhoCorasick(char        joker)        :
matchPatterns(1100000) {
        Top root;
```

```
        root.prev = -1;
        Tops.push_back(root);
        this->joker = joker;
        countTerminalVertex = 0;
    }

    void readPattern(std::string& str) {
        patternLen = str.size();
        split(str);
        for (const auto& pattern : patternArr) {
            addPattern(pattern);
        }



    }

    void search(const std::string& str)
    {
        int curr = 0;
        bool terminalVertexFound;
        for (int i = 0; i < str.size(); i++) {
            curr = getGo(curr, str[i]);
            terminalVertexFound = false;
            for (int tmp = curr; tmp != 0; tmp =
getSuffix(tmp)) {
                if (Tops[tmp].isLeaf) {
                    for (int j = 0; j <
Tops[tmp].number.size(); j++) {
                        if (i + 1 -
patternsLength[Tops[tmp].number[j] - 1] - Tops[tmp].deep
>= 0 &&
```

```
                            i + 1 -
patternsLength[Tops[tmp].number[j] - 1] - Tops[tmp].deep
<= str.size() - patternLen) {
                            matchPatterns[i + 1 -
patternsLength[Tops[tmp].number[j] - 1] -
Tops[tmp].deep]++;
                            terminalVertexFound = true;
                            break;
                        }
                    }
                }
            }
        }

    void printResult(const std::string& text) const {
        printMaxArcs();
        std::vector<bool> cutStr(text.size());
        std::string textRest;
        for (int i = 0; i < matchPatterns.size(); i++) {
            if (matchPatterns[i] ==
patternsLength.size()) {
                std::cout << i + 1 << "\n";
                for (int j = 0; j < patternLen; j++)
                    cutStr[i + j] = true;
            }
        }
        for (int i = 0; i < cutStr.size(); i++) {
            if (!cutStr[i])
                textRest.push_back(text[i]);
        }
```

```
    }


private:

    void printMaxArcs() const {
        auto current = Tops.begin();
        int maxArcs = 0;
        while (current != Tops.end()) {
            if (current->next.size() > maxArcs)
                maxArcs = current->next.size();
            current++;
        }
    }

    void split(std::string str) {
        std::string buf = "";
        for (int i = 0; i < str.size(); i++) {
            if (str[i] == joker) {
                if (!buf.empty()) {
                    patternArr.push_back(buf);
                    patternsLength.push_back(i -
buf.size());
                    buf = "";
                }
            }
            else {
                buf.push_back(str[i]);
                if (i == str.size() - 1) {
                    patternArr.push_back(buf);
```

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

    void addPattern(const std::string& str)
    {
        int current = 0;
        for (char i : str) {
            if (Tops[current].next.find(i) == Tops[current].next.end()) {
                Top ver;
                ver.suffix = -1;
                ver.prev = current;
                ver.prevChar = i;
                Tops.push_back(ver);
                Tops[current].next[i] = Tops.size() - 1;
            }
            current = Tops[current].next[i];
        }
        countTerminalVertex++;

Tops[current].number.push_back(countTerminalVertex);
        Tops[current].isLeaf = true;
        Tops[current].deep = str.size();
    }

    int getSuffix(int index)
    {
```

```
        if (Tops[index].suffix == -1) {
            if (index == 0 || Tops[index].prev == 0) {
                Tops[index].suffix = 0;
            }
            else {
                Tops[index].suffix                          =
getGo(getSuffix(Tops[index].prev), Tops[index].prevChar);
            }
        }
        return Tops[index].suffix;
    }


    int getGo(int index, char ch){
        if           (Tops[index].go.find(ch)           ==
Tops[index].go.end()) {
            if        (Tops[index].next.find(ch)        !=
Tops[index].next.end()) {
                Tops[index].go[ch]                          =
Tops[index].next[ch];
            }
            else {
                if (index == 0) {
                    Tops[index].go[ch] = 0;
                }
                else {
                    Tops[index].go[ch]                      =
getGo(getSuffix(index), ch);
                }
            }
        }
        return Tops[index].go[ch];
```

```
    }

private:
    std::vector<Top> Tops;
    char joker;
    int countTerminalVertex;
    std::vector<std::string> patternArr;
    int patternLen{};
    std::vector<int> matchPatterns;
    std::vector<int> patternsLength;
};



int main() {
    std::string str;
    std::string pattern;
    char joker;
    std::cin >> str;
    std::cin >> pattern;
    std::cin >> joker;

    auto* ahoCorasick = new AhoCorasick(joker);
    ahoCorasick->readPattern(pattern);
    ahoCorasick->search(str);
    ahoCorasick->printResult(str);

    return 0;
}
```