

МГТУ им. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №7

По курсу: "АНАЛИЗ АЛГОРИТМОВ"

## **Поиск подстроки в строке**

Работу выполнил: Гаврилов Дмитрий, ИУ7-56Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

*Москва, 2019*

# Оглавление

<b>Введение</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Общие сведения об алгоритмах поиска подстроки . . . . .	4
1.1.1 Стандартный алгоритм . . . . .	4
1.1.2 Алгоритм Бойера-Мура . . . . .	5
1.1.3 Алгоритм Кнута-Морриса-Пратта . . . . .	5
Вывод . . . . .	6
<b>2 Конструкторская часть</b>	<b>7</b>
2.1 Требования к программе . . . . .	7
2.2 Пример работы алгоритмов . . . . .	7
2.2.1 Алгоритм Кнута-Морриса-Пратта . . . . .	7
2.2.2 Алгоритм Бойера-Мура . . . . .	8
Вывод . . . . .	8
<b>3 Технологическая часть</b>	<b>9</b>
3.1 Выбор ЯП . . . . .	9
3.2 Сведения о модулях программы . . . . .	9
3.3 Листинг кода алгоритмов . . . . .	9
3.4 Тестирование программы . . . . .	12
Вывод . . . . .	13
<b>4 Исследовательская часть</b>	<b>15</b>
4.1 Сравнительный анализ на основе замеров времени . . . . .	15
Вывод . . . . .	16
<b>Заключение</b>	<b>17</b>



# Введение

Цель работы: изучение алгоритмов поиска подстроки в строке.

Задачи данной лабораторной работы:

1. изучить алгоритмы Бойера-Мура и Кнута-Морриса-Пратта;
2. реализовать эти алгоритмы;
3. провести тестирование ПО.

# 1 | Аналитическая часть

В данной части будут рассмотрены алгоритмы поиска подстроки в строке.

## 1.1 Общие сведения об алгоритмах поиска подстроки

Поиск подстроки в строке — одна из простейших задач поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах, языках программирования, программы определения плагиата осуществляют онлайн-проверку, используя алгоритмы поиска подстроки среди большого количества документов, хранящихся в собственной базе[1]. На сегодняшний день существует огромное разнообразие алгоритмов поиска подстроки. Программисту приходится выбирать подходящий в зависимости от таких факторов: длина строки, в которой происходит поиск, необходимость оптимизации, размер алфавита, возможность проиндексировать текст, требуется ли одновременный поиск нескольких строк. В данной лабораторной работе будут рассмотрены два алгоритма сравнения с образцом, алгоритм Кнута-Морриса-Пратта и алгоритм Бойера-Мура.

### 1.1.1 Стандартный алгоритм

Стандартный алгоритм начинает 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 встретят-

ся 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[2].

### 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[2].

### 1.1.3 Алгоритм Кнута-Морриса-Пратта

Алгоритм Кнута-Морриса-Пратта основан на принципе конечного автомата, однако он использует более простой метод обработки неподходящих символов. 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. В программной реализации этого алгоритма применяется массив сдвигов, который создается для каждой подстроки, которая ищется в тексте. Для каждого символа из подстроки рассчитывается значение, равное максимальной длине совпадающего префикса и суффикса относительно конкретного элемента подстроки. Создание этого массива позволяет при несовпадении строки сдвигать ее на расстояние, большее, чем 1 (в отличие от стандартного алгоритма).

## Вывод

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

## 2 | Конструкторская часть

В данном разделе будут рассмотрены основные требования к программе и пошаговая работа алгоритмов.

### 2.1 Требования к программе

**Требования к вводу:** Длина подстроки должна быть больше, чем длина строки.

**Требования к программе:**

- каждая из функций должна выдавать первый индекс вхождения подстроки в строку;
- если строка не содержит подстроку, то функция выдает -1.

### 2.2 Пример работы алгоритмов

В таблице 1 и таблице 2 буде рассмотрена пошаговая работа алгоритмов Кнута-Морриса-Пратта и Бойера-Мура на значениях строки  $s$  и подстроки  $sub$ .

```
string s = "ababacabaa"; string sub = "abaa";
```

#### 2.2.1 Алгоритм Кнута-Морриса-Пратта

Для алгоритма Кнута-Морриса-Пратта вычисленный массив префиксов для заданой подстроки  $sub$  имеет значение:  $prefix = [0, 0, 1, 1]$

Таблица 1 отображает пошаговую работу алгоритма Кнута-Морриса-Пратта при данном массиве префиксов.



Таблица 1. Пошаговая работа алгоритма Кнута-Морриса-Пратта.

a	b	a	b	a	c	a	b	a	a
a	b	a	a						
		a	b	a	a				
				a	b	a	a		
					a	b	a	a	
						a	b	a	a

### 2.2.2 Алгоритм Бойера-Мура

Для алгоритма Бойера-Мура вычисленный массив суффиксов для заданной подстроки sub имеет значение:  $\text{suffix} = [2, 5, 5, 6]$ . Переходы алфавита для подстроки sub:  $\text{letters} = ['a' = 0, 'b' = 2]$  Если буквы нет в letters, будет считаться, что переход равен длине sub.

Таблица 2. Пошаговая работа алгоритма Бойера-Мура.

a	b	a	b	a	c	a	b	a	a
a	b	a	a						
		a	b	a	a				
						a	b	a	a

## Вывод

В данном разделе были рассмотрены основные требования к программе, разобрана работа алгоритмов на конкретной строке и подстроке.

## 3 | Технологическая часть

Замеры времени были произведены на: Intel(R) Core(TM) i5-8300H, 4 ядра, 8 логических процессоров.

### 3.1 Выбор ЯП

В качестве языка программирования был выбран Java [3]. Средой разработки Visual Studio. Время работы алгоритмов было замерено с помощью системного вызова `System.currentTimeMillis()`. Тестирование было реализовано с помощью стандартного шаблона модульных тестов[4].

### 3.2 Сведения о модулях программы

Программа состоит из:

- `Main.java` - главный файл программы, в котором располагается точка входа в программу
- `StrMatching.java` - функции поиска подстроки
- `SubstrTest.java` - проект для тестирования программы

### 3.3 Листинг кода алгоритмов

В этой части будут рассмотрены листинги кода (листинг 3.1 - 3.5) реализованных алгоритмов.

Листинг 3.1: Стандартная функция

```

1 public static int standard(String str, String substr) {
2     for (int i = 0; i <= str.length() - substr.length(); i++) {
3         boolean correct = true;
4         for (int j = 0; j < substr.length() && correct; j++) {
5             if (str.charAt(i + j) != substr.charAt(j))
6                 correct = false;
7         }
8         if (correct)
9             return i;
10    }
11    return -1;
12 }

```

Листинг 3.2: Алгоритм КМП

```

1 public static int KMP(String str, String substr) {
2     int[] prefix = prefixFunction(substr);
3     int last_prefix = 0;
4     for (int i = 0; i < str.length(); i++) {
5         while (last_prefix > 0 && substr.charAt(
6             last_prefix) != str.charAt(i))
7             last_prefix = prefix[last_prefix - 1];
8
9         if (substr.charAt(last_prefix) == str.charAt(i))
10            last_prefix++;
11
12        if (last_prefix == substr.length()) {
13            return i + 1 - substr.length();
14        }
15    }
16    return -1;
17 }

```

Листинг 3.3: Функция нахождения массива сдвигов

```

1 private static int[] prefixFunction(String substr) {

```

```

2      int[] prefix = new int[substr.length()];
3
4      int lastPrefix = prefix[0] = 0;
5      for (int i = 1; i < substr.length(); i++) {
6          while (lastPrefix > 0 && substr.charAt(
7              lastPrefix) != substr.charAt(i))
8              lastPrefix = prefix[lastPrefix - 1];
9
10         if (substr.charAt(lastPrefix) == substr.charAt(
11             i))
12             lastPrefix++;
13
14         prefix[i] = lastPrefix;
15     }
16     return prefix;
17 }

```

Листинг 3.4: Алгоритм Бойера-Мура

```

1 public static int BM(String str, String substr) {
2     if (substr.length() == 0)
3         return -1;
4
5     Map<Character, Integer> letters = new HashMap<
6         Character, Integer>();
7
8     for (int i = 0; i < substr.length(); i++) {
9         letters.putIfAbsent(substr.charAt(i), substr.
10             length() - 1 - i);
11     }
12
13     int[] suffix = getSuffix(substr);
14
15     for (int i = substr.length() - 1; i < str.length();
16         ) {
17         int j = substr.length() - 1;
18         while (substr.charAt(j) == str.charAt(i)) {
19             if (j == 0)
20                 return i;
21             j--;
22             i--;
23         }
24     }
25 }

```

```

20     }
21     int a = letters.containsKey(str.charAt(i)) ?
        letters.get(str.charAt(i)) : substr.length();
22     ;
23     int b = suffix[substr.length() - 1 - j];
24     i += Math.max(a, b);
25 }
26 return -1;
}

```

Листинг 3.5: Функция вычисления сдвигов суффиксов

```

1 private static int[] getSuffix(String substr) {
2     int[] table = new int[substr.length()];
3     int lastPrefixPosition = substr.length();
4
5     for (int i = substr.length() - 1; i >= 0; i--) {
6         if (isPrefix(substr, i + 1))
7             lastPrefixPosition = i + 1;
8         table[substr.length() - 1 - i] =
9             lastPrefixPosition - i + substr.length() -
10                1;
11     }
12
13     for (int i = 0; i < substr.length() - 1; i++) {
14         int slen = getSuffixLength(substr, i);
15         table[slen] = substr.length() - 1 - i + slen;
16     }
17     return table;
}

```

## 3.4 Тестирование программы

В этой части будет рассмотрен листинг функций тестирования алгоритмов (листинг 3.6 -3.7).

Листинг 3.6: Тестирование функции случайными значениями

```

1 @Test

```

```

2      public void testStandardRandom() {
3          for (int i = 0; i < N; i++) {
4              String s = Character.toString(random.nextInt
5                  (999999) + 1000);
6              String sub = Character.toString(random.nextInt
7                  (999) + 100);
8
9              int correctIndex = s.indexOf(sub);
10             int resultIndex = StrMatching.standard(s, sub);
11             Assert.assertEquals("str: " + s + " sub: " +
                sub, correctIndex, resultIndex);
            }
        }

```

Листинг 3.7: Тестирование функции случайными значениями одной длины

```

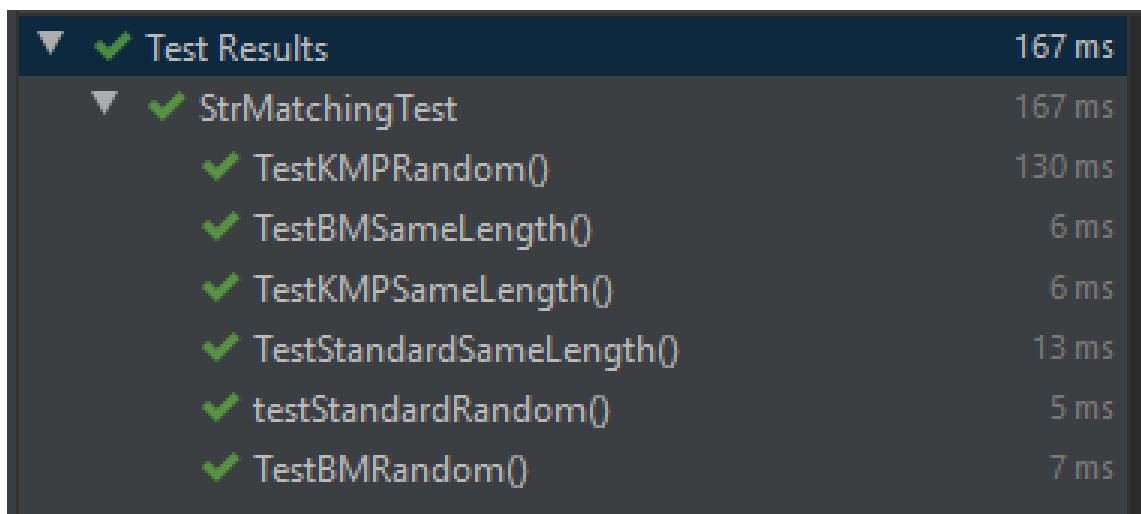
1  @Test
2      public void testStandardRandom() {
3          for (int i = 0; i < N; i++) {
4              String s = Character.toString(random.nextInt
5                  (999999) + 1000);
6              String sub = Character.toString(random.nextInt
7                  (999) + 100);
8
9              int correctIndex = s.indexOf(sub);
10             int resultIndex = StrMatching.standard(s, sub);
11             Assert.assertEquals("str: " + s + " sub: " +
                sub, correctIndex, resultIndex);
            }
        }

```

На рис. 3.1 предоставлен скриншот результатов работы тестов, на котором видно, что алгоритмы работают правильно.

## Вывод

В данном разделе были рассмотрены основные сведения о модулях программы, листинг кода алгоритмов и тестов, результаты тестирования, которое показало, что алгоритмы реализованы корректно.



▼	✓	Test Results	167 ms
▼	✓	StrMatchingTest	167 ms
	✓	TestKMPRandom()	130 ms
	✓	TestBMSameLength()	6 ms
	✓	TestKMPSameLength()	6 ms
	✓	TestStandardSameLength()	13 ms
	✓	testStandardRandom()	5 ms
	✓	TestBMRandom()	7 ms

Рис. 3.1: Результат работы тестов

## 4 | Исследовательская часть

В данном разделе будет проведен временной анализ работы алгоритмов.

### 4.1 Сравнительный анализ на основе замеров времени

Был проведен замер времени работы алгоритмов при разных размерах строки и фиксированном размере подстроки.

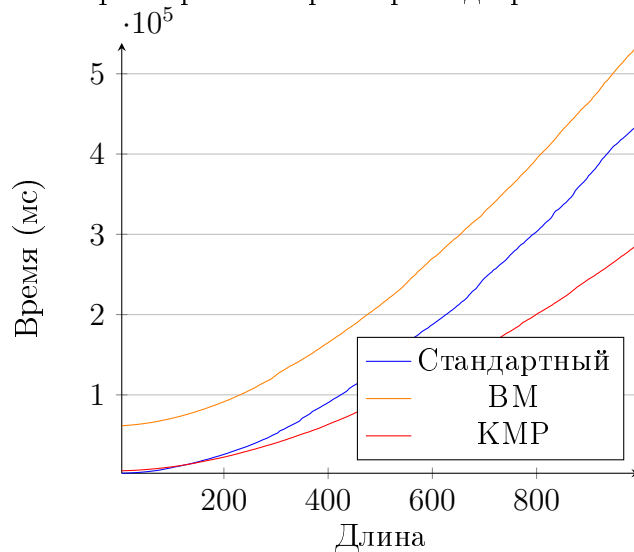


Рис. 4.1: Сравнение времени работы алгоритмов при увеличении длины строки. Длина подстроки 3

На рисунке 4.1 видно, что моя реализация алгоритма Бойера-Мура проигрывает стандартному и Кнута-Морриса-Пратта. Это происходит



потому что в моей реализации Бойера-Мура используется словарь и в каждом цикле происходит проверка наличия ключа в словаре, из-за чего и замедляется работа.

## Вывод

Сравнительный анализ по времени показал, что внедрение словаря сильно замедляет алгоритм Бойера-Мура.

# Заключение

В ходе лабораторной работы я изучила возможности применения и реализовала алгоритмы поиска подстроки в строке.

Было проведено тестирование, показавшее, что алгоритмы реализованы правильно.

Временной анализ показал, что неэффективно использовать структуру словаря для реализации алгоритма Бойера-Мура.

# Литература

- [1] Окулов С. М. Алгоритмы обработки строк. — М.: Бином, 2013. — 255 с.
- [2] Дж. Макконнелл. Анализ лгоритмов. Активный обучающий подход
- [3] Руководство по языку C#[Электронный ресурс], - режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/csharp/>
- [4] Основные сведения о модульных тестах [Электронный ресурс], - режим доступа: <https://docs.microsoft.com/ru-ru/visualstudio/test/unit-test-basics?view=vs-2019>