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»

Отчёт по лабораторной работе №1

По дисциплине: «Анализ алгоритмов»

Тема: «Вычисление редакционного расстояния»

Студент Мередова Айджахан

Группа ИУ7-56Б

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

Москва - 2020 г.

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. Впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

1. Аналитическая часть

1.1. Постановка задачи

Цель данной лабораторной работы: разработать и сравнить алгоритмы поиска расстояний Левенштейнаи Демерау-Левенштейна с использованием метода динамического динамического программирования.
В данной работе требуется:

1) описать алгоритмы поиска Левенштейна и Демерау - Левенштейна лдля нахождения редакционного расстояния между строками;

2) реализовать данные алгоритмы на одном из языков программирования;

3) провести сравнительный анализ алгоритмов по затраченному времени и памяти;

4) привести пример работы всез указанных алгоритмов;

1.2. Описание расстояния

1.2.1. Расстояние Левенштейна

Для поиска расстояния Левенштейна, чаще всего используют алгоритм, в котором необходимо заполнить матрицу D, размером n+1 на m+1, где n, m - длины сравниваемых строк A и B, по следующим правилам:

- $D_{0,0} = 0$;
- $D_{i,j} =$ минимальное из:

– $D_{i-1,j-1+0}$(мисволы одиноковые), либо $Di-1,j-1+$ Creplace(замена символа);

– $D_{i,j-1}$ + Cdelete(удаление символа);

– $D_{i-1,j}$ + Cinsert(Вставка символа);

где Creplace, Cdelete, Cinsert - цена или вес замены, удаления и вставки символа. При этом $D_{n-1,m-1}$ - содержит значение расстояния Левенштейна.

Пусть S1 и S2 - две строки(длиной n и m соответственно) над некоторым алфавитом, тогда редакционное расстояние D(S1,S2) можно подсчитать по следующей рекуррентной формуле:

$$D(S_1[i]S_2[j]) = min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1; \\ D(S_1[1..i-1], S_2[1..j]) + 1; \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{cases} 0, \text{если} S_1[i] = S_2[j]; \\ 1, \text{иначе}; \end{cases} \end{cases} \tag{1}$$

Рассмотрим формулу подробно. Здесь шаг по i символизирует удаление (D) из первой строки, по j - вставку (I) в первую строку, а шаг по обоим индексам символизирует замену символа (R) или отсутствие изменений (M). Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Так же очевидно слелдующее: чтобы получить пустую строку из строки длиной i, следует совершить i операций удаления, а чтобы получить строку длиной j из пустой, трубуется произвести j операций вставки. В нетривиальном случае необходимо выбрать минимальную «стоимость» из трёх вариантов. Вставка или удаление будет в любом случае имеют стоимость в одну операцию, а вот замена может не понадобится, если символы равны. Тогда шаг по обоим индексам бесплатный. Формализация этих рассуждений приводит к формуле (1).

1.2.2. Расстояние Дамерау - Левентейшна

В автоматической обработке естественного языка(например, при автоматической проверке орфографии) часто бывает нужно определить, насколько различны два написанных слова. Одна из количественных мер, используемых для этого, называется расстоянием Дамерау - Левенштейна - в честь Владимира Левенштейна и Фредерика Дамерау. Левенштейн придумал способ измерения «расстояний» между словами, а Дамерау независимо от него выделил несколько классов, в которые попадает большинство опечаток.

Эта вариация алгоритма вносит в опередедение расстояния Левенштейна ещё одно правило - транспозиция(перестановка) двух соседних букв также учитывается как одна операция, наряду с вставками, удалениями и заменами.

Чтобы вычислить такое расстояние, достаточно немного модифицировать алгоритм нахождения обычного расстояния Левенштейна следующим образом: хранить не две, а три последних строки матрицы, а также добавить соответсвующее дополнительное условие - в случае обнаружения транспозиции при расчёте расстояния также учитывать и её стоимость.

$$D(S_1[i]S_2[j]) = min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1; \\ D(S_1[1..i-1], S_2[1..j]) + 1 \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{cases} 0, \text{если} S_1[i] = S_2[j]. \\ 1, \text{иначе.} \end{cases} \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{cases} 1, \text{если} S_1[i] = S_2[j-1] \text{и} \\ S_1[i] = S_2[j-1]. \\ \infty, \text{иначе.} \end{cases} \end{cases} \tag{2}$$

2. Конструкторская часть

2.1. Разработка алгоритмов

2.1.1. Схема алгоритмов

На рисунках 1-4 показаны схемы итеративной и рекурсивной алгоритмов Левенштейна, рекурсивной реализации с заполнением матрицы и схема алгоритма Дамерау - Левентштейна.

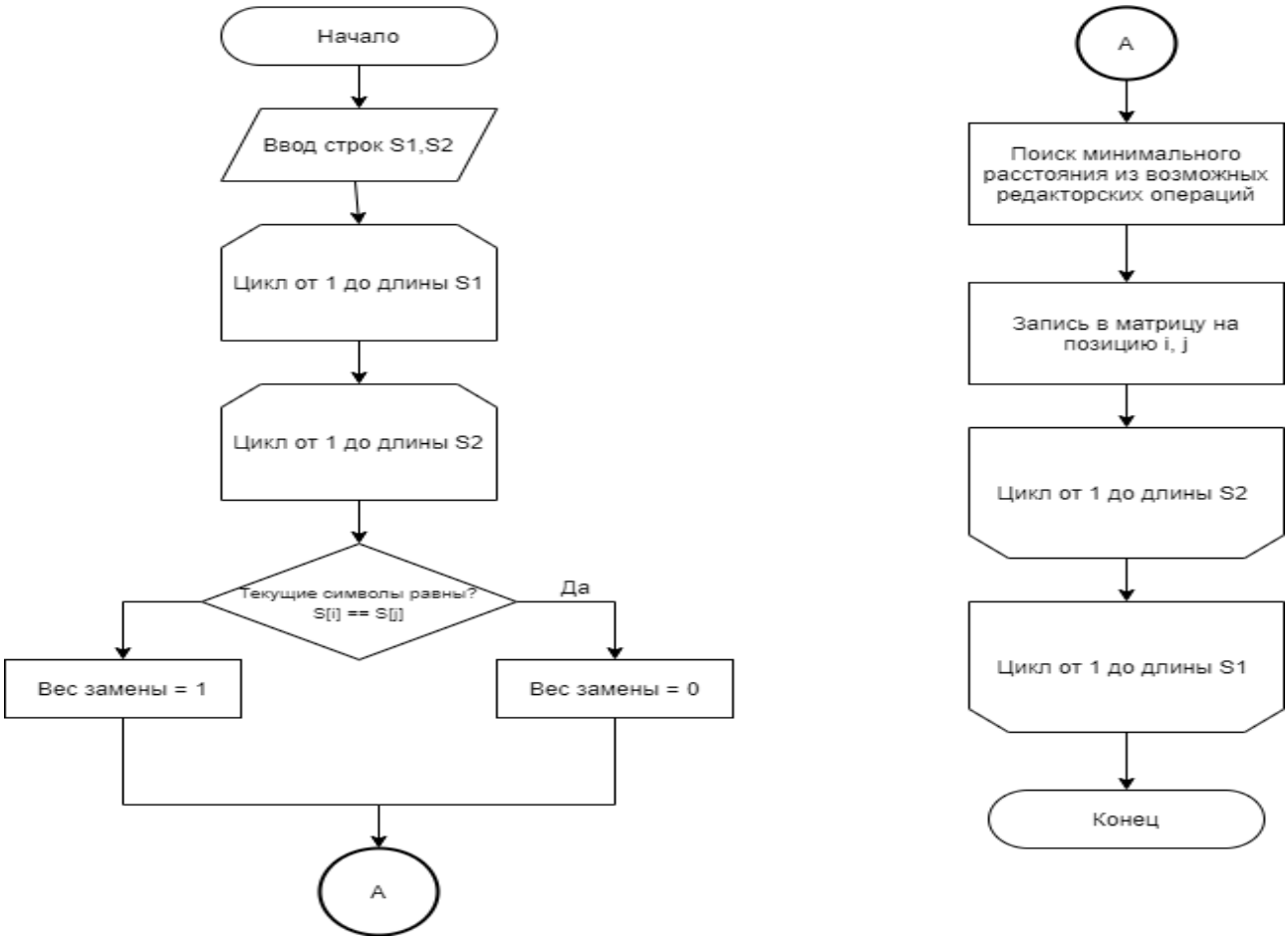

Рис. 1. Схема алгоритма нахождения расстояния Левенштейна. Сделана по реализации кода в приложении.

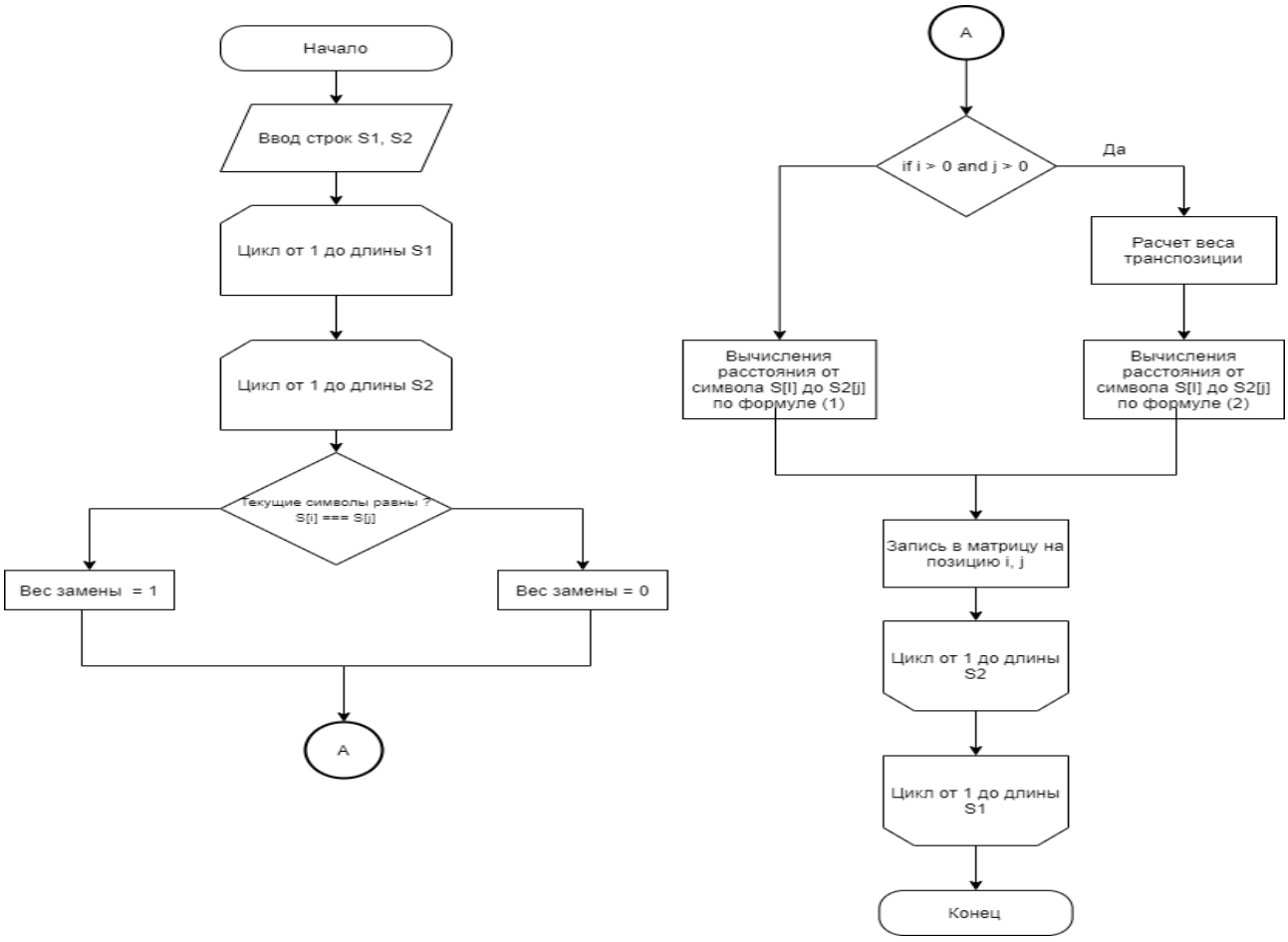

Рис. 2. Схема алгоритма нахождения расстояния Дамерау - Левенштейна. Сделана по реализации кода в приложении.

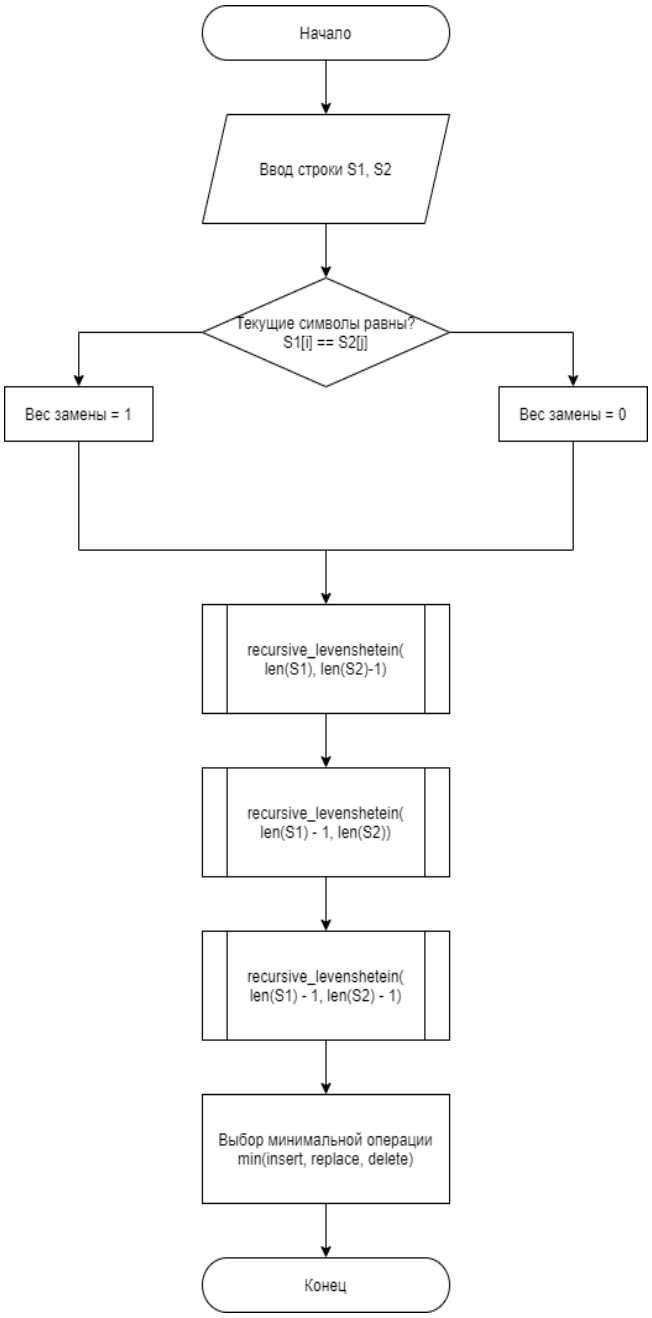

Рис. 3. Схема рекурсивного алгоритма нахождения расстояния Левенштейна. Сделана по реализации кода в приложении.

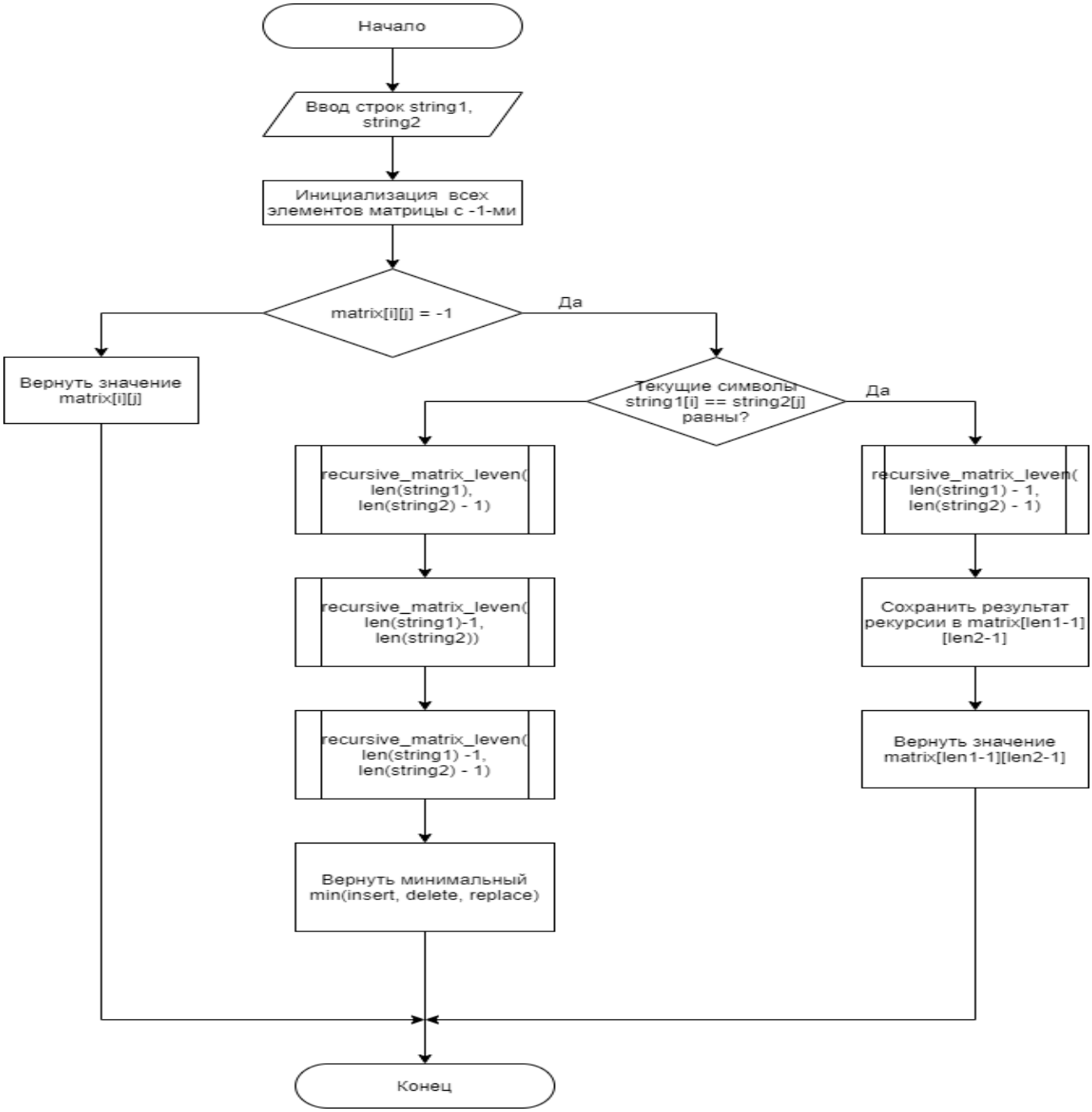

Рис. 4. Схема алгоритма нахождения расстояния Левенштейна с рекурсивным заполнением матрицы.Сделана по реализации кода в приложении.

3. Технологическая часть

3.1. Выбор языка программирования

Python[1] — это высокоуровневый язык программирования, который используется в различных сферах IT, таких как машинное обучение, разработка приложений, web, парсинг и другие. С ним легко работать, что сокращает время разработки. Написанный в удобочитаемом формате, Python делает процесс разработки программного обеспечения быстрым, удобным и максимально упрощенным. По сравнению с другими языками, Python в 5-10 раз быстрее по времени разработки, однако медленный при выполнении программ. Он обеспечивает расширенные возможности управления процессами и объектно-ориентированный дизайн, помогая как в скорости, так и в производительности.

3.2. Методы замера времени в программе

3.2.1. Время

Сушествует несколько способов измерения процессорного времени исполнения программы. Помимо стандратного модуля *time* есть библиотека *timeit* [2]. Этот модуль предоставляет простой способ найти время выполнения маленьких битов кода Python. *timeit* запускает фрагмент кода миллионы раз(значение по умолчанию - 1000000), так что получаем наиболее статистически значимое измерение времени выполнения кода.

3.2.2. Улучшение точности замеров времени

Чтобы получить более точные результаты, каждый тест запускается несколько раз, все полученные значения времени(в тиках) суммируются и делятся на количество запусков кода. Таким образом, получаем среднее время выполнения кода.

4. Экспериментальная часть

4.1. Листинг кода

На Листингах показаны 2 - 5 показаны реализации алгоритмов для нахождения редакционного расстояния.

Листинг 1. Расстояние Левенштейна(итеративная реализация)

```
def non_recursive_levenshtein (string1, string2):
 n1 = len(string1)
 n2 = len(string2)


 d = [[0  for j in range(n2+1)] for i in range(n1+1)]

 for i in range(n2+1):
   d[0][i] = i

 for i in range(n1+1):
   d[i][0] = i

 if n1 == 0 or n2 ==0:
   return max(n1, n2)

 for i in range(1, n1):
   for j in range(1, n2):
     d[i][j] = min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] +
        is_weight(string1[i], string2[j]))


 for i in range(n1+1):
   for j in range(n2+1):
     print(d[i][j], end = ' ')
   print("\n")
```

Листинг 2. Расстояние Левенштейна(рекурсивная реализация)

```
def recursive_leven(s1, s2):
    def recursive(n1, n2):
      if n1 == 0 or n2 == 0:
        return max(n1, n2)

      elif s1[n1-1] == s2[n2-1]:
        return recursive(n1-1, n2-1)
      else:
        return 1 + min(
          recursive(n1, n2 -1), #delete
          recursive(n1-1, n2), #insert
          recursive(n1-1, n2-1) # replace
        )
return recursive(len(s1), len(s2))
```

Листинг 3. Расстояние Левенштейна(рекурсивное заполнение матрицы)

```
  def recursive_matrix_leven(string1, string2, matrix):
    def recursive(n1, n2):
      if n1 == 0 or n2 == 0:
        return max(n1, n2)
      for i in range(n2+1):
        matrix[0][i] = i

      for i in range(n1+1):
        matrix[i][0] = i

      if matrix[n1-1][n2-1] != -1:
        return matrix[n1-1][n2-1]

      if string1[n1-1] == string2[n2-1]:
        matrix[n1-1][n2-1] = recursive(n1-1, n2-1)
        return matrix[n1-1][n2-1]
      matrix[n1-1][n2-1] = 1 + min(recursive(n1, n2-1), recursive(
         n1-1, n2), recursive(n1-1, n2-1))
      return matrix[n1-1][n2-1]
  return recursive(len(string1),len(string2))
```

Листинг 4. Алгоритм для нахождения расстояния Дамерау-Левенштейна(итеративная реализация)

```
    def damerau_levenshtein(string1, string2):
    n1, n2 = len(string1), len(string2)
    if n1 == 0:
      return n2
    if n2 == 0:
      return n1
    d = [[0  for j in range(n2+1)] for i in range(n1+1)]
    for i in range(n2+1):
      d[0][i] = i

    for i in range(n1+1):
      d[i][0] = i
    for i in range(1, n1):
      for j in range(1, n2):
        insert = d[i-1][j]+1
        delete = d[i][j-1]+1
        replace = d[i-1][j-1]+is_weight(string[i-1], string[j-1])
        minimum = min(insert, delete, replace)
        if i>1 and j >1 and
                is_weight(string[i-1], string2[j-2]) and
                is_weight(string[i-1], string2[j-1]) and
                is_weight([i-1][j-1]):
          minimum = min(minimum, d[i-2][j-2]+1)
        d[i][j] = minimum
    return d[n1][n2]
```

4.2. Примеры работы

В листингах 5 - 8 приведены примеры работы алгоритмов.

Листинг 5. Пример работы алгоритмов

```
Input 1 string: aaaaa
Input 2 string: bbb
    b b b

  0 1 2 3

a 1 1 2 3

a 2 2 2 3

a 3 3 3 3

a 4 4 4 4

a 5 5 5 5
Levenshtein: 5
Recursive Levenshtein: 5
Damerau-Levenshtein: 5
Recursive matrix = 5
```

Листинг 6. Пример работы алгоритмов

```
Input 1 string: mama
Input 2 string: papa
    p a p a

  0 1 2 3 4

m 1 1 2 3 4

a 2 2 1 2 3

m 3 3 2 2 3

a 4 4 3 3 2
Levenshtein: 2
Recursive Levenshtein: 2
Damerau-Levenshtein: 2
Recursive matrix = 2
```

Листинг 7. Пример работы алгоритмов

```
Input 1 string: ok
Input 2 string: ko
    k o

  0 1 2

o 1 1 1

k 2 1 2
Levenshtein:  2
Recursive Levenshtein:  2
Damerau-Levenshtein:  2
Recursive matrix =  2
```

Листинг 8. Пример работы алгоритмов

```
Input 1 string: cat
Input 2 string: cat
    c a t

  0 1 2 3

c 1 0 1 2

a 2 1 0 1

t 3 2 1 0
Levenshtein:  0
Recursive Levenshtein:  0
Damerau-Levenshtein:  0
Recursive matrix =  0
```

4.3. Показательное сравнение временных характеристик

Сравнение быстродействия алгоритмов Левенштейна и Дамерау-Левенштейнав матричной и рекурсивной формой в тиках приведено в таблице 1.

Таблица 1. Временные сравнения алгоритмов.

Слова	L	LR	DL	LMR
5	6777	18105180	7421	4715
10	36872	332723388	26163	5407
15	21574	104631506	25570	4546
100	237151	1045942000	186527	5570
200	458068	2297609000	316668	10059
350	1823692	5569742500	664259	18948
500	2064513	27433621866999994	1170698	29770

Характеристики компьютера, на котором были производились замеры времени:

1) операционная система - Майкрософр Windows 10 для образовательных учреждений 10.0.18363

2) процессор - Intel(R) Core(TM) i5-6300U CPU @2.30GHz 2.40 GHz

3) видеокарта - Intel(R) HD Graphics 520

4.4. Показательное сравнение по памяти

Алгоритмы Левенштйна и Дамерау-Левенштейна не отличаются друг от друга с точки зрения использования памяти. Следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов. Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти равен:

$$C(S1) + C(S2) * (2 * C(str) + 3 * C(int)) \tag{3}$$

где C - оператор вычисления размера, $S1, S2$ - строки, int - целочисленный тип, str - строковый тип.

Использование памяти при итеративной реализации теорически равен:

$$C(S1 + 1) + C(S2 + 1) * C(int) + 10 * C(int) + 2 * C(str) \tag{4}$$

Заключение

В ходе работы были сделаны следующие выводы:

- рекурсивная реализация алгоритма Левенштейна и Дамерау-Левенштейна выполняются быстрее только в случаях, когда замер одной из строк крайне мал;
- итеративные реализации алгоритмов поиска расстояний Дамерау-Левенштейна и Левенштейна имеют схожую конструкцию, но алгоритм поиска расстояния Дамерау-Левенштейна из-за более сложной внутренней логики в среднем работает медленнее.
- Итеративный алгоритм поиска расстояний Левенштейна и Дамерау-Левенштейна являются намного эффективнее по времени, чем их рекурсивные реализации.

Была достигнута цель и решены следующие задачи:

1) были рассмотрены алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна для нахождения редакционного расстояния между строками;
2) были реализованы данные алгоритмы на языке программирования Python;
3) был проведен сравнительный анализ алгоритмов по затраченному времени и памяти;
4) были приведены примеры работы всех указанных алгоритмов;

Список литературы

[1] Python:что нужно знать. [ЭЛ. РЕСУРС]
Режим доступа: https://skillbox.ru/media/code/story_buzunov/
(Дата обращения: 05.10.2020)

[2] Timeit в Python с примерами. [ЭЛ. РЕСУРС]
Режим доступа: http://espressocode.top/timeit-python-examples/
(Дата обращения: 05.10.2020)

[3] Расстояние Левенштейна на Python. [ЭЛ. РЕСУРС]
Режим доступа: https://tirinox.ru/levenstein-python/
(Дата обращения: 05.10.2020)

[4] Программная реализация алгоритма Левенштейна для устранения опечаток в записях баз данных.[ЭЛ. РЕСУРС]
Режим доступа: https://moluch.ru/archive/19/1966/
(Дата обращения: 05.10.2020)