МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 9382 _____ Иерусалимов Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

Познакомиться с одним из часто используемых на практике алгоритма поиска подстрок в строке. Получить навыки решения задач на этот алгоритм.

Индивидуализация.

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

Первое задание.

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.

Sample Input:

```
NTAG
3
TAGT
TAG
T
```

Sample Output:

```
2 2
2 3
```

Второе задание.

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 шаблоныобразцу P необходимо найти все вхождения P в текст T.

Например, образец ab??c?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.

Sample Input:

```
ACTANCA
A$$A$
$
```

Sample Output:

```
1
```

Описание алгоритма.

Постройка бора.

Из строк-паттернов строим сначала бор. Рассматривается в каждом паттерне каждый символ. Если у рассматриваемой вершины бора есть дети которые соответствую рассматриваемому символу, то мы переходим к этому ребенку, если такого ребенка нет, то создаём нового и идем к нему. Если паттерн закончился, то помечаем текущую вершину терминированной и возвращаемся к корню дерева.

Суффиксные ссылки.

Далее надо заполнить бор суффиксными ссылками. Для вершины X суффиксная ссылка будет указывать на вершину Y, такую, что наибольшийсуффикс текущего пути (от корня до X) будет префиксом (от корня до Y) в боре. Вершины дерева

перебираются в ширину. Для каждой вершины смотрятся дети. Для каждого ребенка берется суффиксная ссылка его родителя. Потом проходимся по суффиксной ссылке выше по дереву пока не встретится корень или вершина среди детей которой есть рассматриваемый элемент. Если найден корень, то суффиксная ссылка для этого ребенка будет указывать на корень, а если же был найден такой же ребенок, то будет указывать на него. Также, если вершина, на которую указывает новая суффиксная ссылка была терминирована, то эта терминированность также передается рассматриваемому ребенку.

Пропускаем текст через дерево.

Теперь пропускаем текст через это дерево. Смотрим каждый символ строки. Если среди детей текущейвершины был найден этот символ, то делается переход в эту вершину. Достается из вершины информация о терминированности. Если же нужной вершины среди детей не было найдено, то делается переход по терминированной ссылке.

Вторая задача.

Тут паттерн разбиваем на подстроки где нет джокеров. Потом из них строится бор и запоминаются начальные позиции этих подстрок в паттерне. Далее с помощью алгоритма Ахо-Карасика находим все вхождения этих подстрок в тексте. Для каждого такого вхождения, в векторе res длины равной длине главной строки увеличиваются на единицу значение с индексом j - l + 1, где l – начальная позиция подстроки в паттерне.Паттерн содержитя в главной строке если res(n) = k, где n – индекс вхождения, а k – количество подстрок в паттерне.

Сложность алгоритма.

Бор строится за время $O(n)$, где n – общая длина паттернов.

Построение суффиксных ссылок происходит через обход в ширину, так как это дерево ребер там столько же сколько и вершин-1 тогда O(n). Поиск в строке происходит за O(k), k — длина строки. Тогда время работы будет O(n + k).

Суффиксное дерево будет занимать O(n), где n — длина всех паттернов. Информация о найденных положениях подстрок в строке O(m * k), где m – длина строки, k - это количество паттернов. По итогу O(n + m * k)

Описание функций и структур данных.

std::vector<std::unordered_map<char, int>> tree – бор, вектор карт с ключом и значением – символ и индекс вершины.

std::vector<std::vector<int>> term – вектор, который показывает терминированность вершины с индексом i.

std::vector<int> fail – вектор суффиксных ссылок

Karas(std::vector<std::string> &pattern) – конструктор класса алгоритма.
std::vector<std::string> &pattern – паттерны

void initTreeForPatterns(std::vector<std::string> &patterns) — инициализация бора.
std::vector<std::string> &pattern – паттерны

std::vector<int> letterChecker(char letter) – Находит в боре letter и дает информацию о терменированости вершины letter
char letter – искомая вершина

Выходные значения:
std::vector<int> - номера паттернов, которые заканчиваются на символ letter.

Тестирование.

Задание 1:

№	Входные данные	Выходные данные
1	abcdefghegheawdwefsdfwefwefkillwefwefwefme 2	16 1 29 2

	kill me	
2	Russiaisagreatpower,andisconsideredapotentialsuperpower. 4 Russia great superpower power	1 1 10 2 15 4 46 3 51 4
3	BOOBAABOOBA 3 BOO BAA BA	1 1 4 2 4 3 7 1 10 3
4	YESPAPANOPAPA 2 PAPANO PAPAYES	4 1

Задание 2:

№	Входные данные	Выходные данные
1	DUNGENMASTER D****N* *	1
2	BUDMBS B% %	1 5
3	AAAAAA A*A *	1 2 3 4
4	2+2!=5ChangeMyMind 2+2!=* *	1

Выводы.

Был исследован часто используемый на практике алгоритм - поиск подстрок в строке. Также были получены навыки решения задач на этот алгоритм.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

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


bool interData = 0;

class Karasi {
public:

    void initTreeForPatterns(std::vector<std::string> &patterns) {
        //Строим дерево
        info += "Building a prefix tree...\n";
        Display();

        int counterElemInTree = 0;
        std::unordered_map<char, int> root;
        this->tree.push_back(root);
        this->fail.push_back(0);
        this->term.emplace_back(std::vector<int>());

        int wordCounter = 0;

        info += "We take a word from the list of templates: \n[";
        Display();
        for (std::string wordDispaly : patterns) {
            info += wordDispaly + ", ";
        }
        info += "]\n";
        Display();
        //берем слово из списка паттернов
        for (std::string word : patterns) {

            info += "********************* " + word + "
**********************\n";
            Display();
```

```
            int current = 0;
            info += "\t    Take a letter from the word\n";
            Display();

            //Просматриваем буквы в слове для постройки дерева
            for (char letter : word) {
                info += "\t\t\t" + std::string(1, letter) + "\n";
                info += "Current index in tree - " + std::to_string(current) +
"\n";
                Display();


                //Если буква уже является ребенком корня(cur)
                if (this->tree[current].find(letter) != this-
>tree[current].end()) {
                    info += std::string(1, letter) + " - is child for current
root, go to the " +
                            std::string(1, letter) + "\n";
                    current = this->tree[current][letter];

                } else { // Буква не является ребенком корня, добавляем ее
                    //Создаем новую вершину.
                    std::unordered_map<char, int> newNode;
                    this->tree.push_back(newNode);
                    this->term.emplace_back(std::vector<int>());
                    //Увеличиваем значение о количестве вершин в дереве
                    ++counterElemInTree;

                    //Помечаем вершину новым номером
                    this->tree[current][letter] = counterElemInTree;
                    current = counterElemInTree;

                    info += std::string(1, letter) + " is not child for current
root, make a new node with index " +
                            std::to_string(current) + "\n\n";
                    Display();
                }
            }
            info += "The pattern is over, let's mark the last vertex as the end
of the word. Index - " +
                    std::to_string(current) + "\n";
```

```
            Display();
            this->term[current].push_back(wordCounter++);
            info += "************************************************\n";
            Display();
        }
    }

    std::vector<int> letterChecker(char letter) {
        while (this->curPos > 0 && this->tree[this->curPos].find(letter) == this-
>tree[this->curPos].end()) {
            this->curPos = this->fail[this->curPos];
        }
        if (this->tree[this->curPos].find(letter) != this->tree[this-
>curPos].end()) {
            this->curPos = this->tree[this->curPos][letter];
        }
        return this->term[this->curPos];
    }

    Karasi(std::vector<std::string> &patterns) {
        //Инициализируем дерево слов/патернов
        initTreeForPatterns(patterns);


        info += "Filling in the links of failures.\n";
        Display();
        //Далее инициализируем нулями точки с ошибками, т.е теперь ошибки
указывают на корень
        for (int i = 0; i < this->tree.size(); ++i) {
            this->fail.push_back(0);
        }

        info += "********************** BFS **********************\n";
        //Заводим очередь где будет хранится все вершины
        std::queue<std::pair<int, std::unordered_map<char, int>>> nodes;

        //Рассматриваем всех детей корня дерева
        for (auto child : this->tree[0]) {
            nodes.push(std::make_pair(child.second, this->tree[child.second]));
        }

        //Проходимся по всем вершинам в очереди пока не дойдем до конца
```

```
        while (!nodes.empty()) {
            info += "Breadth First Search Queue Status [";
            Display();
            auto que = nodes;
            while (!que.empty()) {
                info += std::to_string(que.front().first) + ", ";
                que.pop();
            }
            info += "]\n";
            Display();

            //Берем вершину из очереди
            auto vertex = nodes.front();
            nodes.pop();
            auto i = vertex.first;
            auto node = vertex.second;

            info += "Considering the top \'" + std::to_string(i) + "\' and
childs\n";
            Display();

            //Проходимя по всем детям текущей вершины, записывая их в очередь для
поиска в ширину
            for (auto child : node) {
                //запись в очередь
                nodes.push(std::make_pair(child.second, this-
>tree[child.second]));

                char childName = child.first;
                int childPos = child.second;

                info += "Child is " + std::string(1, childName) + ", with index "
+
                        std::to_string(childPos) + "\n";
                Display();
                int failPointer = this->fail[i];

                info += "Failure link for index " + std::to_string(i) + " is " +
std::to_string(failPointer) + "\n";
                Display();

                while (failPointer != 0 && this-
```

```
>tree[failPointer].find(childName) == tree[failPointer].end()) {

                    info += "Among children current failing pointer " +
std::string(1,failPointer) + " not contained " + std::string(1,childName) + "
going next\n";
                    Display();
                    failPointer = this->fail[failPointer];
                }

                if (this->tree[failPointer].find(childName) !=
tree[failPointer].end()) {
                    failPointer = this->tree[failPointer][childName];
                }
                this->fail[childPos] = failPointer;

                info += "Now fail pointer for " + std::string(1, childName) + " -
it's vertex with index " + std::to_string(failPointer) + "\n\n";

                if (!this->term[this->fail[childPos]].empty()) {



                    for (auto item : this->term[this->fail[childPos]]) {
                        this->term[childPos].push_back(item);
                    }
                }
            }
        }
        this->curPos = 0;
        printAk(patterns);

    }


    void printAk(std::vector<std::string> &patterns){
        info += "\n\nResulting automaton\n";
        Display();
        for (int i = 0; i < tree.size(); i++) {
            info += "_________________ " + std::to_string(i) + "
___________________\n";
            info += "Vertex with index " + std::to_string(i);
            if (term[i].empty()) {
```

```
                info += " not terminated\n";
            } else {
                info += " is terminated\n Patterns that end at this top: ";
                for (auto item : term[i]) {
                    info += patterns[item] + " ";
                }
                info +=  "\n";
            }
            Display();
            info += "Suffix link points to index: " + std::to_string(fail[i]) +
"\n";
            if (tree[i].empty()) {
                info += "No children\n";
            } else {
                info += "List of children: ";
                for (auto item : tree[i]) {
                    info += "[" + std::string(1,item.first) + ", " +
std::to_string(item.second) + "] ";
                }
            }
            info += "\n";
            Display();
        }
    }

    void Task1(std::string txt) {
        int count = 0;
        std::cin >> count;

        std::vector<std::string> patterns;
        for (int i = 0; i < count; ++i) {
            std::string word;
            std::cin >> word;
            patterns.push_back(word);
        }
        Karasi *object = new Karasi(patterns);
        std::vector<std::pair<int, int>> result;
        for (int i = 0; i < txt.length(); ++i) {
            std::vector<int> pats = object->letterChecker(txt[i]);
            if (!pats.empty()) {
                for (auto point : pats) {
                    int patternLength = patterns[point].length();
```

```
                    result.emplace_back(std::make_pair(i - patternLength + 2,
point + 1));
                }
            }
        }

        std::sort(result.begin(), result.end());
        for (auto item : result) {
            std::cout << item.first << " " << item.second << std::endl;
        }

    }

    void Task2(std::string txt) {
        char joker;
        std::string word;

        std::cin >> word;
        std::cin >> joker;

        std::vector<int> res(txt.length(), 0);
        std::vector<std::string> patterns;
        std::vector<int> index;
        int j = 0;
        while (j < word.length()) {
            std::string small;
            int i = j;
            while (j < word.length() && word[j] != joker) {
                small += word[j];
                j++;
            }
            if (!small.empty()) {
                patterns.push_back(small);
                index.push_back(i);
            }
            j++;
        }
        Karasi *obj = new Karasi(patterns);
        for (int pos = 0; pos < txt.length(); pos++) {
            std::vector<int> pats = obj->letterChecker(txt[pos]);
            if (!pats.empty()) {
                for (auto point : pats) {
```

```
                    int firstLetter = pos - patterns[point].length() + 1;
                    int idx = firstLetter - index[point] + 1;
                    if (idx >= 0 && idx < res.size()) {
                        res[firstLetter - index[point] + 1]++;
                    }
                }
            }
        }

        for (int indexOfAnswer = 0; indexOfAnswer < res.size() - word.size() + 2;
indexOfAnswer++) {
            if (res[indexOfAnswer] == patterns.size()) {
                std::cout << indexOfAnswer << std::endl;
            }
        }
    }

    void Display() {
        if (interData) {
            std::cout << info;
        }
        info.clear();
    }


private:
    std::string info;
    std::vector<std::unordered_map<char, int>> tree;
    std::vector<std::vector<int>> term;
    std::vector<int> fail;
    int curPos;

};

int main() {
    int choise = 1;
    std::string txt;

    std::cout << "Include intermediate data?\n0 - No; 1 - Yes\n";
    std::cin >> interData;

    std::cout << "Choise task.\n Task1 - 1; Task2 - 2\n";
```

```
    std::cin >> choise;

    std::cout << "Enter Data: \n";
    std::cin >> txt;

    Karasi *Do;

    if (choise == 1) {
        Do->Task1(txt);
    } else {
        Do->Task2(txt);
    }
    return 0;
}
```