Федеральное государственное бюджетное образовательное учреждение
высшего образования «Московский государственный технический университет
имени Н. Э. Баумана» (национальный исследовательский университет)

Дисциплина: «Анализ алгоритмов»
Отчет по лабораторной работе №7

Тема работы:
«Алгоритмы поиска подстроки в строке»

Студент: Левушкин И. К.
Группа: ИУ7-52Б
Преподаватели: Волкова Л. Л.,
Строганов Ю. В.

Москва, 2019 г.

Содержание

Введение

Поиск подстроки в строке — важная задача поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах и языках программирования.

Такой поиск приходится проводить довольно часто, поэтому необходимо, чтобы он осуществлялся как можно быстрее. Становится ясно, что наивный алгоритм с полным перебором всех частей строки, длины которых соответствуют длине шаблона, является не самым эффективным способом решения такой задачи.

Существует немалое количество алгоритмов, справляющихся с поиском подстроки, лучше, чем примитивный перебор. Особенно красиво реализуются алгоритмы, основанные на применении конечных автоматов.

Цель лабораторной работы: изучение метода динамического программирования на материале алгоритмов поиска подстроки в строке.

Задачи работы:

1) изучить алгоритмы поиска подстроки в строке Кнута-Морриса-Пратта и Бойера-Мура;
2) применить метод динамического программирования для реализации указанных алгоритмов;
3) провести сравнительный анализ алгоритмов по времени выполнения;
4) экспериментально подтвердить различия во временно́й эффективности алгоритмов при помощи разработанного программного обеспечения на материале замеров процессорного времени на строках различных размеров;
5) описать и обосновать полученные результаты в отчете о лабораторной работе, выполненного как расчётно-пояснительная записка.

1 Аналитический раздел

В данном разделе будут рассмотрены выбранные алгоритмы поиска подстроки в строке.

1.1 Описание алгоритмов

1.1.1 Алгоритм Кнута-Морриса-Пратта

Алгоритм Кнута-Морриса-Пратта основан на принципе конечного автомата. В этом алгоритме состояния помечаются символами, совпадение с которыми должно в данный момент произойти. Из каждого состояния имеется два перехода: один соответствует успешному сравнению, другой — несовпадению. Успешное сравнение переводит нас в следующий узел автомата, а в случае несовпадения мы попадаем в предыдущий узел, отвечающий образцу. Пример автомата Кнута-Морриса-Пратта для подстроки *ababcb* приведен на рис. 1. [1]

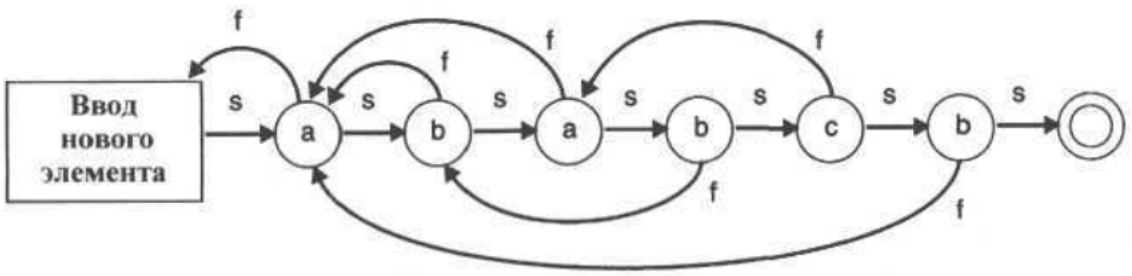

Рис. 1: Полный автомат Кнута-Морриса-Пратта для подстроки *ababcb*

При всяком переходе по успешному сравнению в конечном автомате Кнута-Морриса-Пратта происходит выборка нового символа из текста. Переходы, отвечающие неудачному сравнению, не приводят к выборке нового символа; вместо этого они повторно используют последний выбранный символ. Если мы перешли в конечное состояние, то это означает, что искомая подстрока найдена.

1.1.2 Алгоритм Бойера-Мура

Алгоритм Бойера-Мура осуществляет сравнение с образцом справа налево, а не слева направо, как алгоритм Кнута-Морриса-Пратта. Исследуя искомый образец, можно осуществлять более эффективные прыжки в тексте при обнаружении несовпадения [1].

Рассмотрим строку *there they are* и подстроку *they*. Здесь сначала сравнивается y с r и обнаруживается несовпадение. Поскольку известно, что буква r

вообще не входит в образец, можно сдвинуться в тексте на целых четыре буквы (т. е. на длину образца) вправо. Затем сравнивается буква y с h и вновь обнаруживается несовпадение. Однако поскольку на этот раз p входит в образец, есть возможность сдвинуться вправо только на две буквы так, чтобы буквы h совпали. Затем начинается сравнение справа и обнаруживается полное совпадение кусочка текста с образцом. В алгоритме Бойера—Мура делается 6 сравнений вместо 13 сравнений в примитивном алгоритме, в котором рассматриваются все подстроки исходной строки, совпадающие по длине с шаблоном.

Пример проиллюстрирован в таблице 1.

Таблица 1: Поиск образца they в тексте *there they are*

	t	h	e	r	e		t	h	e	y		a	r	e
1	t	h	e	y										
2					t	h	e	y						
3							t	h	e	y				

Алгоритм Бойера-Мура обрабатывает образец двумя способами. Во-первых, можно вычислить величину возможного сдвига при несовпадении очередного символа. Во-вторых, вычисляется величина прыжка. Эти действия помогают эффективно подстраивать шаблон под новую подстроку.

Выводы

Рассмотрены алгоритмы поиска подстроки в строке, предоставляющие, очевидно, более эффективное решение в сравнении с наивным алгоритмом. Обоснованы различия между приведенными алгоритмами и выделены ключевые моменты каждого из них.

2 Конструкторский раздел

В разделе приводятся схемы выбранных алгоритмов поиска подстроки в строке, производится их теоретический сравнительный анализ.

2.1 Разработка алгоритмов

На рис. 2-4 приведены схемы алгоритмов поиска подстроки.

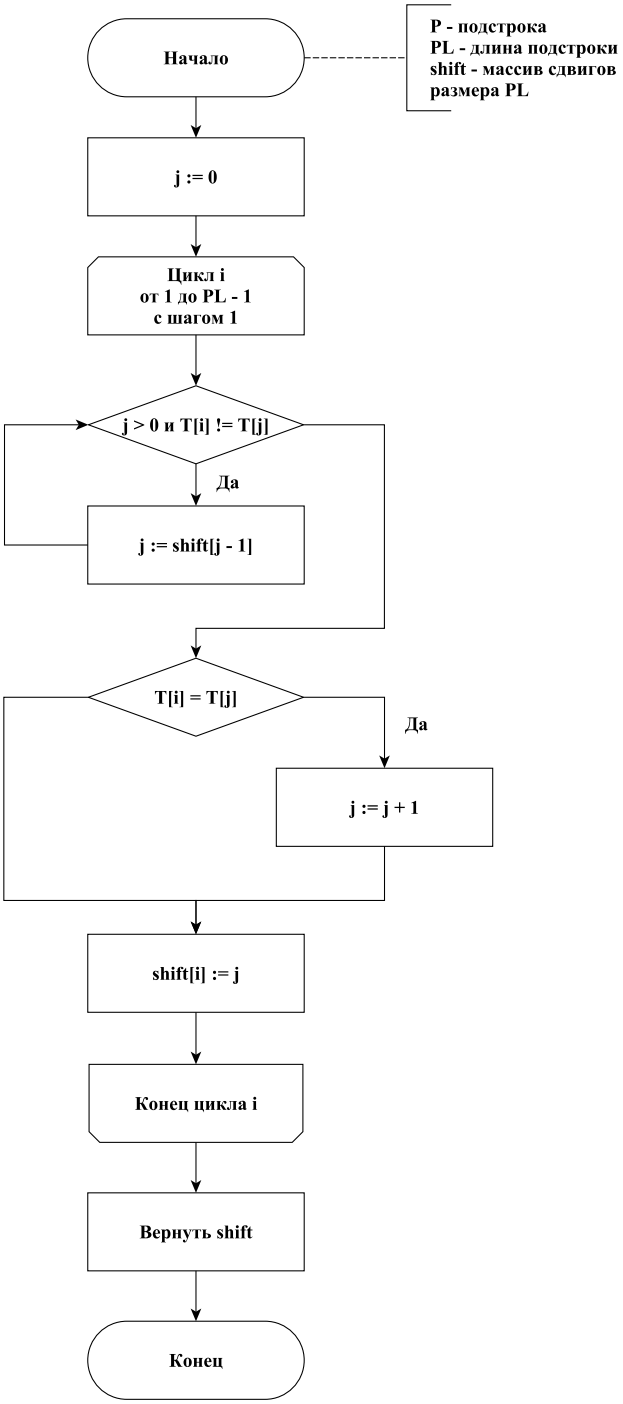

Рис. 2: Заполнение массива сдвигов

2.2 Сравнительный анализ алгоритмов

2.2.1 Пример работы алгоритма Кнута-Морриса-Пратта

Рассмотрим пример работы алгоритмов на строке *ababqcababababakababacqw* длиной 22 символа и подстроке *ababac* длиной 6 символов.

Сперва необходимо построить префикс-функцию для подстроки. Для этого последовательно пройдем по подстроке в поисках очередного максимального суффикса, равного префиксу. Пошаговый разбор представлен в таблице 2: сле-

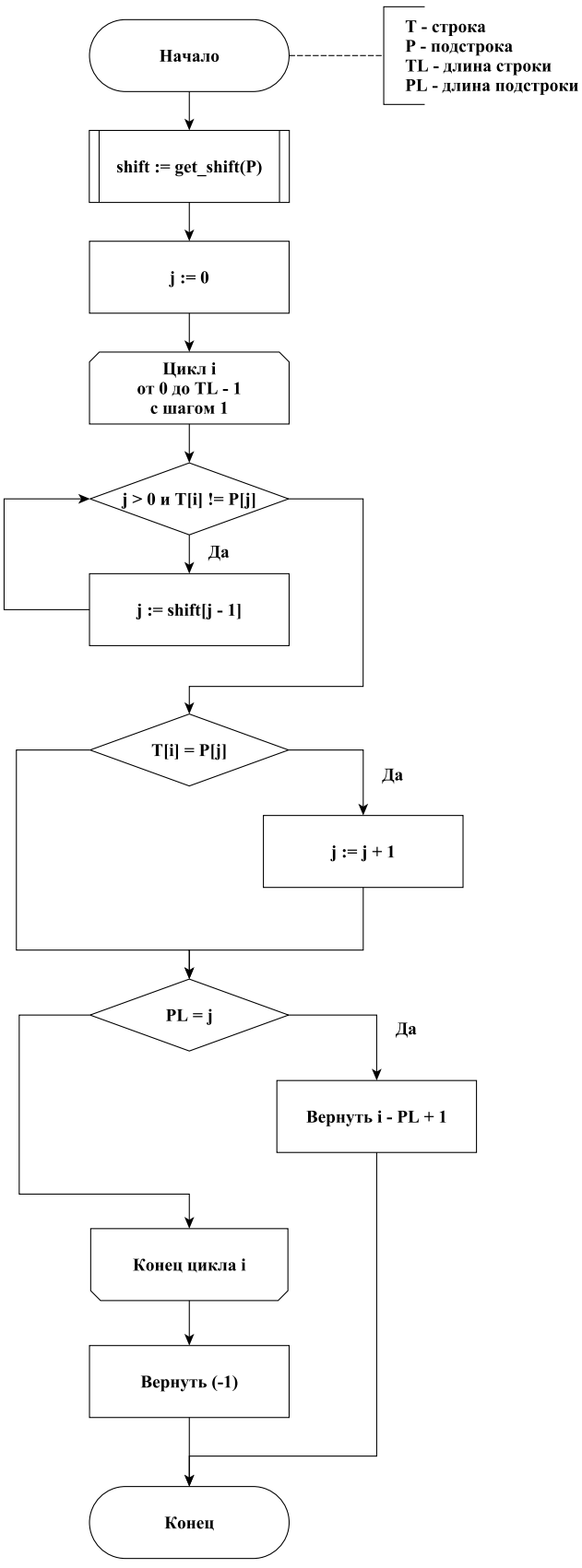

Рис. 3: Алгоритм Кнута-Морриса-Пратта

ва отмечен индекс нового входящего символа, посередине — рассматриваемая строка, справа — значение префикс-функции. *Курсивом* и **жирным** выделены совпадающие максимальные *префиксы* и **суффиксы**.

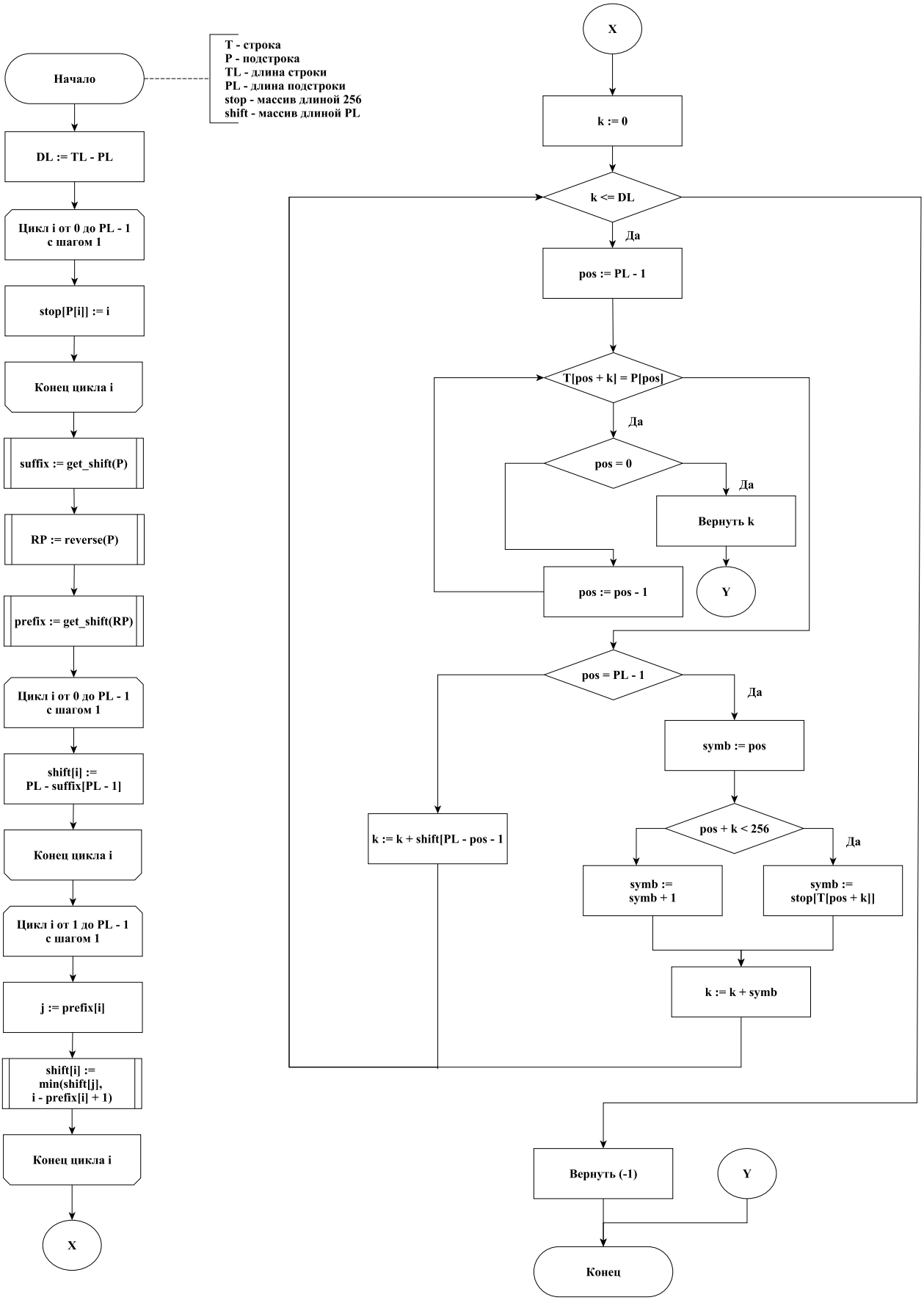

Рис. 4: Алгоритм Бойера-Мура

Далее уже применим похожий подход для поиска подстроки. При несов-

Таблица 2: Нахождение префикс-функции

Индекс	Строка	Префикс-функция
0	a	0
1	a b	0
2	*a* b **a**	1
3	*a* *b* **a** **b**	2
4	*a* *b* **a** **b** **a**	3
5	a b a b a c	0

падении символов, будем смещать подстроку таким образом, чтобы префикс строки (до нового рассматриваемого символа этой строки) был равен некоторому префиксу подстроки (табл. 3).

Таблица 3: Пример работы алгоритма Кнута-Морриса-Пратта

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	a	b	a	b	**q**	**c**	a	b	a	b	a	**b**	a	**k**	a	b	a	b	a	c	q	w
1	**a**	**b**	**a**	**b**	**a**	c																
2						**a**	b	a	b	a	c											
3							**a**	**b**	**a**	**b**	**a**	**c**										
4									**a**	**b**	**a**	**b**	**a**	**c**								
5															**a**	**b**	**a**	**b**	**a**	**c**		

1) несовпадение происходит на символе с индексом 4, подходящего префикса нет, поэтому следующее сравнение начинаем с 5 символа;

2) аналогичная ситуация;

3) несовпадение на 11 символе, суффикс *abab* равен префиксу подстроки, значение префикс-функции для данной части подстроки равно 2, поэтому имеем соответствующее смещение;

4) аналогичный случай в 1-2 этапах;

5) полное совпадение.

Для нахождения потребовалось 5 сравнений на первом этапе, 1 — на втором, по 6 — на остальных этапах. Итого, 24 сравнения. В наивном алгоритме потребовалось бы 38 сравнений.

2.2.2 Пример работы алгоритма Бойера-Мура

Посмотрим, как работает алгоритм Бойера-Мура на тех же входных данных.

В алгоритме Бойера-Мура необходимо дополнительно найти массив стоп-символов и массив сдвигов. Длина массива стоп-символов равна мощности алфавита, над которым производятся действия. Для каждого уникального символа подстроки вычислим индекс самого последнего вхождения и занесем его в этот массив, для остальных символов поставим значение равное -1. Итак, для a – 4, b – 3, c – 5.

Для каждого возможного суффикса шаблона указываем наименьшую величину, на которую нужно сдвинуть вправо шаблон, чтобы он снова совпал с префиксом и при этом символ, предшествующий этому вхождению, не совпадал бы с символом, предшествующим суффиксу. Если такой сдвиг невозможен, ставится длина подстроки (табл. 4).

Таблица 4: Массив сдвигов в алгоритме Бойера-Мура

Индекс	0	1	2	3	4	5
Строка	a	b	a	b	a	c
Сдвиг	6	2	3	4	5	6

Теперь можно перейти непосредственно к поиску подстроки (табл. 5).

Таблица 5: Пример работы алгоритма Бойера-Мура

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	a	b	a	b	**q**	c	a	**b**	a	**b**	a	**b**	a	**k**	a	b	a	b	a	c	q	w
1	a	b	a	b	**a**	**c**																
2			a	b	a	b	a	**c**														
3					a	b	a	b	a	**c**												
4							a	b	a	b	a	**c**										
5									a	b	a	b	a	**c**								
6															**a**	**b**	**a**	**b**	**a**	**c**		

1) несовпадение происходит на символе с индексом 4, префикс ab совпадает с суффиксом, поэтому смещаемся на 2 в соответствии с массивом сдвигов;

2-5) аналогичная ситуация несовпадения, только смещение происходит по стоп-таблице, так как нет подходящего суффикса;

6) полное совпадение.

Для нахождения потребовалось 2 сравнения на первом этапе, 4 — на последнем этапе, по 1 — на всех остальных. Всего 12 сравнений.

2.2.3 Оценка времени работы алгоритма Кнута-Морриса-Пратта

Рассмотрим сравнение строк на позиции i, где образец $S[0, m-1]$ сопоставляется с частью текста $T[i, i+m-1]$. Предположим, что первое несовпадение

произошло между $T[i+j]$ и $S[j]$, где $1 < j < m$. Тогда $T[i, i+j-1] = S[0, j-1] = P$ и $a = T[i+j] \neq S[j] = b$.

При сдвиге вполне можно ожидать, что префикс (начальные символы) образца S сойдется с каким-нибудь суффиксом (конечные символы) текста P. Длина наиболее длинного префикса, являющегося одновременно суффиксом, есть значение префикс-функции от строки S для индекса j.

Это приводит нас к следующему алгоритму: пусть $\pi[j]$ – значение префикс-функции от строки $S[0, m-1]$ для индекса j. Тогда после сдвига мы можем возобновить сравнения с места $T[i+j]$ и $S[\pi[j]]$ без потери возможного местонахождения образца. Можно показать, что таблица π может быть вычислена (амортизационно) за $\mathbf{O(m)}$ сравнений перед началом поиска. А поскольку строка T будет пройдена ровно один раз, суммарное время работы алгоритма будет равно $\mathbf{O(m+n)}$, где n – длина текста T. [2]

2.2.4 Оценка времени работы алгоритма Бойера-Мура

Ниже мы обозначаем через $\mathbf{A}$ — число символов в алфавите.

При подсчете массива сдвигов по одному присваиванию приходится на каждый элемент массива и еще по одному – на каждый символ образца. Поэтому общее число присваиваний равно $\mathbf{O(m+A)}$. При вычислении массива прыжков в худшем случае каждый символ образца будет сравниваться со всеми последующими его символами.

Можно подсчитать, что число сравнений в худшем случае будет $\mathbf{O(mn)}$. В лучшем случае асимптотика равна $\mathbf{O(m+n)}$. [6]

Выводы

В разделе представлены схемы алгоритмов поиска подстроки в строке Кнута-Морриса-Пратта и Бойера-Мура. Произведен теоретический анализ временной сложности, обозначены различия в работе алгоритмов, пояснена их эффективность.

3 Технологический раздел

Здесь описываются требования к программному обеспечению и средства реализации, приводятся листинги программы и тестовые данные.

3.1 Требования к программному обеспечению

Входные данные:

- *text* — строка, в которой ведется поиск;
- *pattern* — искомая подстрока.

Выходные данные: индекс первого вхождения подстроки *pattern* в строке *text*; при отсутствии подстроки либо неверных данных (длина подстроки превышает длину строки) — сообщение об этом.

3.2 Средства реализации

Программа написана на языке Python [9], который предоставляет программисту мощные инструменты для реализации различных алгоритмов и является достаточно надежным, эффективным и удобным для реализации сложных алгоритмов. Для написания использовался редактор исходного кода *PyCharm* [10].

Замер времени выполнения программы производится с помощью функции *process_time()* из библиотеки *time*, функционал которой позволяет подсчитывать процессорное время в тиках, а затем конвертировать полученный результат в секунды.

3.3 Листинг программы

Реализованная программа представлена в листингах 1, 2 и 3.

Листинг 1: Реализация заполнения массива сдвигов

```
def get_fail(substring):
  fail = [0 for i in range(len(substring))]
  fail[0] = -1
  for i in range(1, len(substring)):
    temp = fail[i - 1]
    while ((substring[temp] != substring[i - 1]) and (temp > -1)):
      temp = fail[temp]
    fail[i] = temp + 1
  return fail
```

Листинг 2: Реализация алгоритма Кнута-Морриса-Пратта

```
def knut_moris_prat(text, substring):
  subLoc = 0
  textLoc = 0

  fail = get_fail(substring)

```

```
  while (textLoc < len(text) and subLoc < len(substring)):
    if (subLoc == -1 or text[textLoc] == substring[subLoc]):
      textLoc += 1
      subLoc += 1
    else:
      subLoc = fail[subLoc]

  if (subLoc > len(substring) - 1):
    return textLoc - len(substring)
  else:
    return -1
```

Листинг 3: Реализация алгоритма Бойера-Мура

```
def create_dict(pattern):
  d = {}
  len_pattern = len(pattern)
  for i in pattern:
    if (d.get(i) is None):
      d.update({i : len_pattern})

  return d


def get_slide_mas(pattern):
  slide = create_dict(pattern)

  len_pattern = len(pattern)

  for i in range(len(pattern)):
    slide.update({pattern[i] : len_pattern - i - 1})

  slide.update({pattern[len(pattern) - 1] : len_pattern})

  return slide


def get_jump_mas(pattern):

  jump = [0 for i in range(len(pattern))]

  for i in range(len(pattern)):
    jump[i] = 2 * len(pattern) - i - 1

  test = len(pattern) - 1
  target = len(pattern)

  link = [0 for i in range(len(pattern))]

  while (test > -1):
    link[test] = target
    while (target < len(pattern) and (pattern[test] != pattern[target])):
      jump[target] = min(jump[target], len(pattern) - 1 - test)
      target = link[target]
    test -= 1
    target -= 1
```

```
43
44    for i in range(target + 1):
45      jump[i] = min(jump[i], len(pattern) + target - i)
46
47    temp = link[target]
48    while (target < len(pattern) - 1):
49      while (target <= temp):
50        jump[target] = min(jump[target], temp - target + len(pattern))
51        target += 1
52      temp = link[temp]
53
54    return jump
55
56
57  def boyer_mur(text, pattern):
58    textLoc = len(pattern) - 1
59    patternLoc = len(pattern) - 1
60
61    slide = get_slide_mas(pattern)
62    jump = get_jump_mas(pattern)
63
64    while (textLoc < len(text) and (patternLoc > -1)):
65      if (text[textLoc] == pattern[patternLoc]):
66        textLoc -= 1
67        patternLoc -= 1
68      else:
69        if (slide.get(text[textLoc]) is None):
70          textLoc += max(len(pattern), jump[patternLoc])
71        else:
72          textLoc += max(slide.get(text[textLoc]), jump[patternLoc])
73        patternLoc = len(pattern) - 1
74
75    if (patternLoc == -1):
76      return textLoc + 1
77    else:
78      return -1
```

3.4 Тестовые данные

Для тестирования выделены следующие случаи, представленные в табл. 6.

Выводы

В данном разделе были рассмотрены требования к программному обеспечению, обоснован выбор средств реализации, приведены листинги программы и тестовые данные.

Таблица 6: Тестовые данные

№	Строка	Подстрока	Результат
1	a	abc	Неверная длина
2	abc	abcd	Неверная длина
3	abc	def	Не найдено
4	abc	aa	Не найдено
5	abbaabbab	aaab	Не найдено
6	abc	a	0
7	abbaabbab	abba	0
8	abc	b	1
9	abbaabbab	baab	2
10	there they are	they	6
11	abc	c	2
12	abbaabbab	bbab	5
13	there they are	are	11
14	abcabc	a	0
15	abcabc	c	2
16	abbaabbabbbab	bbab	5
17	there they are are here are	are	11
18	a	a	0
19	abc	abc	0
20	abbaabbabbbab	abbaabbabbbab	0

4 Исследовательский раздел

В разделе представлены примеры выполнения программы, а также результаты исследования эффективности сортировок.

4.1 Примеры работы

На рис. 5-11 приведены примеры работы программы.

```
Введите строку: abc
Введите подстроку: abcd
Выберете алгоритм:
    1)Кнут-Моррис-Пратт
    2)Бойер-Мур
1
-1
Время выполнения в секундах: 3.5999999999784e-05 seconds
```

Рис. 5: Некорректная длина

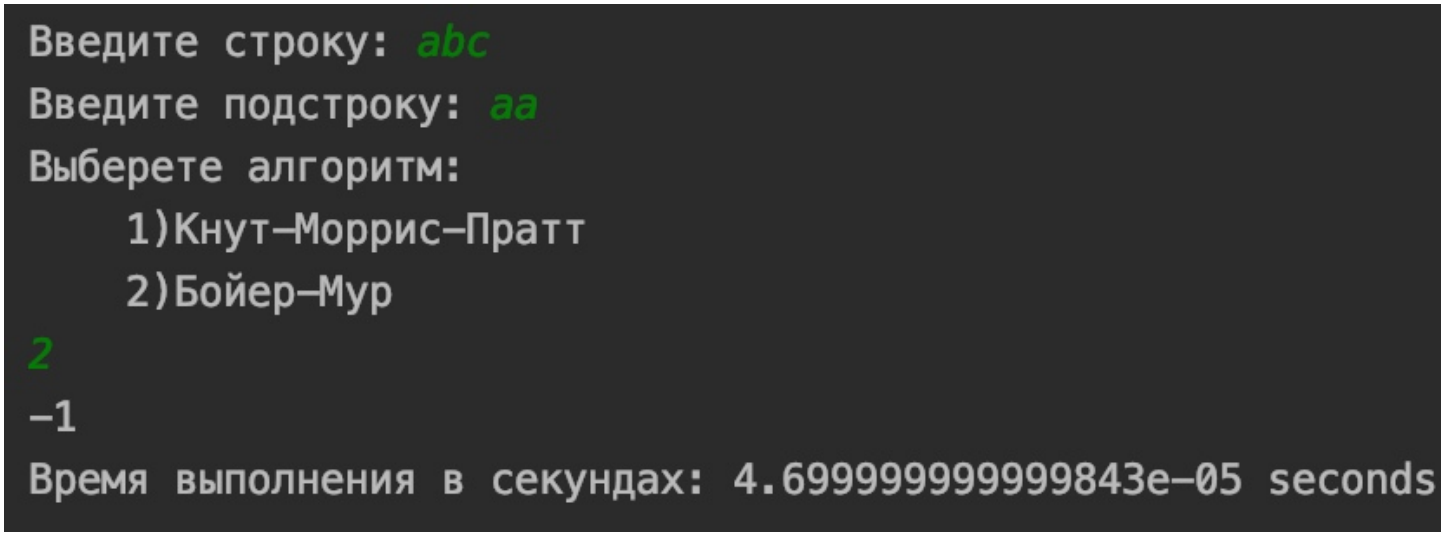

Рис. 6: Подстрока не найдена

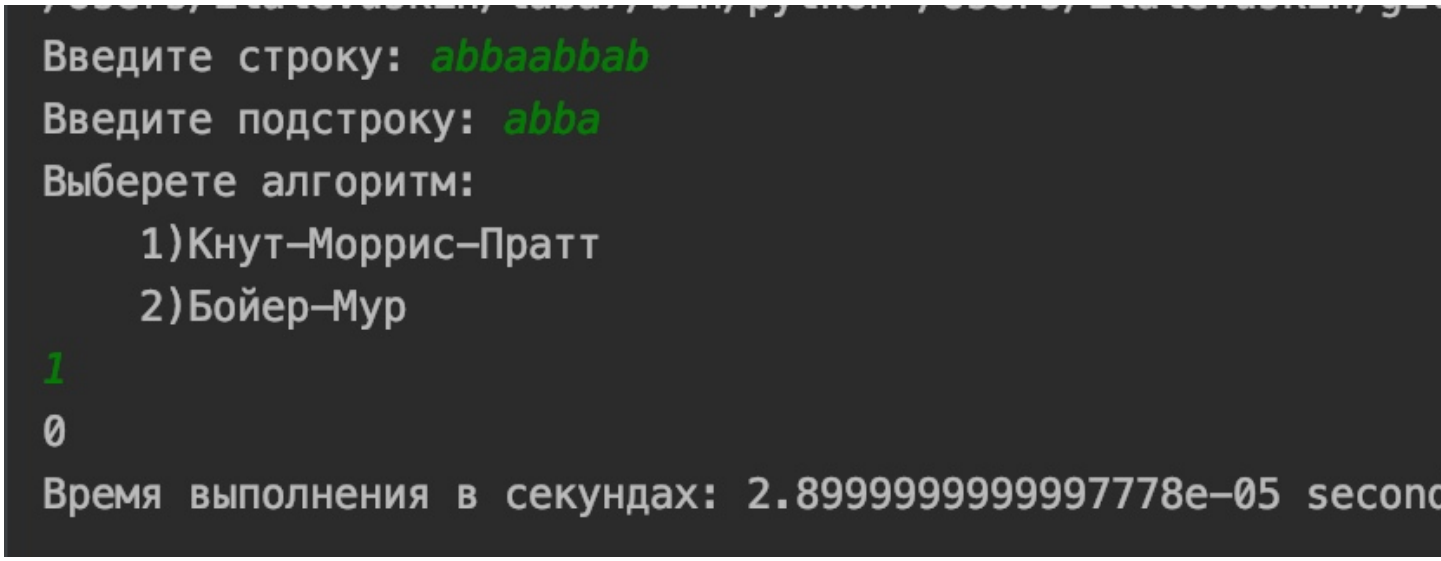

Рис. 7: Подстрока в начале

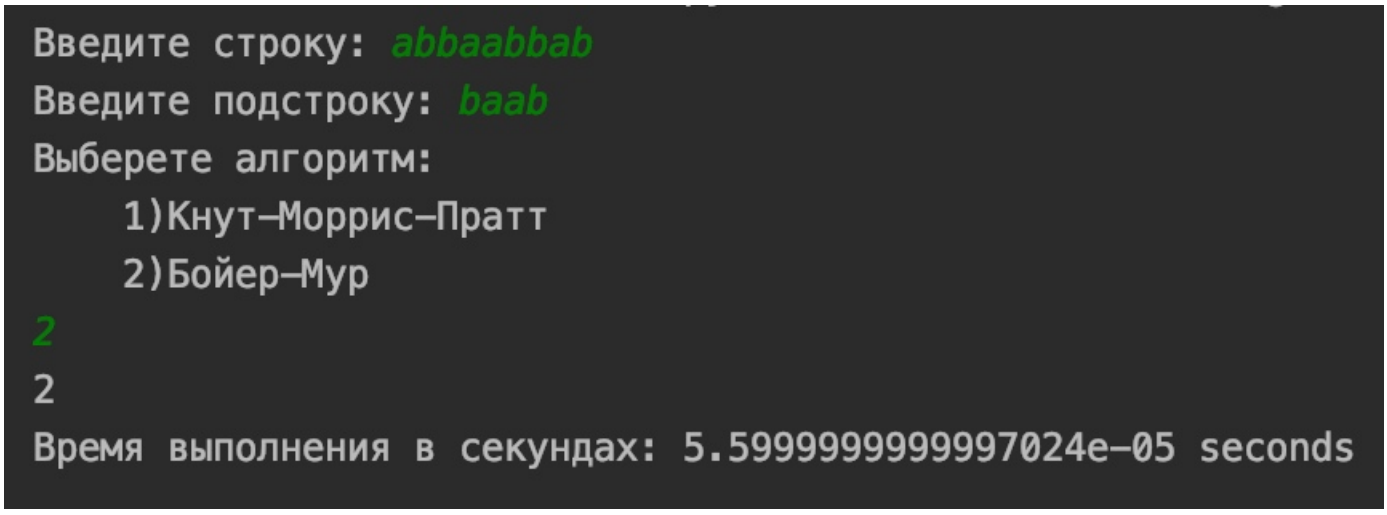

Рис. 8: Подстрока в середине

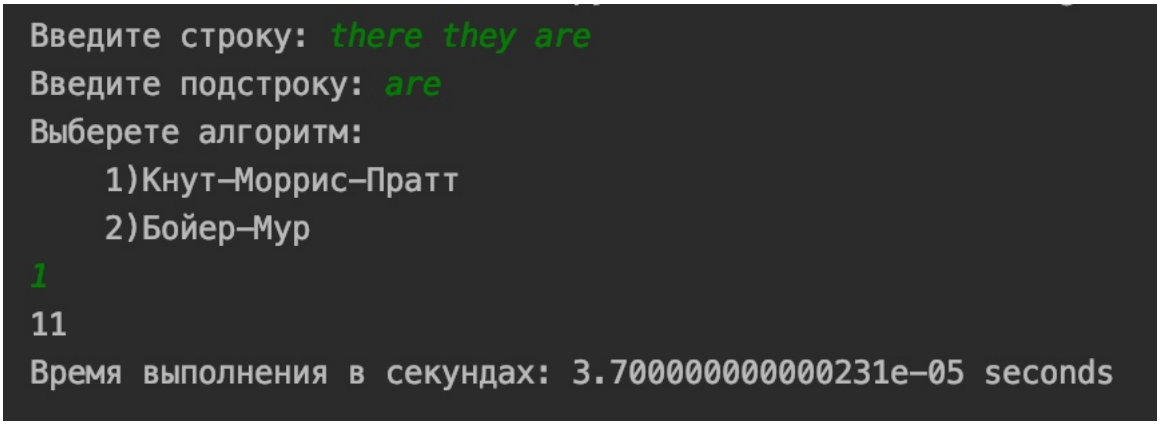

Рис. 9: Подстрока в конце

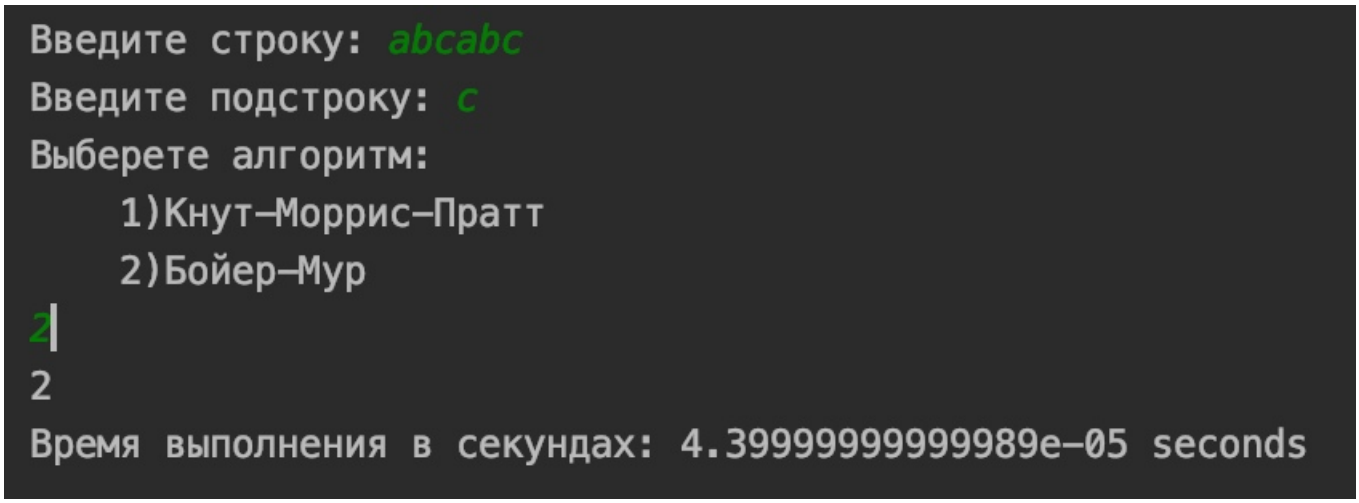

Рис. 10: Повтор подстроки

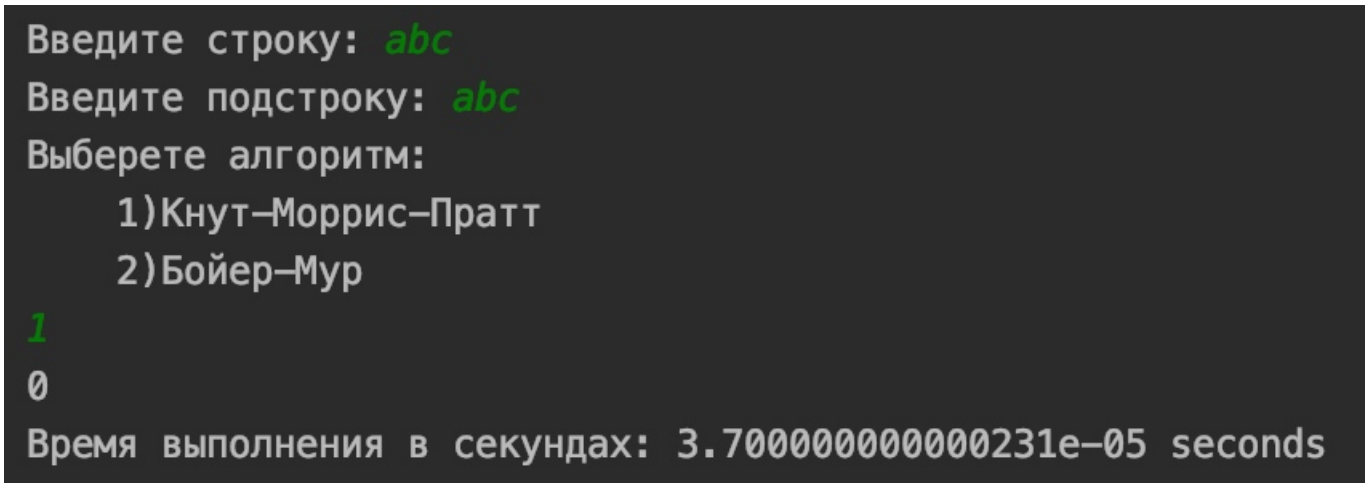

Рис. 11: Строка равна подстроке

4.2 Постановка эксперимента

Необходимо сравнить время работы реализованных алгоритмов поиска подстроки в строке. Размер строки меняется от 100 тыс. до 1 млн. символов, длина шаблона – 100. Строка содержит только символы a, в подстроке символы b и a чередуются (реализуется худший случай для алгоритма Бойера-Мура).

4.3 Сравнительный анализ на основе эксперимента

Замеры произведены на 4-ядерном процессоре *Intel Core i7* с тактовой частотой 2,4 ГГц, оперативная память — 8 ГБ.

На рис. 12 приводятся графики сравнения времени выполнения выбранных алгоритмов.

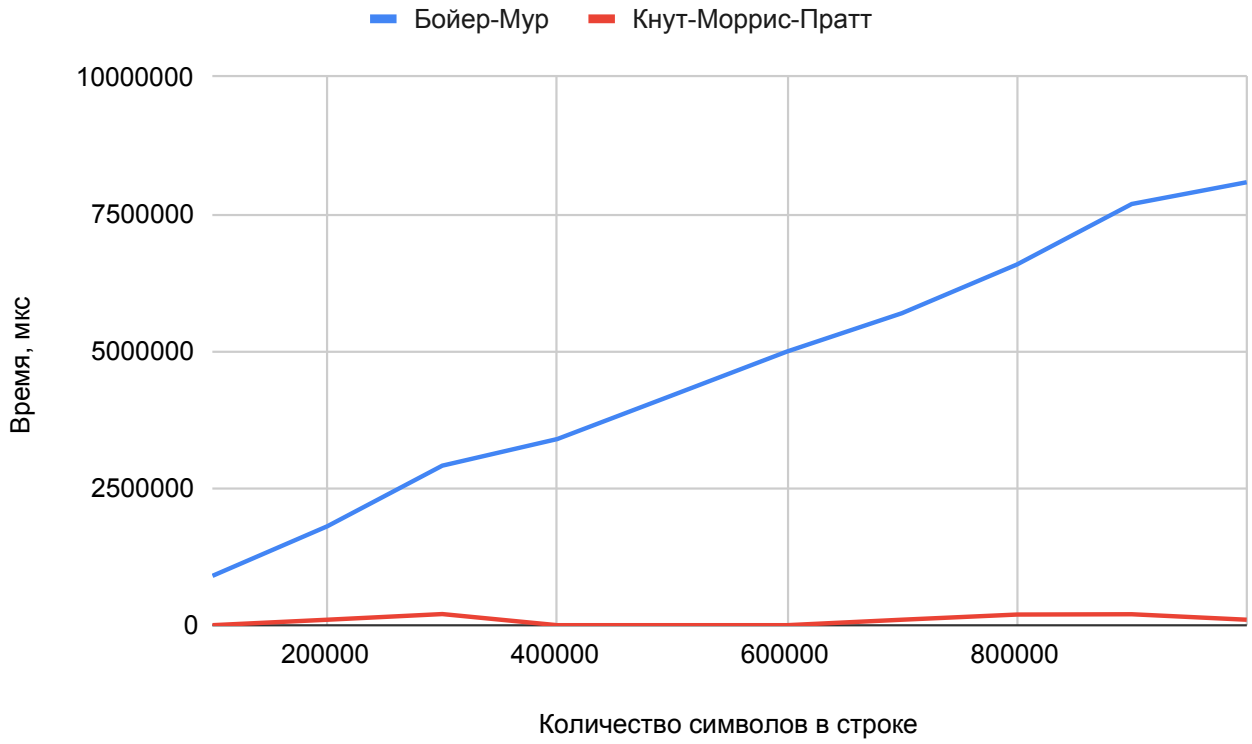

Рис. 12: Сравнение времени выполнения алгоритмов поиска подстроки в строке

Как видно, подтверждены теоретические расчеты. В худшем случае асимптотика алгоритма Бойера-Мура квадратична, а Кнута-Морриса-Пратта – линейна. Из-за того, что все символы a из текста повторяются в шаблоне m раз, эвристика хорошего суффикса будет пытаться сопоставить шаблон в каждой позиции (суммарно, n раз), а эвристика плохого символа в каждой позиции будет двигать строку m раз. Итого, $\mathbf{O(mn)}$ [3].

Выводы

Программа успешно прошла все заявленные тесты. Эксперименты замера времени подтвердили предположения и теоретические расчеты из раздела 2.2.

Заключение

В ходе работы выполнено следующее:

1) изучены алгоритмы поиска подстроки в строке;

2) применен метод динамического программирования для реализации указанных алгоритмов;

3) проведен сравнительный анализ алгоритмов по времени выполнения;

4) экспериментально подтверждены различия во временно́й эффективности алгоритмов при помощи разработанного программного обеспечения на материале замеров процессорного времени на строках различных размеров;

5) описаны и обоснованы полученные результаты в отчете о лабораторной работе, выполненного как расчётно-пояснительная записка.

Список литературы

[1] Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход.- М.:Техносфера, 2009.

[2] Д. Кнут. Искусство программирования, М., Мир, 1978

[3] Алгоритмы поиска в строке [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/111449//, свободный – (26.11.2019)

[4] Поиск подстроки. Алгоритм Кнута–Морриса-Пратта [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/191454/, свободный – (21.11.2019)

[5] Алгоритм Бойера-Мура для поиска по шаблону [Электронный ресурс]. – Режим доступа: http://espressocode.top/boyer-moore-algorithm-for-pattern-searching/, свободный – (26.11.2019)

[6] Алгоритм Бойера-Мура для поиска по шаблону [Электронный ресурс]. – Режим доступа: http://www-igm.univ-mlv.fr/ lecroq/string/node14.html, свободный – (26.11.2019)

[7] ISO/IEC 14882:2017 [Электронный ресурс]. – Режим доступа: https://www.iso.org/standard/68564.html, свободный – (27.11.2019)

[8] <chrono> [Электронный ресурс]. – Режим доступа: http://www.cplusplus.com/reference/chrono/, свободный – (20.11.2019)

[9] Python 3.8.2rc1 documentation [Электронный ресурс]. - Режим доступа: https://docs.python.org/3/, свободный - (28.11.2019)

[10] PyCharm documentation [Электронный ресурс]. - Режим доступа: https://www.jetbrains.com/pycharm/documentation/ - (28.11.2019)