Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по Лабораторной работе №7

по курсу «Анализы Алгоритмов»

на тему: «Поиск подстроки в строке»

Студент группы ИУ7-56Б _____ (Подпись, дата) Мамврийский И. С. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Строганов Д. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

В программах, предназначенных для редактирования текста, часто необходимо найти все фрагменты текста, совпадающие с заданным образцом. Обычно, текст — это редактируемый документ, а образец — искомое слово, введённое пользователем.

Целью данной лабораторной работы является исследование алгоритмов поиска подстроки в строке.

Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать стандратный алгоритм и алгоритм Бойера-Мура для поиска подстроки в строке;

— разработать программное обеспечение, реализующее описанные алгоритмы;

— выполнить замеры процессорного времени работы алгоритмов;

— провести сравнительный анализ по времени работы реализаций алгоритмов.

1 Аналитическая часть

В данной части будут рассмотрены алгоритмы поиска подстроки в строке.

1.1 Стандартный алгоритм

Стандартный алгоритм начинает работу со сравнения первого символа текста с первым символом подстроки. Если они совпадают, то происходит переход ко второму символу текста и подстроки. При совпадении сравниваются следующие символы. Так продолжается до тех пор, пока не окажется, что подстрока целиком совпала с отрезком текста, или пока не встретятся несовпадающие символы. В первом случае задача решена, во втором мы сдвигаем указатель текущего положения в тексте на один символ и заново начинаем сравнение с подстрокой [1].

1.2 Алгоритм Бойера-Мура

Алгоритм Бойера-Мура осуществляет сравнение с образцом справа налево, а не слева направо. Исследуя искомый образец, можно осуществлять более эффективные прыжки в тексте при обнаружении несовпадения. В этом алгоритме кроме таблицы суффиксов применяется таблица стоп-символов. Она заполняется для каждого символа в алфавите. Для каждого встречающегося в подстроке символа таблица заполняется по принципу максимальной позиции символа в строке, за исключением последнего символа. При определении сдвига при очередном несовпадении строк, выбирается максимальное значение из таблицы суффиксов и стоп-символов [1].

Вывод

В данном разделе были рассмотрены основные алгоритмы поиска подстроки в строке.

2 Конструкторская часть

В данном разделе будут рассмотрены основные требования к программному обеспечению и схемы алгоритмов.

2.1 Требования к программному обеспечению

Программа должна предоставлять следующие возможности:

— возможность ввода строки и подстроки;

— вывод результата поиска подстроки в строке.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема стандартного алгоритма поиска подстроки в строке. На рисунках 2.2 – 2.5 показаны схемы алгоритма Бойера-Мура и дополнительных функций необходимых для поиска подстроки в строке.

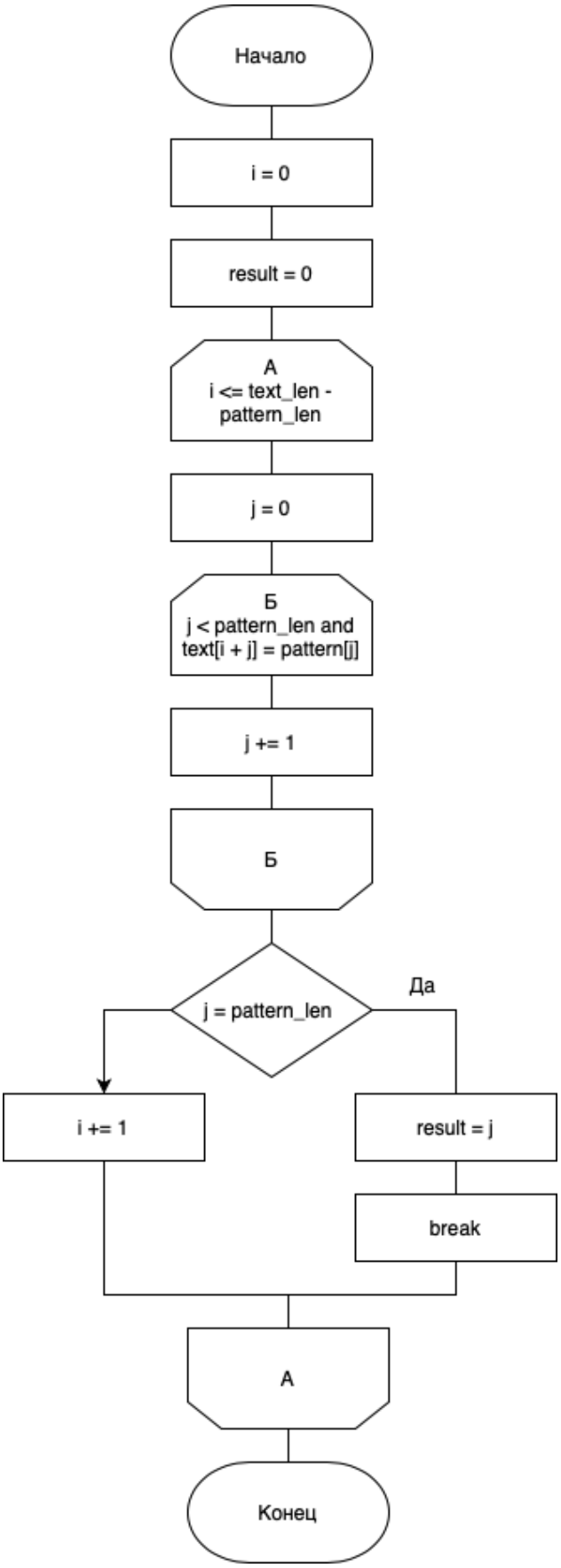

Рисунок 2.1 – Схема стандартного алгоритма поиска подстроки в строке

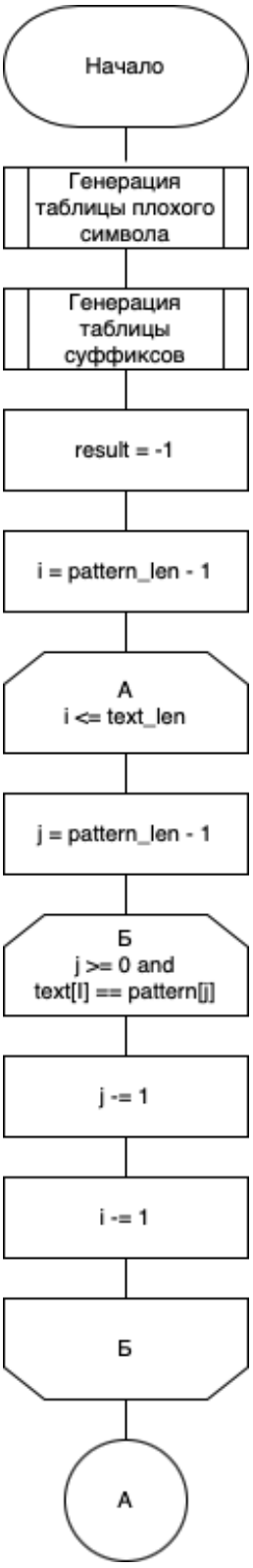

Рисунок 2.2 – Схема алгоритма Бойера-Мура

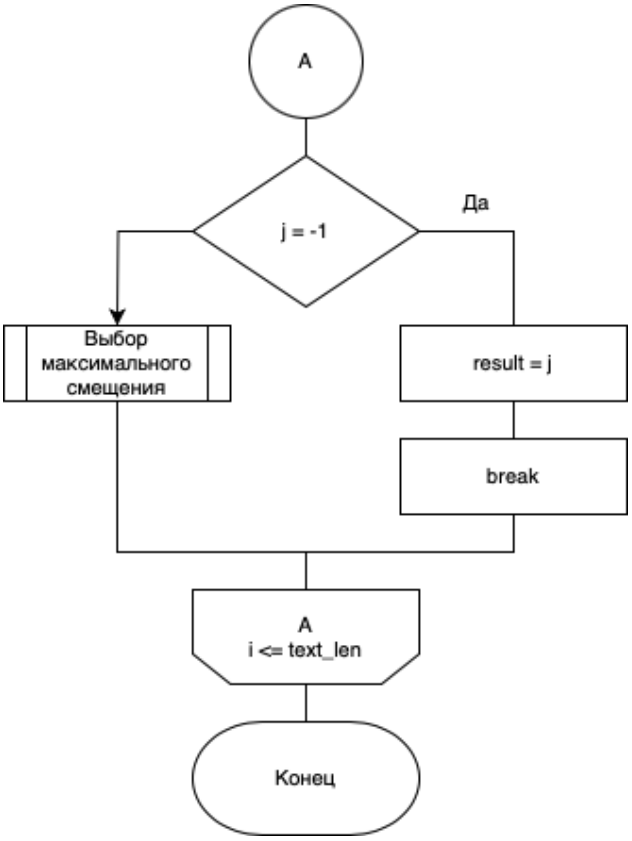

Рисунок 2.3 – Схема алгоритма Бойера-Мура

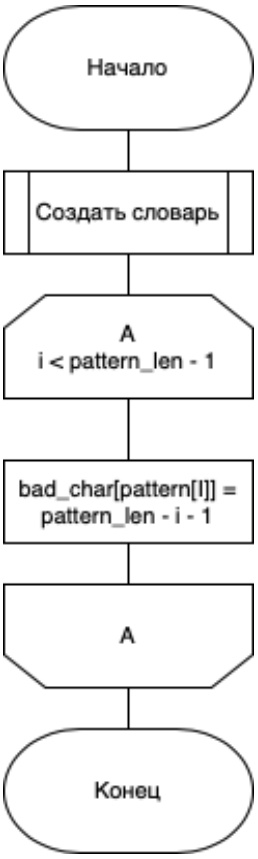

Рисунок 2.4 – Схема создания таблицы плохого символа

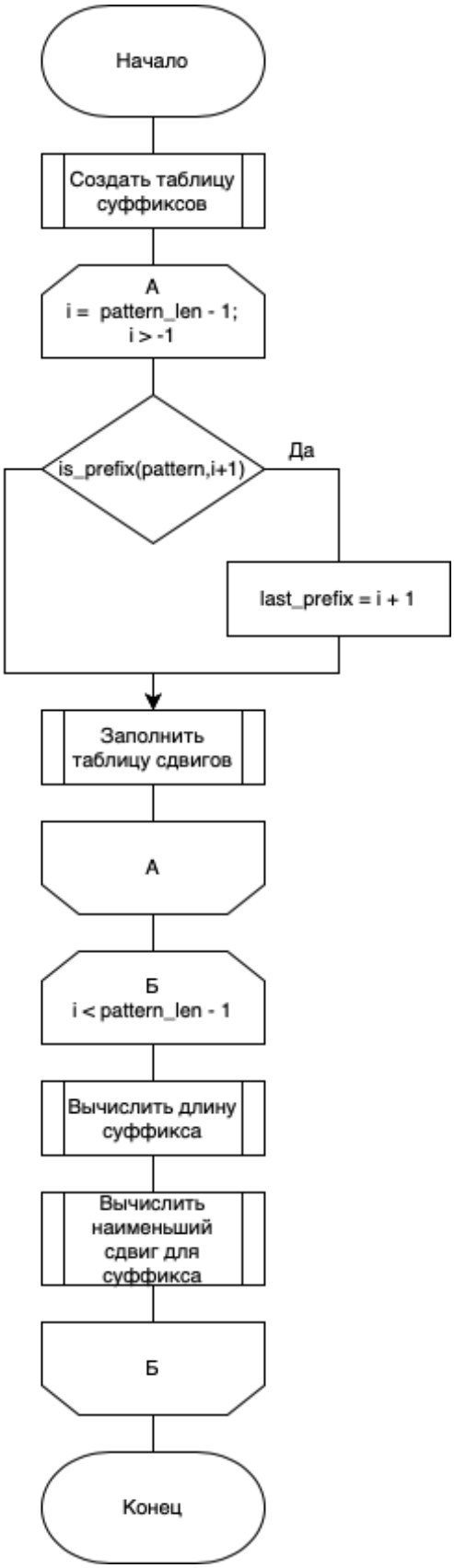

Рисунок 2.5 – Схема создания таблицы суффиксов

2.3 Псевдокоды рассматриваемых алгоритмов

В листинге 2.1 представлен псевдокод стандартного алгоритма поиска подстроки в строке. В листингах 3.2 – 2.4 показаны псевдокоды алгоритма Бойера-Мура и дополнительных функций необходимых для поиска подстроки в строке.

Листинг 2.1 – Псевдокод стандартного алгоритма поиска подстроки в строке

```
1  i <- 0
2  result <- -1
3  text_len <- Длина строки
4  pattern_len <- Длина подстроки
5  До тех пор пока i <= text_len - pattern_len выполнять
6      j <- 0
7
8      До тех пор пока j < pattern_len и text[i + j] == pattern[j]
           выполнять
9          j <- j + 1
10     Конец цикла
11
12     Если j == pattern_len тогда
13         result <- i
14         Выйти из цикла
15     Конец условия
16
17     i <- i + 1
18 Конец цикла
19
20 Возвратить result
```

Листинг 2.2 – Псевдокод функции поиска подстроки в строке в алгоритме Бойера-Мура

```
1  text_len <- Длина строки
2  pattern_len <- Длина подстроки
3
4  bad_char_table <- Сгенерировать таблицу стоп символа
5  good_suffix_table <- Сгенерировать таблицу суффиксов
```

```
6
7  i <- pattern_len - 1
8
9  result <- -1
10 До тех пор пока i < text_len выполнять
11     j <- pattern_len - 1
12
13     До тех пор пока j >= 0 и text[i] == pattern[j] выполнять
14         i <- i - 1
15         j <- j - 1
16     Конец цикла
17
18     Если j == -1 тогда
19         result <- i + 1
20         Выйти из цикла
21     Конец условия
22
23     bad_char_shift <- Найти смещение в таблице стоп символа
24     good_suffix_shift <- Найти смещение в таблице суффиксов
25
26     i <-i + max(bad_char_shift, good_suffix_shift)
27 Конец цикла
28
29 Возвратить result
```

Листинг 2.3 – Псевдокод создания таблицы суффиксов

```
1  pattern_len <- длина подстроки
2  suffix_table <- массив равный длине подстроки
3  last_prefix_position <- pattern_le
4
5  i <- pattern_len
6  До тех пор пока i > -1 выполнять
7      Если подстрока является префиксом тогда
8          last_prefix_position <- i + 1
9      Конец условия
10     suffix_table[pattern_len - 1 - i] <- last_prefix_position -
           i + pattern_len - 1
11 Конец цикла
12
13 i <- 0
14 До тех пока i < pattern_len - 1 выполнять
```

```
15      j <- получить длину суффикса
16      suffix_table[j] = min(suffix_table[j], pattern_len - 1 - i
            + j)
17
18  Возвратить suffix_table
```

Листинг 2.4 – Псевдокод создания таблицы стоп-символа

```
1  bad_char_table <- создать словарь
2  pattern_len = <- длина подстроки
3
4  До тех пор пока i < pattern_len - 1 выполнять
5      bad_char_table[pattern[i]] <- pattern_len - 1 - i
6
7  Возвратить bad_char_table
```

Вывод

В данном разделе были разработаны алгоритм стандартного поиска подстроки в строке и алгоритм Бойера-Мура, также были выдвинуты требования к программному обеспечению.

3 Технологическая часть

В данном разделе описаны средства реализации, приведены листинги кода и данные, на которых будет проводиться тестирование.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python* [2]. Выбор обусловлен наличием опыта работы с ним.

Замеры времени будут происходить с помощью функции *process_time* из библиотеки *time* [3].

3.2 Реализация алгоритмов

В листинге 3.1 представлен алгоритм стандартного поиска подстроки в строке, а в листингах 3.2 – алгоритм Бойера-Мура и дополнительные к нему функции.

Листинг 3.1 – Стандартный алгоритм поиска подстроки в строки

```
1  def standart_alg(text, pattern):
2      text_len = len(text)
3      pattern_len = len(pattern)
4
5      if pattern_len == 0:
6              return "Подстрока пустая"
7
8      if pattern_len > text_len:
9          return "Подстрока больше, чем строка"
10
11     i = 0
12     result = -1
13     while i <= text_len - pattern_len:
14         j = 0
15
16         while j < pattern_len and text[i + j] == pattern[j]:
17             j += 1
18
```

```
19          if j == pattern_len:
20              result = i
21              break
22
23          i += 1
24
25      return result
```

Листинг 3.2 – Алгоритм Бойера-Мура для поиска подстроки в строке

```
1  def generate_bad_char_table(pattern):
2      bad_char_table = dict()
3      pattern_len = len(pattern)
4
5      for i in range(pattern_len - 1):
6          bad_char_table[pattern[i]] = pattern_len - 1 - i
7
8      return bad_char_table
9
10 def generate_good_suffix_table(pattern):
11     pattern_len = len(pattern)
12     suffix_table = [0] * pattern_len
13     last_prefix_position = pattern_len
14
15     for i in range(pattern_len - 1, -1, -1):
16         if is_prefix(pattern, i + 1):
17             last_prefix_position = i + 1
18         suffix_table[pattern_len - 1 - i] =
               last_prefix_position - i + pattern_len - 1
19
20     for i in range(pattern_len - 1):
21         j = get_suffix_length(pattern, i)
22         suffix_table[j] = min(suffix_table[j], pattern_len - 1
               - i + j)
23
24     return suffix_table
25
26 def is_prefix(pattern, p):
27     pattern_len = len(pattern)
28     j = 0
29
30     for i in range(p, pattern_len):
```

```
31          if pattern[i] != pattern[j]:
32              return False
33          j += 1
34
35      return True
36
37  def get_suffix_length(pattern, p):
38      length = 0
39      pattern_len = len(pattern)
40      j = pattern_len - 1
41
42      while p >= 0 and pattern[p] == pattern[j]:
43          length += 1
44          p -= 1
45          j -= 1
46
47      return length
48
49  def boyer_moore_search(text, pattern):
50      text_len = len(text)
51      pattern_len = len(pattern)
52
53      if pattern_len == 0:
54          return "Подстрока пустая"
55
56      if pattern_len > text_len:
57          return "Подстрока больше, чем строка"
58
59      bad_char_table = generate_bad_char_table(pattern)
60      good_suffix_table = generate_good_suffix_table(pattern)
61
62      i = pattern_len - 1
63      result = -1
64      while i < text_len:
65          j = pattern_len - 1
66
67          while j >= 0 and text[i] == pattern[j]:
68              i -= 1
69              j -= 1
70
71          if j == -1:
```

```
72                result = i + 1
73                break
74
75            bad_char_shift = bad_char_table.get(text[i],
                  pattern_len)
76            good_suffix_shift = good_suffix_table[pattern_len - 1 -
                  j]
77
78            i += max(bad_char_shift, good_suffix_shift)
79
80        return result
```

3.3 Функциональные тесты

В таблице 3.1 представлены функциональные тесты для стандартного алгоритма и алгоритма Бойера-Мура.

Входными данными являются строка и подстрока. Выходными данными является индекс, с которого начинается подстрока в строке.

Таблица 3.1 – Функциональные тесты

Входная строка	Входная подстрока	Результат	Ожидаемый результат
data	data	0	0
bombambum	bem	-1	-1
aaaaaaaa	bbbb	-1	-1
a	a	0	0
test data	ta	6	6

Алгоритмы, реализованные в данной лабораторной работе, функциональные тесты прошли успешно.

Вывод

В данном разделе были реализованы стандартный алгоритм поиска подстроки в строке и алгоритм Бойера-Мура. Для проверки правильности работы программы были написаны функциональные тесты.

4 Исследовательская часть

В данном разделе будет приведен пример работы реализации программы, проведен сравнительный анализ реализаций алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее:

— процессор – 2 ГЦ 4-ядерный процессор Intel Core i5;

— оперативная память – 16 ГБайт;

— операционная система – macOS Venura 13.5.2.

4.2 Пример работы реализации программы

В листинге 4.1 приведен пример работы реализации программы.

Листинг 4.1 – Демонстрация работы реализации программы

```
ivanmamvriyskiy@MacBook-Pro-Ivan-2 main % python3 main.py
Введите строку: Все будет хорошо!
Введите подстроку: плохо
Подстрока не найдена
Подстрока не найдена

ivanmamvriyskiy@MacBook-Pro-Ivan-2 main % python3 main.py
Введите строку: Все будет хорошо!
Введите подстроку: хорошо
Стандартный алгоритм. Подстрока найдена на позиции 10
Алгоритм Бойера–Мура. Подстрока найдена на позиции 10
```

4.3 Цель исследования

Целью исследования является проведение измерений количества сравнений и времени работы реализаций алгоритмов, необходимых для решения задачи поиска первого вхождения подстроки в строку в лучшем и худшем случае. На основе полученных оценок следует обосновать, какой из двух случаев представляет собой худший вариант: сценарий отсутствия подстроки длиной S или сценарий нахождения её в строке на последних S позициях.

4.4 Ход исследования

Исследование было проведено для двух различных сценариев расположения образца в строке: когда образец находился в конце строки и когда его не было в строке вообще. Замеры времени производились для строк разных размеров: 512, 1024, 2048, 4096 и 8192. Строка состояла из одинаковых элементов русского алфавита. В ходе замеров измерялись время выполнения реализаций алгоритмов и количество сравнений.

4.5 Результаты исследования

На рисунках 4.1 – 4.2 представлены столбчатые диаграммы результатов замера времени для стандартного алгоритма и алгоритма Бойера-Мура. Замеры времени производятся в микросекундах. Также на рисунках 4.3 – 4.4 представлены столбчатые диаграммы результатов замера количества сравнений.

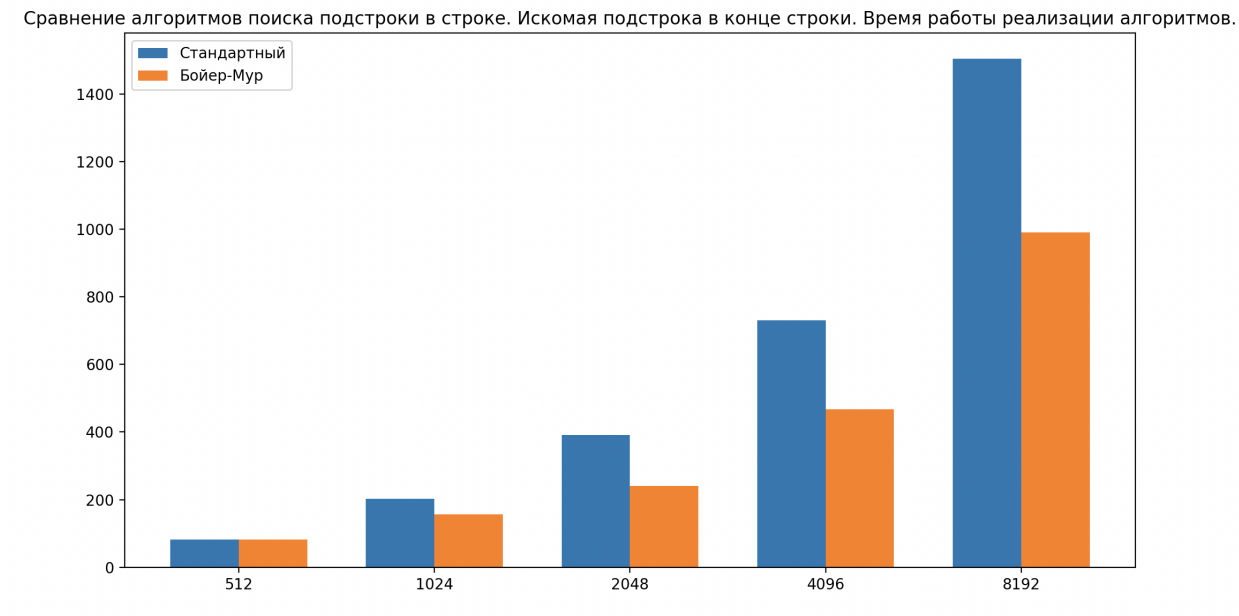

Рисунок 4.1 – Результаты замера времени реализаций алгоритмов в случае, когда искомая подстрока находится в конце строки

Рисунок 4.2 – Результаты замера времени реализаций алгоритмов в случае, когда искомой подстроки нет в строке

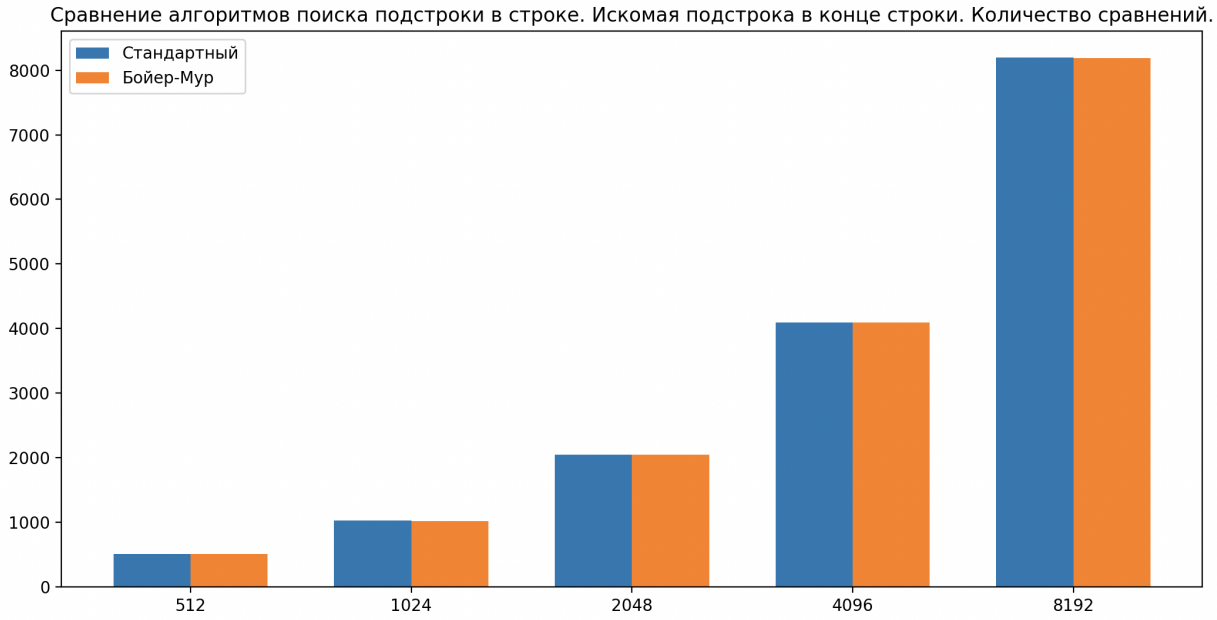

Рисунок 4.3 – Результаты замера количества сравнений в случае, когда искомая подстрока находится в конце строки

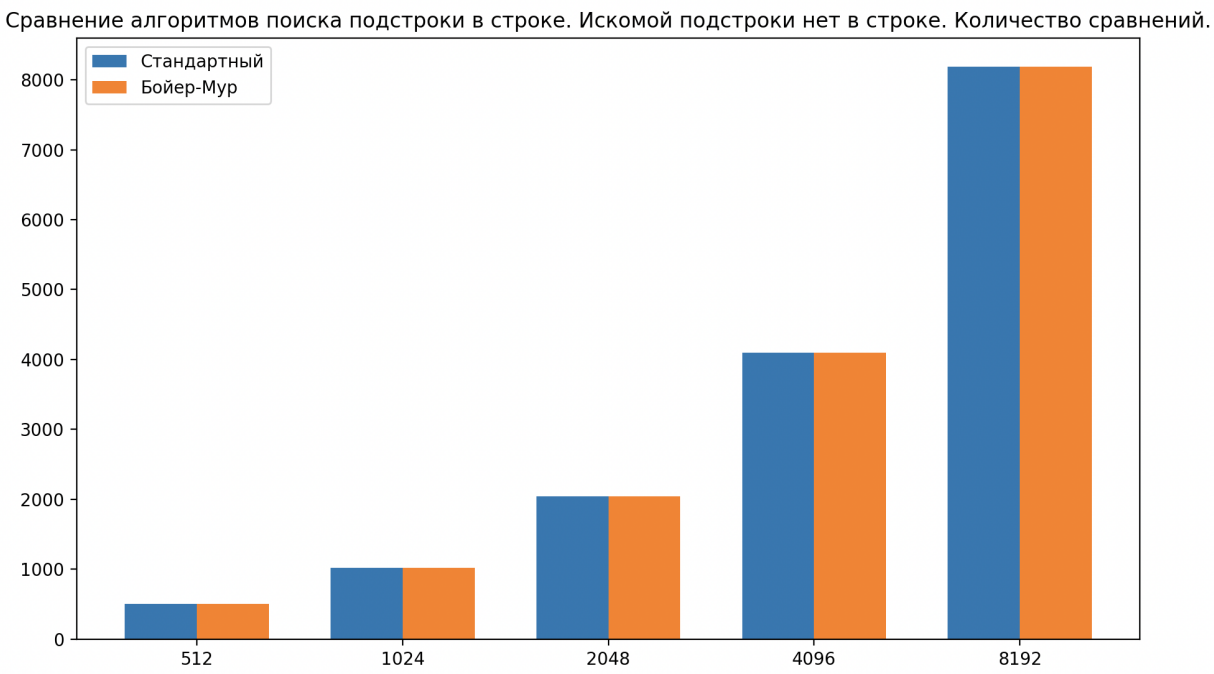

Рисунок 4.4 – Результаты замера количества сравнений в случае, когда искомой подстроки нет в строке

По данным замерам можно сделать следующие выводы:

— худшим случаем по времени выполнения как для реализации стандартного алгоритма, так и для реализации алгоритма Бойера-Мура является случай, когда подстроки нет в строке;

— худшим случаем по количеству сравнений для реализации стандартного алгоритма является случай, когда образец находится в конце строки, а для реализации алгоритма Бойера-Мура худшим является случай, когда подстроки нет в строке.

4.6 Вывод

Таким образом, наихудшим сценарием по количеству сравнений для реализации стандартного алгоритма является тот, в котором подстрока расположена в конце строки, а наихудшим с точки зрения времени — тот, где подстрока отсутствует в строке. В случае реализации алгоритма Бойера-Мура наихудшим сценарием как по количеству сравнений, так и по временным затратам является ситуация, когда подстрока отсутствует в строке.

Заключение

В ходе исследования был проведен сравнительный анализ реализаций стандартного алгоритма и алгоритма Бойера-Мура, по результатам которого можно сделать вывод, что для реализации стандартного алгоритма наихудшим сценарием с точки зрения количества сравнений является ситуация, когда искомая подстрока находится в конце строки, а с учетом затрат времени, худшим случаем является отсутствие подстроки в строке. В отношении реализации алгоритма Бойера-Мура наихудшим сценарием как по количеству сравнений, так и по временным затратам является ситуация, когда искомая подстрока отсутствует в строке.

Цель данной лабораторной работы, которая заключается в исследовании алгоритмов поиска подстроки в строке выполнена, также были выполнена следующие задачи:

— описаны стандартный алгоритм и алгоритм Бойера-Мура для поиска подстроки в строке;

— разработано программное обеспечение, реализующее описанные алгоритмы;

— выполнены замеры процессорного времени работы алгоритма;

— проведен сравнительный анализ по времени работы реализаций алгоритма.

Список используемых источников

1. Дж. Макконнел. Анализ алгоритмов. Активный обучающий подход. Техносфера, 2018.

2. Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 20.12.2023).

3. time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 20.12.2023).