МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

# Расстояние Левенштейна

Работу выполнил: Рязанов М. С., ИУ7-52Б

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

*Москва, 2019*

# Оглавление

# Введение

В теории информации и компьютерной лингвистике часто возникают следующие задачи, связанные с сравнением строк:

- исправления ошибок в слове
- сравнения текстовых файлов
- в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы является изучение алгоритмов, решающих приведенные задачи. Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать существующие методы решения задачи;
- разработать рекурсивные и нерекурсивные версии алгоритмов;
- выбрать технологии для последующей реализации и иследования алгоритмов;
- реализовать разработанные алгоритмы;
- произвести тестирование корректности работы реализаций;
- сравнить быстродействие реализаций;
- опиать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

# 1 | Аналитическая часть

**Расстояние Левенштейна** - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.[3]

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

**Действия обозначаются так:**

1. D (англ. delete) — удалить;
2. I (англ. insert) — вставить;
3. R (replace) — заменить;
4. M(match) - совпадение.

Пусть $S_1$ и $S_2$ — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min( & \\ D(i,j-1)+1, & \\ D(i-1,j)+1, & j > 0, i > 0 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ ), & \end{cases} \tag{1.1}$$

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов. С точки зрения приложений определение расстояния между словами или текстовыми полями по Левенштейну обладает следующими недостатками:

- при перестановке местами слов или частей слов получаются сравнительно большие расстояния;
- расстояния между совершенно разными короткими словами оказываются небольшими, в то время как расстояния между очень похожими длинными словами оказываются значительными.

Если к списку разрешённых операций добавить транспозицию (два соседних символа меняются местами), получается расстояние Дамерау — Левенштейна. Дамерау показал, что 80 % ошибок при наборе текста человеком являются транспозициями. [2]

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min( & \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ \begin{cases} D(i-2, j-2) + 1, & \text{if } i,j > 1 \text{ and } a_i = b_{j-1}, a_{i-1} = b_j \\ INF \end{cases} & \\ ) & \end{cases} \tag{1.2}$$

**Выводы:**

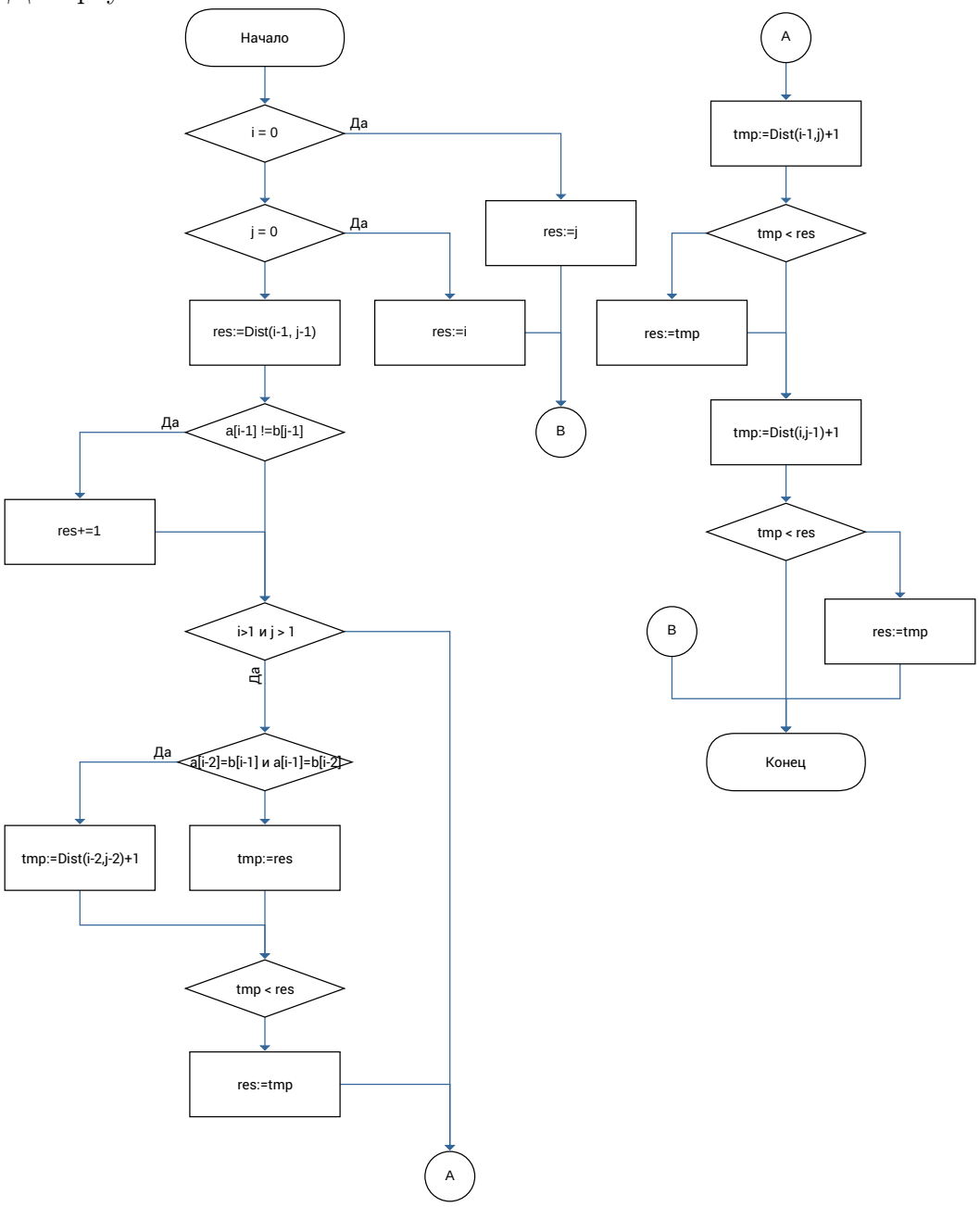
- В качестве алгоритмов для последующего анализа были выбраны алгоритмы Левенштейна и Домерау-Левенштейна;
- Расчетные соотношения к выбранным алгоритмам приведены в формулах 1.1 и 1.2.

# 2 | Конструкторская часть

## 2.1 Разработка алгоритмов

Общая идея рекурсивного алгоритма заключается в том, чтобы выразить расстояние между строками через расстояние между строками меньших длин, из которых возможен переход в текущие строки. Рекурсивный алгоритм в явном видем реализует вычисление формул 1.1 и 1.2.

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Домерау-Левенштейна



Недостатком данного алгоритма является то, что значения для некоторых значенией длин строк буду вычисляться несколько раз.

Избежать этого можно, используя матрицу для хранения редакторского расстояния между строками разными длины. Таким образом элемент d[i][j] матрицы расстояний будет хранить редакторское расстояние для строк длины i и j.

Из формул 1.1 и 1.2 можно заметить, что для вычисления расстояния для текущих длин строк необходимо знать расстояния для строк длины, которых на 1 (для расстояния Левенштейна) или 2 (для расстояния Домерау-Левенштейна) меньше. Значит нет необходимости хранить всю матрицу расстояний, нужно запоминать только последнюю и предпоследнии строки этой матрицы.

Рис. 2.2: Схема итерационного алгоритма нахождения расстояния Левенштейна

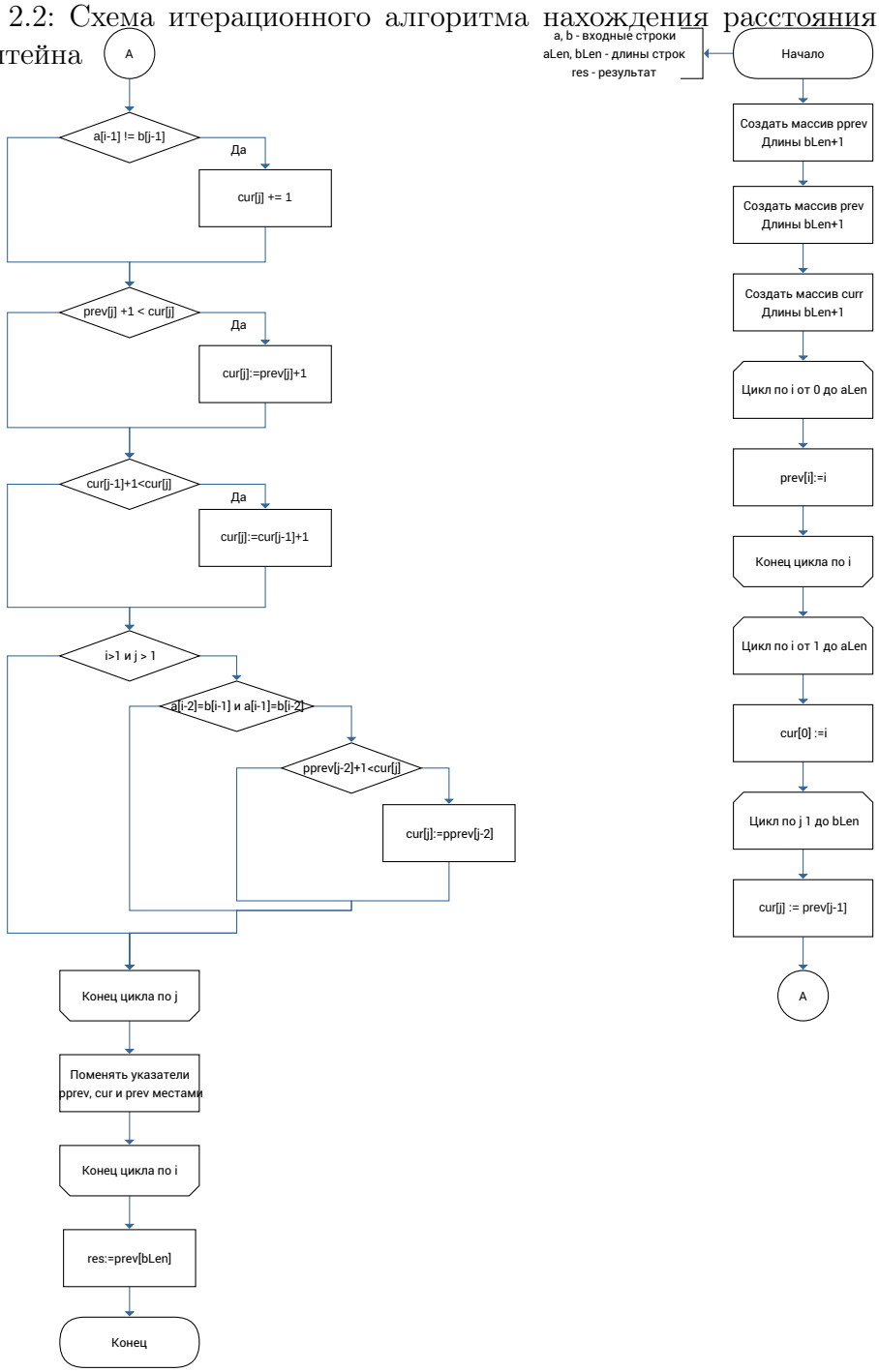
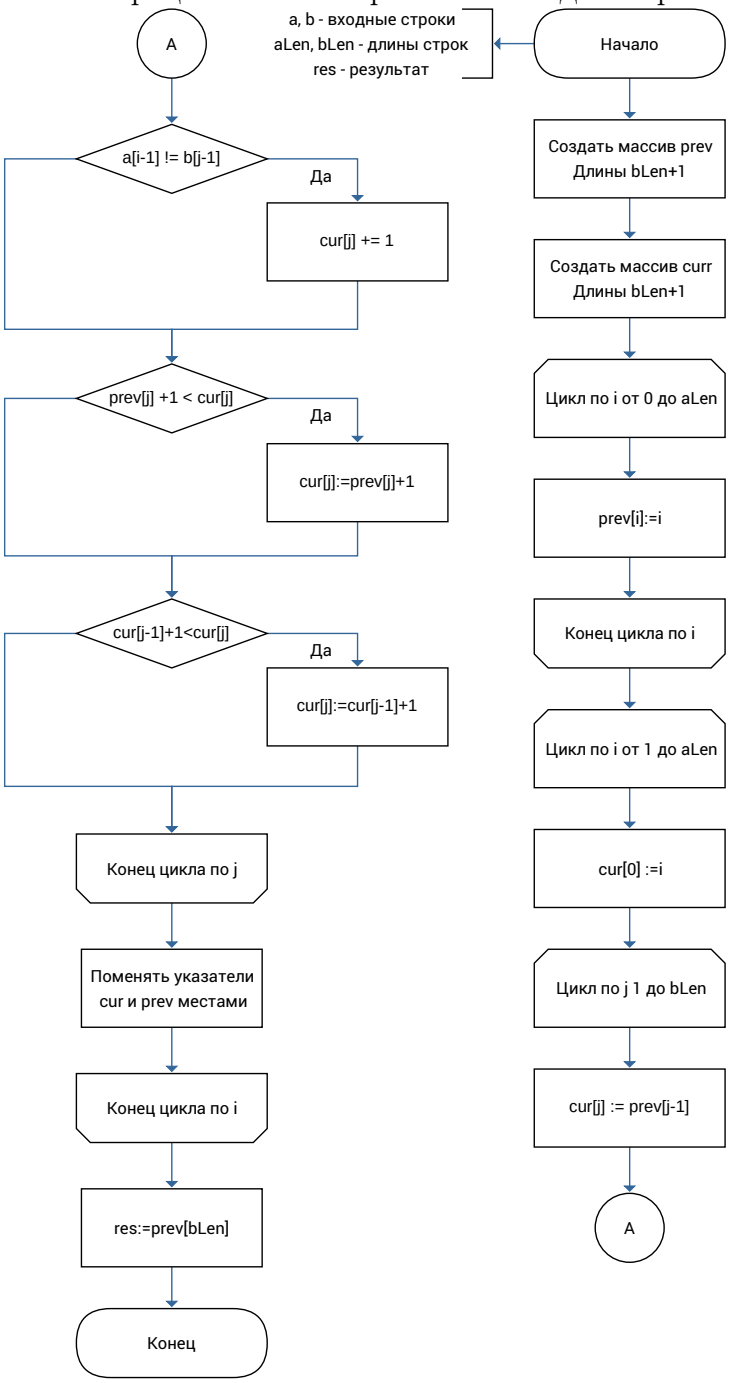


Рис. 2.3: Схема итерационного алгоритма нахождения расстояния Левенштейна



## 2.2 Оценка используемой памяти

Для итеративных алгоритмов затрачиваемая память определяется размером и количеством используемых массивов

$$M = 2 * sizeof(int) * N + R = 8 * N + 8 \tag{2.1}$$

Для расстояния Левенштейна

$$M = 3 * sizeof(int) * N + R = 12 * N + 12 \tag{2.2}$$

Для расстояния Дамерау-Левенштейна

Где $N$ - длина меньшей из строк, а R - общий объем дополнительных переменных.

Расход памяти рекурсивного алгоритма определяется числом рекурсивных вызовов и объемом локальных переменных, используемых в каждом вызове.

$$M = R * N = 16 * N = 16 * (N + K) \tag{2.3}$$

Где N, K - длины строк

Так как при каждом новом вызове, суммарная длина строк уменьшается на 1, следовательно максимальная глубина дерева рекурсии равна сумме длин двух строк.

**Выводы:**

- Разработан рекурсиный алгоритм нахождения расстояния Домерау-Левенштейна рисунок 2.1
- Разработан итеративный алгоритм нахождения расстояния Домерау-Левенштейна рисунок 2.2
- Разработан итеративный алгоритм нахождения расстояния Левенштейна рисунок 2.3
- Проведена оценка расходуемоей алгоритмами памяти (расчетные соотношения приведены в формулах 2.1, 2.2, 2.3)

# 3 | Технологическая часть

## 3.1 Выбор ЯП

В качестве языка программирования был выбран go[1] т.к он сочетает в себе эффективность, которая позволит оценить характеристики работы алгоритмов максимально точно, и удобство реализации. Также имеет встроенную библиотеку для проведение модульного тестирования.

Время работы алгоритмов было замерено с помощью функции Now() из пакета time.

## 3.2 Сведения о модулях программы

Программа состоит из:

- main.go - главный файл программы - точка входа программы, обработка входных данных;
- distFinders.go - файл с реализацией алгоритмов;
- utils.go - файл с вспомогательными функциями;
- distFinders_test.go - файл с тестами.

Листинг 3.1: Функция нахождения расстояния Левенштейна итеративно

```
1 func editorDistance(outstream io.Writer, a, b string) (dist
      int) {
2   var (
3     // Число символов строки a
4     aLen   = utf8.RuneCountInString(a)
```

```
5     // Число символов строки b
6       bLen    = utf8.RuneCountInString(b)
7     // Предыдущая строка матрицы вычисления расстояния
8       dpPrev = make([]int, bLen+1)
9     // Текущая строка матрицы вычисления расстояния
10      dpCurr = make([]int, bLen+1)
11    // Массив символов строки a
12      aRunes = []rune(a)
13      // Массив симоволов строки b
14      bRunes = []rune(b)
15    )
16
17  // Инициализация значений при длине строки = 0
18    for i := 0; i < bLen+1; i++ {
19      dpPrev[i] = i
20    }
21    printSlice(outstream, dpPrev)
22
23  // Цикл по строке a
24    for i := 1; i < aLen+1; i++ {
25      dpCurr[0] = i // Начальное значение при длине строки = 0
26
27      // Цикл по строке b
28      for j := 1; j < bLen+1; j++ {
29        dpCurr[j] = dpPrev[j-1]
30  // Если текущие символы не равны инкрементируем стоимость перехода
31        if aRunes[i-1] != bRunes[j-1] {
32          dpCurr[j]++
33        }
34  // Выбираем минимальную стоимость из переходов вида: D, I, R
35        dpCurr[j] = min(dpCurr[j], min(dpCurr[j-1]+1, dpPrev[
      j]+1))
36      }
37
38      printSlice(outstream, dpCurr)
39
40      // Заменяем предыдущую строку на текущую
41      dpPrev, dpCurr = dpCurr, dpPrev
42    }
43
```

```
44    return dpPrev[bLen]
45  }
```

Листинг 3.2: Функция нахождения расстояния Дамерау-Левенштейна итеративно

```
1  func domerauEditorDistance(outstream io.Writer, a, b string
       ) (dist int) {
2  var (
3    // Число символов строки a
4      aLen   = utf8.RuneCountInString(a)
5    // Число символов строки b
6      bLen   = utf8.RuneCountInString(b)
7    // Строка перед предыдущей матрицы вычисления расстояния
8      dpPrePrev = make([]int, bLen+1)
9    // Предыдущая строка матрицы вычисления расстояния
10     dpPrev = make([]int, bLen+1)
11   // Текущая строка матрицы вычисления расстояния
12     dpCurr = make([]int, bLen+1)
13   // Массив символов строки a
14     aRunes = []rune(a)
15   // Массив симоволов строки b
16     bRunes = []rune(b)
17   )
18
19 // Инициализация значений при длине строки = 0
20   for i := 0; i < bLen+1; i++ {
21     dpPrev[i] = i
22   }
23   printSlice(outstream, dpPrev)
24
25   for i := 1; i < aLen+1; i++ {
26     dpCurr[0] = i // Начальное значение при длине строки = 0
27
28     for j := 1; j < bLen+1; j++ {
29       dpCurr[j] = dpPrev[j-1]
30 // Если текущие символы не равны инкрементируем стоимость перехода
31       if aRunes[i-1] != bRunes[j-1] {
32         dpCurr[j]++
33       }
34
```

```
35 // Проверка возможности транспозиции
36         if i > 1 && j > 1 {
37           if aRunes[i-2] == bRunes[j-1] && aRunes[i-1] ==
       bRunes[j-2] {
38             dpCurr[j] = min(dpCurr[j], dpPrePrev[j-2]+1)
39           }
40         }
41 // Выбираем минимум из переходов вида: I, D, R, T
42         dpCurr[j] = min(dpCurr[j], min(dpCurr[j-1]+1, dpPrev[
       j]+1))
43       }
44       printSlice(outstream, dpCurr)
45
46 // Замена предпредыдущей строки на предыдущую,
47 // Предыудущей на текущую
48       dpPrePrev, dpPrev, dpCurr = dpPrev, dpCurr, dpPrePrev
49     }
50
51     return dpPrev[bLen]
52 }
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1 func _domerauRecursive(a, b []rune, i, j int) (dist int) {
2   // Если одна одной из строк нулевая, расстояние — длина другой
     строки
3   if i == 0 {
4     return j
5   }
6   if j == 0 {
7     return i
8   }
9
10    // Проверка возможности транспозиции
11  transition := math.MaxInt64
12  if i > 2 && j > 2 && a[i-1] == b[j-2] && a[i-2] == b[j-1]
      {
13    transition = _domerauRecursive(a, b, i-2, j-2) + 1
14  }
15
```

```
16      // Проверка стоимости замены
17    replace := _domerauRecursive(a, b, i-1, j-1)
18    if a[i-1] != b[j-1] {
19      replace++
20    }
21  // Возвращаем минимальную из стоимостей действий вида: I, D, R, T
22    return min(transition, min(replace, min(_domerauRecursive
      (a, b, i, j-1)+1, _domerauRecursive(a, b, i-1, j)+1)))
23  }
```

## 3.3 Тестовые случаи

Тестирование было организовано с помощью пакета **testing**.

Для каждой из функции были составленые следующие наборы тестов:

- пустые входные строки
- пуста одна из входных строк
- одинаковые входные строки
- входные строки в кириллице
- тест покрывающий все допустимые действия по изменению строки

## 3.4 Интерфейс программы

Листинг 3.4: "Способ запуска программы"

```
1     app <string> <string> <funcKey>
```

Где funcKey - ключ выбора определенной функции расчета редакторского расстояния:

- -l - Итеративно вычисляемое расстояние Левенштейна
- -d - Итеративно вычисляемое расстояние Домерау-Левенштейна

- -r - Рекурсивно вычисляемое расстояние Домерау-Левенштейна

После этого программой будет выведена матрица расчета(для итеративных алгоритмов), искомое расстояние между 2 строками и время исполнения программы.

**Выводы:**

- В качестве языка программирования выбран Golang, так как прост в использовании, а также достаточно эффективен для проведения анализа времени работы разработанных алгоритмов
- Алгоритмы реализованы в виде подпрограмм, код которых приведен в листингах : 3.1, 3.2, 3.3
- Разработаны тестовые случаи для проверки корректности работы функций
- Был выбран консольный интерфейс программы, так как прост в реализации и использовании при небольшом функционале программмы.

# 4 | Исследовательская часть

Был проведен замер времени работы каждого из алгоритмов.

Таблица 4.1: Время исполнения различных реализаций

| Длина(символ) | Lev(I)(c) | DamLev(I)(c) | DamLev(R)(c) |
|---|---|---|---|
| 3 | 0.000002834 | 0.000000353 | 0.000002834 |
| 4 | 0.000000303 | 0.000000382 | 0.000004887 |
| 5 | 0.000000354 | 0.000000461 | 0.000015351 |
| 6 | 0.000000412 | 0.000000538 | 0.000070558 |
| 7 | 0.000000466 | 0.000000630 | 0.000381820 |
| 8 | 0.000000535 | 0.000000735 | 0.002082971 |
| 9 | 0.000000535 | 0.000000846 | 0.010481199 |

**Выводы:**

- Итеративные реализации одного порядка, однако реализация расстояния Левенштейна оказывается на константу быстрее;
- При большей длине строк рекурсивная реализация заметно медленнее(при длине в 9 символов итеративная реализация в 50,000 раз быстрее.)

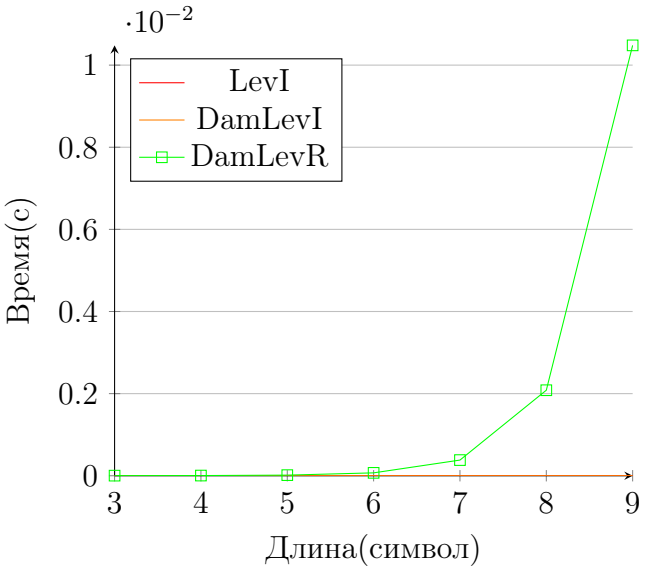



Рис. 4.1: Сравнение времени исполения различных реализаций

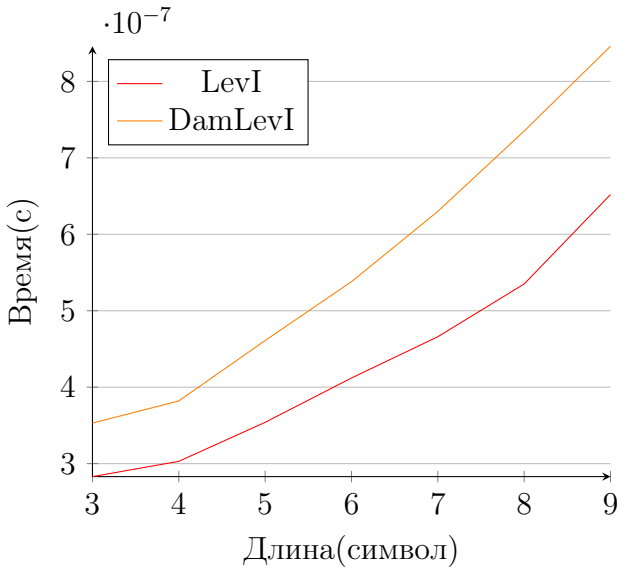



Рис. 4.2: Сравнение времени исполнения итеративных реализаций

# Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк.

# Список литературы


[1] *Golang documentation*. URL: `https://godoc.org/`. (accessed: 02.10.2019).

[2] Гасфилд. *Строки, деревья и последовательности в алгоритмах.* Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.

[3] В. И. Левенштейн. «Двоичные коды с исправлением выпадений, вставок и замещений символов». в: Доклады Академий Наук СССР. (1965).