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 _____ Субботин М. О.

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 шаблоны образцу 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.

Сначала надо построить бор из заданных строк-паттернов. Рассматривается в каждом паттерне каждый символ. Если среди детей текущей вершины бора есть рассматриваемый символ, то идет переход к этому ребенку, если такого ребенка нет, то создается нового и идем к нему. Если паттерн закончился, то помечаем текущую вершину терминированной и возвращаемся к корню дерева.

Затем следует заполнить бор суффиксными ссылками. Для вершины v суффиксная ссылка будет указывать на вершину t, такую, что наибольший суффикс текущего пути (от корня до v) будет префиксом (от корня до t) в боре.

Вершины дерева перебираются в ширину. Для каждой вершины смотрятся дети. Для каждого ребенка берется суффиксная ссылка его родителя. И производится проход по суффиксной ссылке выше по дереву до тех пор, пока не встретится корень или вершина среди детей которой есть рассматриваемый ребенок. 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. С помощью такого алгоритма происходит заполнение бора суффиксными ссылками.

Теперь через такое дерево можно пропускать строку, в которой нужно находить заданные паттерны. Читается каждый символ строки. Если среди детей текущей вершины был найден этот символ, то делается переход в соответствующую вершину. И достается из вершины информация о терминированности. Если же нужной вершины среди детей не было найдено, то делается переход по терминированной ссылке.

Во второй задаче паттерн с джокерами разбивается на подстроки без джокеров и с помощью этих подстрок строится бор и запоминаются начальные позиции этих подстрок в паттерне. Затем алгоритмом Ахо-Корасик находятся все вхождения этих подстрок в главной строке. Для каждого такого вхождения в j-ой позиции строки, в векторе C длины равной длине главной строки инкрементируется значение с индексом $j - l_i + 1$, где l_i – начальная позиция подстроки в паттерне. Паттерн входит в главную строку в том случае, если $C[p] = k$, где p – индекс вхождения, а k – количество подстрок в паттерне.

Сложность алгоритма.

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

Для построения суффиксных ссылок используется обход в ширину, его сложность $O(V + E)$, но из-за того, что количество ребер не совсем произвольное (оно зависит от количества вершин), то можно написать $O(n)$. Поиск в строке происходит за время $O(m)$, где m – длина строки. В итоге общее время работы $O(n + m)$.

Для хранения суффиксного дерева требуется $O(n)$ памяти, где n – количество вершин в дереве (общая длина паттернов). Также хранится информация о найденных положениях подстрок в строке и это требует $O(m * k)$ памяти, где $m -$ длина строки, k - это количество паттернов.

Описание функций и структур данных.

```
class Korasik {
public:
    std::vector<std::unordered_map<char, int>> trie;
    std::vector<std::vector<int>> term;
    std::vector<int> fail;
    int curPos;

    Korasik(std::vector<std::string> &words);
    std::vector<int> check(char letter);
};
```

std::vector<std::unordered_map<char, int>> trie – вектор карт с ключом и значением – символ и индекс вершины. Эта структура представляет бор.
std::vector<std::vector<int>> term – вектор, который показывает терминированность вершины с индексом i. На одну вершину могут заканчиваться несколько паттернов, поэтому структура имеет формат вектор векторов.
std::vector<int> fail – вектор суффиксных ссылок
Korasik(std::vector<std::string> &words) – конструктор класса алгоритма.
Параметры:
std::vector<std::string> &words – паттерны

std::vector<int> check(char letter) – функция, которая находит в боре следующий символ letter и выводит информацию о терминированности вершины letter.
Параметры:
char letter – искомая вершина

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

Тестирование.

Задание 1:

№	Входные данные	Выходные данные	Результат
1	abcdefgheghe 4 cde de gh fe	1 3 1 4 2 7 3 10 3	Правильно
2	ccca 1 cc	1 1 2 1	Правильно
3	abababababab 1 aba	1 1 3 1 5 1 7 1 9 1 11 1	Правильно
4	NTAG 3 TAGT TAG T	2 2 2 2 3	Правильно

Задание 2:

№	Входные данные	Выходные данные	Результат
1	ACTANCA A$$A$ $	1	Правильно
2	ACTANCAB A%%A% %	1 4	Правильно
3	AAAAAA A*A *	1 2 3 4	Правильно
4	shot sh*t *	1	Правильно

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

//#define TASK1
#define TASK2
//#define DEBUG

class Korasik {
public:
    std::vector<std::unordered_map<char, int>> trie;
    std::vector<std::vector<int>> term;
    std::vector<int> fail;
    int curPos;

    Korasik(std::vector<std::string> &words);
    std::vector<int> check(char letter);
};

Korasik::Korasik(std::vector<std::string> &words) {
#ifdef DEBUG
    std::cout << "Строим префиксное дерево для набора паттернов: " << std::endl;
    for (std::string word : words) {
        std::cout << word << " ";
    }
    std::cout << std::endl;
#endif
    int total = 0;
    std::unordered_map<char, int> root;
    this->trie.push_back(root);
    this->fail.push_back(0);
    this->term.emplace_back(std::vector<int>());

    int wordCounter = 0;
    for (std::string word : words) {
```

```
#ifdef DEBUG
        std::cout << "Рассматриваем паттерн: " << word << std::endl;
#endif
        int cur = 0;
        for (char ch : word) {
#ifdef DEBUG
            std::cout << "Текущий индекс в дереве: " << cur << std::endl;
            std::cout << "Рассматриваем символ: " << ch << std::endl;
#endif
            // если ch - уже ребенок для cur, то просто переходим в ch
            if (this->trie[cur].find(ch) != this->trie[cur].end()) {
                cur = this->trie[cur][ch];
#ifdef DEBUG
                std::cout << ch << " - ребенок для текущего элемента, переходим
в " << ch << std::endl;
#endif
            }
            // если ch не ребенок для cur
            else {
                //создаем новую вершину
                std::unordered_map<char, int> newNode;
                this->trie.push_back(newNode);
                this->term.emplace_back(std::vector<int>());
                //инкрементируем кол-во вершин в trie
                total++;
                //помечаем вершину соответствующему ей номеру
                this->trie[cur][ch] = total;
                cur = total;
#ifdef DEBUG
                std::cout << ch << " не ребенок для текущего элемента, добавили
новую вершину с индексом " << cur << std::endl;
#endif
            }
        }
#ifdef DEBUG
        std::cout << "Паттерн закончился, помечаем последнюю символ-вершину
паттерна с индексом " << cur << std::endl;
#endif
        //помечаем вершину, которая является концом слова
        this->term[cur].push_back(wordCounter++);
    }
```

```
#ifdef DEBUG
    std::cout << "Заполняем связи неудач. " << std::endl;
    std::cout << "Проходимся по дереву в ширину: " << std::endl;
#endif

    // заполняем failing pointers нулями (указывают пока на корневую вершину)
    for (int i = 0; i < this->trie.size(); i++) {
        this->fail.push_back(0);
    }

    std::queue<std::pair<int, std::unordered_map<char, int>>> nodes;
    //рассматриваем всех детей корня
    for (auto child : this->trie[0]) {
        // first - номер вершинки, second - unordered_map соотв. этой вершине, в
которой хранятся ее дети
        auto o = std::make_pair(child.second, this->trie[child.second]);
        nodes.push(o);
    }

    while (!nodes.empty()) {
#ifdef DEBUG
        std::cout << "Состояние очереди поиска в ширину: " << std::endl;
        auto printQueue = nodes;
        while (!printQueue.empty()) {
            std::cout << printQueue.front().first << " ";
            printQueue.pop();
        }
        std::cout << std::endl;
#endif
        //рассматриваем следующую вершину из очереди
        auto vertex = nodes.front();
        nodes.pop();
        auto i = vertex.first;
        auto node = vertex.second;
#ifdef DEBUG
        std::cout << "Рассматриваем вершину " << i << " и ее детей: " << std::endl;
#endif
        //каждого ребенка текущей вершины записываем в очередь для BFS.
        for (auto child : node) {
```

```
            auto      bfsNode      =      std::make_pair(child.second,      this-
>trie[child.second]);
            nodes.push(bfsNode);
        }
        //рассматриваем каждого ребенка для текущей вершины
        for (auto child : node) {

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

#ifdef DEBUG
            std::cout << "Ребенок " << childName << " с индексом " << childPos <<
std::endl;
#endif

            // failing pointer – индекс вершины, в которую нужно перейти в случае,
если дальше по trie идти некуда
            int fPointer = this->fail[i];
#ifdef DEBUG
            std::cout << "Связь неудачи для индекса " << i << " – это " << fPointer
<< std::endl;
#endif

            // пока failing pointer не указывает на корень и пока среди детей
failing pointer
            // нет вершины схожей с текущим ребенком
            // просто передвигаемся по поинтерам вверх
            while  (fPointer  !=  0  &&  this->trie[fPointer].find(childName)  ==
trie[fPointer].end()) {
#ifdef DEBUG
                std::cout << "Среди детей текущего failing pointer'а " << fPointer
<< " нет " << childName << " идем выше" << std::endl;
#endif
                fPointer = this->fail[fPointer];
            }

            //если loop выше закончился из-за встречи схожего ребенка
            // то failing pointer для текущего ребенка будет индексом
            //схожего ребенка для 'сдвинутого' failing pointer'а
            if (this->trie[fPointer].find(childName) != trie[fPointer].end()) {
                fPointer = this->trie[fPointer][childName];
```

```
            }
            this->fail[childPos] = fPointer;
#ifdef DEBUG
            std::cout << "Теперь fpointer для " << childName << " – это вершина с индексом " << fPointer << std::endl;
#endif

            //если же еще этот failing pointer указывает на терминированную вершину
            //то пометим текущего ребенка тоже терминированным
            if (!this->term[this->fail[childPos]].empty()) {
#ifdef DEBUG
                std::cout << "Ребенок наследует от failing pointer терминированные метки: ";
#endif
                for (auto item : this->term[this->fail[childPos]]) {
                    std::cout << item << " ";
                    this->term[childPos].push_back(item);
                }
                std::cout << std::endl;
            }
        }
    }
    this->curPos = 0;

#ifdef DEBUG
    std::cout << "\n Построенный автомат: \n";
    for (int i = 0; i < trie.size(); i++) {
        std::cout << "Вершина с индексом " << i << std::endl;
        if (term[i].empty()) {
            std::cout << "Не терминированная" << std::endl;
        } else {
            std::cout << "Вершина терминированна, номера паттернов, которые заканчиваются на этой вершине: " << std::endl;
            for (auto item : term[i]) {
                std::cout << item << " ";
            }
            std::cout << std::endl;
        }
        std::cout << "Суффиксная ссылка указывает на вершину с индексом: " << fail[i] << std::endl;
```

```
        if (trie[i].empty()) {
            std::cout << "Детей нет";
        } else {
            std::cout << "Дети: ";
            for (auto item : trie[i]) {
                std::cout << "{" << item.first << ", " << item.second << "} ";
            }
        }
        std::cout << std::endl
                  << "-------------------------------" << std::endl;
    }
#endif
}

std::vector<int> Korasik::check(char letter) {
#ifdef DEBUG
    std::cout << "Ищем вхождение символа " << letter << std::endl;
#endif
    while (this->curPos > 0 && this->trie[this->curPos].find(letter) == this->trie[this->curPos].end()) {
#ifdef DEBUG
        std::cout << "Для индекса " << curPos << " среди детей не нашлось " << letter << std::endl;
#endif
        this->curPos = this->fail[this->curPos];
#ifdef DEBUG
        std::cout << "Переходим дальше по fpointer'у в " << curPos << std::endl;
#endif
    }
    if (this->trie[this->curPos].find(letter) != this->trie[this->curPos].end()) {

        this->curPos = this->trie[this->curPos][letter];
#ifdef DEBUG
        std::cout << "Один из детей совпал с " << letter << ", текущий индекс " << curPos << std::endl;
#endif
    }
#ifdef DEBUG
    std::cout << "Возвращаем паттерны, которые заканчиваются на индексе " << curPos << " : " << std::endl;
```

```
    for (auto item : this->term[this->curPos]) {
        std::cout << item << " ";
    }
    std::cout << std::endl;
#endif
    return this->term[this->curPos];
}

int main() {
    std::string seq;
    std::cin >> seq;

#ifdef TASK1
    int n = 0;
    std::cin >> n;

    std::vector<std::string> words;
    for (int i = 0; i < n; i++) {
        std::string word;
        std::cin >> word;
        words.push_back(word);
    }
    auto obj = new Korasik(words);
    std::vector<std::pair<int, int>> res;
    for (int pos = 0; pos < seq.length(); pos++) {
        std::vector<int> pats = obj->check(seq[pos]);
        if (!pats.empty()) {
            for (auto point : pats) {
                int patternLength = words[point].length();
                res.emplace_back(std::make_pair(pos - patternLength + 2, point +
1));
            }
        }
    }

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

```
#ifdef TASK2
    std::string word;
    std::cin >> word;
    char joker;
    std::cin >> joker;

    std::vector<int> C(seq.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++;
    }
    auto obj = new Korasik(patterns);
    for (int pos = 0; pos < seq.length(); pos++) {
        std::vector<int> pats = obj->check(seq[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 < C.size()) {
                    C[firstLetter - index[point] + 1]++;
                }
            }
        }
    }

    for (int i = 0; i < C.size() - word.size() + 2; i++) {
        if (C[i] == patterns.size()) {
            std::cout << i << std::endl;
```

```
        }
    }
#endif
    return 0;
}
```