Национальный исследовательский университет
«Высшая школа экономики»

Отчет по индивидуальному домашнему заданию №2

«Практический анализ алгоритмов поиска вхождений строки-шаблона в тексте.»

Выполнил:
студент 213 группы
Мовшин М. А.

Москва
2023

Содержание

1. Цель работы

- Реализация различных алгоритмов поиска вхождений заданного паттерна (шаблона) в исходном тексте
- Экспериментальный анализ временной сложности алгоритмов
  - алгоритм Кнута-Морриса-Прата
  - алгоритм Кнута-Морриса-Прата с применением уточненных граней
  - наивный алгоритм
- Сопоставление теоретических и практических оценок временной сложности алгоритмов
- Интерпретация экспериментальных результатов

2. Тестовые данные

В соответствии с запросами условия мною были проведены тесты всех трех алгоритмов со всеми возможными вариациями следующих параметров:

- Длина текста: 10.000/100.000 символов
- Размер алфавита: 2/4 символов
- Количество символов подстановки: 1/2/3/4 символа

И размером паттерна от 100 до 3000 символов с шагом в 100 символов. Эти тесты, однако, не позволяли построить репрезентативные графики для КМП-алгоритмов - те отрабатывали слишком быстро, либо прирост размера паттерна был непоказателен. Поэтому я также провел тесты с теми же вариациями по размеру алфавита и количеству символов подстановки, но другими размерами текстов и паттернов.

Размер текста: 1.000.000 символов. Размер паттерна - до половины размера текста с шагом в $\frac{1}{50}$ размера текста.

3. Особенности проведения исследования

Для усреднения замеров за время прохождения теста бралось среднее по пяти запускам.

Для проверки паттерна с учетом символа подстановки все сравнения символов производились с помощью специальной функции, верно трактующей символ подстановки.

Генерация тестовых данных и выполнение тестов реализованы внутри одной программы. Время прохождения тестов не учитывает время генерации данных.

Результирующие данные были экспортированы в текстовые файлы. После они были загружены в среду Jupiter, где была проделана аналитическая работа - построение графиков и т. д.

4. Исследование

Для усреднения замеров за время прохождения теста бралось среднее по пяти запускам.

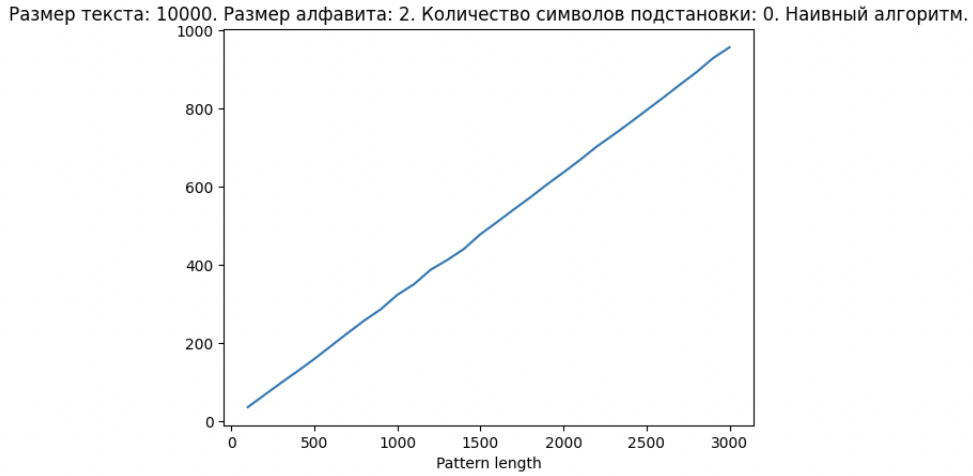
Для проверки паттерна с учетом символа подстановки все сравнения символов производились с помощью специальной функции, верно трактующей символ подстановки.

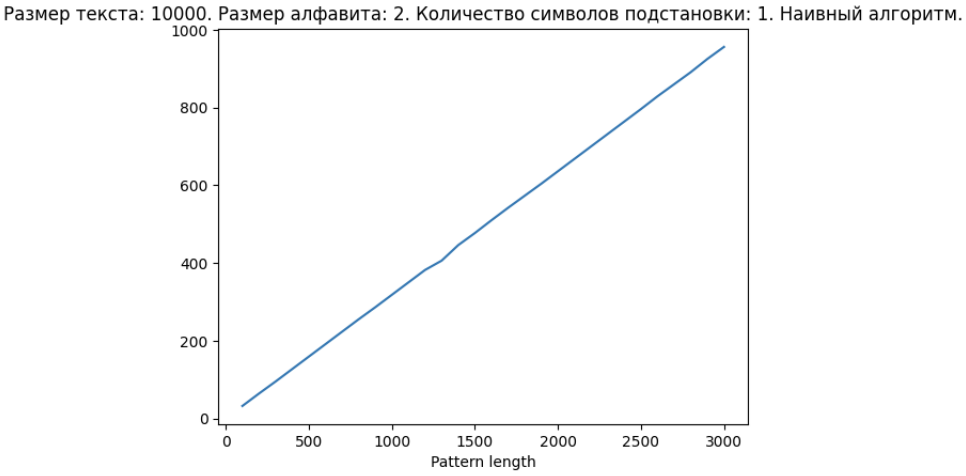
4..1 Наивный алгоритм

Ожидаемая асимптотика для наивного алгоритма:

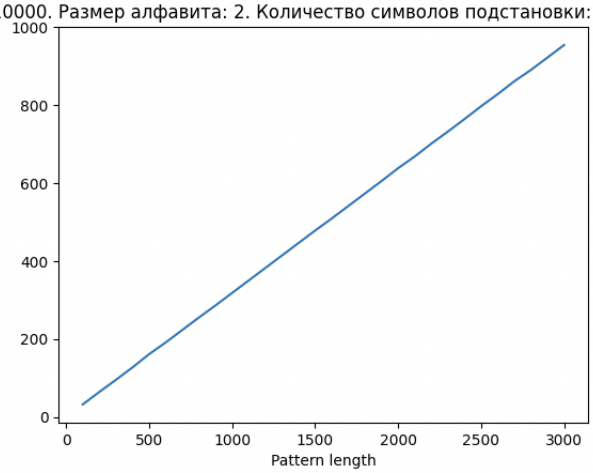
$$O(text\_size \cdot pattern\_size)$$

Взглянем на графики:

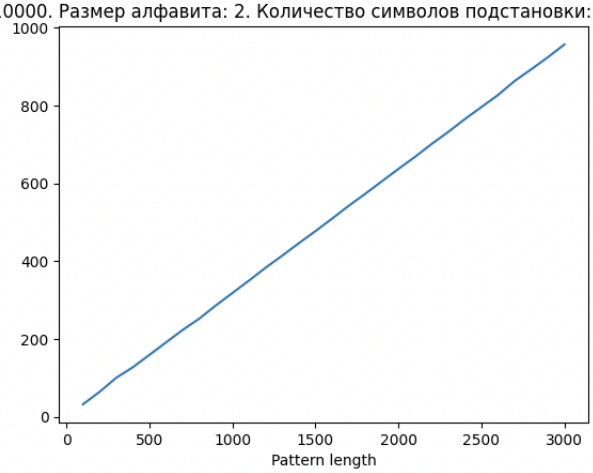




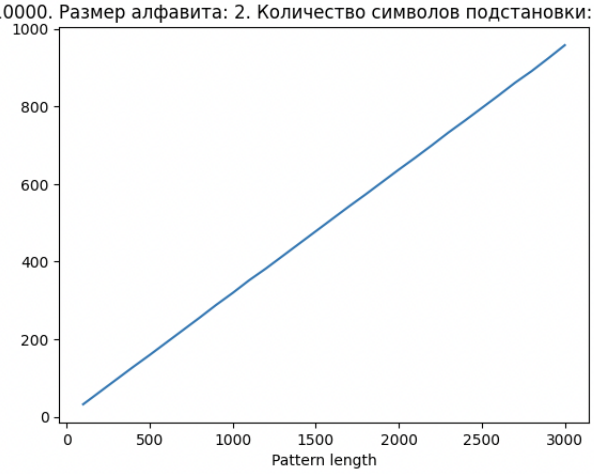
Размер текста: 10000. Размер алфавита: 2. Количество символов подстановки: 2. Наивный алгоритм.



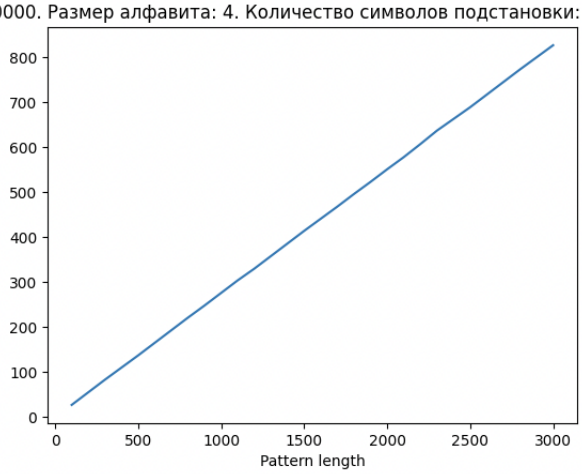
Размер текста: 10000. Размер алфавита: 2. Количество символов подстановки: 3. Наивный алгоритм.



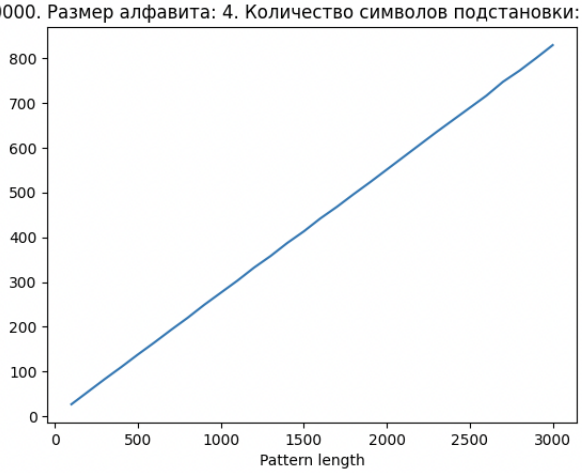
Размер текста: 10000. Размер алфавита: 2. Количество символов подстановки: 4. Наивный алгоритм.



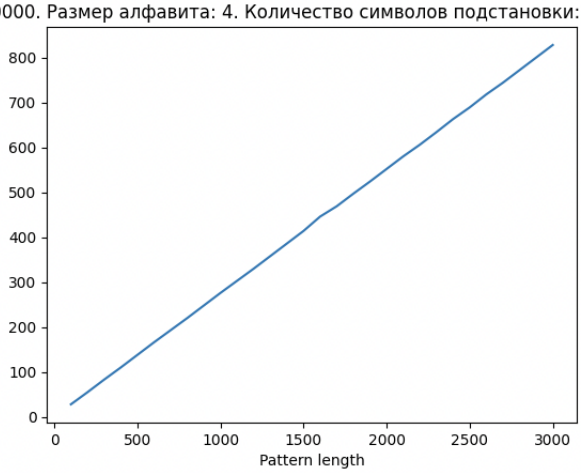
Размер текста: 10000. Размер алфавита: 4. Количество символов подстановки: 0. Наивный алгоритм.



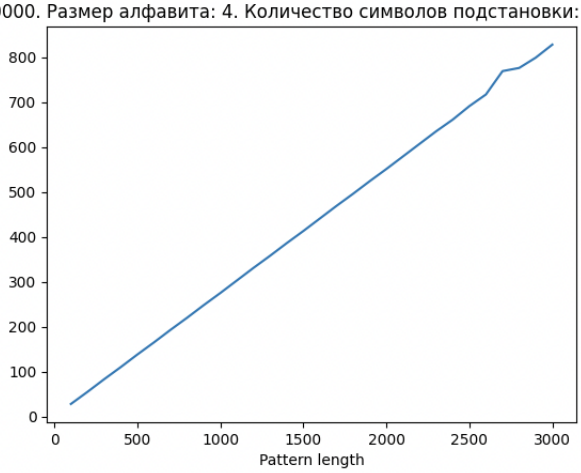
Размер текста: 10000. Размер алфавита: 4. Количество символов подстановки: 1. Наивный алгоритм.



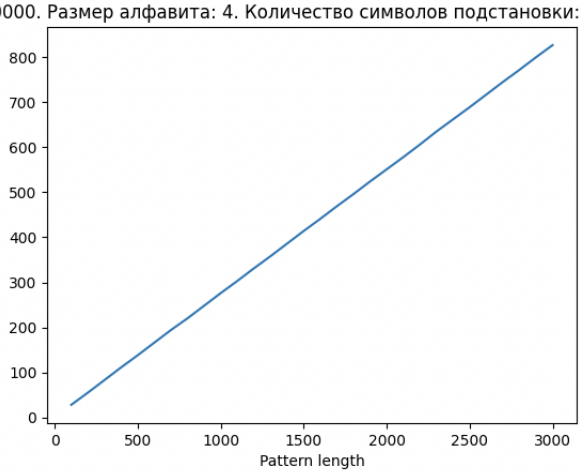
Размер текста: 10000. Размер алфавита: 4. Количество символов подстановки: 2. Наивный алгоритм.



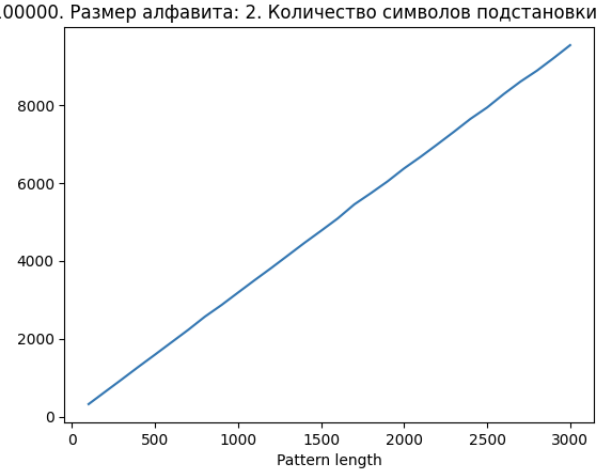
Размер текста: 10000. Размер алфавита: 4. Количество символов подстановки: 3. Наивный алгоритм.



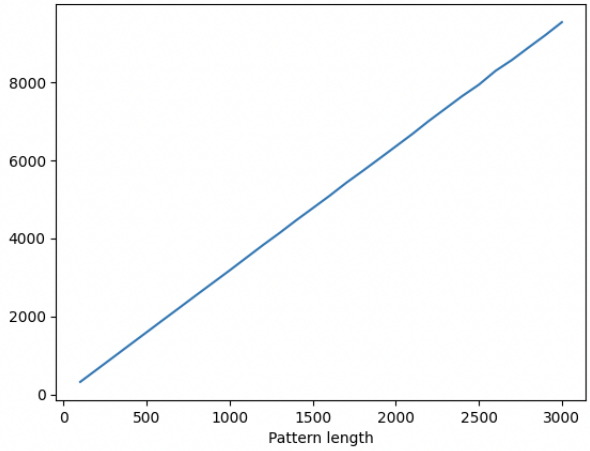
Размер текста: 10000. Размер алфавита: 4. Количество символов подстановки: 4. Наивный алгоритм.



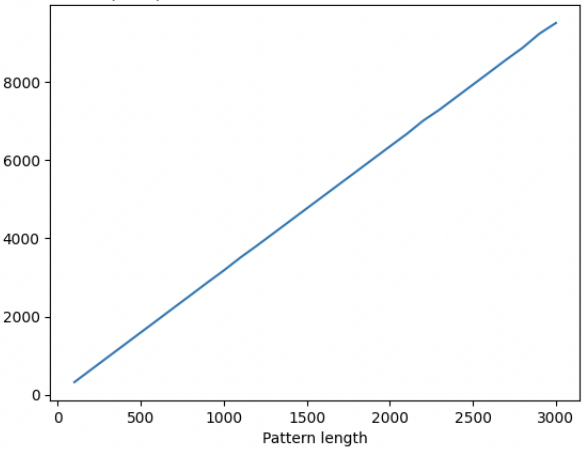
Размер текста: 100000. Размер алфавита: 2. Количество символов подстановки: 0. Наивный алгоритм.



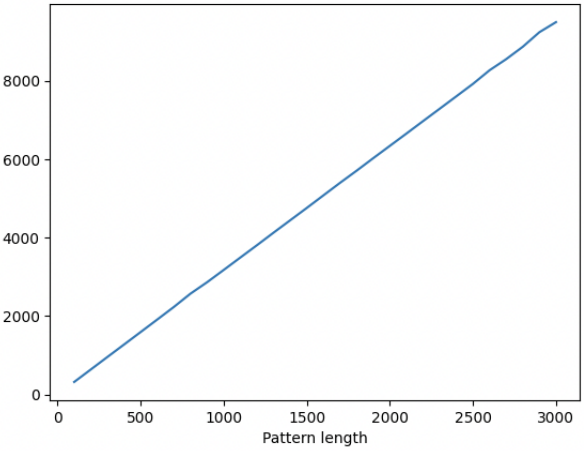
Размер текста: 100000. Размер алфавита: 2. Количество символов подстановки: 1. Наивный алгоритм.



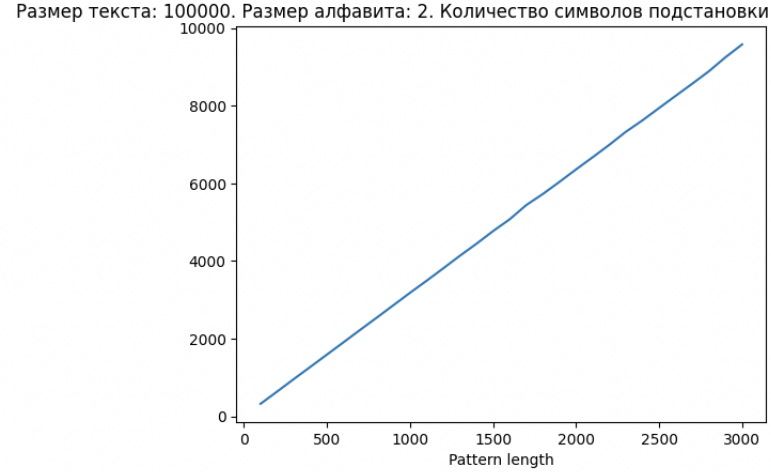
Размер текста: 100000. Размер алфавита: 2. Количество символов подстановки: 2. Наивный алгоритм.



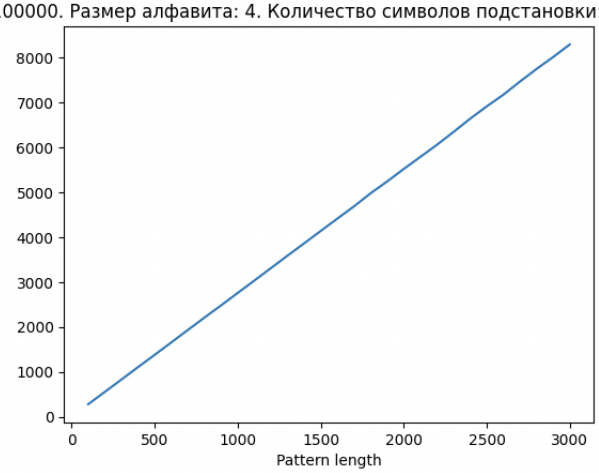
Размер текста: 100000. Размер алфавита: 2. Количество символов подстановки: 3. Наивный алгоритм.



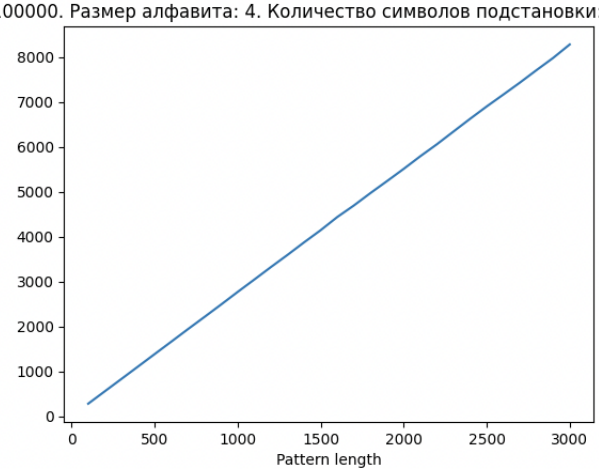
Размер текста: 100000. Размер алфавита: 2. Количество символов подстановки: 4. Наивный алгоритм.



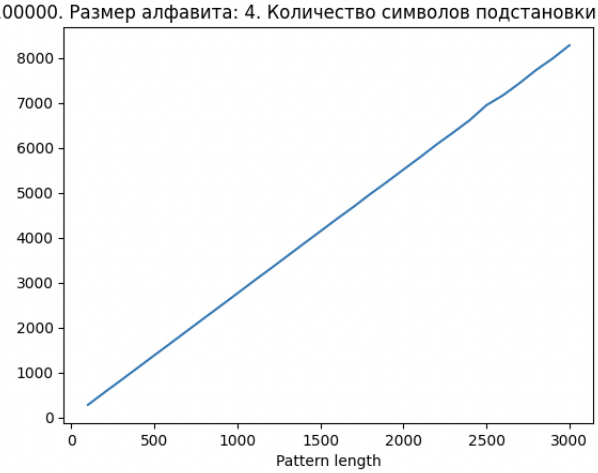
Размер текста: 100000. Размер алфавита: 4. Количество символов подстановки: 0. Наивный алгоритм.



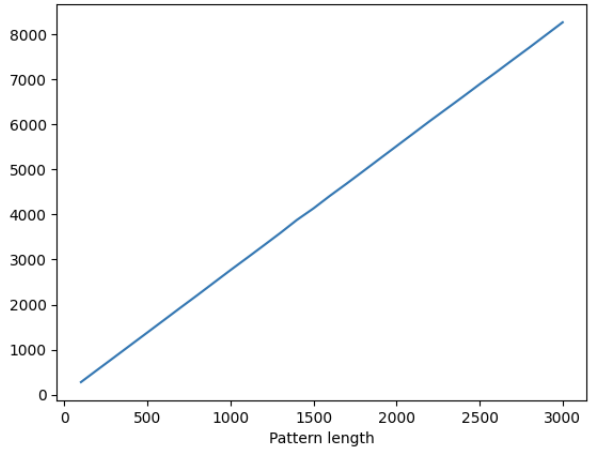
Размер текста: 100000. Размер алфавита: 4. Количество символов подстановки: 1. Наивный алгоритм.



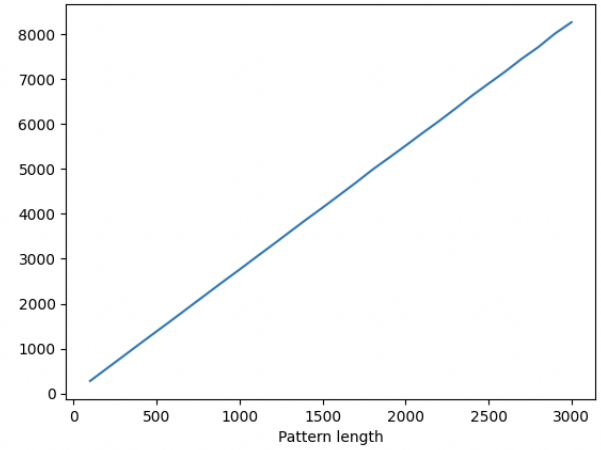
Размер текста: 100000. Размер алфавита: 4. Количество символов подстановки: 2. Наивный алгоритм.



Размер текста: 100000. Размер алфавита: 4. Количество символов подстановки: 3. Наивный алгоритм.



Размер текста: 100000. Размер алфавита: 4. Количество символов подстановки: 4. Наивный алгоритм.



На всех графиках прослеживается одна и та же картина - почти идеальная прямая стремящаяся началом к началу координат. Во-первых, легко заметить линейную зависимость от размера паттерна. Во-вторых, то, что прямая исходит из начала координат говорит о том, что размер паттерна выступает множителем в асимптотической формуле. По предложенным в условии входным данным

я сделал вывод, что линейная зависимость от размера текста считается очевидной (не было предложено делать графики с константным размером паттерна и изменяющимся размером текста). Поэтому можно сделать общий вывод об асимптотике: раз размер паттерна входит как множитель в первой степени, асимптотика для наивного алгоритма действительно:

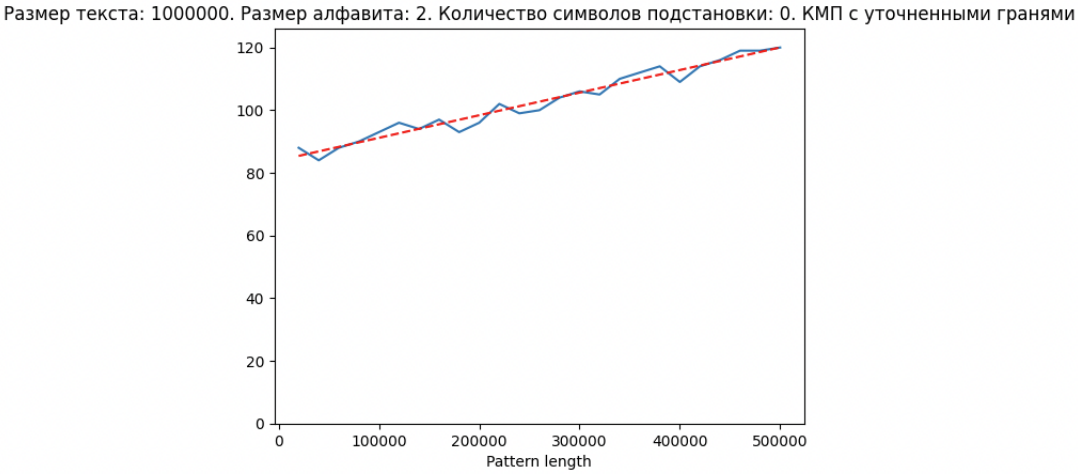
$$O(text\_size \cdot pattern\_size)$$

4..2 КМП алгоритмы

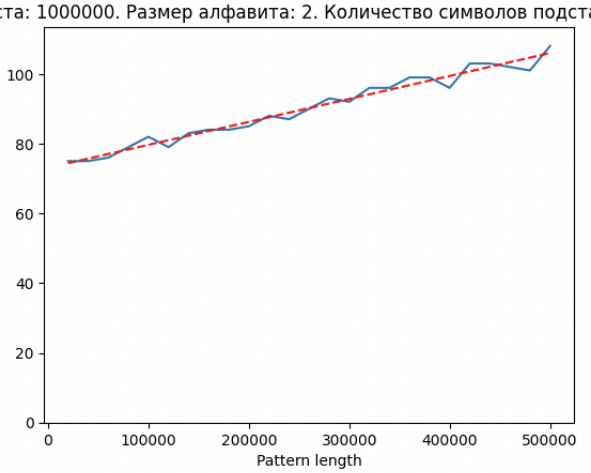
Ожидаемая асимптотика для КМП алгоритмов (обоих):

$$O(text\_size + pattern\_size)$$

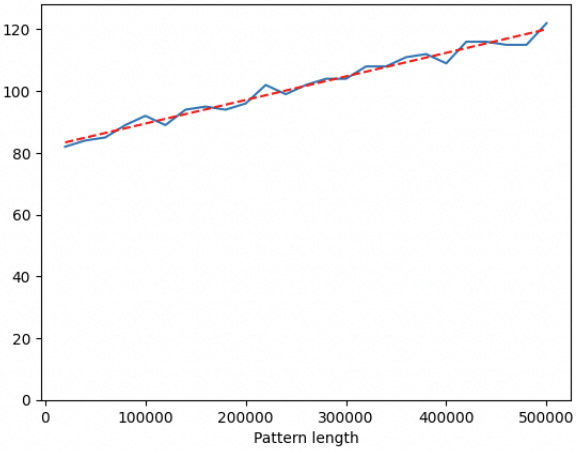
Взглянем на графики:



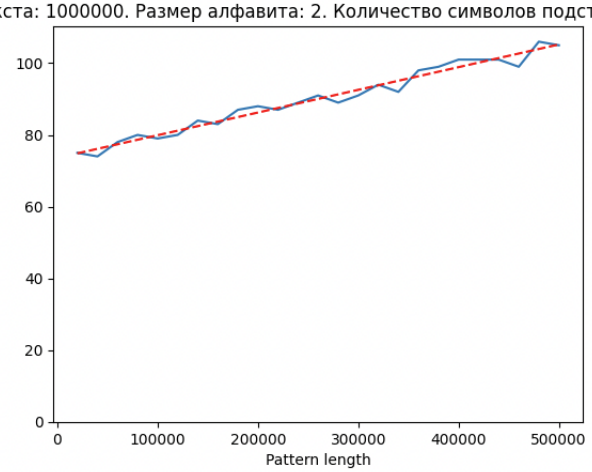
Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 1. КМП



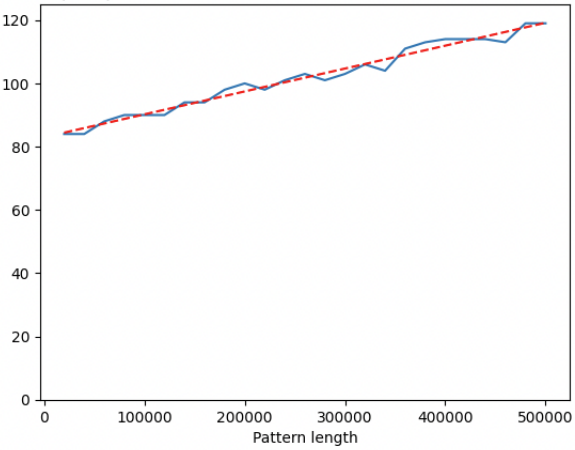
Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 1. КМП с уточненными гранями



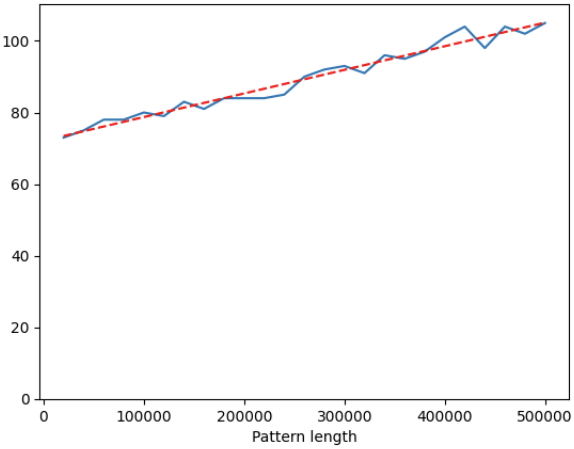
Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 2. КМП



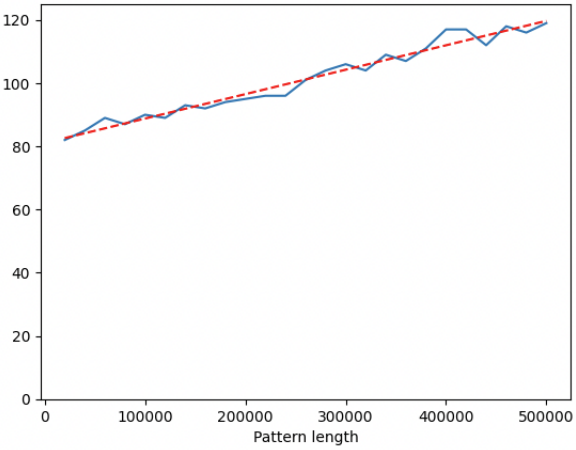
Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 2. КМП с уточненными гранями



Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 3. КМП



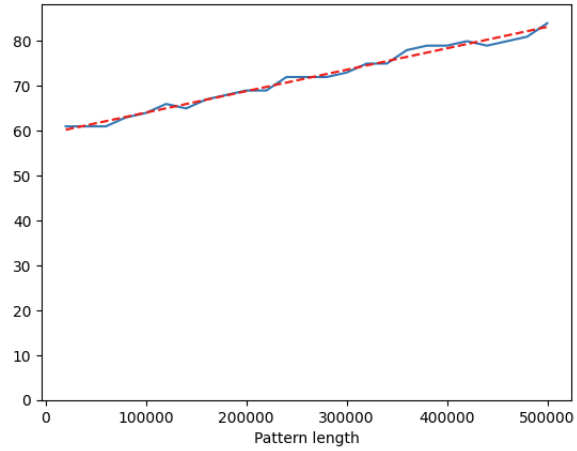
Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 3. КМП с уточненными гранями



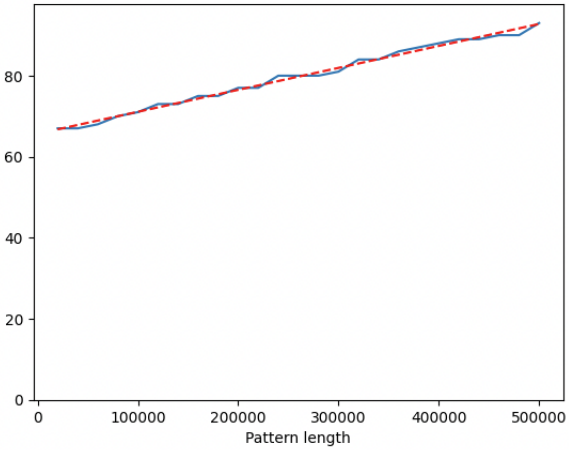
Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 4. КМП

Размер текста: 1000000. Размер алфавита: 2. Количество символов подстановки: 4. КМП с уточненными гранями

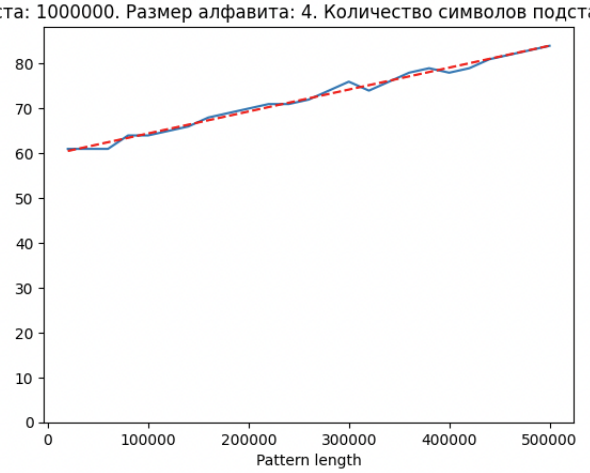
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 0. КМП



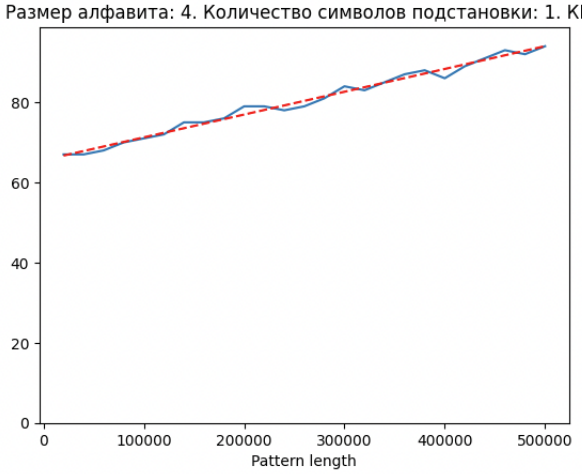
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 0. КМП с уточненными гранями



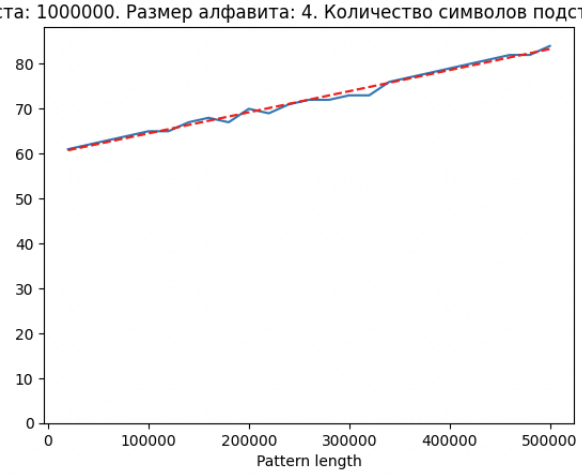
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 1. КМП



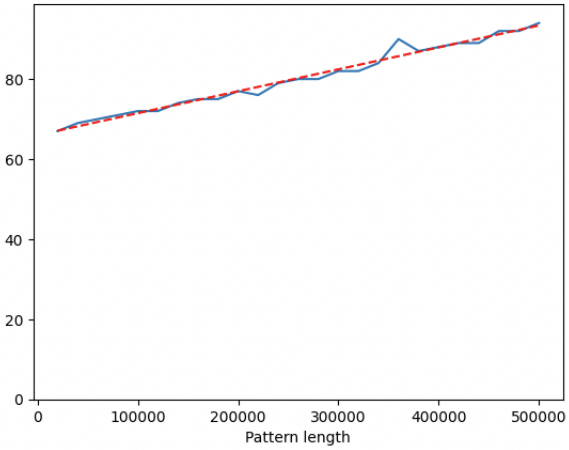
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 1. КМП с уточненными гранями



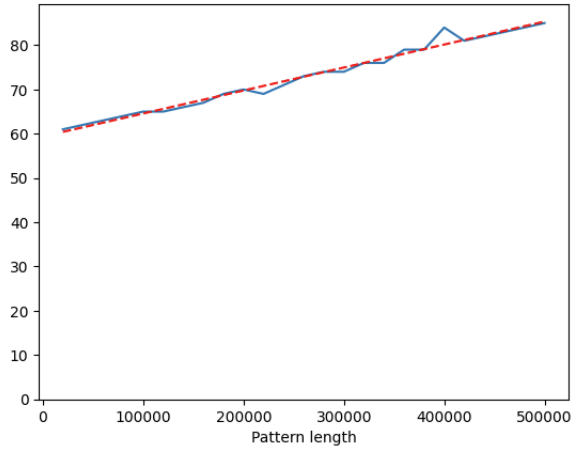
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 2. КМП



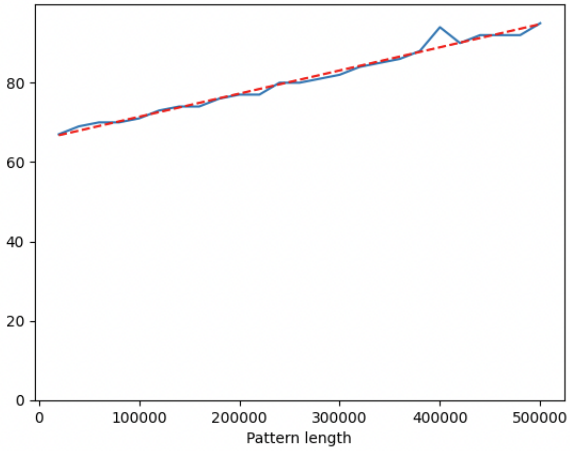
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 2. КМП с уточненными гранями



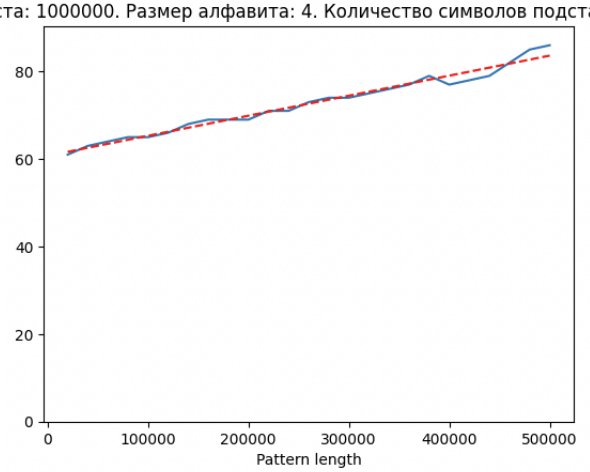
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 3. КМП



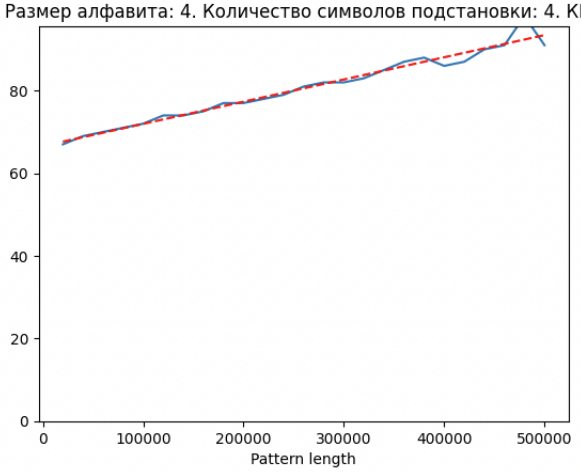
Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 3. КМП с уточненными гранями



Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 4. КМП



Размер текста: 1000000. Размер алфавита: 4. Количество символов подстановки: 4. КМП с уточненными гранями



На всех графиках видна довольно близкая к прямой картина (линии красным пуктиром - аппроксимация полиномом первой степени). Прямая имеет наклон, заметна линейная зависимость между размером паттерна и временем работы. Однако при минимальном размере паттерна сложность стремится к константе (не нулю) причем. Это позволяет говорить о том, что размер паттерна выступает слагаемым в первой степени в асимптотической формуле. Опять же,

использовав наши предположения о зависимости от размера текста, можем заключить, что сложность действительно:

$$O(text\_size \cdot pattern\_size)$$

4..3 Замечания

Оценка временной сложности алгоритма КМП с уточненными гранями легко доказуема. С одной стороны, оценка сверху доказывается также, как и для обычного алгоритма КМП. С другой же, очевидно невозможно построить алгоритм, которому не пришлось бы как минимум считать входные данные. Поэтому сложность минимум $O(text\_size + pattern\_size)$.

Исследование на зависимость от размера алфавита не предполагались, судя по предлагаемым входным данным. Поэтому даже если это число влияет на сложность, оно предполагается константым коэффициентом, и, следовательно, опускается. Аналогично с количеством символов паттерна.

Все графики, различающиеся лишь количеством символов паттерна, почти идентичны. Можно сделать вывод, что количество символов паттерна, по крайней мере в таком небольшом количестве, практически не влияют на время работы алгоритма.

5. Выводы

- Алгоритмы были корректно реализованы
- Эксперементальный анализ временной сложности алгоритмов подтвердил теоретические оценки
- Влияние количества символов паттерна и размер алфавита невозможно исследовать с предлагаемыми входными данными.

6. Листинги

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <unordered_map>

inline bool compareSymbols(const char& i, const char& j) {
    return i == j || i == '?' || j == '?';
}

std::vector<int> prefixFunction(const std::string& s) {
    std::vector<int> pi(s.length());
    pi[0] = 0;
    int j;
    for (int i = 1; i < static_cast<int>(s.length()); ++i) {
        j = pi[i - 1];

        while (j > 0 && !compareSymbols(s[i], s[j])) {
            j = pi[j - 1];
        }

        if (compareSymbols(s[i], s[j])) {
            j++;
        }

        pi[i] = j;
    }
    return pi;
}

std::vector<int> prefixFunctionSpecial(const std::string& s) {
    std::vector<int> pi = prefixFunction(s);
    for (int i = 0; i < static_cast<int>(pi.size()) - 1; ++i) {
        if (pi[i + 1] == pi[i] + 1) {
            pi[i] = 0;
        }
    }
    return pi;
}

std::vector<int> kmp(const std::string& text, const std::string&
    pattern, bool special = false) {
    std::string data = pattern + "#" + text;
    std::vector<int> pi = (special) ? prefixFunctionSpecial(data) :
    prefixFunction(data);
    std::vector<int> answers;
    int pattern_size = static_cast<int>(pattern.length());
    for (int i = 0; i < static_cast<int>(text.length()); ++i) {
        if (pi[i + pattern_size + 1] == static_cast<int>(pattern.
    length())) {
            answers.push_back(i - pattern_size + 1);
        }
    }
```

```
51      return answers;
52  }
53
54  std::vector<int> naiveSearch(const std::string& text, const std::
        string& pattern) {
55      std::vector<int> answers;
56      bool flag;
57      for (int i = 0; i < text.length(); ++i) {
58          flag = true;
59          for (int j = 0; j < pattern.length(); ++j) {
60              if (!compareSymbols(pattern[j], text[i + j])) {
61                  flag = false;
62              }
63              if (flag  && j == pattern.length() - 1) {
64                  answers.push_back(i);
65              }
66          }
67      }
68      return answers;
69  }
70
71  std::string createRandomString(int size, int alphabet_size) {
72
73      std::string res;
74      for (int i = 0; i < size; ++i) {
75          auto t = rand() % alphabet_size;
76          res += 'A' + t;
77      }
78      return res;
79  }
80
81  struct Point {
82      int pattern_length;
83      long long time; // in milliseconds
84  };
85
86  void insertToFile(const std::vector<std::vector<Point>>& vv, const
        std::string& file_name) {
87      std::ofstream file;
88      file.open(file_name);
89      std::string output;
90      for (const auto& v : vv) {
91          for (const auto& p : v) {
92              output += std::to_string(p.pattern_length) + ',' + std
        ::to_string(p.time) + ';';
93          }
94          output = output.substr(0, output.length() - 1);
95          output += '|';
96      }
97      output = output.substr(0, output.length() - 1);
98      file << output;
99      file.close();
100 }
101
102 long long process(std::function<void()> const& function) {
103     long long aggr = 0;
```

```
104     long long start, end;
105     for (int i = 0; i < 5; ++i) {
106         auto time_start = std::chrono::system_clock::now();
107         auto start_since_epoch = time_start.time_since_epoch();
108         start = std::chrono::duration_cast<std::chrono::
    milliseconds>(start_since_epoch).count();
109         function();
110         auto time_end = std::chrono::system_clock::now();
111         auto end_since_epoch = time_end.time_since_epoch();
112         end = std::chrono::duration_cast<std::chrono::milliseconds
    >(end_since_epoch).count();
113         aggr += (end - start);
114     }
115     return aggr / 5;
116 }
117
118 void doTest(int text_size, int alphabet_size, int
    magic_symbols_amount = 0, bool with_naive = true, int
    pattern_step = 100, int pattern_max_size = 3000, const std::
    string& suffix = "") {
119     std::vector<Point> v_naive, v_kmp, v_kmp_special;
120     for (int pattern_size = pattern_step; pattern_size <=
    pattern_max_size; pattern_size += pattern_step) {
121         std::string text = createRandomString(text_size,
    alphabet_size);
122         std::string pattern = createRandomString(pattern_size,
    alphabet_size);
123         if (magic_symbols_amount != 0) {
124             std::unordered_map<int, int> m;
125             std::vector<int> magic_indexes;
126             int max_size = pattern_size;
127             int randomized;
128             for (int i = 0; i < magic_symbols_amount; ++i) {
129                 randomized = rand() % (max_size - i);
130                 if (m.find(randomized) == m.end()) {
131                     magic_indexes.push_back(randomized);
132                     m[randomized] = max_size - 1;
133                 } else {
134                     magic_indexes.push_back(m[randomized]);
135                     m[randomized] = max_size - 1;
136                 }
137             }
138             for (const auto& i : magic_indexes) {
139                 pattern[i] = '?';
140             }
141         }
142
143         if (with_naive) {
144             auto naive_exec = [&text, &pattern]() { naiveSearch(
    text, pattern); };
145             v_naive.push_back({pattern_size, process(naive_exec)});
146         }
147
148         auto kmp_exec = [&text, &pattern]() { kmp(text, pattern);};
149         v_kmp.push_back({pattern_size, process(kmp_exec)});
150
```

```
151         auto kmp_exec_special = [&text, &pattern]() { kmp(text,
    pattern, true);};
152         v_kmp_special.push_back({pattern_size, process(
    kmp_exec_special)});
153     }
154     std::vector<std::vector<Point>> vv = {v_kmp, v_kmp_special};
155     if (with_naive) {
156         vv.push_back(v_naive);
157     }
158     insertToFile(vv, "text_size_" + std::to_string(text_size) +
159                                                 "_alphabet_size_"
    + std::to_string(alphabet_size) +
160                                                 "_magic_symbols_"
    +
161                                                 std::to_string(
    magic_symbols_amount) + "__" + suffix + ".txt");
162 }
163
164 int main() {
165     srand(time(NULL));
166
167     // 2 * 2 * 5 = 20 in total
168     // input data as it was requested
169     int count = 0;
170     int now;
171     int start = time(NULL);
172     for (const auto& text_size : {10000, 100000}) {
173         for (const auto& alphabet_size : {2, 4}) {
174             for (int i = 0; i < 5; ++i) {
175                 doTest(text_size, alphabet_size, i);
176                 count++;
177                 now = time(NULL);
178                 std::cout << (count * 5) << "%  | " << ((now -
    start) / 60) << " min from start\n";
179             }
180         }
181     }
182     std::cout << "requested data processed\n";
183
184     count = 0;
185     start = time(NULL);
186     // data is too small for kmp algorithms - need special tests
187     for (const auto& text_size : {1000000}) {
188         for (const auto& alphabet_size : {2, 4}) {
189             for (int i = 0; i < 5; ++i) {
190                 doTest(text_size, alphabet_size, i, false,
    text_size / 50, text_size / 2, "kmp");
191                 count++;
192                 now = time(NULL);
193                 std::cout << (count * 10) << "%  | " << ((now -
    start) / 60) << " min from start\n";
194             }
195         }
196     }
```

```
197 }
```

Листинг 1: Тесты

```
1
2 file_names = []
3 text_sizes = [10000, 100000, 1000000]
4 alphabet_sizes = [2, 4]
5 prefixes = ["", "__kmp"]
6 magic_symbols = list(range(5))
7
8 for text_size in text_sizes:
9   for alphabet_size in alphabet_sizes:
10    for magic_symbol in magic_symbols:
11      for prefix in prefixes:
12        file_names.append('text_size_' + str(text_size) +
13                          '_alphabet_size_' + str(alphabet_size) +
14                          '_magic_symbols_' + str(magic_symbol) +
    prefix +
15                          '.txt')
16
17
18 import os.path
19 graph_data = {}
20 for k, file_name in enumerate(file_names):
21   if os.path.isfile(file_name):
22     with open(file_name, 'r') as file:
23       print(file_name)
24       text = file.read()
25       lines = text.split('|')
26       lines = [[dot.split(',') for dot in line.split(';')] for line
    in lines]
27       graph_data[file_name] = lines
28
29
30 from matplotlib import pyplot as plt
31 import numpy as np
32
33
34 def extract_data_from_name(name):
35   paths = name.split('_')
36   return {
37       'text_size': paths[2],
38       'alphabet_size': paths[5],
39       'magic_symbols': paths[8] if len(paths) > 9 else paths[8][:-
    len('.txt')],
40       'suffix': '' if len(paths) == 9 else ''
41   }
42
43 def create_title(data):
44   return f"Razmer teksta: {data['text_size']}. Razmer alphavita: {
    data['alphabet_size']}. Kolichestvo simvolov podstanovki: {data
    ['magic_symbols']}."
45
46
47 for k, v in filter(lambda name: not name[0].endswith('kmp.txt'),
```

```
    graph_data.items()):
  XX = [[dot[0] for dot in line] for line in v]
  YY = [[dot[1] for dot in line] for line in v]
  plot_data = list(zip(XX, YY))
  naive = plot_data[0]

  title_data = extract_data_from_name(k)
  ax = plt.subplot()
  ax.plot(np.asarray(naive[0], int), np.asarray(naive[1], int))
  ax.set_title(create_title(title_data) + " Naivniy algoritm.")
  ax.set_xlabel('Pattern length')
  plt.show()



counter = 0
kmp_names = ['KMP', 'KMP s utochnennimi granyami']
for k, v in filter(lambda data: len(data[0]) == len('
    text_size_1000000_alphabet_size_4_magic_symbols_4__kmp.txt'),
    graph_data.items()):

  XX = [[dot[0] for dot in line] for line in v]
  YY = [[dot[1] for dot in line] for line in v]
  plot_data = list(zip(XX, YY))
  for i, graph in enumerate(plot_data):
    XX_asarr = np.asarray(graph[0], int)
    YY_asarr = np.asarray(graph[1], int)

    ax = plt.subplot()
    ax.plot(XX_asarr, YY_asarr)

    ax.set_xlabel('Pattern length')
    plt.ylim([0, YY_asarr[-1] * 1.05])

    title_data = extract_data_from_name(k)
    ax.set_title(create_title(title_data) + ' ' + kmp_names[i])


    p = np.polyfit(XX_asarr, YY_asarr, 1)

    ya = np.polyval(p, XX_asarr)

    plt.plot(XX_asarr, ya, '--', color='r')

    plt.show()
```

Листинг 2: Построение графиков