**Федеральное государственное бюджетное образовательное учреждение   
высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

**Анализ алгоритмов**

**Лабораторная работа №1**

**Отчёт на тему:**

# «Расстояния Левенштейна и Дамерау-Левенштейна»

Выполнила:

Янова Даниэлла

ИУ7-53

**Москва, 2018**

**Введение**

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Впервые задачу упомянул в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей 0-1. Впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд. Расстояние Левенштейна и его обобщения активно применяется для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи); для сравнения текстовых файлов утилитой diff и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы); в биоинформатике для сравнения генов, хромосом и белков. 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. Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов. Расстояние Дамерау-Левенштейна, как и метрика Левенштейна, является мерой "схожести"двух строк. Алгоритм его поиска находит применение в реализации нечёткого поиска, а также в биоинформатике (сравнение ДНК), несмотря на то, что изначально алгоритм разрабатывался для сравнения текстов, набранных человеком (Дамерау показал, что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе. Поэтому метрика Дамерау-Левенштейна часто используется в редакторских программах для проверки правописания).

**Задачи работы:**

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;

2. Применение метода динамического программирования для матричной реализации указанных алгоритмов;

3. Получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;

4. Сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5. Экспериментальное подтверждение различий во временн´oй 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;

6. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

**1. Аналитическая часть**

В этом разделе затронуты темы, описанные ниже.

1. Определение и формула нахождения расстояния левенштейна.
2. Определение и формула нахождения расстояния Дамерау-Левенштейна.
3. Принцип поиска расстояния для обоих случаев.
4. Применение

**1.1 Описание алгоритмов**

Ниже будут изложены словесные описания алгоритмов.

**1.1.1 Расстояние Левенштейна**

(также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике - это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Формула, для нахождения расстояния Левенштейна:

Для нахождения кратчайшего расстояния необходимо вычислить матрицу D, используя вышеприведённую формулу. Её можно вычислять как по строкам, так и по столбцам. Для восстановления редакционного предписания требуется вычислить матрицу D, после чего идти из правого нижнего угла (M,N) в левый верхний, на каждом шаге ища минимальное из трёх значений:

• если минимально (D(i-1, j) + цена удаления символа S1[i]), добавляем удаление символа S1[i] и идём в (i-1, j);

• если минимально (D(i, j-1) + цена вставки символа S2[j]), добавляем вставку символа S2[j] и идём в (i, j-1);

• если минимально (D(i-1, j-1) + цена замены символа S1[i] на символ S2[j]), добавляем замену S1[i] на S2[j] (если они не равны; иначе ничего не добавляем), после чего идём в (i-1, j-1) Здесь (i, j) — клетка матрицы, в которой мы находимся на данном шаге. Если минимальны два из трёх значений (или равны все три), это означает, что есть 2 или 3 равноценных редакционных предписания.

**1.1.2 Расстояние Дамерау-Левенштейна**

Расстояние Дамерау-Левенштейна - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Формула, для нахождения расстояния:

Для нахождения кратчайшего расстояния необходимо вычислить матрицу D, используя вышеприведённую формулу. Её можно вычислять как по строкам, так и по столбцам. Для восстановления редакционного предписания требуется вычислить матрицу D, после чего идти из правого нижнего угла (M,N) в левый верхний, на каждом шаге ища минимальное из трёх значений:

• если минимально (D(i-1, j) + цена удаления символа S1[i]), добавляем удаление символа S1[i] и идём в (i-1, j);

• если минимально (D(i, j-1) + цена вставки символа S2[j]), добавляем вставку символа S2[j] и идём в (i, j-1);

• если минимально (D(i-1, j-1) + цена замены символа S1[i] на символ S2[j]), добавляем замену S1[i] на S2[j] (если они не равны; иначе ничего не добавляем), после чего идём в (i-1, j-1) Здесь (i, j) — клетка матрицы, в которой мы находимся на данном шаге. Если минимальны два из трёх значений (или равны все три), это означает, что есть 2 или 3 равноценных редакционных предписания.

• если транспозиция возможна, то возвращаем (D(i-2, j-2) + 1

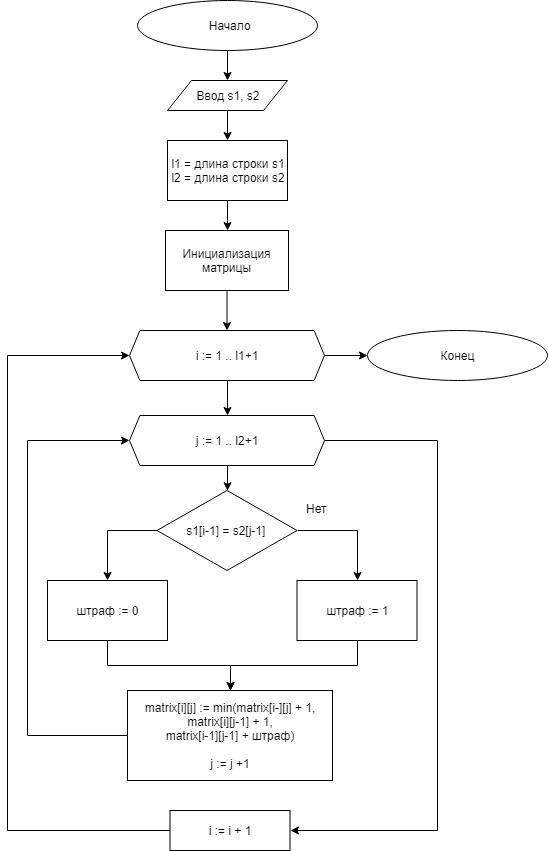
**1.2.3 Область применения алгоритмов**

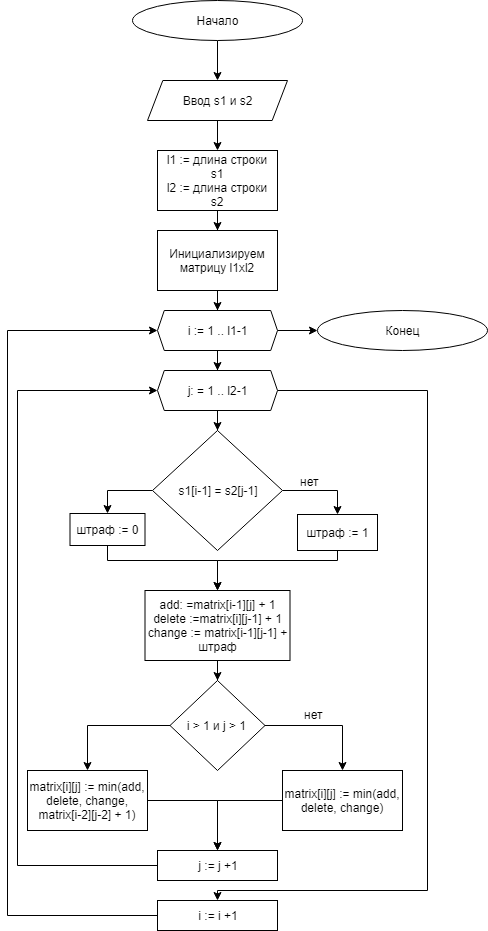
Данные алгоритмы применяются при поиске информации по запросу, с помощью них можно, найти, если пользователь допустил одну или несколько ошибок в слове, наиболее подходящее(имеющее наименьшее расстояние) к нему слово и заменить его в поисковой строке.

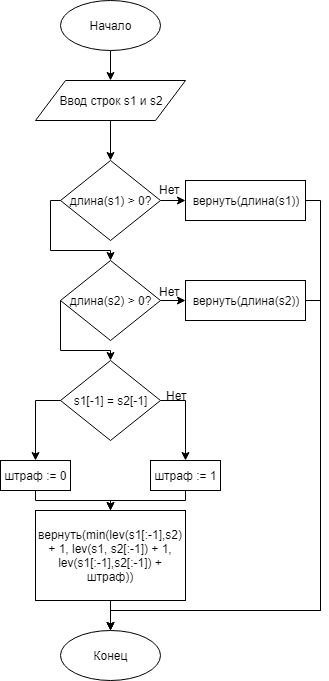
**2. Констукторский раздел**

В данном разделе размещены блоксхемы алгоритмов и сравнительный анализ линейной и рекурсивной реализаций.

**2.1 Разработка реализаций алгоритма**

 Рисунок 1.1 – Схема классической реализация алгоритма Левенштейна

Рисунок 1.2 – Схема алгоритма Дамерау-Левенштейна

Рисунок 1.3 – Схема рекурсивной реализации алгоритма Левенштейна

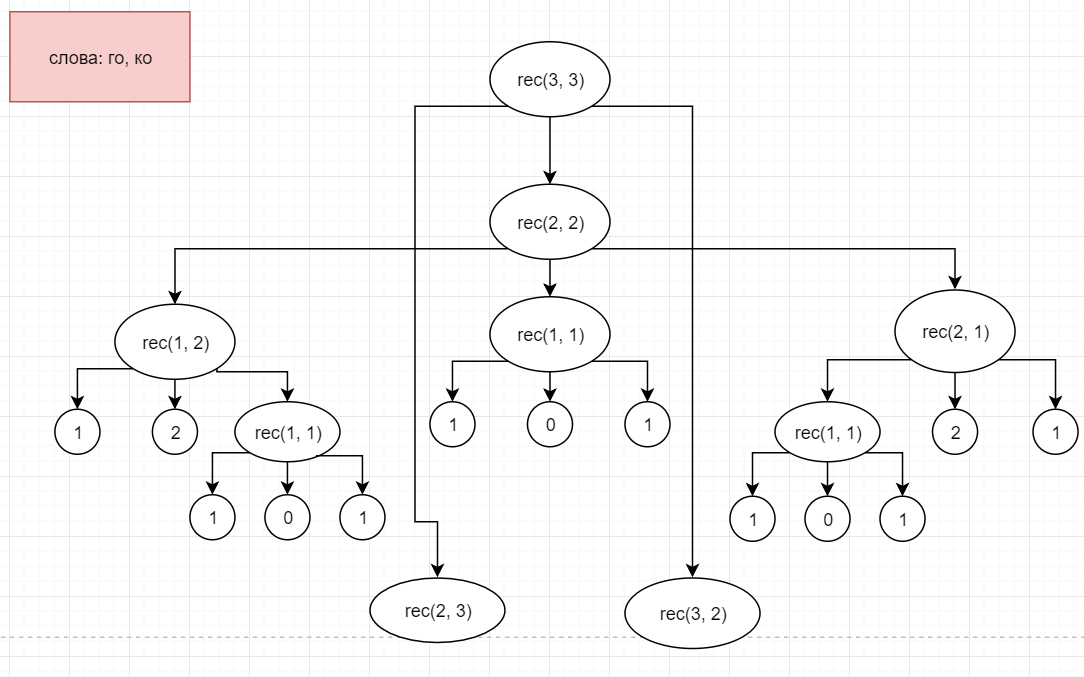
**2.2 Сравнительный анализ рекурсивной и нерекурсивной реализаций** Рекурсивная реализация работает медленнее по сравнение с линейной из-за повторных вычислений, возникающих в ходе работы рекурсивного алгоритма, это наглядно видно на Рис. 2.1, иллюстрирующей дерево рекурсивных вызовов выбранной реализации. Функция rec() принимает на вход 2 строки и 2 индекса, но чтобы не нагружать схему, 2 строки при вызове не подписываются. Для высчитывания цены перехода из диагонального элемента в текущий надо учитывать совпадают ли предыдущие символы строк, но поскольку это не влияет на дерево рекурсии, в схеме я опускаю этот момент и пишу просто число/вызов функции  
 

Рисунок 2.1- Дерево рекурсивных вызовов 1

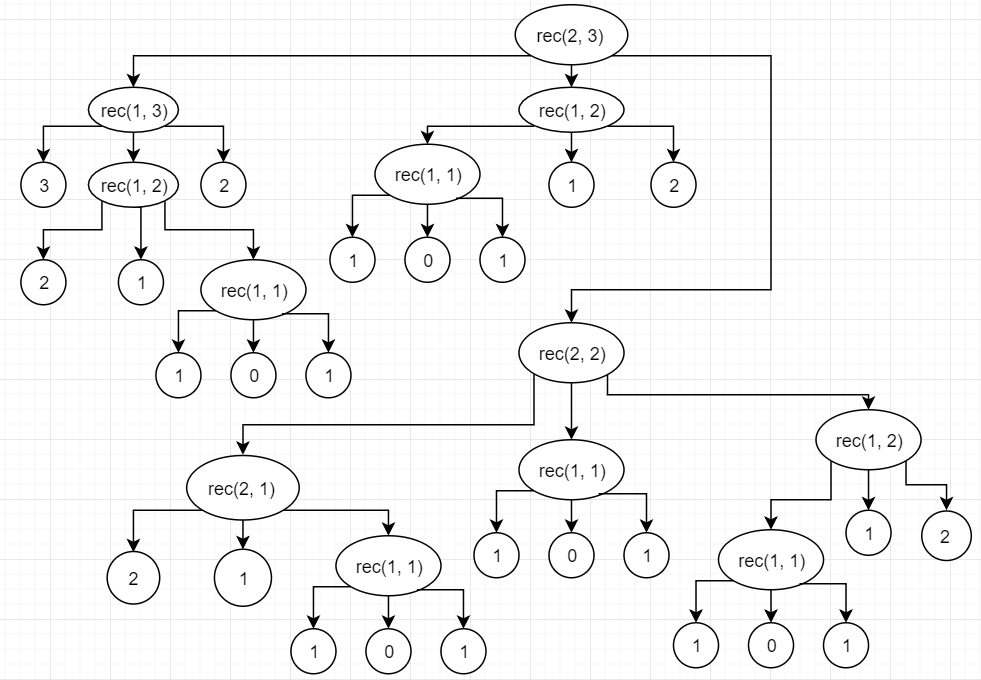


Рисунок 2.2- Дерево рекурсивных вызовов 2

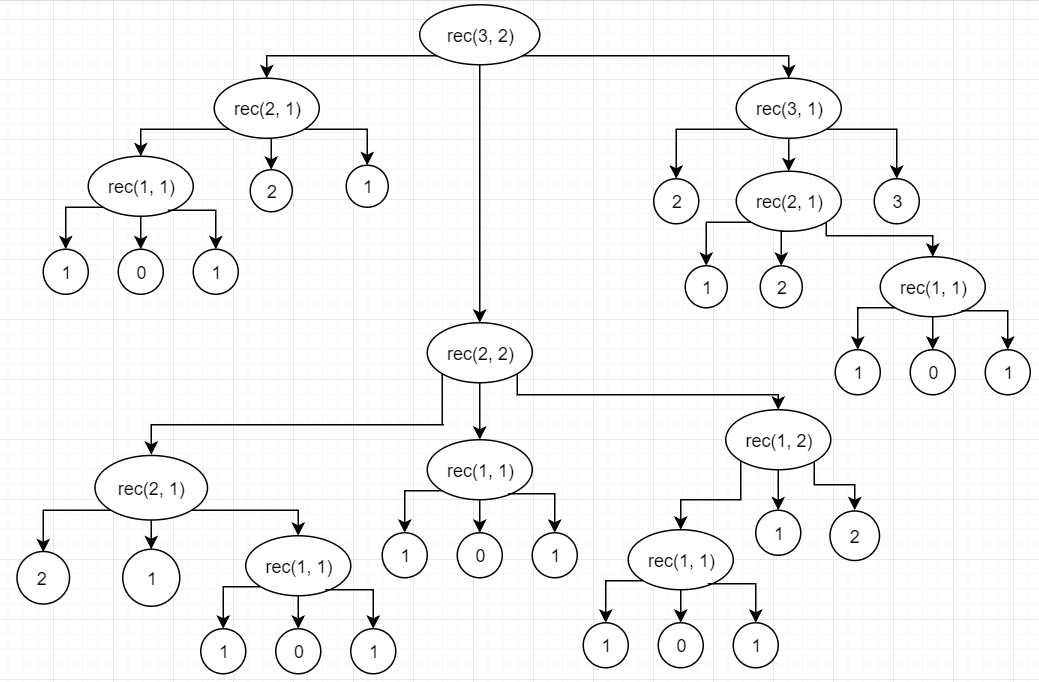


Рисунок 2.3- Дерево рекурсивных вызовов 3

**3. Технологический раздел**

В этом разделе будут изложены требования к программному обеспечению и листинги алгоритмов.

**3.1 Требования к программному обеспечению**

Данная программа разрабатывалась на языке Python 3.7, поддерживаемом многими операционными системами. Для запуска программы необходим интерпретатор, поддерживающий Python 3.7 для 64-битной системы.

**3.2 Листинг кода 1**

**def levenstain(a, b, print\_flag):**

n, m = len(a), len(b)

**if** n > m:

a, b = b, a

n, m = m, n

matrix = []

current\_row = [x **for** x **in** range(n+1)]

matrix.append(current\_row)

**for** i **in** range(1, m+1):

previous\_row, current\_row = current\_row, [i]+[0]\*n

**for** j **in** range(1, n+1):

add = previous\_row[j]+1

delete = current\_row[j-1]+1

change = previous\_row[j-1] + (a[j-1] != b[i-1])

current\_row[j] = min(add, delete, change)

matrix.append(current\_row)

**if** print\_flag != 0:

print\_matrix(matrix,a,b)

**return** current\_row[n]

Листинг кода 1.1 — Классический алгоритм Левенштейна

**def** damerau\_levenstain(a, b, print\_flag):

n, m = len(a), len(b)

**if** n > m:

a, b = b, a

n, m = m, n

matrix = []

previous\_row = [x **for** x **in** range(n+1)]

current\_row = [1]+[0]\*n

**for** i **in** range(1, n+1):

add = previous\_row[i]+1

delete = current\_row[i-1]+1

change = previous\_row[i-1] + (a[i-1] != b[0])

current\_row[i] = min(add, delete, change)

matrix.append(previous\_row)

matrix.append(current\_row)

**for** i **in** range(2, m+1):

preprevious\_row, previous\_row, current\_row = previous\_row, current\_row, [i]+[0]\*n

**for** j **in** range(1, n+1):

add = previous\_row[j]+1

delete = current\_row[j-1]+1

change = previous\_row[j-1] + (a[j-1] != b[i-1])

**if** j == 1:

current\_row[j] = min(add, delete, change)

**else**:

**if** a[j-2] == b[i-1] **and** a[j-1] == b[i-2]:

current\_row[j] = min(preprevious\_row[j-2] + 1, add, delete, change)

**else**:

current\_row[j] = min(add, delete, change)

matrix.append(current\_row)

**if** print\_flag != 0:

print\_matrix(matrix,a,b)

**return** current\_row[n]

Листинг кода 1.2 — Алгоритм Дамерау-Левенштейна

**def** recurr\_levenstain(a, b):

n, m = len(a), len(b)

**if** n > m:

a, b = b, a

n, m = m, n

**return** find\_eom(n, m, a, b)

**def** find\_eom(i, j, a, b):

find\_eom.count += 1

**if** i == 0:

**return** j

**elif** j == 0:

**return** i

**else**:

el = min(find\_eom(i-1, j, a, b)+1, find\_eom(i, j-1, a, b)+1,

find\_eom(i-1, j-1, a, b)+ (a[i-1] != b[j-1]))

**return** el

Листинг кода 1.3 — Рекурсивный алгоритм Левенштейна

**4. Исследовательский раздел**

В этом разделе будут изложены примеры работы и исследования быстродействия алгоритмов.

**4.1 Пример работы**

Далее представлены матрицы, выводимые алгоритмами:

а) Алгоритм Левенштейна

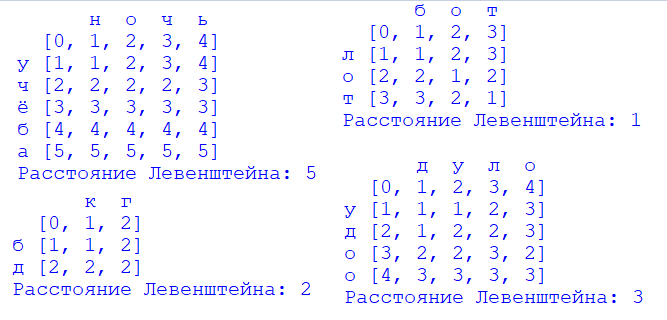


Рисунок 3.1 — Примеры работы классического алгоритма Левенштейна

б) Алгоритм Дамерау-Левенштейна

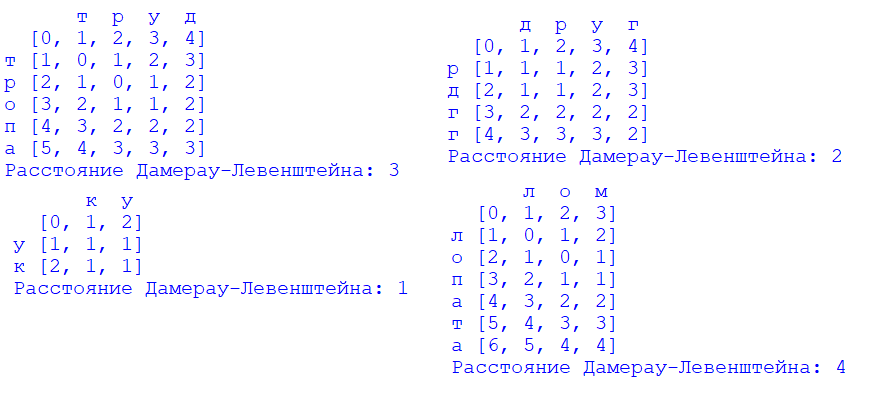


Рисунок 3.2 — Примеры работы Алгоритма Дамерау-Левенштейна

Таблица 1

Примеры работы алгоритмов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **s1** | **s2** | **Ожидание** | **Результат** | **Оценка (0, 1)** |
| 1 | сон | мечта | 5 5 5 | 5 5 5 | 1 1 1 |
| 2 | анализ | алгоритмов | 8 8 8 | 8 8 8 | 1 1 1 |
| 3 | выхода | нет | 6 6 6 | 6 6 6 | 1 1 1 |
| 4 | угол | гулл | 3 2 3 | 3 2 3 | 1 1 1 |
| 5 | первый | превый | 1 1 1 | 1 1 1 | 1 1 1 |
| 6 | письмо | почта | 5 5 5 | 5 5 5 | 1 1 1 |
| 7 | фигура | карикатура | 6 6 6 | 6 6 6 | 1 1 1 |
| 8 | армяне | нахичевань | 8 8 8 | 8 8 8 | 1 1 1 |

1. алгоритм Левенштейна;

2. алгоритм Дамерау-Левенштейна;

3. рекурсивный алгоритм Левенштейна.

**4.2 Постановка эксперимента.**   
  
 При сравнении быстродействия алгоритмов были использованы строки длинной в диапазоне от 100 до 1000 с шагом 100. результат одного эксперимента рассчитывался как средний из результатов проведенных испытаний с одинаковыми входными данными. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными.

**4.3 Сравнительный анализ рекурсивной и нерекурсивной реализации**

В данном разделе будет проведен сравнительный анализ алгоритмов. График для итеративной реализацией алгоритма изображен на 4.2. Там же расположен График времени работы рекурсивной реализации изображен на 4.2



Таблица 2: Результаты замеров времени, затрачиваемого итеративной и рекурсивной реализацией алгоритма Левенштейна



Рисунок 4.2 – Графики времени работы (в мкс) классического и рекурсивного алгоритмов расстояния Левенштейна.



Таблица 2: Результаты замеров времени, затрачиваемого итеративной реализацией алгоритма Левенштейна и Дамерау-Левенштейна

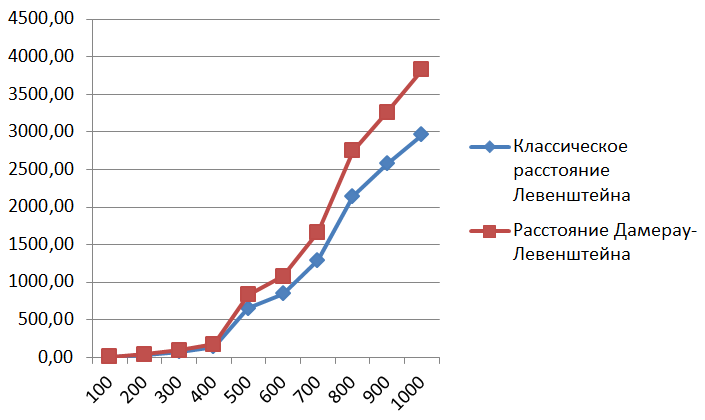


Рисунок 4.3 – Графики времени работы (в мкс) классического алгоритма расстояния Левенштейна и алгоритма Дамерау-Левенштейна.

В результате проведенного эксперимента был получен следующий вывод: рекурсивный алгоритм Левенштейна работает гораздо дольше итеративной реализации, начиная с длины строк = 2, время его работы увеличивается в геометрической прогрессии. Итеративный алгоритм значительно превосходит его по эффективности. Алгоритм Дамерау-Левенштейна работает дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки.

**Заключение**

В ходе работы были изучены и реализованы в матричной и рекурсивной форме алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, применены методы динамического программирования для матричной реализации данных алгоритмов. Также был проведен сравнительный анализ линейной и рекурсивной реализаций алгоритма Левенштейна по затрачиваемым ресурсам. Экспериментально подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций путем замеров процессорного времени работы алгоритмов. Рекурсивный алгоритм Левенштейна работает на несколько порядков медленнее матричной реализации. Если длина сравниваемых строк превышает 10, рекурсивный алгоритм становится неприемлемым для использования. Матричная реализация алгоритма Дамерау-Левенштейна работает дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки.