



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 (ИУ7)

НАПРАВЛЕНИЕ ПОДГОТОВКИ 09.03.04 Программная инженерия

## ОТЧЕТ

по лабораторной работе № 1

Название: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-51Б  
(Группа)

О.А. Тюрин  
(Подпись, дата) (И.О. Фамилия)

Преподаватель

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

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Описание расстояний и алгоритмов . . . . .	5
1.1.1 Расстояние Левенштейна, рекурсивное определение расстояния . . . .	5
1.1.2 Расстояние Левенштейна, матричное определение расстояния . . . .	5
1.1.3 Расстояние Левенштейна, рекурсивное матричное определение рас- стояния . . . . .	6
1.1.4 Алгоритм поиска расстояния Дамерау-Левенштейна . . . . .	6
1.2 Вывод . . . . .	6
<b>2 Конструкторская часть</b>	<b>7</b>
2.1 Разработка алгоритмов . . . . .	8
2.2 Вывод . . . . .	11
<b>3 Технологическая часть</b>	<b>12</b>
3.1 Требования к программному обеспечению . . . . .	12
3.2 Средства реализации . . . . .	12
3.3 Листинг кода . . . . .	12
3.3.1 Рекурсивный алгоритм Левенштейна . . . . .	12
3.3.2 Матричный алгоритм Левенштейна . . . . .	13
3.3.3 Рекурсивный матричный алгоритм Левенштейна . . . . .	13
3.3.4 Алгоритм Дамерау-Левенштейна . . . . .	14
3.4 Сравнительный анализ матричной и рекурсивной реализаций . . . . .	14
3.4.1 Теоретический анализ затрачиваемой памяти . . . . .	14
3.5 Интерфейс программы . . . . .	15
3.6 Тестирование . . . . .	16
3.7 Вывод . . . . .	18
<b>4 Экспериментальная часть</b>	<b>19</b>
4.1 Примеры работы . . . . .	19
4.2 Постановка эксперимента по замеру времени . . . . .	20
4.3 Сравнительный анализ на материале экспериментальных данных . . . . .	21
4.4 Вывод . . . . .	21
<b>Заключение</b>	<b>22</b>

# Введение

Расстояние Левенштейна (редакционное расстояние) - это минимальное кол-во редакторских операций, которое необходимо для превращения одной строки в другую.

Существуют следующие применения редакционных расстояний.

- 1) Поисковики (Google), автоисправление текста.
- 2) Биоинформатика.

Задачи для данной ЛР:

- 1) изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- 2) получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
- 3) сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- 4) экспериментальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
- 5) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

# 1 Аналитическая часть

Задача по нахождению расстояния Левенштайна заключается в поиске минимального количества операций с единичным штрафом.

**Вставка (I - Insert)**

**Удаление (D - Delete)**

**Замена (R - Replace)**

**Совпадение (M - Match)**

Для превращения одной строки в другую.

В алгоритме Далмерау-Левенштайна добавляется ещё одна операция:

**Транспозиция(T)**

Все операции, кроме "Совпадения" имеют штраф 1. Операция "Совпадения" имеет штраф 0.

## 1.1 Описание расстояний и алгоритмов

### 1.1.1 Расстояние Левенштейна, рекурсивное определение расстояния

Введём понятие  $D(s_1, s_2)$  = минимальному количеству редакторских операций, с помощью которых строка  $s_1$  преобразуется в строку  $s_2$ . Тогда расстояние Левенштейна можно записать следующим образом:

$$D(i, j) = \begin{cases} 0, i = 0, j = 0 \\ i, j = 0, i > 0 \\ j, i = 0, j > 0 \\ \min(D(S_1[1, \dots, i], S_2[1, \dots, j-1]) + 1, \\ D(S_1[1, \dots, i-1], S_2[1, \dots, j]) + 1, \\ D(S_1[1, \dots, i-1], S_2[1, \dots, j-1]) + \\ \begin{cases} 0, \text{if } S_1[i] = S_2[j], \\ 1, \text{else} \end{cases} \end{cases} \quad \text{Расстояние можно искать рекурсивно}$$

по представленной формуле.

### 1.1.2 Расстояние Левенштейна, матричное определение расстояния

Вводится матрица, размерностью  $[Len(S_1) + 1 \times Len(S_2) + 1]$

Первая строки и столбец матрицы заполняются от 0 до  $Len(S)$  (первые 3 пункта системы из предыдущего пункта).

$$A = \begin{pmatrix} & \emptyset & С & Т & О & Л & Б \\ \emptyset & 0 & 1 & 2 & 3 & 4 & 5 \\ Т & 1 & & & & & \\ Е & 2 & & & & & \\ Л & 3 & & & & & \\ О & 4 & & & & & \end{pmatrix}$$

Далее для нахождения ответа применяется последняя формула из системы, описанной в предыдущем пункте.

$$A = \begin{pmatrix} & \emptyset & С & Т & О & Л & Б \\ \emptyset & 0 & 1 & 2 & 3 & 4 & 5 \\ Т & 1 & 1 & 1 & 2 & 3 & 4 \\ Е & 2 & 2 & 2 & 2 & 3 & 4 \\ Л & 3 & 3 & 3 & 3 & 2 & 3 \\ О & 4 & 4 & 4 & 3 & 3 & \mathbf{3} \end{pmatrix}$$

Ответ в правом нижнем углу.

Чтобы определить, какая именно цепочка преобразований привела к ответу представим матрицу как карту высот: нужно спуститься на санках из клетки с ответом в левый верхний угол. В нашем случае:

**I:** ТЕЛО  $\rightarrow$  СТЕЛО

**M:** T = T

**R:** E  $\rightarrow$  O

**M:** Л = Л

**R:** O  $\rightarrow$  Б

### 1.1.3 Расстояние Левенштейна, рекурсивное матричное определение расстояния

Аналогичен алгоритму из предыдущего пункта с той лишь разницей, что матрица начинает заполнение "с конца". Вычисляем значение ячейки матрицы только в том случае, если значения там ещё нет (аналогично  $\infty$  в алгоритме Дейкстры). Ответ всё так же в правом нижнем углу.

### 1.1.4 Алгоритм поиска расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна вычисляется по следующей формуле:

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min( & \\ \quad D(S_1[i], S_2[j - 1]) + 1, & j > 0 \quad //I \\ \quad D(S_1[i - 1], S_2[j]) + 1, & i > 0 \quad //D \\ \quad D(S_1[i - 1], S_2[j - 1]) + & \\ \quad \begin{cases} 0, & \text{если } S_1[i] == S_2[j], \quad //M \\ 1, & \text{иначе} \quad //R \end{cases} & \\ \quad D(S_1[i - 2], S_2[j - 2]) + 1 & \text{если } i = 1, j = 1, S_1[i] = S_2[j - 1], S_1[i - 1] = S_2[j] \\ ) & \end{cases}$$

## 1.2 Вывод

Были рассмотрены алгоритмы нахождения расстояния Левенштейна и нахождения расстояния Дамерау-Левенштейна. Главное отличие - наличие операции транспозиции.

## 2 Конструкторская часть

### **Требования к вводу:**

- 1) на вход подаются две строки;
- 2) одна и та же буква в разном регистре считается как разный символ.

### **Требования к программе::**

- 1) Две пустые строки являются корректным вводом, который программа должна обрабатывать.

## 2.1 Разработка алгоритмов

В данном разделе представлены схемы реализуемых алгоритмов. Схема рекурсивного алгоритма поиска расстояния Левенштейна представлена на рис. 1

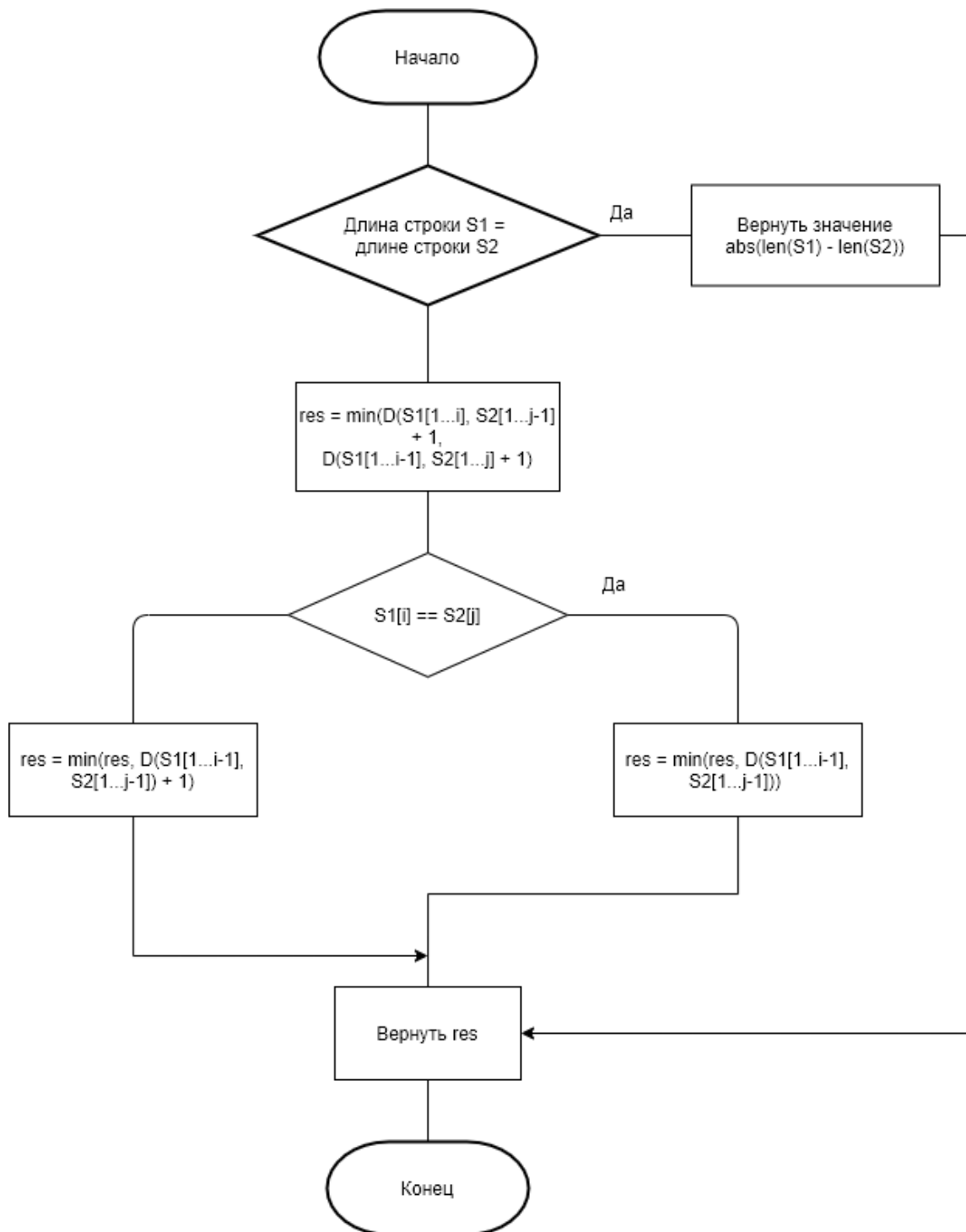


Рис. 1: Схема рекурсивного алгоритма поиска расстояния Левенштейна



Схема матричной реализации алгоритма поиска расстояния Левенштейна представлена на рис. 2

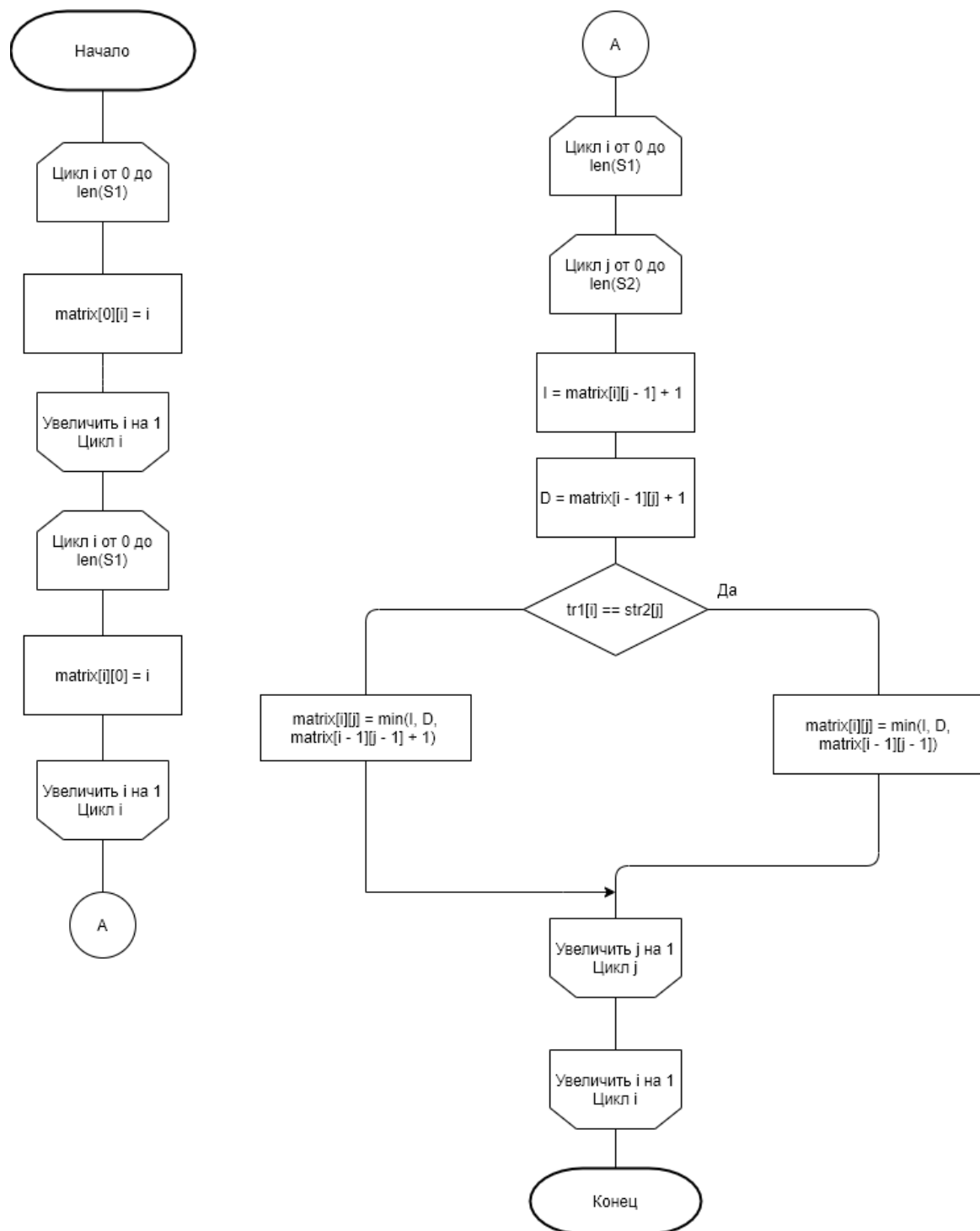


Рис. 2: Схема матричной реализации алгоритма поиска расстояния Левенштейна

Схема рекурсивного матричного алгоритма поиска расстояния Левенштейна представлена на рис. 3

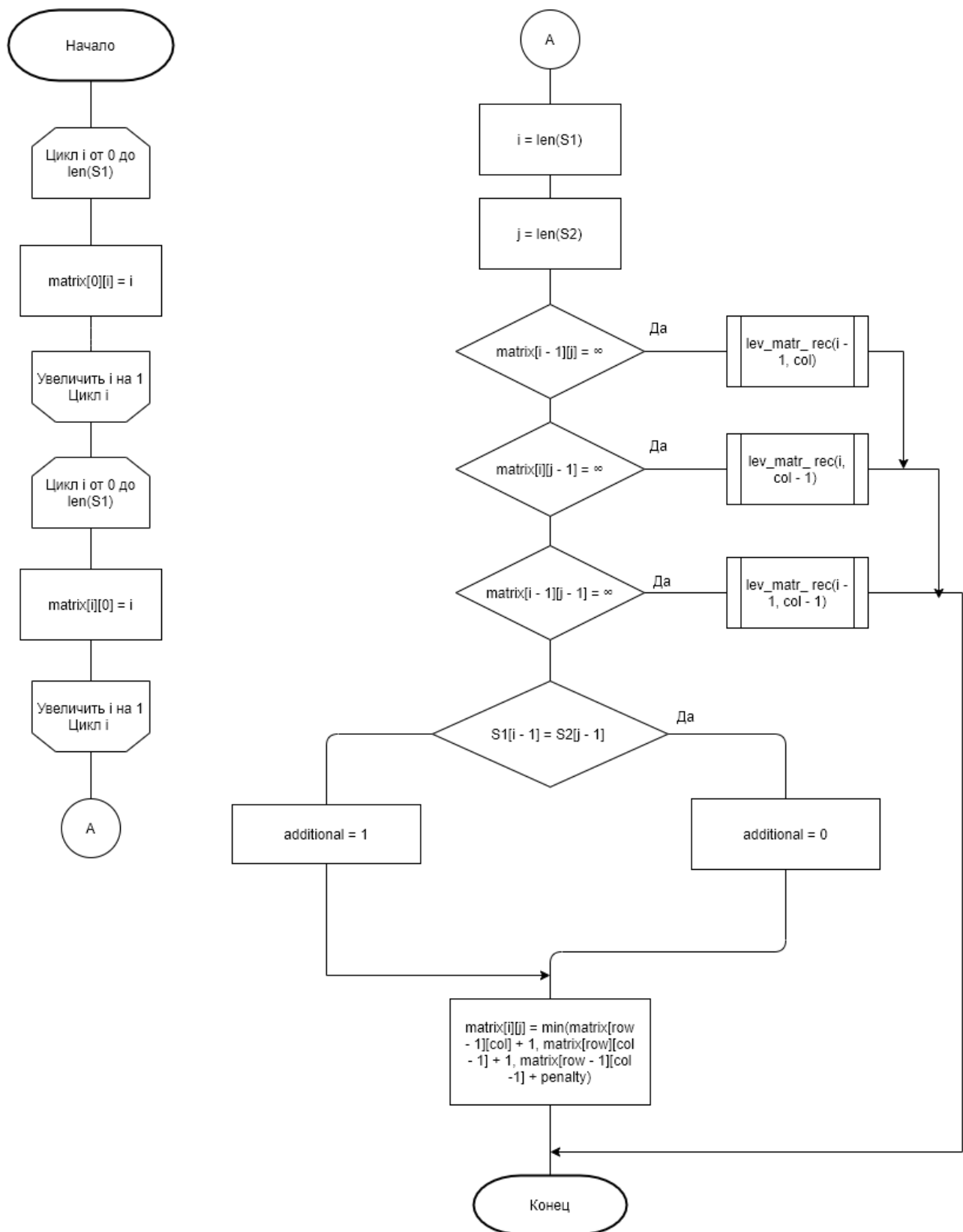


Рис. 3: Схема рекурсивной матричной реализации алгоритма поиска расстояния Левенштейна

Схема алгоритма поиска расстояния Дameraу-Левенштейна представлена на рис. 4

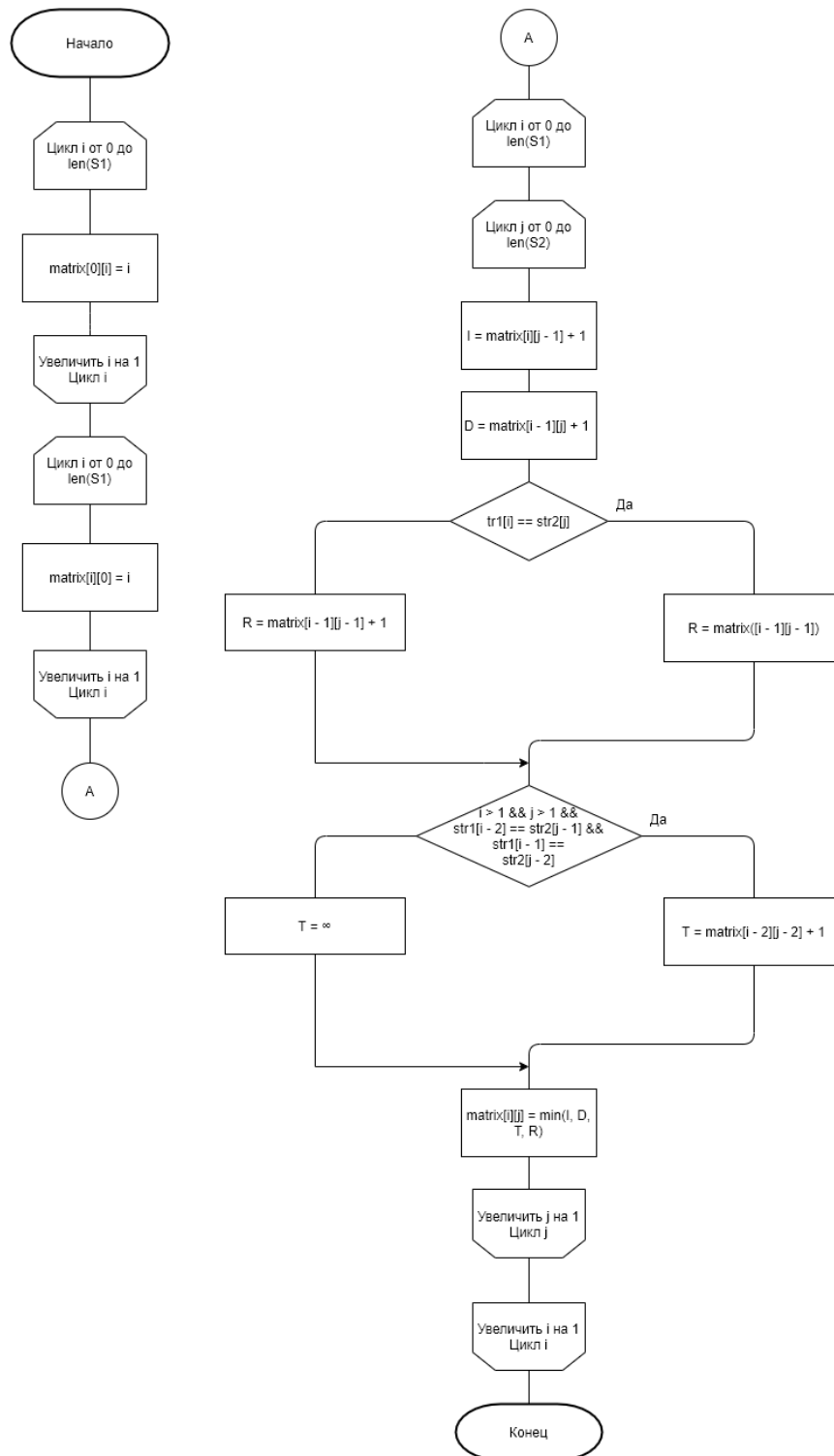


Рис. 4: Схема алгоритма поиска расстояния Дameraу-Левенштейна

## 2.2 Вывод

Были рассмотрены и обозначены требования к программе, а также к входным и выходным параметрам в программе. Также были рассмотрены и представлены схемы всех рассматриваемых алгоритмов.

## 3 Технологическая часть

В данном разделе будет описана технологическая часть лабораторной работы: требования к ПО, листинг кода, сравнительный анализ всех алгоритмов.

### 3.1 Требования к программному обеспечению

Входные данные: два слова: str1, str2

Выходные данные: редакционное расстояние данных слов, а также матрица решения для матричных реализаций

Среда выполнения: Windows 10 x64

### 3.2 Средства реализации

Для выполнения данной лабораторной работы использовался ЯП Python 3.9.0

### 3.3 Листинг кода

В данном разделе будет представлен листинг кода разработанных алгоритмов (листинги 1 - 4).

#### 3.3.1 Рекурсивный алгоритм Левенштейна

Листинг 1: Рекурсивный алгоритм Левенштейна

```
1 def lev_rec(source , target):
2     if len(source) == 0 or len(target) == 0:
3         return abs(len(source) - len(target))
4
5     if (source[-1] == target[-1]):
6         additional = 0
7     else:
8         additional = 1
9
10    return min(lev_rec(source , target[:-1]) + 1,
11               lev_rec(source[:-1], target) + 1,
12               lev_rec(source[:-1], target[:-1]) + additional)
```

### 3.3.2 Матричный алгоритм Левенштейна

Листинг 2: Матричный алгоритм Левенштейна

```
1 def lev_matrix(source, target)
2     data = [[i + j for j in range(len(target) + 1)]
3             for i in range(len(source) + 1)]
4
5     for i in range(1, len(source) + 1)
6         for j in range(1, len(target) + 1)
7             if source[i - 1] == target[j - 1]:
8                 additional = 0
9             else:
10                additional = 1
11
12            data[i][j] = min(data[i - 1][j] + 1,
13                             data[i][j - 1] + 1,
14                             data[i - 1][j - 1] + additional)
15
16     return data[-1][-1]
```

### 3.3.3 Рекурсивный матричный алгоритм Левенштейна

Листинг 3: Рекурсивный матричный алгоритм Левенштейна

```
1 def lev_matrix_rec(matrix, row, column, source, target):
2     if row == 0:
3         return column
4     if column == 0:
5         return row
6     if matrix[row][column] == -1:
7         matrix[row][column] = min(lev_matrix_rec(matrix, row,
8             column - 1, source, target) + 1,
9             lev_matrix_rec(matrix, row - 1, column, source, target) + 1,
10            lev_matrix_rec(matrix, row - 1, column - 1, source, target) +
11            int(source[row - 1] != target[column - 1]))
12
13     return matrix[row][column]
14
15 def lev_matrix_recursion(source, target):
16     matrix = [[-1 for j in range(len(target) + 1)]
17              for i in range(len(source) + 1)]
18     lev_matrix_rec(matrix, len(source), len(target), source, target)
19
20     return matrix[-1][-1]
```

### 3.3.4 Алгоритм Дамерау-Левенштейна

Листинг 4: Алгоритм Дамерау-Левенштейна

```
1 def damer_lev(source, target):
2     data = [[i + j for j in range(len(target) + 1)]
3             for i in range(len(source) + 1)]
4
5     for i in range(1, len(source) + 1):
6         for j in range(1, len(target) + 1):
7             if source[i - 1] == target[j - 1]:
8                 additional = 0
9             else:
10                additional = 1
11            data[i][j] = min(data[i - 1][j] + 1,
12                             data[i][j - 1] + 1,
13                             data[i - 1][j - 1] + additional)
14
15            if (i > 1 and j > 1 and
16                source[i - 1] == target[i - 2] and
17                source[i - 2] == target[i - 1]):
18                data[i][j] = min(data[i][j], data[i - 2][j - 2] + 1)
19
20    return data[-1][-1]
```

## 3.4 Сравнительный анализ матричной и рекурсивной реализаций

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти.

**Рассмотрим разницу между рекурсивной и матричной реализациями:**

Рекурсивная версия алгоритма работает существенно медленнее матричной реализации ввиду многократного вызова функции. На каждый вызов необходимо производить соответствующие операции со стеком. Более того, главным недостатком является - повторное вычисление тех значений, которые были посчитаны на более ранних этапах рекурсии. В матричных реализациях будет затрачена дополнительная память на хранение матриц и дополнительных переменных в цикле, однако время работы подобной реализации будет значительно быстрее рекурсивной.

### 3.4.1 Теоретический анализ затрачиваемой памяти

**Рекурсивная реализация алгоритма Левенштейна.** Для получения конечной оценки затрачиваемой памяти необходимо память, затрачиваемую на единичный вызов функции умножить на максимальную глубину рекурсии, то есть на  $n + m$ , где  $n$  и  $m$  - длины сравниваемых строк  $s1$  и  $s2$  соответственно.

1. ссылки на строки  $s1$ ,  $s2$ :  $(m + n) * \text{sizeof}(\text{str})$ ,
2. длины строк:  $2 * \text{sizeof}(\text{int})$ ,
3. дополнительная переменная внутри алгоритма:  $\text{sizeof}(\text{int})$
4. адрес возврата

$\text{memory} = (m + n) * ((m + n) * \text{sizeof}(\text{str}) + 2 * \text{sizeof}(\text{int}) + 4 \text{ bytes})$

### Матричная реализация алгоритма Левенштейна

1. строки:  $\text{sizeof}(\text{str}) * (n + m)$
2. матрица:  $\text{sizeof}(\text{int}) * (n + 1) * (m + 1)$
3. дополнительная переменная внутри алгоритма:  $\text{sizeof}(\text{int})$

$\text{memory} = \text{sizeof}(\text{str}) * (n + m) + \text{sizeof}(\text{int}) * (n + 1) * (m + 1) + \text{sizeof}(\text{int})$

**Рекурсивный матричный алгоритм Левенштейна.** Аналогично обычному рекурсивному алгоритму для получения конечной оценки затрачиваемой памяти необходимо память, затрачиваемую на каждом рекурсивном вызове умножить на максимальную глубину рекурсии.

1. строки:  $\text{sizeof}(\text{str}) * (n + m)$
2. матрица:  $\text{sizeof}(\text{int}) * (n + 1) * (m + 1)$

$\text{memory} = (m + n) * (\text{sizeof}(\text{str}) * (n + m) + \text{sizeof}(\text{int}) * (n + 1) * (m + 1))$

При каждой необходимости предварительного подсчёта значения (рек. вызова)

1. передача строки и столбца:  $2 * \text{sizeof}(\text{int})$
2. дополнительная переменная:  $\text{sizeof}(\text{int})$
3. адрес возврата

$\text{memory} = (m + n) * (2 * \text{sizeof}(\text{int}) + \text{sizeof}(\text{int}) + 4 \text{ bytes})$

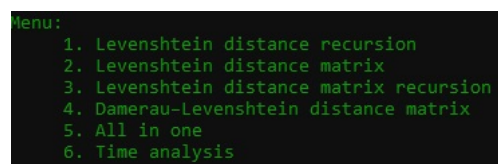
### Матричная реализация алгоритма Дамерау-Левенштейна

1. строки:  $\text{sizeof}(\text{str}) * (n + m)$
2. матрица:  $\text{sizeof}(\text{int}) * (n + 1) * (m + 1)$
3. дополнительная переменная внутри алгоритма:  $\text{sizeof}(\text{int})$

$\text{memory} = \text{sizeof}(\text{str}) * (n + m) + \text{sizeof}(\text{int}) * (n + 1) * (m + 1) + \text{sizeof}(\text{int})$

## 3.5 Интерфейс программы

При запуске программы пользователя встречает меню выбора реализаций алгоритма:



```
Menu:
1. Levenshtein distance recursion
2. Levenshtein distance matrix
3. Levenshtein distance matrix recursion
4. Damerau-Levenshtein distance matrix
5. All in one
6. Time analysis
```

Рис.5: Меню программы

После выбора необходимой реализации пользователю предлагают ввести строки s1 и s2. После ввода программа выдаёт результат:

```

Menu:
  1. Levenshtein distance recursion
  2. Levenshtein distance matrix
  3. Levenshtein distance matrix recursion
  4. Damerau-Levenshtein distance matrix
  5. All in one
  6. Time analysis
1
Input source: abc
Input target: abl
Distance = 1

```

Рис.6: Ввод строк и результат работы программы

### 3.6 Тестирование

Тестирование проводилось по методу черного ящика. При сравнении результатов двух функций использовалась функция random string, которая генерирует случайную строку нужной длины.

Листинг 5: Функция random string

```

1 def random_string(str_len):
2     letters = string.ascii_lowercase
3     return ''.join(random.choice(letters) for i in range(str_len))

```

Листинг 6: Тестирование

```

1 import unittest
2
3 from main import lev_rec, lev_matrix, lev_matrix_recursion, damer_lev
4 from main import random_string
5
6 class TestDistanse(unittest.TestCase):
7
8     def testEmpty(self):
9         self.assertEqual(self.function("", ""), 0)
10
11     def testSame(self):
12         self.assertEqual(self.function("abc", "abc"), 0)
13         self.assertEqual(self.function("0", "0"), 0)
14
15     def testDifferent(self):
16         self.assertEqual(self.function("a", ""), 1)
17         self.assertEqual(self.function("", "1"), 1)
18         self.assertEqual(self.function("b", "c"), 1)
19         self.assertEqual(self.function("bc", "b"), 1)
20         self.assertEqual(self.function("bc", "c"), 1)
21         self.assertEqual(self.function("ab", "cd"), 2)
22
23
24 class TestLevDistanse(TestDistanse):
25     def setUp(self):
26         self.function = lev_matrix
27     def testTypo(self):
28         self.assertEqual(self.function("ac", "ca"), 2)
29         self.assertEqual(self.function("abc", "cba"), 2)

```



```

30
31
32 class TestDamLevDistanse( TestDistanse ):
33     def setUp( self ):
34         self.function = damer_lev
35     def testTypo( self ):
36         self.assertEqual( self.function( "ac", "ca" ), 1 )
37         self.assertEqual( self.function( "abc", "cba" ), 2 )
38
39
40 class TestTwoFunctions( unittest.TestCase ):
41
42     n = 15
43     def testCompareSameLen( self ):
44         for i in range( TestTwoFunctions.n ):
45             str1 = random_string( 5 )
46             str2 = random_string( 5 )
47             self.assertEqual( self.f1( str1, str2 ), self.f2( str1, str2 ) )
48
49     def testCompareDifLen( self ):
50         for i in range( TestTwoFunctions.n ):
51             str1 = random_string( 3 )
52             str2 = random_string( 5 )
53             self.assertEqual( self.f1( str1, str2 ), self.f2( str1, str2 ) )
54
55     def testCompareEmpty( self ):
56         for i in range( TestTwoFunctions.n ):
57             str1 = random_string( 4 )
58             str2 = random_string( 5 )
59             self.assertEqual( self.f1( str1, str2 ), self.f2( str1, str2 ) )
60
61
62 class TestLev( TestTwoFunctions ):
63     def setUp( self ):
64         self.f1 = lev_rec
65         self.f2 = lev_matrix
66
67
68 class TestDamLev( TestTwoFunctions ):
69     def setUp( self ):
70         self.f1 = lev_matrix_recursion
71         self.f2 = damer_lev
72
73
74 if __name__ == '__main__':
75     suite = unittest.TestLoader().loadTestsFromTestCase( TestLev )
76     suite.addTests( unittest.TestLoader().
77                     loadTestsFromTestCase( TestDamLev ) )
78     suite.addTests( unittest.TestLoader().
79                     loadTestsFromTestCase( TestDamLevDistanse ) )

```

```
80 suite.addTests(unittest.TestLoader().
81                 loadTestsFromTestCase(TestLevDistance))
82 unittest.TextTestRunner().run(suite)
83 #unittest.main()
```

Все тесты пройдены успешно

### 3.7 Вывод

Были разработаны, протестированы и проанализированы спроектированные алгоритмы: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау-Левенштейна с заполнением матрицы.

## 4 Экспериментальная часть

В данной части работы будут приведены примеры работы программ, а также анализ алгоритмов на основе экспериментальных данных.

### 4.1 Примеры работы

Проверка на пустые строки:

```
1      source = ""
2      target = ""
3      Levenshtein Recursive: 0
4      Levenshtein Matrix: 0
5      Levenshtein Matrix Recursive: 0
6      Damerau Levenshtein: 0
```

Проверка на равенство строк:

```
1      source = "abc"
2      target = "abc"
3      Llevenshtein Recursive: 0
4      Levenshtein Matrix: 0
5      Levenshtein Matrix Recursive: 0
6      Damerau Levenshtein: 0
```

Операция удаления:

```
1      source = "abc"
2      target = "ab"
3      Levenshtein Recursive: 1
4      Levenshtein Matrix: 1
5      Levenshtein Matrix Recursive: 1
6      Damerau Levenshtein: 1
```

Операция замены:

```
1      source = "abf"
2      target = "abc"
3      Levenshtein Recursive: 1
4      Levenshtein Matrix: 1
5      Levenshtein Matrix Recursive: 1
6      Damerau Levenshtein: 1
```

Операция вставки:

```
1      source = "ab"
2      targer = "abc"
3      Levenshtein Recursive: 1
4      Levenshtein Matrix: 1
5      Levenshtein Matrix Recursive: 1
6      Damerau Levenshtein: 1
```

## Операция перестановки

```
1 source = "abc"
2 target = "acb"
3 Levenshtein Recursive: 2
4 Levenshtein Matrix: 2
5 Levenshtein Matrix Recursive: 2
6 Lamerau Levenshtein: 1
```

## 4.2 Постановка эксперимента по замеру времени

Были произведены замеры для строк длиной от 0 до 9.

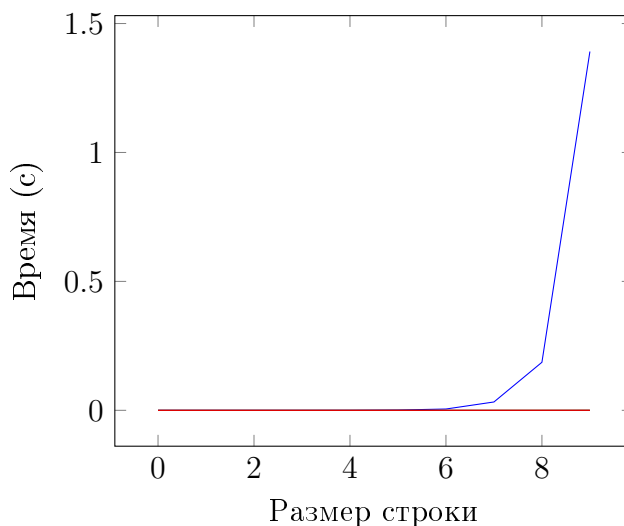
Для каждой размерности было проведено 100 вызовов функции. После чего получившееся время было поделено на 100. Таким образом было получено аппроксимированное значение времени выполнения функции

Результаты замеров процессорного времени:

Был проведен замер времени работы каждого из алгоритмов.

len	Lev(R)	Lev(M)	Lev(MR)	DamLev
3	0.00003	0.00001	0.00001	0.00001
4	0.00017	0.00001	0.00002	0.00001
5	0.00105	0.00002	0.00014	0.00002
6	0.00505	0.00003	0.00004	0.00005
7	0.03235	0.00004	0.00006	0.00006
8	0.18632	0.00006	0.00007	0.00008
9	1.39152	0.00011	0.00024	0.00012

График 1: Замеры времени работы алгоритмов



Легенда:

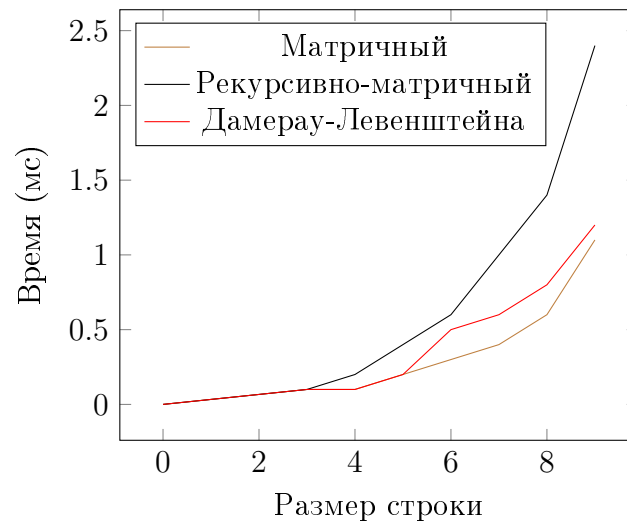
Синий цвет - Рекурсивная реализация

Коричневый цвет - Матричная реализация

Чёрный цвет - Рекурсивная матричная реализация

Красный цвет - Алгоритм Дамерау-Левенштейна

График 2: Замеры времени работы алгоритмов  
·10<sup>-4</sup>



### 4.3 Сравнительный анализ на материале экспериментальных данных

Рекурсивный алгоритм Левенштейна работает дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. При увеличении длины строк становится очевидна выигрышность по времени матричного варианта.

Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный на аналогичных данных. Алгоритм Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна. В нём добавлены дополнительные проверки, и по сути он является алгоритмом другого смыслового уровня.

По расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

### 4.4 Вывод

Теоретические расчёты подтвердились результатами, полученными на практике: рекурсивный алгоритм ввиду многократного вызова функции и пересчёта уже известных значений выполняется крайне долго, рекурсивная матричная реализация выполняется быстрее, но всё равно из-за операций со стеком и вызовом самой себя уступает по времени обычной матричной реализации. Алгоритм Дамерау-Левенштейна уступает по времени обычной матричной реализации ввиду дополнительной проверки на перестановку символов.

# Заключение

Цель достигнута и все задачи выполнены.

В ходе работы были изучены алгоритмы поиска расстояния Левенштейна и Дamerau-Левенштейна. Реализованы алгоритмы поиска расстояния Левенштейна с заполнением матрицы, а также реализован рекурсивный алгоритм поиска расстояния Левенштейна. Экспериментально было установлено, что из трех алгоритмов Левенштейна самым медленным является рекурсивный, а самым требовательным по памяти - рекурсивный алгоритм с заполнением матрицы, однако так же было установлено, что за счет ее заполнения не происходит повторных вычислений, что существенно повышает скорость выполнения. Сравнение матричного алгоритма Левенштейна и Дamerau-Левенштейна показало, что последний работает медленнее, в силу того, что на каждой итерации цикла выполняется дополнительная проверка и в случае ее справедливости еще и дополнительные вычисления.

Из написанного выше можно сделать вывод, что именно матричный алгоритм Левенштейна следует использовать при решении задач на нахождение минимального редакционного расстояния.

## Список литературы

- [1] Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. – М.: Техносфера, 2017. – 267с
- [2] Нечёткий поиск в тексте и словаре [электронный ресурс].Режим доступа: <https://habr.com/ru/post/114997/>, свободный – (Дата обращения: 5.10.20)
- [3] Официальный сайт Python, документация [электронный ресурс]. Режим доступа: <https://docs.python.org/3.9>, свободный – (Дата обращения: 8.10.20)