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»

Лабораторная работа № 1

Дисциплина Анализ алгоритмов

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Куликов Д. А.

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Волкова Л. Л.

Москва — 2020 г.

Оглавление

Введение

Расстояние Левенштейна — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены [1].

Расстояние Левенштейна широко используется в теории информации и компьютерной лингвистике. Примеры использования:

- исправление ошибок в слове;
- в биоинформатике для сравнения генов, хромосом и белков;
- сравнение текстовых файлов утилитой diff и ей подобными.

Цель работы: изучение и применение на практике алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также получение практических навыков реализации этих алгоритмов и сравнении их между собой.

Задачи работы:

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, выполненного как расчётно-пояснительная записка к работе.

1 | Аналитическая часть

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

При нахождении расстояния Дамерау-Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Операции обозначаются следующим образом:

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

Операция M имеет штраф 0, другие операции — штраф 1.

1.1. Описание расстояний

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:

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

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

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

2 | Конструкторская часть

В этом разделе содержатся схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и сравнительный анализ матричной и рекурсивной реализаций.

2.1. Схемы алгоритмов

На рис. 2.1-2.4 приведены схемы следующих алгоритмов:

1) матричный алгоритм нахождения расстояния Левенштейна;
2) рекурсивный алгоритм нахождения расстояния Левенштейна;
3) рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы;
4) матричный алгоритм нахождения расстояния Дамерау-Левенштейна.

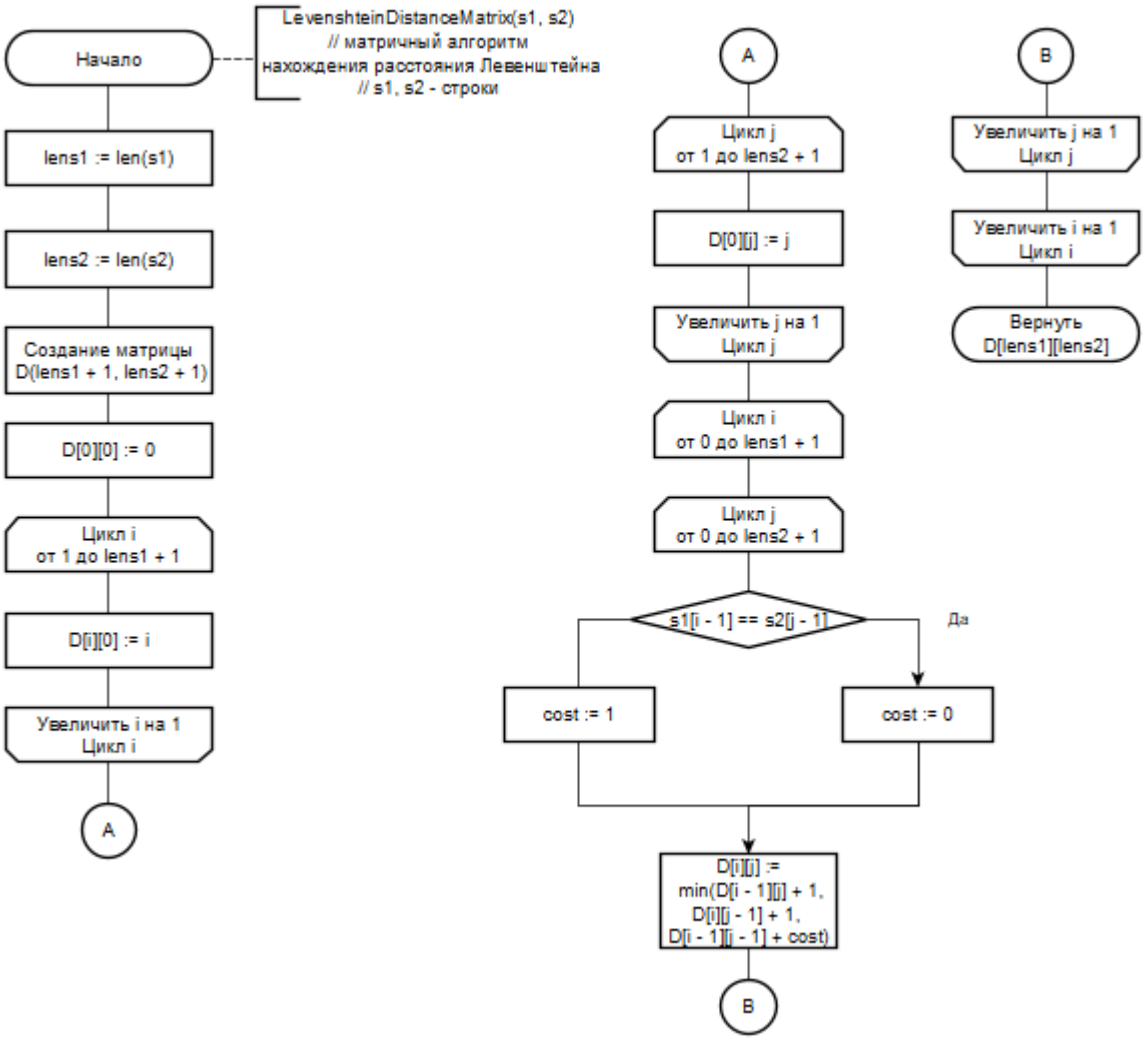

Рис. 2.1. Матричный алгоритм нахождения расстояния Левенштейна

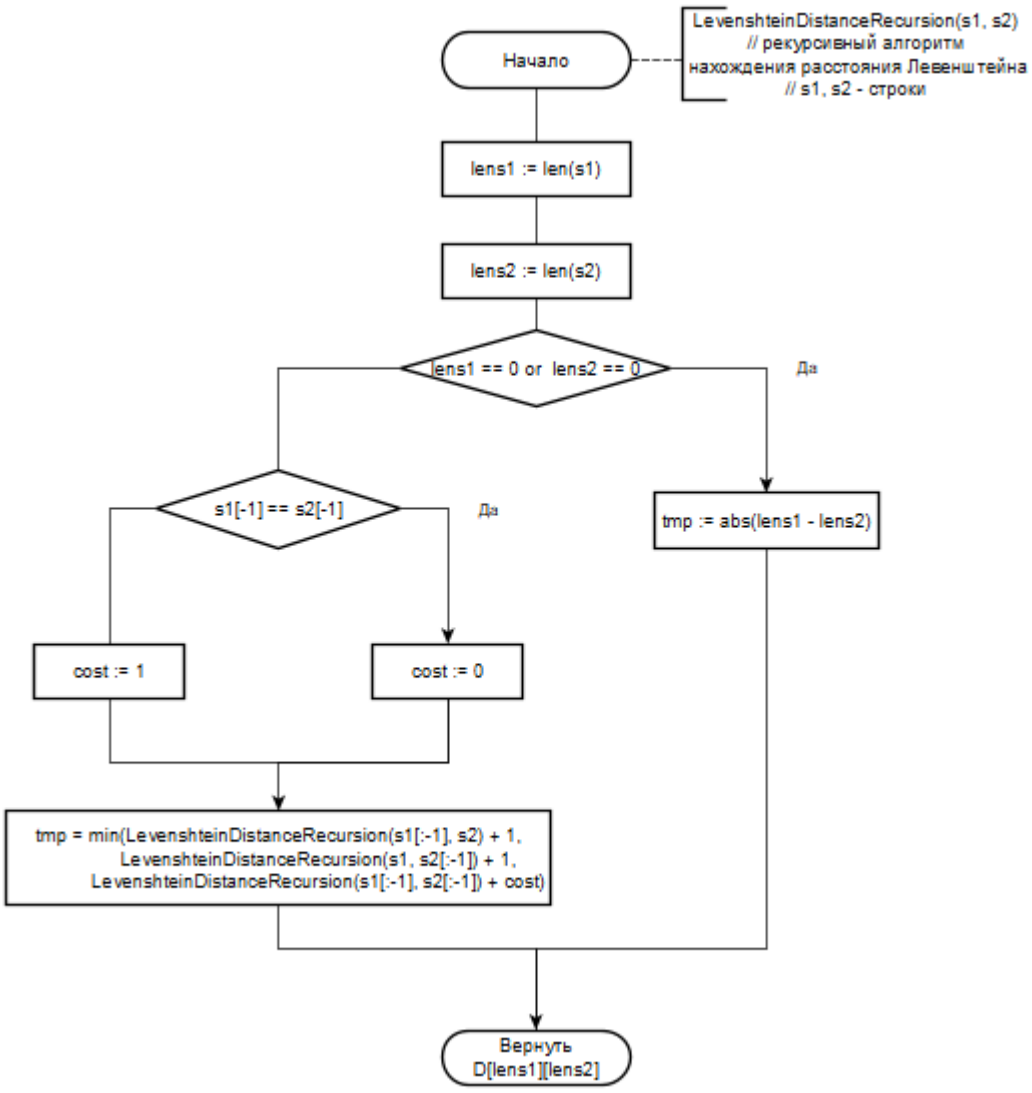

Рис. 2.2. Рекурсивный алгоритм нахождения расстояния Левенштейна

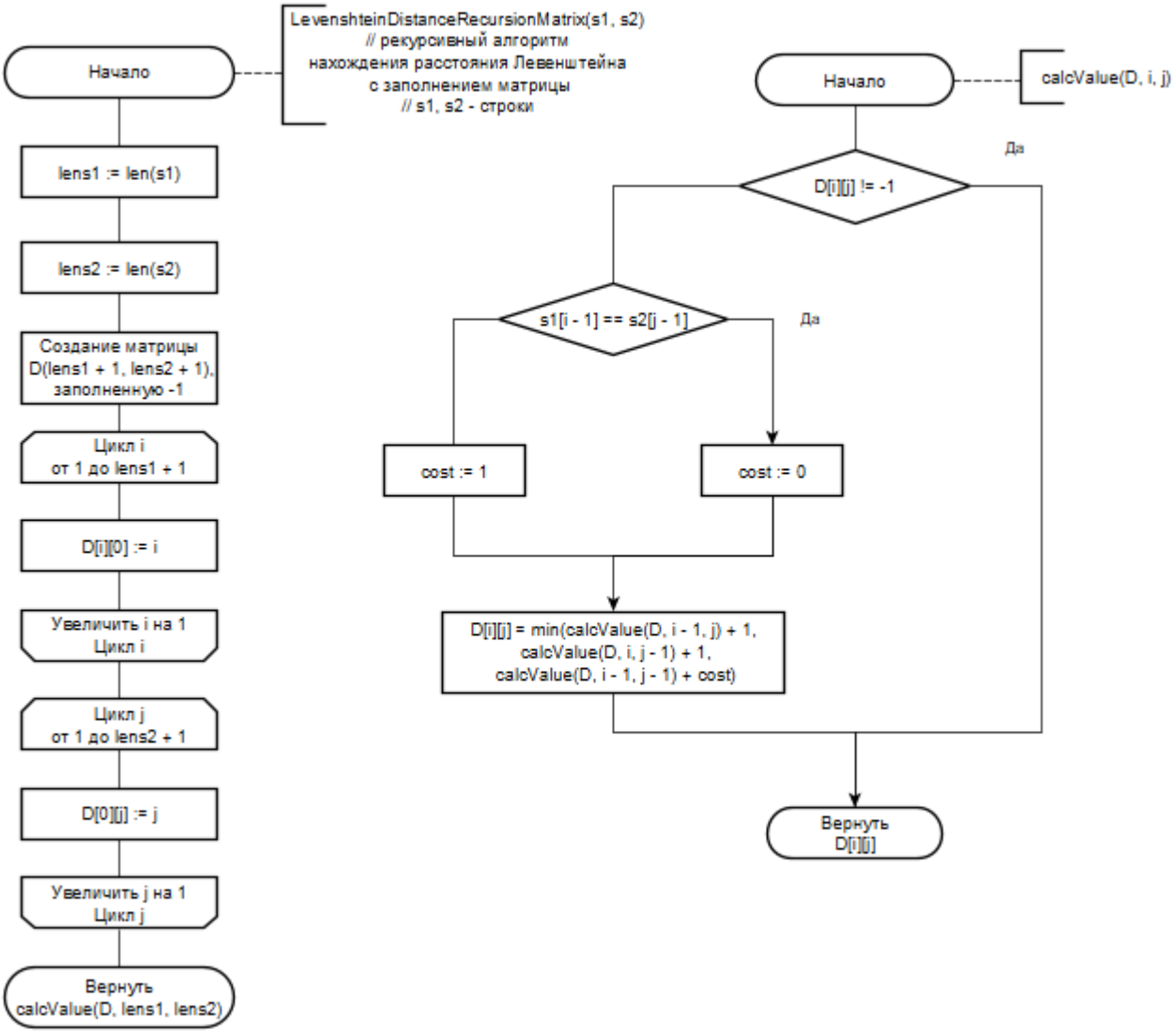

Рис. 2.3. Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

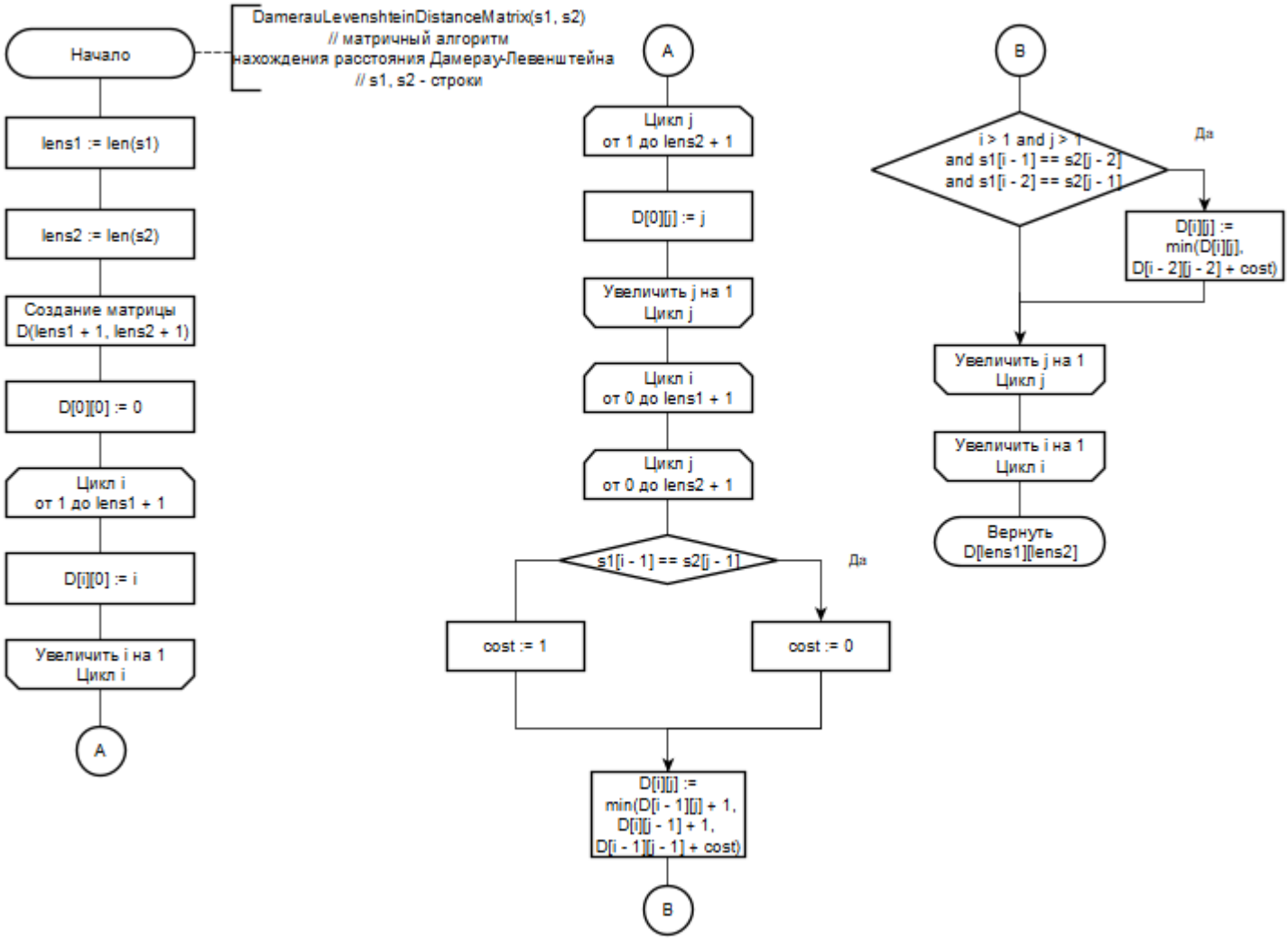

Рис. 2.4. Рекурсивный алгоритм нахождения расстояния Левенштейна

3 | Технологическая часть

В данном разделе будут рассмотрены требования к программному обеспечению, средства реализации, представлен листинг кода, сравнительный анализ матричной и рекурсивной реализаций, описание тестирования.

3.1. Требования к программному обеспечению

Требования к вводу:

1) на вход подаются две строки;

2) uppercase и lowercase буквы считаются разными.

Требования к программе:

1) две пустые строки — корректный ввод, программа не должна аварийно завершаться;

2) на выходе необходимо получить число, являющиеся результатом работы алгоритма;

3) для матричных реализаций требуется вывести матрицу решений;

4) требуется замерить время работы каждого из алгоритмов.

3.2. Средства реализации

В качестве языка программирования был выбран Python т.к. я знаком с данным языком, он простой и лаконичный, имеющий немногословный и понятный синтаксис, похожий на псевдокод, обладающий сильной динамической типизацией, которая способствует быстрому написанию кода.

Среда разработки — PyCharm, которая предоставляет умную проверку кода, быстрое выявление ошибок и оперативное исправление, вкупе с автоматическим рефакторингом кода, и богатыми возможностями в навигации.

Матрица создается с помощью функции eye(m, n) из библиотеки numpy [2].

Время работы алгоритмов было замерено с помощью функции process_time() из библиотеки time [3].

3.3. Листинг кода

В листингах 3.1-3.4 представлена реализация алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна

Листинг 3.1. Матричный алгоритм нахождения расстояния Левенштейна

```
def LevenshteinDistanceMatrix(s1, s2, printMatrix = False):
  lens1 = len(s1)
  lens2 = len(s2)
  D = np.eye(lens1 + 1, lens2 + 1)
  D[0][0] = 0
  for i in range(1, lens1 + 1):
  D[i][0] = i
  for j in range(1, lens2 + 1):
  D[0][j] = j
  for i in range(1, lens1 + 1):
    for j in range(1, lens2 + 1):
      cost = 0 if s1[i - 1] == s2[j - 1] else 1
      D[i][j] = min(D[i - 1][j] + 1,
                D[i][j - 1] + 1, D[i - 1][j - 1] + cost)

  if printMatrix:
    print('Matrix:')
    outputMatrix(s1, s2, D)
    operations(s1, s2, D)

  return D[lens1][lens2]
```

Листинг 3.2. Рекурсивный алгоритм нахождения расстояния Левенштейна

```
def LevenshteinDistanceRecursion(s1, s2):
  lens1 = len(s1)
  lens2 = len(s2)
  if lens1 == 0 or lens2 == 0:
    tmp = abs(lens1 - lens2)
  else:
    cost = 0 if s1[-1] == s2[-1] else 1
    tmp = min(LevenshteinDistanceRecursion(s1[:-1], s2) + 1,
    LevenshteinDistanceRecursion(s1, s2[:-1]) + 1,
    LevenshteinDistanceRecursion(s1[:-1], s2[:-1]) + cost)
  return tmp
```

Листинг 3.3. Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

```
def LevenshteinDistanceRecursionMatrix(s1, s2):
  def calcValue(D, i, j):
    if D[i][j] != -1:
      return D[i][j]
    else:
      cost = 0 if s1[i - 1] == s2[j - 1] else 1
      D[i][j] = min(calcValue(D, i - 1, j) + 1,
      calcValue(D, i, j - 1) + 1,
      calcValue(D, i - 1, j - 1) + cost)
      return D[i][j]

  lens1 = len(s1)
  lens2 = len(s2)
  D = np.full((lens1 + 1, lens2 + 1), -1)
  for i in range(lens1 + 1):
    D[i][0] = i
  for j in range(lens2 + 1):
    D[0][j] = j

  return calcValue(D, lens1, lens2)
```

Листинг 3.4. Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

```
def DamerauLevenshteinDistanceMatrix(s1, s2, printMatrix = False):
  lens1 = len(s1)
  lens2 = len(s2)
  D = np.eye(lens1 + 1, lens2 + 1)
  D[0][0] = 0
  for i in range(1, lens1 + 1):
    D[i][0] = i
  for j in range(1, lens2 + 1):
    D[0][j] = j
  for i in range(1, lens1 + 1):
    for j in range(1, lens2 + 1):
    cost = 0 if s1[i - 1] == s2[j - 1] else 1
    D[i][j] = min(D[i - 1][j] + 1,
              D[i][j - 1] + 1, D[i - 1][j - 1] + cost)
    if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and
     s1[i - 2] == s2[j - 1]:
      D[i][j] = min(D[i][j], D[i - 2][j - 2] + cost)

  if printMatrix:
    print('Matrix:')
    outputMatrix(s1, s2, D)
    operations(s1, s2, D)
  return D[lens1][lens2]
```

3.4. Сравнительный анализ матричной и рекурсивной реализаций

Рекурсивная реализация работает медленнее по сравнению с матричной из-за повторных вычислений, возникающих в ходе работы рекурсивного алгоритма. Это наглядно видно на Рис. 3.1, иллюстрирующем дерево рекурсивных вызовов. При каждом рекурсивном вызове необходимо передавать в функцию подстроки исходных строк, что затратно по памяти. Эту проблему возможно избежать, если занести строки в глобальные переменные, что является плохой практикой.

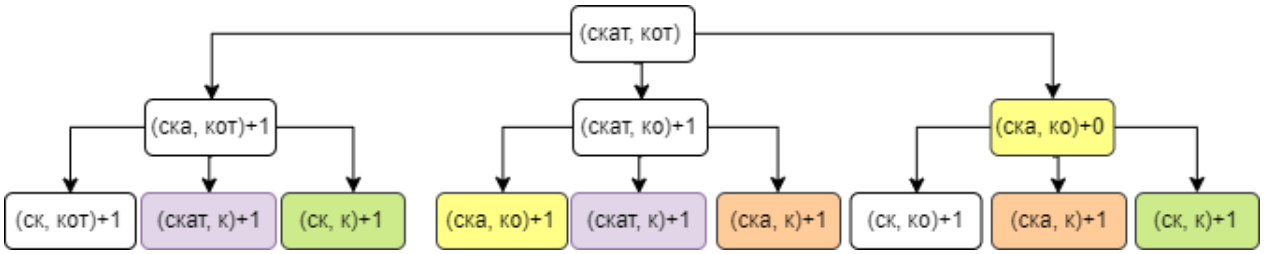

Рис. 3.1. Дерево рекурсивных вызовов

Для сравнения, вызов функции, которая реализует матричный алгоритм, происходит один раз и в функцию только один раз передаются обе строки. В этом алгоритме память будет затрачена на хранение матрицы, а время на вложенные циклы, однако затраты будут существенно меньше, чем при многократных вызовах рекурсивной функции.

Пусть длина строки S1 - n, длина строки S2 - m, тогда затраты памяти на приведенные выше алгоритмы будут следующими. Затраты памяти матричного алгоритма нахождения расстояния Левенштейна:

- строки S1, S2 - (m + n) * sizeof(char)
- матрица - ((m + 1) * (n + 1)) * sizeof(int)
- длины строк - 2 * sizeof(int)
- вспомогательная переменная - sizeof(int)

Затраты памяти рекурсивного алгоритма нахождения расстояния Левенштейна(для каждого вызова):

- строки S1, S2 - (m + n) * sizeof(char)
- длины строк - 2 * sizeof(int)
- вспомогательные переменные - 2 * sizeof(int)
- адрес возврата

Чтобы получить итоговую оценку затрачиваемой памяти необходимо затрачиваемую память для одного вызова умножить на максимальную глубину рекурсии, которая равна сложению длин входных строк.

Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы:

- матрица - ((m + 1) * (n + 1)) * sizeof(int)
- строки S1, S2 - (m + n) * sizeof(char)

Для каждого рекурсивного вызова:

- передача данных - 2 * sizeof(int*) + sizeof(int**)
- вспомогательная переменная - sizeof(int)
- адрес возврата

Чтобы получить итоговую оценку затрачиваемой памяти на рекурсивные вызовы необходимо затрачиваемую память для одного рекурсивного вызова умножить на максимальную глубину рекурсии, которая равна сложению длин входных строк.

Затраты памяти матричного алгоритма нахождения расстояния Дамерау-Левенштейна:

- строки S1, S2 - (m + n) * sizeof(char)
- матрица - ((m + 1) * (n + 1)) * sizeof(int)
- длины строк - 2 * sizeof(int)
- вспомогательная переменная - sizeof(int)

3.5. Описание тестирования

Реализовано модульное тестирование отдельным файлом test.py с помощью библиотеки unittest [4]. Полученные результаты функций сравниваются с контрольными значениями.

Тестирование происходит по следующим данным:

1) проверка работы с пустыми строками;
2) проверка работы с идентичными строками;
3) проверка работы со строками, имеющие совпадающие символы;
4) проверка работы с полностью несовпадающими строками.

4 | Экспериментальная часть

В данном разделе приведены примеры работы программы и сравнительный анализ алгоритмов на основе экспериментальных данных.

4.1. Примеры работы

На рис. 4.1 представлено главное меню программы.

```
Меню:
1) Расстояние Левенштейна матричный алгоритм
2) Расстояние Левенштейна рекурсивный алгоритм по формуле
3) Расстояние Левенштейна рекурсивный алгоритм, заполняющий матрицу
4) Расстояние Дамерау-Левенштейна матричный алгоритм
5) Сравнение по времени расстояние Левенштейна

Выберите пункт меню:
```

Рис. 4.1. Главное меню программы

На рис. 4.2-4.5 приведены примеры работы программы при вводе строк «тело» и «столб» при выборе пунктов меню 1-4.

```
Выберите пункт меню: 1
Введите первую строку: тело
Введите вторую строку: столб
Matrix:
    с т о л б
  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 3
Operation:
I M R M R
3.0
```

Рис. 4.2. Результат работы матричного алгоритма нахождения расстояния Левенштейна

```
Выберите пункт меню: 2
Введите первую строку: тело
Введите вторую строку: столб
3
```

Рис. 4.3. Результат работы рекурсивного алгоритма нахождения расстояния Левенштейна

```
Выберите пункт меню: 3
Введите первую строку: тело
Введите вторую строку: столб
3
```

Рис. 4.4. Результат работы рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы

```
Выберите пункт меню: 4
Введите первую строку: тело
Введите вторую строку: столб
Matrix:
    с т о л б
  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 3
Operation:
I M R M R
3.0
```

Рис. 4.5. Результат работы матричного алгоритма нахождения расстояния Дамерау-Левенштейна

4.2. Результаты тестирования

Было проведено тестирование программы, результаты которого занесены в Таблицу 4.1, 1 столбец которой - номер тестового случая; 2 и 3 столбцы - строки, поступающие на вход; 4 и 5 столбец - ожидаемый и полученный результат, в которых приняты следующие обозначения:

- 1-ая цифра – результат работы матричного алгоритма нахождения расстояния Левенштейна
- 2-ая цифра – результат работы рекурсивного алгоритма нахождения расстояния Левенштейна
- 3-ая цифра – результат работы рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы;
- 4-ая цифра – результат работы матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

Таблица 4.1. Результаты тестирования

№	S1	S2	Ожидаемый результат	Полученный результат
1	пустая строка	пустая строка	0, 0, 0, 0	0, 0, 0, 0
2	кот	пустая строка	3, 3, 3, 3	3, 3, 3, 3
3	пустая строка	кот	3, 3, 3, 3	3, 3, 3, 3
4	кот	кот	0, 0, 0, 0	0, 0, 0, 0
5	кот	кит	1, 1, 1, 1	1, 1, 1, 1
6	от	кот	1, 1, 1, 1	1, 1, 1, 1
7	кот	кота	1, 1, 1, 1	1, 1, 1, 1
8	кот	кто	2, 2, 2, 1	2, 2, 2, 1
9	кот	рабы	4, 4, 4, 4	4, 4, 4, 4

Программа успешно прошла все тестовые случаи, все полученные результаты совпали с ожидаемыми.

4.3. Постановка эксперимента по замеру времени

Для произведения замеров времени выполнения реализации алгоритмов будет использована формула:

$$t = \frac{T}{N} \tag{4.1}$$

где t — среднее время выполнения алгоритма, N — количество замеров, T — время выполнения N замеров. Неоднократное измерение времени необходимо для получения более точного результа.

Количество замеров взято равным 100. Эксперимент проводятся на рандомных строках одинаковой длины.

4.4. Сравнительный анализ на материале экспериментальных данных

Для сравнения скорости работы необходимо провести замеры процессорного времени для строк одинаковой длины.

Был проведены замеры времени работы каждого из алгоритмов, результаты которых занесены в Таблицу 4.2, столбцы которой обозначают следующее:

- 1 столбец – длина строк;
- 2 столбец – время работы матричного алгоритма нахождения расстояния Левенштейна;
- 3 столбец – время работы рекурсивного алгоритма нахождения расстояния Левенштейна;
- 4 столбец – время работы рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы;
- 5 столбец – время работы матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

Таблица 4.2. Сравнение алгоритмов по времени(в секундах)

len	LevMatr	LevRec	LevRecMatr	DamLevMatr
3	0.00000750	0.00003239	0.00002969	0.00000837
5	0.00008750	0.00123438	0.00012969	0.00008906
6	0.00009050	0.00155438	0.00014069	0.00009596
8	0.00024553	2.00000438	0.00035378	0.00026596
10	0.00029531	5.73437500	0.00046875	0.00032188

Вывод: У матричных алгоритмов нахождения расстояния Левенштейна время работы пропорционально квадрату длины строк (при увеличении строки в два раза, время увеличивается в четыре раза). Рекурсивный алгоритм по нахождению расстояние Левенштейна показывает наихудшее время, что связано с повторным вычислением одних и тех же значений (пример рис 2.5). Матричный алгоритм по нахождению расстояния Левенштейна и Дамерау-Левенштейна показывают практически одинаковое значение, однако необходимо учитывать то, что Дамерау-Левенштейн решает другую задачу.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна. Реализованы 4 алгоритма поиска этих расстояний, приведен программный код реализации алгоритмов нахождения расстояний.

Было выполнено сравнение разных алгоритмов нахождения расстояния Левенштейна по затраченным ресурсам. Было установлено, что рекурсивный алгоритм занимает гораздо меньше памяти при работе со строками большой длины, чем матричные алгоритмы. Однако матричные алгоритмы отмечаются своим быстродействием.

Цель работы достигнута. Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна применить на практике, получены практические навыки реализации этих алгоритмов.

Список литературы

1. Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. – М.: Техносфера, 2017. – 267 с.

2. Библиотека numpy Python, документация [электронный ресурс] – Режим доступа: https://pythonworld.ru/numpy.html, свободный – (Дата обращения: 16.09.20)

3. Официальный сайт Python, документация [электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html, свободный (Дата обращения: 16.09.20)

4. Официальный сайт Python, документация [электронный ресурс] – Режим доступа: https://docs.python.org/3/library/unittest.html, свободный – (Дата обращения: 16.09.20)