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

Студентка гр. 9383 _____ Лихашва А.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изученить и реализовать алгоритм Ахо-Корасик для поиска вхождений нескольких шаблонов или вхождения шаблона с маской в текст.

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

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
```

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 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
```

Вариант 3.

Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

Описание алгоритмов:

1. Алгоритм Ахо-Корасик (поиск вхождений безмасочных шаблонов).

- Рассматривается каждый шаблон в данном наборе. По шаблонам строится структура данных бора. Создаётся начальная вершина – корень, из которой выходят рёбра, они отмечены символами начал шаблонов.
    1) Если для текущего рассматриваемого символа шаблона не существует вершины, в которую можно перейти по ребру, отмеченному данным символом, то создаётся вершина и ребро.
    2) Выполняется переход по ребру, отмеченному данным символом, в следующую вершину.
    3) Если в шаблоне ещё есть символы, тогда возвращаемся к первому подпункту 1), иначе помечаем вершину терминальной (или конечной) и переходим к следующему шаблону. Продолжается построение, начиная с корня бора.
- На основе бора строится автомат, содержащий для каждой вершины суффиксные и конечные ссылки. Суффиксные ссылки ведут в максимальный префикс, являющийся максимальным суффиксом для строки, построенной на основе пути до данной вершины (могут также вести в ветви бора, построенные для других шаблонов). Конечные ссылки ведут в концы других шаблонов.
    1) Если текущая вершина – корень, то суффиксная ссылка – корень, иначе суффиксная ссылка – это вершина, в которую ведёт ребро с данным символом из суффиксной ссылки родительской вершины. При этом, если ссылка ищется для верши-

ны, следующей за корнем, то для неё ссылка будет корнем. Пока такого ребра нет, возвращаемся снова к первому подпункту а).

2) Пока текущая суффиксная ссылка – не корень: если текущая суффиксная ссылка – терминальная вершина, значит конечная ссылка найдена, иначе перейти к суффиксной ссылке текущей ссылки, переход к пункту 2).

- Посимвольно рассматривается строка, в которой ищутся вхождения. Начальная вершина является корнем. Если из текущей вершины выходит ребро, помеченное текущим рассматриваемым символом, то переходим по нему в следующую вершину, иначе переходимпо суффиксным ссылкам, пока такое ребро не будет найдено или не будет встречен корень, при этом из него также не будет выходить такое ребро (в последнем случае осуществляется переход в корень). Если вершина, в которую осуществлён переход, терминальная, добавляется информация о вхождении в строку соответствующего ей шаблона в список. Если для этой вершины конечная ссылка не пустая, переходим по конечным ссылкам, пока они не пустые, и для каждой также добавляем информацию о вхождении.

2. Алгоритм Ахо-Корасик (поиск вхождений шаблона с маской).

- Для данного алгоритма строится бор для безмасочных подшаблонов, находящихся в нём. Выделяется массив индексов, длина которого равна длине рассматриваемой строки. На основе бора строится автомат, и дальше выполняется посимвольное рассмотрение строки.
- Если в строке нашёлся какой-либо подшаблон, то ячейка массива по адресу, образованному разностью номера начального символа данного вхождения подшаблона в строке и его смещения относительно начала исходного шаблона (если этот адрес не меньше 0),

увеличивается на единицу. Если у подшаблона несколько смещений, то данная операция выолняется для каждого из них.

- В завершении алгоритма индексы тех ячеек массива, значение которых будет равно количеству подшаблонов в исходном шаблоне, и будут индексами вхождения заданного шаблона в строку.

3. ***Поиск длин самых длинных цепочек из суффиксных и конечных ссылок.***

- Рекурсивно рассматриваются вершины бора.
- Для каждой из них выполняется переход по суффиксным ссылкам, пока не встречен корень, подсчитывается длина цепочки из суффиксных ссылок.
- Аналогично подсчитывается длина цепочки конечных ссылок, но переход и увеличение длины осуществляется только при наличии конечной ссылки.

Сложность алгоритма :

1. Алгоритм Ахо-Корасик (поиск вхождений безмасочных шаблонов).

Сложность по памяти: автомат хранится как красно-чёрное дерево, поэтому его сложность по памяти – O(n), где n – суммарная длина шаблонов.

Сложность алгоритма по времени в данном случае – O((T + n)log(s) + k), где T – длина строки, в которой ищутся вхождения, s – размер алфавита, k – общее количество вхождений шаблонов в текст, так как время на построение бора (и автомата) сокращается по сравнению с другими реализациями (лишние символы не добавляются), что влияет и на время обработки строки.

2. ***Алгоритм Ахо-Корасик (поиск вхождений шаблона с маской).***

Сложность по памяти: так как ищется один шаблон, то сложность по памяти составит O(n + T), где n – суммарная длина всех подшаблонов в ша-

блоне с маской, а T – длина строки (на сложность влияет добавление массива индексов).

Сложность по времени: время тратится на заполнение массива индексов, сложность по памяти составляет O((T + n)log(s) + k*p), где s – размер алфавита, где k – общее количество вхождений шаблонов в текст, p – суммарное количество сдвигов подшаблонов относительно исходного шаблона.

Функции и структуры данных:

Структуры данных:

class Bor - класс для реализации алгоритмов.

Bor parent* - родительская вершина

Bor SufLink* - суффиксная ссылка

Bor EndLink* - конечная ссылка

char ToParent - ребро, по которому попали из родительской вершины

bool IsTerminal - терминальая вершина или нет (конец шаблона)

std::map<char, Bor> next* - ребра по которым можно переходить

std::vector<int> NumPattern - номера шаблонов, в которые входит символ, по которому пришли

std::vector<int> shifts - сдвиги подстрок в шаблоне

Функции:

Bor(Bor parent = nullptr, char ToPrnt = 0)* — конструктор

Bor GetLink(const char& symbol)* - получение следующей вершины для перехода

Bor CreateBor(const std::vector<std::pair<std::string, int>>& patterns)* — функция построения бора

void Automaton(Bor bor)* — построение автомата или вычисление ссылок в боре

std::pair<int, int> LongestChain(Bor bor, Bor* root, int& depth)* - вычисление длин наибольших цепочек из суффиксных и конечных ссылок

void AhoCorasick(const std::string& t, const std::vector<std::pair<std::string, int>>& patterns, std::vector<std::pair<int, int>>& result) - алгоритм поиска вхождений в строку

void Partitioning(const std::string& p, const char& joker, std::vector<std::pair<std::string, int>>& patterns) - разбиение шаблона с маской на безмасочные подшаблоны

Тестирование.

Задание 1

Входные данные	Выходные данные
NTAG 3 TAGT TAG T	2 2 2 3
qwerty 3 wer ty t	2 1 5 2 5 3
sleeepp 3 lee pp s	1 3 2 1 6 2
shadoww 2 shaw dow	4 2

Задание 2

Входные данные	Выходные данные
ACTANCA A$$A$ $	1
qwerty qw@r @	1
fafafafafa f$f $	1 3 5 7
aaaaa a$a $	1 2 3

Выводы.

В результате выполнения работы был изучен и реализован алгоритм Ахо-Корасик для поиска вхождений нескольких шаблонов или вхождения шаблона с маской в текст.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл lab5_1.h

```
#pragma once
#include <string>
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>


class Bor {
public:
Bor* parent;   // родительская вершина
Bor* SufLink; // суффиксная ссылка
Bor* EndLink;  // конечная ссылка
char ToParent; // ребро, по которому попали из родительской вершины
bool IsTerminal;  // терминальая вершина или нет (конец шаблона)
std::map<char, Bor*> next;  //ребра по которым можно переходить
std::vector<int> NumPattern;  // номера шаблонов, в которые входит символ, по которому пришли
int TermNumPattern;

Bor(Bor* parent = nullptr, char ToPrnt = 0) :
parent(parent), ToParent(ToPrnt), SufLink(nullptr),
EndLink(nullptr), IsTerminal(false) {
        if (parent == nullptr || ToPrnt == 0) { // создание корня
                this->parent = this;
                this->SufLink = this;
```

```
        }
    }

    ~Bor() {
        for (auto i : this->next) {
            delete i.second;
        }
    }

    Bor* GetLink(const char& symbol) {
        if (this->next.find(symbol) != this->next.end()) { //
если есть путь по заданному символу из текущей вершины
            return this->next.at(symbol);
        }
        if (this->SufLink == this) { // если в корне и путь не
найден
            return this;
        }
        return this->SufLink->GetLink(symbol); // не в корне и
путь не найден
    }
    };


    Bor* CreateBor(const std::vector<std::pair<std::string,
int>>& patterns);
    void Automaton(Bor* bor);
    std::pair<int, int> LongestChain(Bor* bor, Bor* root, int&
depth);
    void AhoCorasick(const std::string& t, const
std::vector<std::pair<std::string, int>>& patterns,
std::vector<std::pair<int, int>>& result);
```

Файл lab5_1.cpp

```
#include "lab5_1.h"
```

```
     Bor* CreateBor(const std::vector<std::pair<std::string,
int>>& patterns) { //создание бора
     std::cout << "\n_Начало построения бора_";
     Bor* bor = new Bor();    // корень бора
     for (auto& pt : patterns) {
          int NumPattern = find(patterns.begin(),
patterns.end(), pt) - patterns.begin();
          std::cout << "\nРассматривается " << NumPattern + 1 <<
" шаблон: " << pt.first << '\n';
          Bor* current = bor; // поиск пути, начиная от корня
          for (auto& c : pt.first) {
               if (current->next.find(c) == current->next.end())
{ //если такого ребра нет, то добавляется
                    current->next.insert({ c, new Bor(current,
c) });
                    std::cout << "В бор добавлена вершина, куда
ведёт символ (" << c << ")\n";
               }
               else {
                    std::cout << "Ребро (" << c << ") для теку-
щего шаблона уже существует. Выполнение перехода.\n";
               }
               current = current->next[c]; // переход по данному
ребру
               current->NumPattern.push_back(NumPattern);
          }
          current->TermNumPattern = NumPattern; // для конечной
вершины
          current->IsTerminal = true;
          std::cout << "Вершина является терминальной, закончено
построение ветви бора.\n";
     }
     return bor;
     }
```

```
    void Automaton(Bor* bor) {  //вычисление суф. и конечных ссылок (построение автомата)
    std::cout << "\n\n_Вычисление суффиксных и конечных ссылок_\n";
    std::queue<Bor*> front({ bor });  // вершины одного уровня в боре
    while (!front.empty()) {
        Bor* current = front.front();
        front.pop();
        Bor* CurrentLink = current->parent->SufLink; // родительская ссылка стала текущей
        const char& key = current->ToParent;    // сохраняется символ, для которого ищется ссылка
        bool IsFound = true;
        while (CurrentLink->next.find(key) == CurrentLink->next.end()) {
            if (CurrentLink == bor) {
                std::cout << "Суффиксные ссылки не найдены, ссылка установлена на корень.\n";
                current->SufLink = bor;  // если из корня нет пути, то ссылка устанавливается в корень
                IsFound = false;  //ссылка, не равная корню, не найдена
                break;
            }
            CurrentLink = CurrentLink->SufLink;
        }
        if (IsFound) {
            CurrentLink = CurrentLink->next.at(key);
            if (current->parent == bor) {
                std::cout << "Текущая вершина - начало слова, поэтому ссылка установлена на корень.\n";
                current->SufLink = bor;  // для вершин первого уровня ссылки ведут в корень (иначе эти вершины будут ссылаться на себя)
```

```
                }
                else {
                    current->SufLink = CurrentLink;
                    Bor* CurEndLink = current->SufLink;
                    while (CurEndLink != bor) {   // поиск ко-
нечной ссылки. если дошли до корня, значит её нет
                        if (CurEndLink->IsTerminal) {
                            current->EndLink = CurEndLink;
                            break;
                        }
                        CurEndLink = CurEndLink->SufLink;
                    }
                }
            }
            if (!current->next.empty()) { // добавляются новые
вершины в очередь
                for (auto& i : current->next) {
                    front.push(i.second);
                }
            }
        }
        }


        std::pair<int, int> LongestChain(Bor* bor, Bor* root, int&
depth) {  //вычисление длин наибольших цепочек из суф. и конеч-
ных ссылок
        std::pair<int, int> longest = { 0, 0 };
        Bor* current = bor;
        while (current->SufLink != root) {
            longest.first++;
            current = current->SufLink;
        }
        longest.first++;
        current = bor;
        while (current->EndLink != nullptr) {
```

```
            longest.second++;
            current = current->EndLink;
        }
        for (auto& i : bor->next) {
            std::pair<int, int> next = LongestChain(i.second,
root, ++depth);
            if (next.first > longest.first) {
                longest.first = next.first;
            }
            if (next.second > longest.second) {
                longest.second = next.second;
            }
        }
        depth--;
        return longest;
        }


        void AhoCorasick(const std::string& t, const
std::vector<std::pair<std::string, int>>& patterns,
std::vector<std::pair<int, int>>& result) {
        int length = 0;
        Bor* bor = CreateBor(patterns);
        Automaton(bor);
        std::cout << "\n\n_Поиск самых длинных цепочек из суффикс-
ных и конечных ссылок_\n";
        int depth = 0;
        std::pair<int, int> longest = LongestChain(bor, bor,
depth);
        std::cout << "Длина наибольшей цепочки из суффиксных ссы-
лок: " << longest.first << '\n';
        std::cout << "Длина наибольшей цепочки из конечных ссылок:
" << longest.second << '\n';
        Bor* current = bor;
        for (int i = 0; i < t.length(); i++) {
```

```
            current = current->GetLink(t.at(i));  // получена
ссылка для перехода
            Bor* EndLink = current->EndLink;
            while (EndLink != nullptr) {  // если у этой вершины
есть конечная ссылка, значит нашелся шаблон
                  result.push_back({ i - patterns.at(EndLink-
>TermNumPattern).first.length() + 2, EndLink->TermNumPattern + 1
});
                  EndLink = EndLink->EndLink;   // и так, пока це-
почка из конечных ссылок не прервётся
            }
            if (current->IsTerminal)  // если вершина терминальная
- шаблон найден
                  result.push_back({ i - patterns.at(current-
>TermNumPattern).first.length() + 2, current->TermNumPattern + 1
});
      }
      delete bor;
      }
```

Файл lab5_2.h

```
#pragma once
#include <string>
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>


class Bor {
public:
Bor* parent;   // родительская вершина
Bor* SufLink; // суффиксная ссылка
Bor* EndLink;  // конечная ссылка
```

```
    char ToParent; // ребро, по которому попали из родительской вершины
    bool IsTerminal;  // терминальая вершина или нет (конец шаблона)
    std::map<char, Bor*> next;
    std::vector<int> NumPattern;  // номера шаблонов, в которые входит символ, по которому пришли
    int TermNumPattern;
    std::vector<int> shifts;  // сдвиги подстрок в шаблоне

    Bor(Bor* parent = nullptr, char ToPrnt = 0) : parent(parent), ToParent(ToPrnt), SufLink(nullptr), EndLink(nullptr), IsTerminal(false) {
        if (parent == nullptr || ToPrnt == 0) { // создание корня
            this->parent = this;
            this->SufLink = this;
        }
    }

    ~Bor() {
        for (auto i : this->next) {
            delete i.second;
        }
    }

    Bor* GetLink(const char& symbol) {
        if (this->next.find(symbol) != this->next.end()) { // если есть путь по заданному символу из текущей вершины
            return this->next.at(symbol);
        }
        if (this->SufLink == this) { // если в корне и путь не найден
            return this;
        }
```

```
        return this->SufLink->GetLink(symbol); // не в корне и путь не найден
    }
    };

    Bor* CreateBor(const std::vector<std::pair<std::string, int>>& patterns);
    void Automaton(Bor* bor);
    std::pair<int, int> LongestChain(Bor* bor, Bor* root, int& depth);
    void AhoCorasick(const std::string& t, const std::vector<std::pair<std::string, int>>& patterns, std::vector<std::pair<int, int>>& result, int length);
    void Partitioning(const std::string& p, const char& joker, std::vector<std::pair<std::string, int>>& patterns);
```

Файл lab5_2.cpp

```
#include "lab5_2.h"

    Bor* CreateBor(const std::vector<std::pair<std::string, int>>& patterns) { //создание бора
    std::cout << "\n_Начало построения бора_";
    Bor* bor = new Bor();    // корень бора
    for (auto& pt : patterns) {
        int NumPattern = find(patterns.begin(), patterns.end(), pt) - patterns.begin();
        std::cout << "\nРассматривается " << NumPattern + 1 << " шаблон: " << pt.first << '\n';
        Bor* current = bor; // поиск пути, начиная от корня
        for (auto& c : pt.first) {
            if (current->next.find(c) == current->next.end()) { //если такого ребра нет, то добавляется
```

```
                    current->next.insert({ c, new Bor(current,
c) });
                    std::cout << "В бор добавлена вершина, куда
ведёт символ (" << c << ")\n";
               }
               else {
                    std::cout << "Ребро (" << c << ") для теку-
щего шаблона уже существует. Выполнение перехода.\n";
               }
               current = current->next[c]; // переход по данному
ребру
               current->NumPattern.push_back(NumPattern);
          }
          current->TermNumPattern = NumPattern; // для конечной
вершины
          current->IsTerminal = true;
          std::cout << "Вершина является терминальной, закончено
построение ветви бора.\n";
          current->shifts.push_back(pt.second);
          std::cout << "Данный шаблон имеет сдвиг " << pt.second
<< " относительно начала шаблона с маской.\n";
     }
     return bor;
     }


     void Automaton(Bor* bor) {  //вычисление суф. и конечных
ссылок (построение автомата)
     std::cout << "\n\n_Вычисление суффиксных и конечных ссы-
лок_\n";
     std::queue<Bor*> front({ bor });  // вершины одного уровня
в боре
     while (!front.empty()) {
          Bor* current = front.front();
          front.pop();
```

```
          Bor* CurrentLink = current->parent->SufLink; // роди-
тельская ссылка стала текущей
          const char& key = current->ToParent;    // сохраняется
символ, для которого ищется ссылка
          bool IsFound = true;
          while (CurrentLink->next.find(key) == CurrentLink-
>next.end()) {
               if (CurrentLink == bor) {
                    std::cout << "Суффиксные ссылки не найдены,
ссылка установлена на корень.\n";
                    current->SufLink = bor;  // если из корня
нет пути, то ссылка устанавливается в корень
                    IsFound = false;  //ссылка, не равная корню,
не найдена
                    break;
               }
               CurrentLink = CurrentLink->SufLink;
          }
          if (IsFound) {
               CurrentLink = CurrentLink->next.at(key);
               if (current->parent == bor) {
                    std::cout << "Текущая вершина - начало сло-
ва, поэтому ссылка установлена на корень.\n";
                    current->SufLink = bor;  // для вершин пер-
вого уровня ссылки ведут в корень (иначе эти вершины будут ссы-
латься на себя)
               }
               else {
                    current->SufLink = CurrentLink;
                    Bor* CurEndLink = current->SufLink;
                    while (CurEndLink != bor) {   // поиск ко-
нечной ссылки. если дошли до корня, значит её нет
                         if (CurEndLink->IsTerminal) {
                              current->EndLink = CurEndLink;
                              break;
                         }
```

```
                    CurEndLink = CurEndLink->SufLink;
                }
            }
        }

        if (!current->next.empty()) { // добавляются новые вершины в очередь
            for (auto& i : current->next) {
                front.push(i.second);
            }
        }
    }
    }


    std::pair<int, int> LongestChain(Bor* bor, Bor* root, int& depth) {  //вычисление длин наибольших цепочек из суф. и конечных ссылок
    std::pair<int, int> longest = { 0, 0 };
    Bor* current = bor;
    while (current->SufLink != root) {
        longest.first++;
        current = current->SufLink;
    }
    longest.first++;
    current = bor;
    while (current->EndLink != nullptr) {
        longest.second++;
        current = current->EndLink;
    }
    for (auto& i : bor->next) {
        std::pair<int, int> next = LongestChain(i.second, root, ++depth);
        if (next.first > longest.first) {
            longest.first = next.first;
        }
```

```
            if (next.second > longest.second) {
                longest.second = next.second;
            }
        }
        depth--;
        return longest;
        }


        void AhoCorasick(const std::string& t, const
std::vector<std::pair<std::string, int>>& patterns,
std::vector<std::pair<int, int>>& result, int length) {
        Bor* bor = CreateBor(patterns);
        Automaton(bor);
        std::cout << "\n\n_Поиск самых длинных цепочек из суффикс-
ных и конечных ссылок_\n";
        int depth = 0;
        std::pair<int, int> longest = LongestChain(bor, bor,
depth);
        std::cout << "Длина наибольшей цепочки из суффиксных ссы-
лок: " << longest.first << '\n';
        std::cout << "Длина наибольшей цепочки из конечных ссылок:
" << longest.second << '\n';
        Bor* current = bor;
        std::vector<int> index(t.length(), 0);
        for (int i = 0; i < t.length(); i++) {
            current = current->GetLink(t.at(i));  // получена
ссылка для перехода
            Bor* EndLink = current->EndLink;
            while (EndLink != nullptr) {  // если у этой вершины
есть конечная ссылка, значит нашелся шаблон
                for (auto& s : EndLink->shifts) {
                    int id = i - patterns.at(EndLink-
>TermNumPattern).first.length() - s + 1;
                    if (!(id < 0)) {
                        index.at(id)++;
```

```
                    }
               }
               EndLink = EndLink->EndLink;   // и так, пока це-
почка из конечных ссылок не прервётся
          }
          if (current->IsTerminal)  // если вершина терминальная
- шаблон найден
               for (auto& s : current->shifts) {
                    int id = i - patterns.at(current-
>TermNumPattern).first.length() - s + 1;
                    if (!(id < 0)) {
                         index.at(id)++;
                    }
               }
     }
     for (int i = 0; i < index.size(); i++) {
          if (index[i] == patterns.size() && i + length <=
t.length()) {
               result.push_back({ i + 1, 0 });
          }
     }
     delete bor;
     }


     void Partitioning(const std::string& p, const char& joker,
std::vector<std::pair<std::string, int>>& patterns) { //разбие-
ние шаблона с маской на безмасочные подшаблоны
     int prev = 0;
     int div;
     do {
          div = p.find(joker, prev);
          if ((div != prev) && (prev != p.length())) {
               patterns.push_back({ p.substr(prev, div - prev),
prev });
          }
```

```
            prev = div + 1;
        } while (div != std::string::npos); //пока не пусто
        }
```

Файл tests.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "../source/lb5_2/lb5_2.h"


TEST_CASE("Тест для функции Partitioning") {

    SECTION("1 случай") {
        std::vector<std::pair<std::string, int>> check;
        check.push_back({ "A", 0 });
        check.push_back({ "A", 3 });
        const std::string& p = "A$$A$";
        const char& joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        Partitioning(p, joker, patterns);
        CHECK(patterns == check);
    }

    SECTION("2 случай") {
        std::vector<std::pair<std::string, int>> check;
        check.push_back({ "A", 0 });
        check.push_back({ "A", 1 });
        check.push_back({ "A", 2 });
        check.push_back({ "A", 3 });
        check.push_back({ "A", 4 });
        const std::string& p = "AAAAA";
        const char& joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        Partitioning(p, joker, patterns);
        CHECK(patterns == check);
```

```
    }

    SECTION("3 случай") {
        std::vector<std::pair<std::string, int>> check;
        const std::string& p = "$$$$$";
        const char& joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        Partitioning(p, joker, patterns);
        CHECK(patterns == check);
    }
}


TEST_CASE("Тест для функции LongestChain") {

    SECTION("1 случай") {
        std::pair<int, int> check = { 1, 0 };
        std::pair<int, int> longest = { 0, 0 };
        const std::string& p = "A$$A$";
        const char& joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        Partitioning(p, joker, patterns);
        Bor* bor = CreateBor(patterns);
        Links(bor);
        int depth = 0;
        longest = LongestChain(bor, bor, depth);
        CHECK(longest == check);
    }

    SECTION("2 случай") {
        std::pair<int, int> check = { 5, 0 };
        std::pair<int, int> longest = { 0, 0 };
        const std::string& p = "AAAAA";
        const char& joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        Partitioning(p, joker, patterns);
```

```
        Bor* bor = CreateBor(patterns);
        Links(bor);
        int depth = 0;
        longest = LongestChain(bor, bor, depth);
        CHECK(longest == check);
    }

    SECTION("3 случай") {
        std::pair<int, int> check = { 1, 0 };
        std::pair<int, int> longest = { 0, 0 };
        const std::string& p = "$$$$$";
        const char& joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        Partitioning(p, joker, patterns);
        Bor* bor = CreateBor(patterns);
        Links(bor);
        int depth = 0;
        longest = LongestChain(bor, bor, depth);
        CHECK(longest == check);
    }
}


TEST_CASE("Тест для функции AhoCorasick") {

    SECTION("1 случай") {
        std::string T = "ACTANCA";
        std::string P = "A$$A$";
        char joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        std::vector<std::pair<int, int>> result;
        std::vector<std::pair<int, int>> check;
        check.push_back({ 1, 0 });
        Partitioning(P, joker, patterns);
        AhoCorasick(T, patterns, result, P.length());
        CHECK(result == check);
```

```
    }

    SECTION("2 случай") {
        std::string T = "ACTANCA";
        std::string P = "AAA";
        char joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        std::vector<std::pair<int, int>> result;
        std::vector<std::pair<int, int>> check;
        Partitioning(P, joker, patterns);
        AhoCorasick(T, patterns, result, P.length());
        CHECK(result == check);
    }

    SECTION("3 случай") {
        std::string T = "ACTANCA";
        std::string P = "$$$";
        char joker = '$';
        std::vector<std::pair<std::string, int>> patterns;
        std::vector<std::pair<int, int>> result;
        std::vector<std::pair<int, int>> check;
        check.push_back({ 1, 0 });
        check.push_back({ 2, 0 });
        check.push_back({ 3, 0 });
        check.push_back({ 4, 0 });
        check.push_back({ 5, 0 });
        Partitioning(P, joker, patterns);
        AhoCorasick(T, patterns, result, P.length());
        CHECK(result == check);
    }
}
```