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»

Отчет по лабораторной работе №1 по дисциплине "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Петрова А.А.

Группа ИУ7-56Б

Оценка (баллы) ________________

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

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

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистики для:

- исправления ошибок в слове;
- сравнения текстовых файлов утилитой diff;
- в биоинформатике для сравнения генов, хромосом и белков.

Целью работы является реализация и оценка алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить алгоритмы Левенштейна и Дамерау-Левенштейна (аналитическая часть);
- создать схемы указанных алгоритмов (матричных и рекусривных) (конструкторская часть);
- реализовать эти алгоритмы (технологическая часть);
- провести анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти) (исследовательская часть);
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка (insert), удаление (delete), замена (replace) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

- $w(a, b)$ — цена замены символа a на символ b.
- $w(\lambda, b)$ — цена вставки символа b.
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

- $w(a, a) = 0$.
- $w(a, b) = 1,\ a \neq b$.
- $w(\lambda, b) = 1$.
- $w(a, \lambda) = 1$.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i, j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \mathrm{i = 0,\ j = 0} \\ i & \mathrm{j = 0,\ i > 0} \\ j & \mathrm{i = 0,\ j > 0} \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \mathrm{i > 0,\ j > 0} \\ \qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases}, \tag{1.1}$$

а функция 1.2 определена как:

$$m(a,b) = \begin{cases} 0 & \text{если a = b,} \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена из следующих соображений:

1. Для перевода из пустой строки в пустую требуется ноль операций;
2. Для перевода из пустой строки в строку a требуется $|a|$ операций;
3. Для перевода из строки a в пустую требуется $|a|$ операций;
4. Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:
   (a) Сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;
   (b) Сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b;
   (c) Сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;
   (d) Цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

   Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы

$A_{|a|,|b|}$ значениями $D(i, j)$.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

1.4 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна [2] может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \qquad d_{a,b}(i,j-1)+1, & \\ \qquad d_{a,b}(i-1,j)+1, & \\ \qquad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе} \\ \qquad \left[\begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.1). Как и в случае с рекурсивным методом, прямое применение этой формулы неэффективно по времени исполнения, то аналогично методу из 1.3 производится добавление матрицы для хранения промежуточных значений рекурсивной формулы.

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов. Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 | Конструкторская часть

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояние Левештейна и Дамерау - Левенштейна. На рисунках 2.1 - 2.4 представлены рассматриваемые алгоритмы.

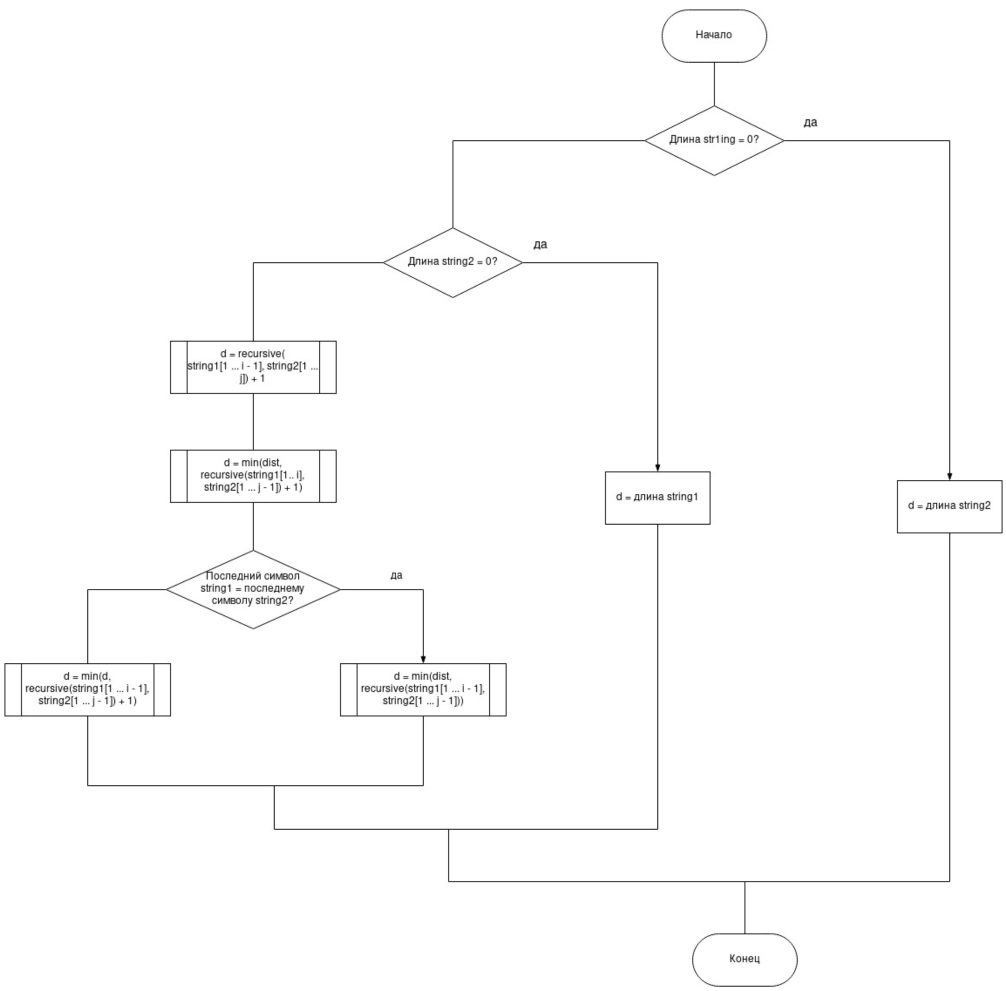

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

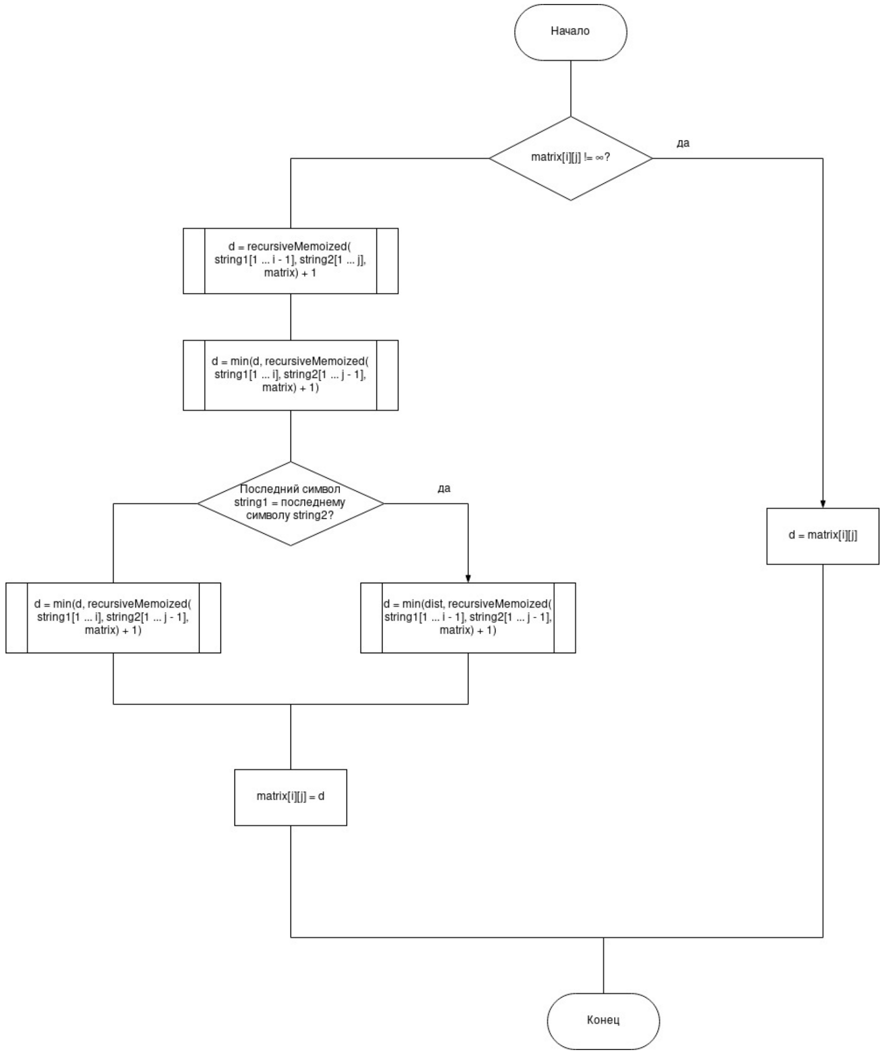

Рис. 2.2: Схема рекурсивного алгоритма Левенштейна с кэшем

2.2 Вывод

На основе теоретических данных, полученные в аналитическом разделе были построены схемы иследуеммых алгоритмов.

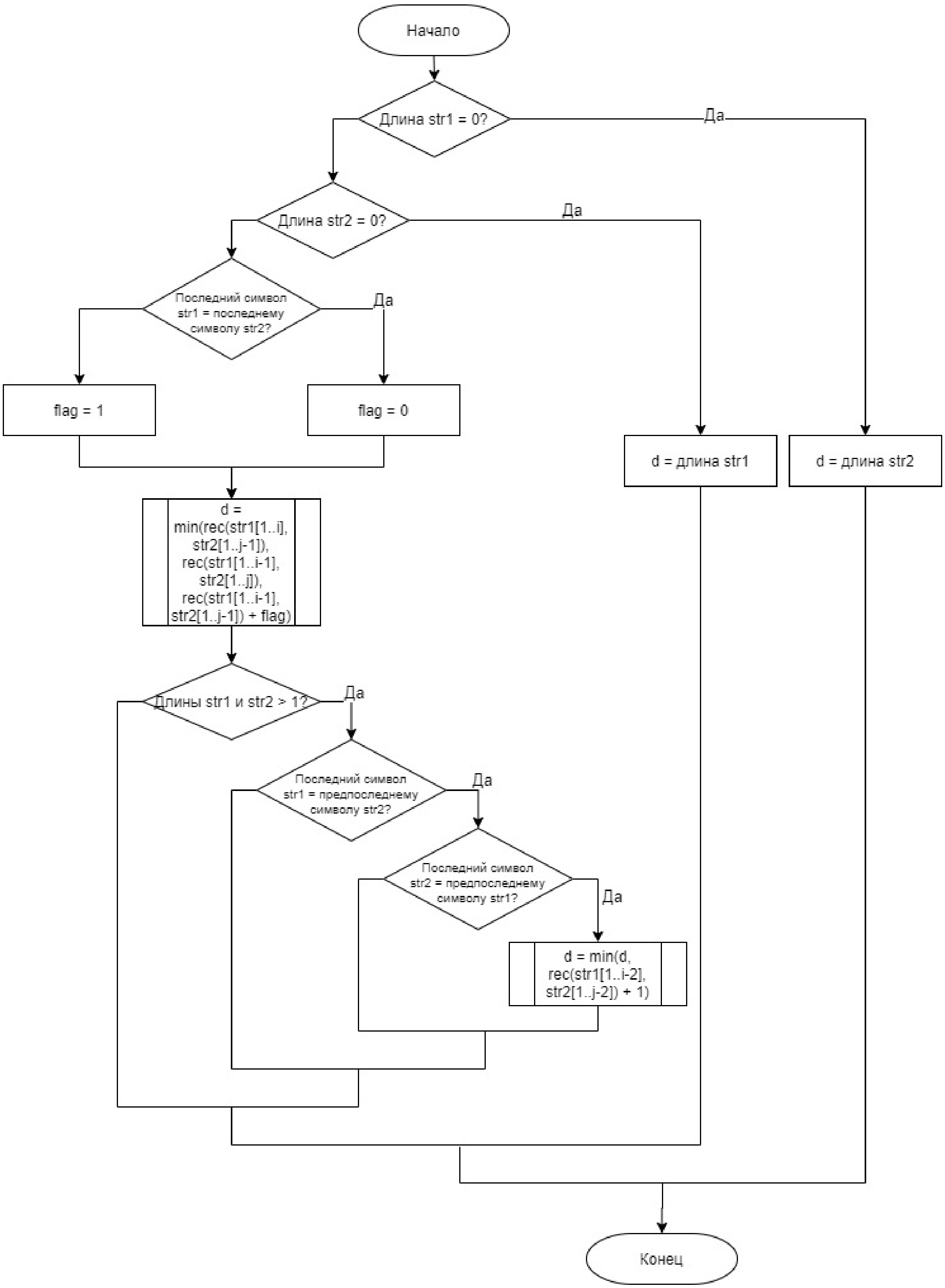

Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

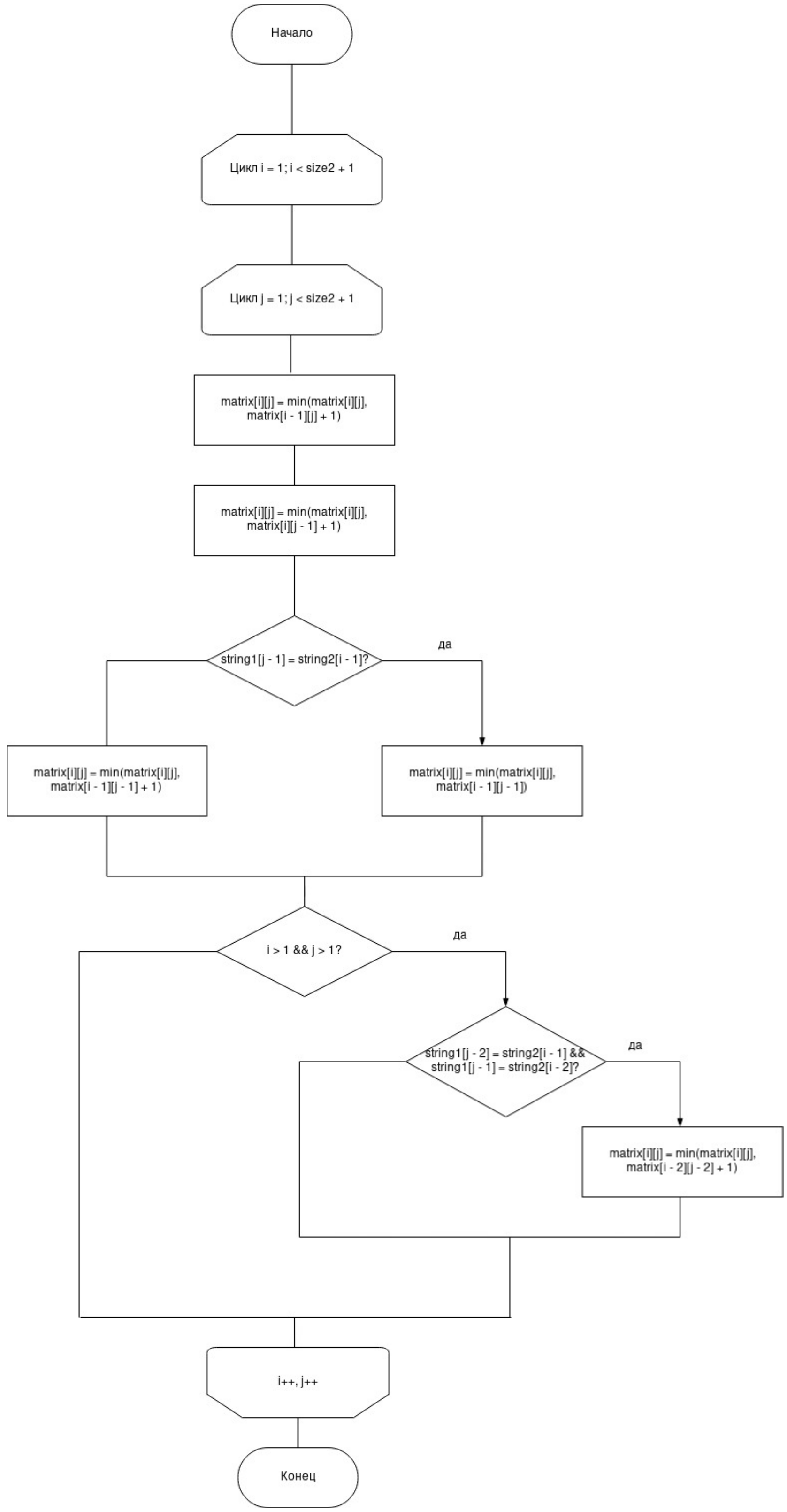

Рис. 2.4: Схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

3.1 Требования к ПО

Требования к вводу:

- на вход подаются две строки в любой раскладке (в том числе и пустые);
- ПО должно выводить полученное расстояние и использованные матрицы;
- ПО должно выводить потраченное время.

Требования к самому ПО:

- ПО должно содержать 2 раздела: пользовательский (ручной ввод) и экспериментальный (для замеров времени);
- в пользовательском разделе должна присутствовать проверка на некорректные данные;
- пустая строка в поле для ввода строки должна считаться корректным данным.

3.2 Средства реализации

Для реализации программы нахождения расстояния Левенштейна был выбран язык программирования Python. Данный выбор обусловлен тем, что этот язык наиболее удобен для работы со строками, а также тем, что в нём присутсвует функция для измерения процессорного времени.

3.3 Реализация алгоритмов

В листингах 3.1 - 3.4 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def lev_recursion(str1, str2, len1, len2):
  if (len1 == len2) and len1 == 0:
    return 0
  elif len1 == 0:
    return len2
  elif len2 == 0:
```

```
    return len1
  else:
    flag = bool(not (str1[len1 - 1] == str2[len2 - 1]))
  return min(min(lev_recursion(str1, str2, len1 - 1, len2) + 1,
           lev_recursion(str1, str2, len1, len2 - 1) + 1),
           lev_recursion(str1, str2, len1 - 1, len2 - 1) + flag)
```

Листинг 3.2: Функция нахождения расстояние Левенштейна рекурсивно с помощью кэша

```
def lev_cache(str1, str2, len1, len2, mtx):
  if not mtx[len1][len2] == 0:
    return mtx[len1][len2]
  elif (len1 == len2) and (len1 == 0):
    mtx[len1][len2] = 0
  elif len1 == 0:
    mtx[len1][len2] = len2
  elif len2 == 0:
    mtx[len1][len2] = len1
  else:
    flag = bool(not(str1[len1 - 1] == str2[len2 - 1]))
    mtx[len1][len2] = min(min(lev_cache(str1, str2, len1 - 1, len2, mtx) + 1,
                 lev_cache(str1, str2, len1, len2 - 1, mtx) + 1),
                 lev_cache(str1, str2,  len1 - 1, len2 - 1, mtx) + flag)
  return mtx[len1][len2]

def rec_lev_cache(str1, str2, len1, len2):
  mtx = [[0 for x in range(len2 + 1)] for y in range(len1 + 1)]
  return lev_cache(str1, str2, len1, len2, mtx)
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def lev_damerau_matrix(str1, str2, len1, len2):
  mtx = [[0 for x in range(len2 + 1)] for y in range(len1 + 1)]
  for i in range(len2 + 1):
    mtx[0][i] = i
  for i in range(1, len1 + 1):
    mtx[i][0] = i
  for i in range(1, len1 + 1):
    for j in range(1, len2 + 1):
      add, delete, change = mtx[i - 1][j] + 1, mtx[i][j - 1] + 1, mtx[i - 1][j - 1]
      if str2[j - 1] != str1[i - 1]:
        change += 1
      mtx[i][j] = min(add, delete, change)
      if ((i > 1 and j > 1) and str1[i - 1] == str2[j - 2] and str1[i - 2] == str2[j - 1]):
        mtx[i][j] = min(mtx[i][j], mtx[i - 2][j - 2] + 1)
  return mtx[len1][len2]
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def lev_damerau_recursion(str1, str2, len1, len2):
  if (len1 == len2) and len1 == 0:
    return 0
  elif len1 == 0:
    return len2
  elif len2 == 0:
    return len1
  else:
    flag = bool(not(str1[len1 - 1] == str2[len2 - 1]))
  res = min(lev_damerau_recursion(str1, str2, len1 - 1, len2) + 1,
        lev_damerau_recursion(str1, str2, len1, len2 - 1) + 1,
        lev_damerau_recursion(str1, str2, len1 - 1, len2 - 1) + flag)
  if (len1 >= 2 and len2 >= 2 and str1[len1 - 1] == str2[len2 - 2] and
     str1[len1 - 2] == str2[len2 - 1]):
    res = min(res, lev_damerau_recursion(str1, str2, len1 - 2, len2 - 2) +
        1)
  return res
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестированно разработанное ПО. Как видно из этой таблицы, все тесты были успешно пройдены, что означает, что программа работает правильно.

Таблица 3.1: Таблица тестовых данных

№	Первая строка	Вторая строка	Ожидаемый результат	Полученный результат
1			0 0 0 0	0 0 0 0
2	кот	скат	2 2 2 2	2 2 2 2
3	утопия	топлес	4 4 4 4	4 4 4 4
4	каска	такса	3 3 2 2	3 3 2 2
5	собака	сбоку	3 3 3 3	3 3 3 3
6	qwerty	queue	4 4 4 4	4 4 4 4
7	apple	aplpe	2 2 1 1	2 2 1 1
8		кот	3 3 3 3	3 3 3 3
9	Linkin Park		11 11 11 11	11 11 11 11

3.5 Вывод

В данном разделе были разработаны исходные коды четырех алгоритмов: вычисления расстояния Левенштейна рекурсивно и рекурсивно с использованием кэша, а также вычисления расстояния Дамерау — Левенштейна итеративно и рекурсивно.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

```
Меню:
1. Ручной ввод
2. Замеры процессорного времени выполнения алгоритмов
0. Выход

Выберите пункт меню: 1

Первая строка: кот
Вторая строка: скат

Рекурсивный алгоритм Левенштейна: 2
Рекурсивный алгоритм Левенштейна с кэшем: 2
Рекурсивный алгоритм Дамерау-Левенштейна: 2
0 1 2 3 4
1 1 1 2 3
2 2 2 2 3
3 3 3 3 2
Нерекурсивный алгоритм Дамерау-Левенштейна (с матрицей): 2

Меню:
1. Ручной ввод
2. Замеры процессорного времени выполнения алгоритмов
0. Выход

Выберите пункт меню: 1

Первая строка: каска
Вторая строка: такса

Рекурсивный алгоритм Левенштейна: 3
Рекурсивный алгоритм Левенштейна с кэшем: 3
Рекурсивный алгоритм Дамерау-Левенштейна: 2
0 1 2 3 4 5
1 1 2 2 3 4
2 2 1 2 3 3
3 3 2 2 2 3
4 4 3 2 2 3
5 5 4 3 3 2
Нерекурсивный алгоритм Дамерау-Левенштейна (с матрицей): 2
```

Рис. 4.1: Работа алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна.

4.2 Технические характеристики

Ниже приведеные технические характеристики устройства, на котором было проведенно тестирование ПО:

- Операционная система: Windows 10 64-bit Home [3].
- Оперативная память: 8 GB.
- Процессор: 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz [4].

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов замерялось с помощью функции process_time модуля time в Python [5]. Данная функция возвращает значение в долях секунды суммы системного и пользовательского процессорного времени текущего процесса.

В таблице 4.1. представлены замеры времени работы для каждого из алгоритмов.

Таблица 4.1: Таблица времени выполнения алгоритмов (в долях секунды)

Длина строк	RecLev	RecLevCache	RecDam	IterDam
2	0.000000	0.000000	0.000000	0.000000
4	0.000156	0.000000	0.000156	0.000000
6	0.002812	0.000000	0.002812	0.000000
8	0.086719	0.000000	0.078437	0.000000
10	2.614063	0.000000	2.390938	0.000000
60	NaN	0.002812	NaN	0.001875
110	NaN	0.009375	NaN	0.006094
160	NaN	0.020000	NaN	0.012969
210	NaN	0.034531	NaN	0.021406
260	NaN	0.053125	NaN	0.033438
310	NaN	0.077188	NaN	0.048438
360	NaN	0.106875	NaN	0.067969
410	NaN	0.140469	NaN	0.089531
460	NaN	0.179375	NaN	0.115156

4.4 Использование памяти

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

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Поэтому, максимальный расход памяти равен:

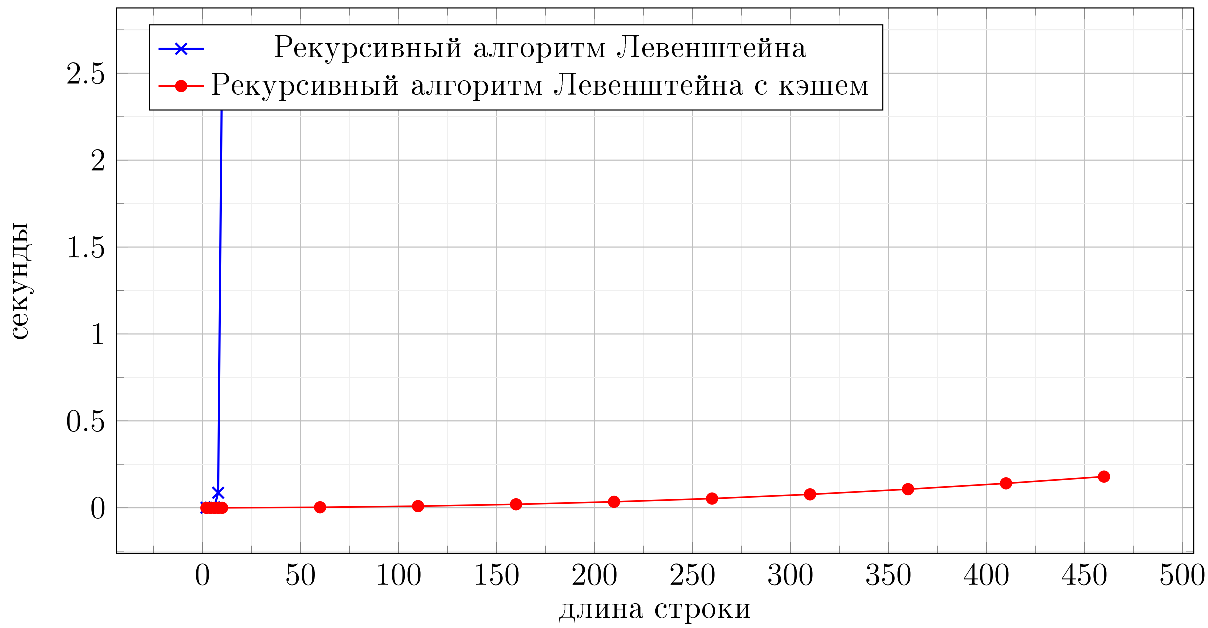

Рис. 4.2: Сравнение рекурсивного алгоритма Левенштейна и рекурсивного с кэшем

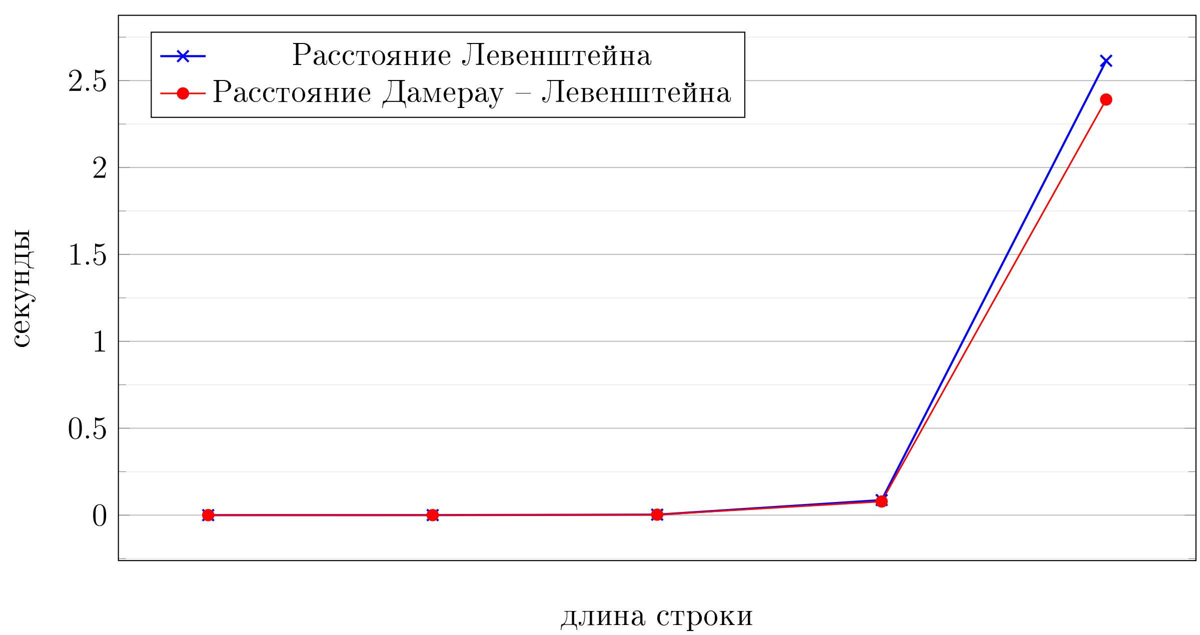

Рис. 4.3: Сравнение рекурсивных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна

$$(\mathcal{S}(STR_1) + \mathcal{S}(STR_2)) \cdot (2 \cdot \mathcal{S}(\text{string}) + 2 \cdot \mathcal{S}(\text{integer})), \tag{4.1}$$

где $\mathcal{S}$ — оператор вычисления размера, STR_1, STR_2 — строки, string — строковый тип, integer — целочисленный тип.

Использование памяти при итеративной реализации теоретически равно:

$$(\mathcal{S}(STR_1) + 1) \cdot (\mathcal{S}(STR_2) + 1) \cdot \mathcal{S}(\text{integer}) + 5 \cdot \mathcal{S}(\text{integer}) + 2 \cdot \mathcal{S}(\text{string}). \tag{4.2}$$

4.5 Вывод

Обычная рекурсивная реализация алгоритма нахождения расстояния Левенштейна работает дольше реализации с кэшем и итеративной реализации, время работы этой реализации увеличивается в геометрической прогрессии. Рекурсивный метод при этом использует больше памяти, чем итеративный.

Заключение

В ходе проделанной работы был изучен метод динамического программирования на материале реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. Также были изучены алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками и получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях, а так же в версиях с мемоизацией.

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. Рекурсивная реализация алгоритма Левенштейна проигрывает нерекурсивной по времени исполнения в несколько десятков раз. Так же стоит отметить, что итеративный алгоритм Левештейна выполняется немного быстрее, чем итеративный алгоритм Дамерау - Левенштейна, но в целом алгоритмы выполняются за примерно одинаковое время.

Теоретически было рассчитано использование памяти в каждой из реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау - Левенштейна.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР. 1965. с. 845.

[2] Damerau Fred J. A technique for computer detection and correction of spelling errors. Communications of the ACM. 1964. с. 171.

[3] Windows 10 Pro и Windows 10 Домашняя. https://www.microsoft.com/ru-ru/windows/compare-windows-10-home-vs-pro. Дата обращения: 18.10.2021.

[4] Процессор Intel® Core™ i3-1115G4 (6 МБ кэш-памяти, до 4,10 ГГц). Режим доступа: https://www.intel.ru/content/www/ru/ru/products/sku/208652/intel-core-i31115g4-processor-6m-cache-up-to-4-10-ghz/specifications.html. Дата обращения: 14.10.2021.

[5] Функция processtime() модуля time в Python. https://docs-python.ru/standart-library/modul-time-python/funktsija-process-time-modulja-time/. Дата обращения: 5.10.2021.