МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)

Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 6.2

по дисциплине «Структуры и алгоритмы обработки данных»

Тема. Поиск образца в тексте

Выполнил студент группы ИКБО-41-23 Гольд Д.В.

Принял старший преподаватель Рысин М.Л.

Москва 2024

ЦЕЛЬ

Освоить приёмы реализации алгоритмов поиска образца в тексте.

ОТЧЕТ ПО ЗАДАНИЮ 1

Формулировка:

1. Дано предложение, слова в котором разделены пробелами и запятыми. Распечатать те пары слов, расстояние между которыми наименьшее. Расстояние – это количество позиций, в которых слова различаются. Например, МАМА и ПАПА, МЫШКА и КОШКА расстояние этих пар равно двум.

КОД ПРОГРАММЫ

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <unordered_map>
#include <algorithm>
#include <string>
#include <limits>

using namespace std;

int calculate_distance(const string& word1, const string& word2) {
    int distance = 0;
    int min_length = min(word1.size(), word2.size());

    for (int i = 0; i < min_length; ++i) {//считаем позиции - в которых символы различаются
        if (word1[i] != word2[i]) {
            distance++;
        }
    }

    distance += abs((int)word1.size() - (int)word2.size());//добавляем разницу в длине- если слова разной длины

    return distance;
}

void find_closest_words(const string& sentence) {
    vector<string> words;
    string word = "";

    for (char ch : sentence) {//проходим по предложению и разбиваем на слова - игнорируя запятые
        if (ch == ' ' || ch == ',') {
            if (!word.empty()) {
                words.push_back(word);
                word.clear();
            }
        }
        else {
            word += ch;
        }
    }
    if (!word.empty()) {//добавляем последнее слово - если оно не пустое
        words.push_back(word);
    }

    int min_distance = numeric_limits<int>::max();
    pair<string, string> closestPair;


    for (size_t i = 0; i < words.size(); ++i) {//линейный поиск - перебираем все пары слов
        for (size_t j = i + 1; j < words.size(); ++j) {
            int distance = calculate_distance(words[i], words[j]);
            if (distance < min_distance) {
                min_distance = distance;
                closestPair = { words[i], words[j] };
            }
        }
    }

    cout << "couple words with minimum distance: " << closestPair.first
        << " and " << closestPair.second << " (distance: " << min_distance << ")\n";
}

```

```
276  int main() {
277      string text;
278      string pattern;
279
280      string sentence = "ajdfkdfjk, kdasd a skdlksl ,  asdd dskdlk";
281      find_closest_words(sentence);
282
```

ТЕСТИРОВАНИЕ

ОТЧЕТ ПО ЗАДАНИЮ 2

Формулировка:

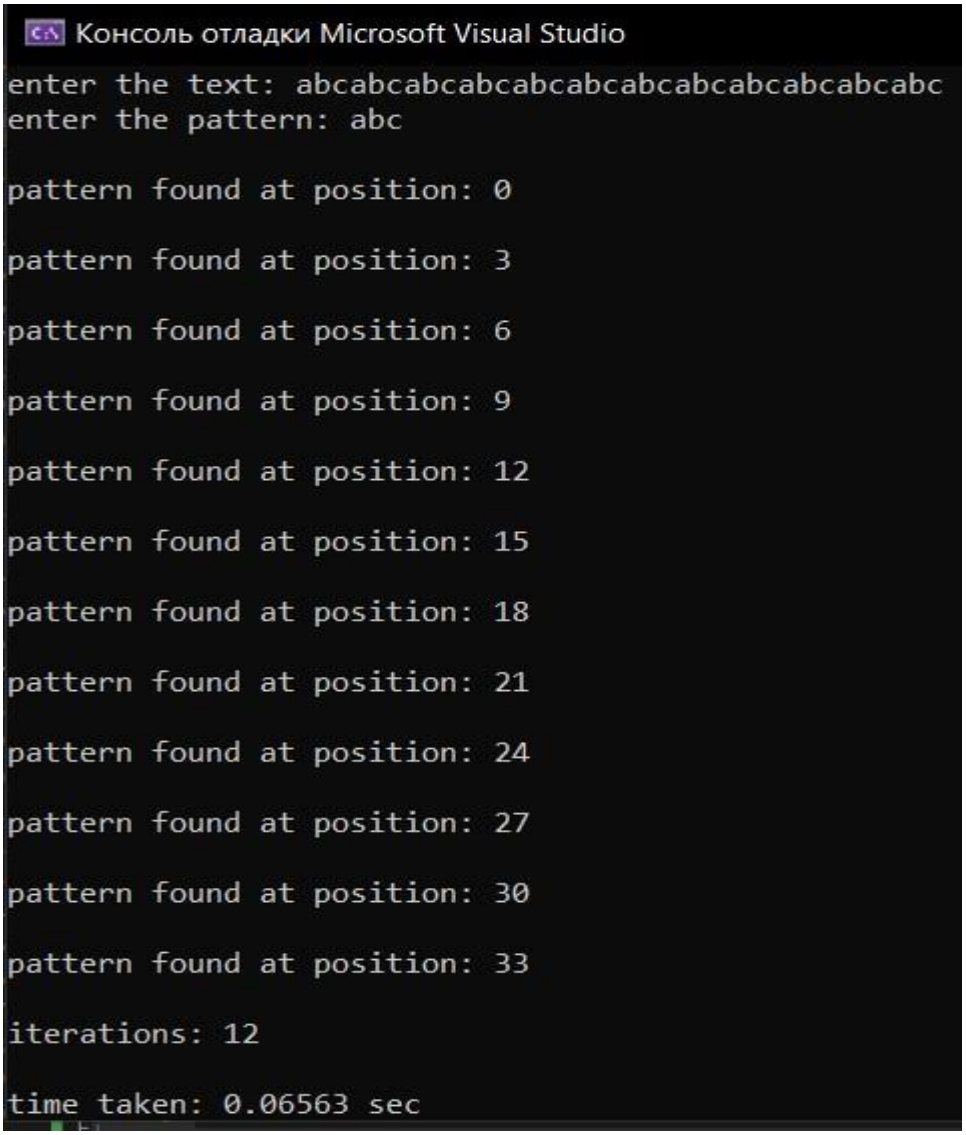
2. Найти все вхождения подстроки в строку, используя алгоритм Бойера-Мура с турбосдвигом.

КОД ПРОГРАММЫ

```
63  class BoyerMoreGoodSuffix {
64  public:
65      vector<int> z_function(const string& s) {//функция для вычисления Z-функции
66          int line_length_n = s.length();//длина строки
67          vector<int> z(line_length_n, 0);//инициализируем массив z нулями
68          int left_index_border = 0, right_index_border = 0;//границы строки
69
70          for (int i = 1; i < line_length_n; ++i) {//проходим по строке начиная с индекса 1
71              if (i <= right_index_border) {
72                  z[i] = min(right_index_border - i + 1, z[i - left_index_border]);
73              }
74              while (i + z[i] < line_length_n && s[z[i]] == s[i + z[i]]) {//расширяем совпадение
75                  z[i]++;//увелич длин совпадения
76              }
77              if (i + z[i] - 1 > right_index_border) {//если новое совпадение расширяет область
78                  left_index_border = i;
79                  right_index_border = i + z[i] - 1;
80              }
81          }
82
83          return z;
84      }
85
86      vector<int> build_good_suffix_table(const string& pattern) {
87          int length_template_substring_m = pattern.length();//длина шаблона
88          string reversed_pattern = pattern;//строка-реверс шаблона
89          reverse(reversed_pattern.begin(), reversed_pattern.end());
90
91          string concatenation_string = pattern + "$" + reversed_pattern;//объединяем шаблон с его реверсией через разделитель
92          vector<int> z = z_function(concatenation_string);
93
94          vector<int> good_suffix(length_template_substring_m + 1, length_template_substring_m);
95          for (int j = 0; j < length_template_substring_m; ++j) {//проходим по всем индексам шаблона
96              good_suffix[j] = length_template_substring_m - z[length_template_substring_m + 1 + j];//заполняем таблицу сдвигами
97              //по совпавшим суффиксам
98          }
99          return good_suffix;
100     }
101
102     void search(const string& text, const string& pattern) {
103         auto start = chrono::high_resolution_clock::now();
104
105         int line_length_n = text.length();//длина текста
106         int length_template_substring_m = pattern.length();//длина шаблона
```

```
101
102     void search(const string& text, const string& pattern) {
103         auto start = chrono::high_resolution_clock::now();
104
105         int line_length_n = text.length();//длина текста
106         int length_template_substring_m = pattern.length();//длина шаблона
107
108         vector<int> good_suffix = build_good_suffix_table(pattern);
109
110         int s = 0;//начальное смещение для шаблона
111         int iterations = 0;
112
113         while (s <= line_length_n - length_template_substring_m) {//в пределах строки
114             int j = length_template_substring_m - 1;//с конца шаблона
115
116             //начинаем сравнение с конца шаблона
117             while (j >= 0 && pattern[j] == text[s + j]) {//если совпад
118                 --j;
119             }
120
121             if (j < 0) {//шаблон совпал
122                 cout << "\npattern found at position: " << s << endl;
123                 s += good_suffix[0]; // сдвиг шаблон = знач табл суффиксов
124             }
125             else {//нет
126                 s += good_suffix[j + 1];//сдвиг = тбл хор суфф
127             }
128             ++iterations;
129         }
130
131         auto end = chrono::high_resolution_clock::now();
132         chrono::duration<double> duration = end - start;
133
134         cout << "\niterations: " << iterations << endl;
135         cout << "\ntime taken: " << duration.count() << " sec\n";
136     }
137 };
138
139 class BoyerMoreBadSuffix { ... };
200
201 class BoyerMoreTurboShift { ... };
276
277 int main() {
```

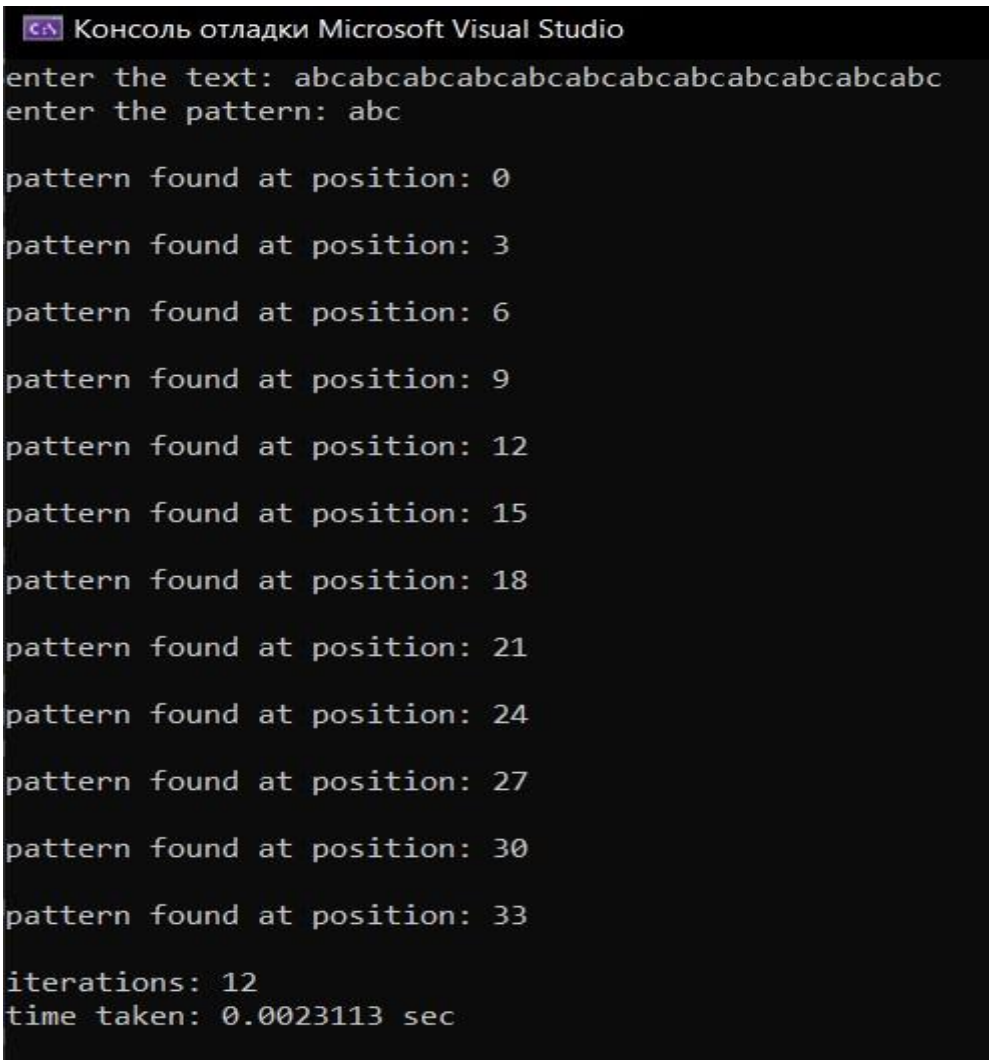
ТЕСТИРОВАНИЕ



```cpp
class BoyerMoreBadSuffix {
public:
    BoyerMoreBadSuffix(const string& pattern) {
        this->pattern = pattern;
        this->m_pattern_length = pattern.length();
        create_bad_suffix_table();
    }

    void search(const string& text) { //поиск подстроки в строке
        auto start_time = chrono::high_resolution_clock::now();
        int n_line_length = text.length();
        int i = 0; //индекс в тексте
        int count = 0; //счетчик итераций

        while (i <= n_line_length - m_pattern_length) {
            int j = m_pattern_length - 1; //индекс в шаблоне
            while (j >= 0 && pattern[j] == text[i + j]) {
                j--;
            }

            if (j < 0) { //совпадение найдено
                cout << "\npattern found at position: " << i << endl;
                i += good_suffix[0];//сдвиг по хорошему суффиксу
            }
            else {
                int bad_char_shift = max(1, j - last_occurence[text[i + j]]); //вычисление сдвигов для плохого символа и
                //хорошего суффикса
                int good_suffix_shift = good_suffix[j + 1];

                i += max(bad_char_shift, good_suffix_shift); //сдвиг
            }
            count++;
        }

        auto end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end_time - start_time;

        cout << "\niterations: " << count << endl;
        cout << "time taken: " << duration.count() << " sec\n";
    }
```

```cpp
private:
    string pattern;
    int m_pattern_length;
    vector<int> good_suffix;
    unordered_map<char, int> last_occurence;

    void create_bad_suffix_table() {
        for (int i = 0; i < m_pattern_length; i++) { //заполняем таблицу последних вхождений каждого символа
            last_occurence[pattern[i]] = i;
        }

        good_suffix = vector<int>(m_pattern_length + 1, m_pattern_length); //создание таблицы хороших суффиксов
        for (int i = m_pattern_length - 2; i >= 0; i--) {
            int j = i;
            while (j >= 0 && pattern[j] == pattern[m_pattern_length - 1 - (i - j)]) { //поиск наибольшего суффикса шаблона
                j--;
            }
            good_suffix[i] = max(1, m_pattern_length - (i - j));
        }
    }
};
```

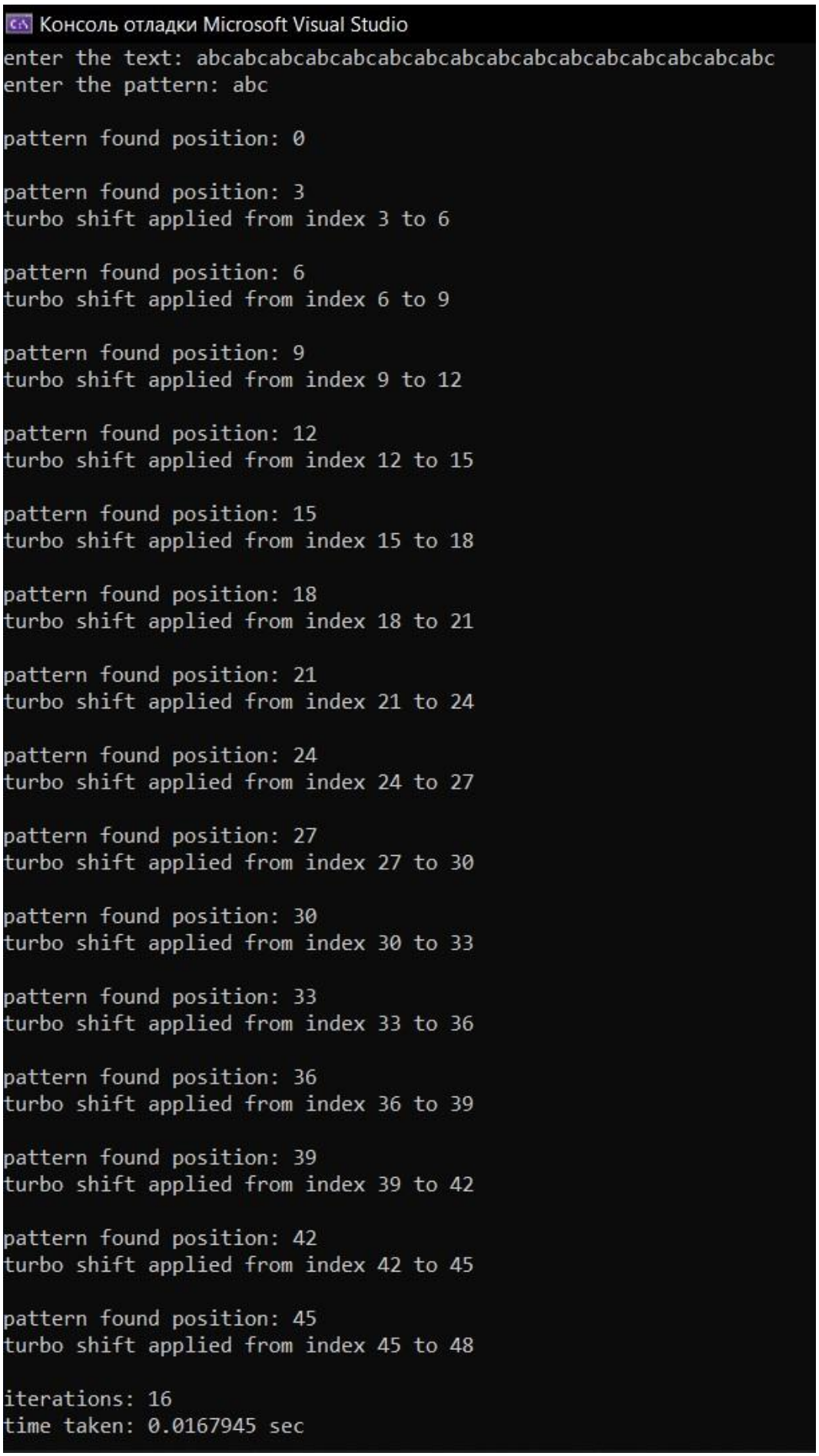
ТЕСТИРОВАНИЕ


Консоль отладки Microsoft Visual Studio
enter the text: abcabcabcabcabcabcabcabcabcabcabc
enter the pattern: abc
pattern found at position: 0
pattern found at position: 3
pattern found at position: 6
pattern found at position: 9
pattern found at position: 12
pattern found at position: 15
pattern found at position: 18
pattern found at position: 21
pattern found at position: 24
pattern found at position: 27
pattern found at position: 30
pattern found at position: 33
iterations: 12
time taken: 0.0023113 sec

```
class BoyerMoreTurboShift {
public:
    BoyerMoreTurboShift(const string& pattern) {
        this->pattern = pattern;
        this->m_pattern_length = pattern.length();
        create_bad_suffix_table();
        create_good_suffix_table();
    }

    void search(const string& text) {
        auto start_time = chrono::high_resolution_clock::now();
        int n_line_length = text.length();
        int i = 0;//индекс в тексте
        int count = 0;//счетчик итераций
        bool turbo_shift_used = false;

        while (i <= n_line_length - m_pattern_length) {
            int j = m_pattern_length - 1;//индекс в шаблоне
            while (j >= 0 && pattern[j] == text[i + j]) {
                j--;
            }

            if (j < 0) {
                cout << "\npattern found position: " << i << endl;

                //применяем турбосдвиг
                if (turbo_shift_used) {
                    i += m_pattern_length;//применяем турбосдвиг
                    cout << "turbo shift applied from index " << i - m_pattern_length << " to " << i << endl;
                }
                else {
                    turbo_shift_used = true;//после первого совпадения применяем турбосдвиг
                    i += m_pattern_length;//стандартный сдвиг
                }
            }
            else {
                int bad_char_shift = max(1, j - last_occurence[text[i + j]]);
                int good_suffix_shift = good_suffix[j + 1];

                int shift = max(bad_char_shift, good_suffix_shift);
                i += shift;//jбычный сдвиг
            }
            count++;
        }
```

```
            else {
                int bad_char_shift = max(1, j - last_occurence[text[i + j]]);
                int good_suffix_shift = good_suffix[j + 1];

                int shift = max(bad_char_shift, good_suffix_shift);
                i += shift;//jбычный сдвиг
            }
            count++;
        }

        auto end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end_time - start_time;

        cout << "\niterations: " << count << endl;
        cout << "time taken: " << duration.count() << " sec\n";
    }

private:
    string pattern;
    int m_pattern_length;
    vector<int> good_suffix;
    unordered_map<char, int> last_occurence;

    void create_bad_suffix_table() {
        for (int i = 0; i < m_pattern_length; i++) {
            last_occurence[pattern[i]] = i;
        }
    }

    void create_good_suffix_table() {
        good_suffix = vector<int>(m_pattern_length + 1, m_pattern_length);//создание таблицы хороших суффиксов
        for (int i = m_pattern_length - 2; i >= 0; i--) {
            int j = i;
            while (j >= 0 && pattern[j] == pattern[m_pattern_length - 1 - (i - j)]) {//поиск наибольшего суффикса шаблона
                j--;
            }
            good_suffix[i] = max(1, m_pattern_length - (i - j));
        }
    }
};
```

ТЕСТИРОВАНИЕ


Консоль отладки Microsoft Visual Studio
enter the text: abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc
enter the pattern: abc
pattern found position: 0
pattern found position: 3
turbo shift applied from index 3 to 6
pattern found position: 6
turbo shift applied from index 6 to 9
pattern found position: 9
turbo shift applied from index 9 to 12
pattern found position: 12
turbo shift applied from index 12 to 15
pattern found position: 15
turbo shift applied from index 15 to 18
pattern found position: 18
turbo shift applied from index 18 to 21
pattern found position: 21
turbo shift applied from index 21 to 24
pattern found position: 24
turbo shift applied from index 24 to 27
pattern found position: 27
turbo shift applied from index 27 to 30
pattern found position: 30
turbo shift applied from index 30 to 33
pattern found position: 33
turbo shift applied from index 33 to 36
pattern found position: 36
turbo shift applied from index 36 to 39
pattern found position: 39
turbo shift applied from index 39 to 42
pattern found position: 42
turbo shift applied from index 42 to 45
pattern found position: 45
turbo shift applied from index 45 to 48
iterations: 16
time taken: 0.0167945 sec

ВЫВОД

Освоил приёмы реализации алгоритмов поиска образца в тексте.