федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Ковалев Дмитрий, ИУ7-53Б

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

Москва, 2020

Оглавление

Введение **2**

1 Аналитическая часть **4**

2 Конструкторская часть **6**
2.1 Схемы алгоритмов 6

3 Технологическая часть **11**
3.1 Выбор ЯП 11
3.2 Реализация алгоритма 11

4 Исследовательская часть **14**
4.1 Сравнительный анализ на основе замеров времени работы алгоритмов 14
4.2 Тесты 15

Заключение **17**

Литература **17**

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 лингвистике в слудующих случаях:

- Исправления ошибок в слове(поисковая строка браузера);
- Сравнения текстовых файлов утилитой diff;
- В биоинформатике для сравнения генов, хромосом и белков.

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

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 двух 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 временно́й 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

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).

ействия обозначаются слдующим образом:

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 (1.1):

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

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

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле (1.2):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \text{, если } i,j > 0 \\ & \text{и } S_1[i] = S_2[j-1] \\ & \text{и } S_1[i-1] = S_2[j] \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases} \tag{1.2}$$

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2. Конструкторская часть

Требования к программе:

1. На вход подаются две строки.
2. Буквы в верхнем регистре и в нижнем регистре считаются разными.
3. Две пустые строки - корректный ввод, программа не должна аварийно завершаться.
4. Для всех алгоритмов выводиться процессорное время исполнения.
5. Для всех алгоритмов кроме Левенштейна с рекурсивной реализацией должна выводиться матрица.

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы следующих алгоритмов Левенштейна [1]: рекурсивного алгоритма Левенштейна (рис. 2.1), рекурсивного алгоритма Левенштейна с матричной оптимизацией (рис. 2.2), матричного алгоритма Левенштейна (рис. 2.3), матричного алгоритма Левенштейна (рис. 2.4).

Вывод

В данном разделе были рассмотрены требования к работе, а также схемы алгоритмов нахождения редакционного расстояния с помощью алгоритмов Левенштейна и Дамерау-Левенштейна.

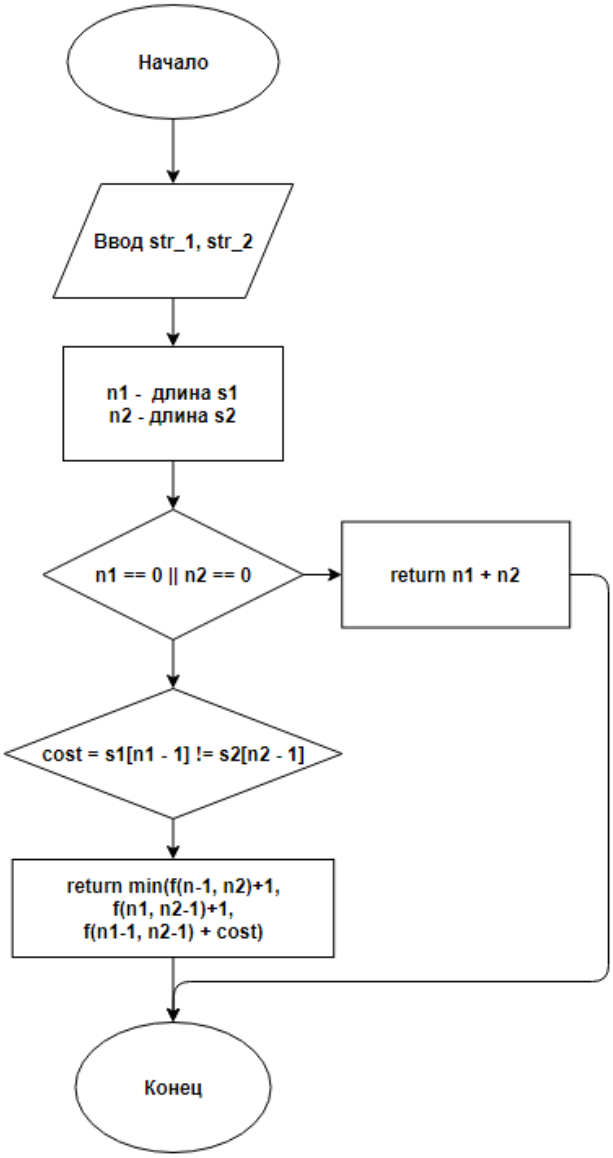


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

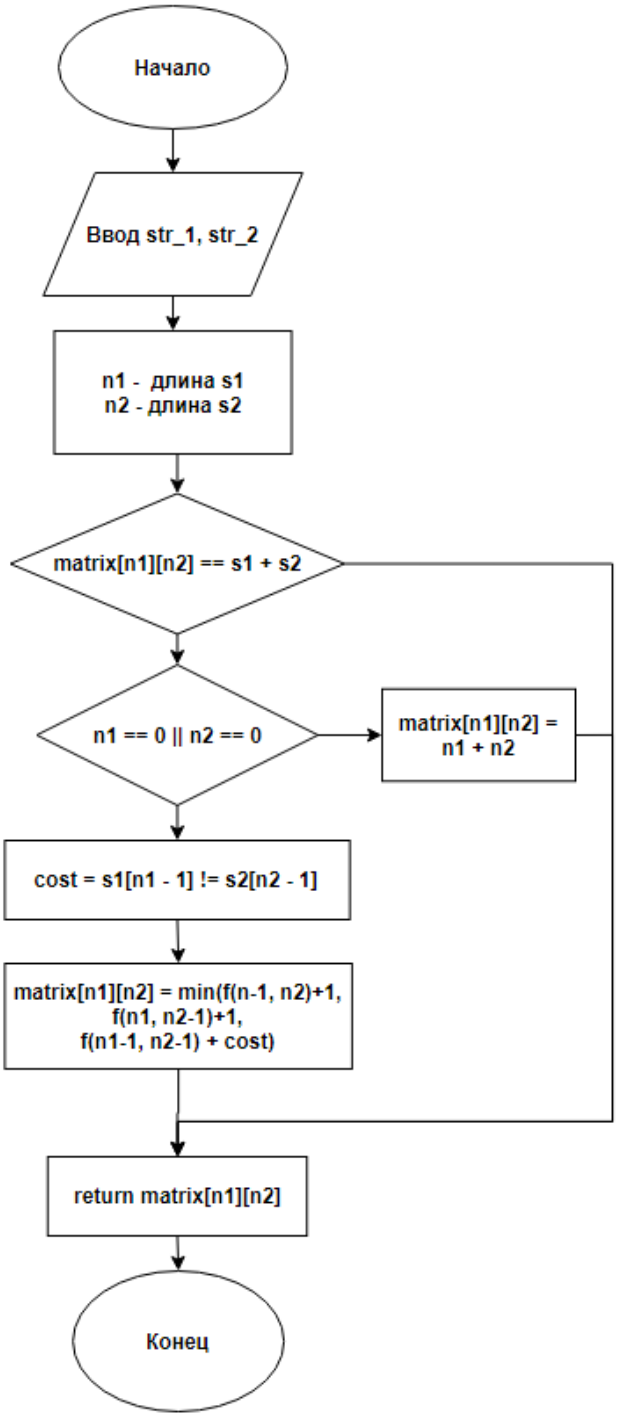


Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с матричной оптимизацией

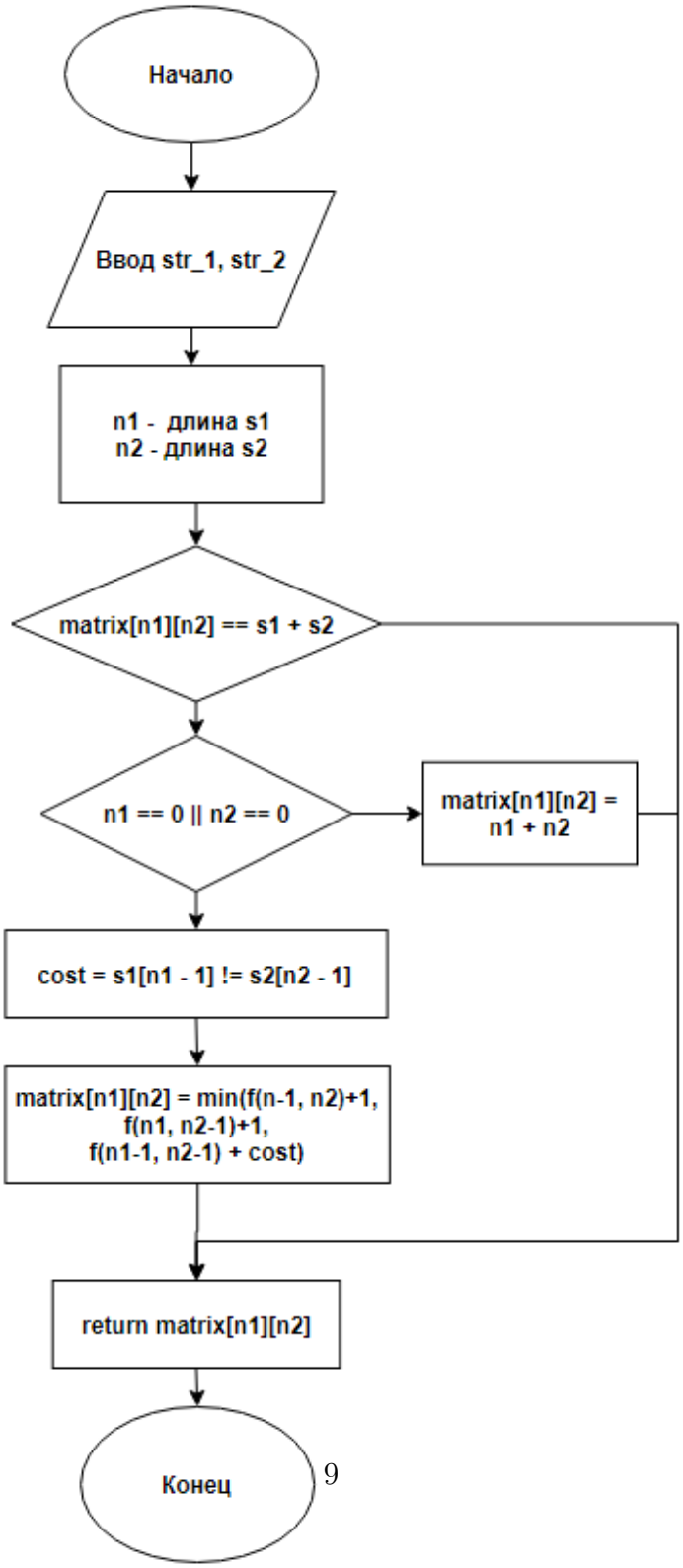


Рис. 2.3: Схема матричного алгоритма нахождения расстояния Левенштейна

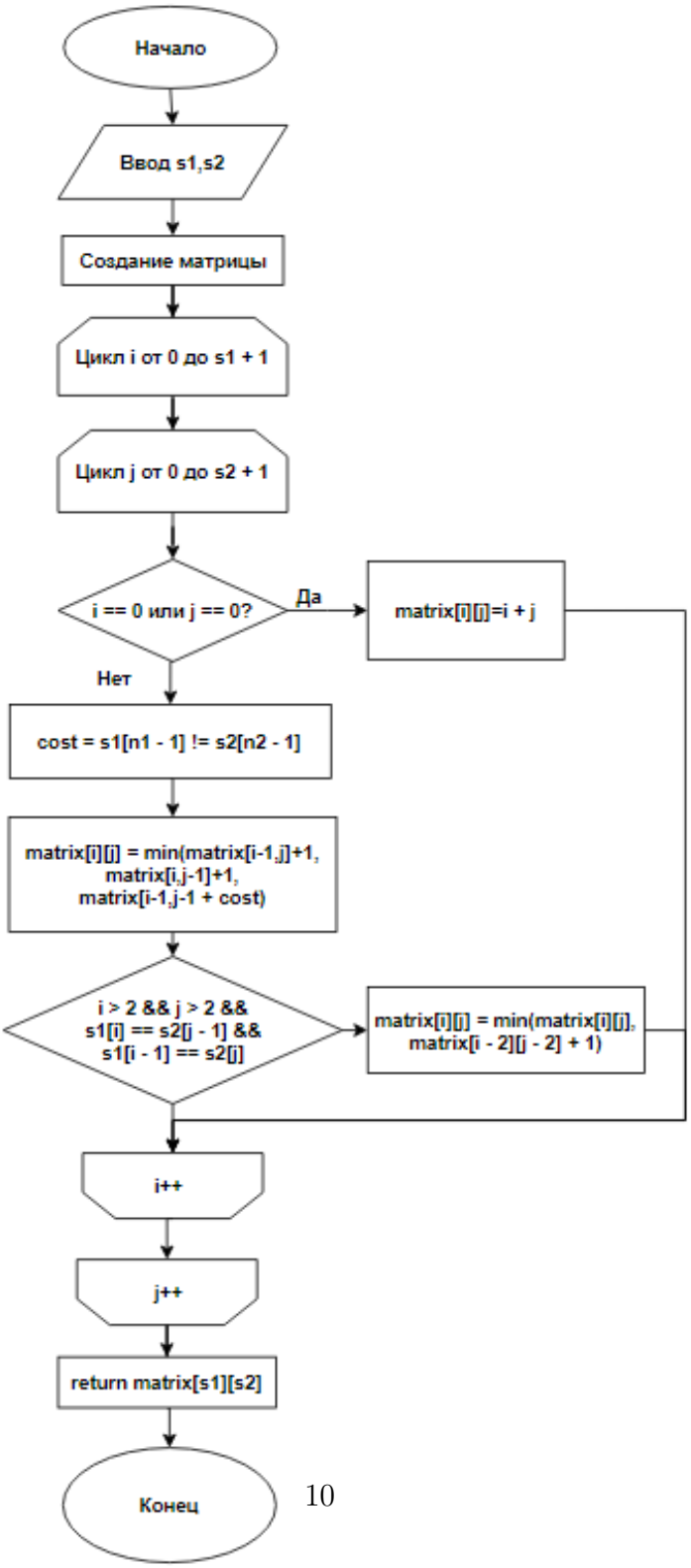


Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3. Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран язык программирования JavaScript [2] в связи с потребностью практики разработки на нем. Среда разработки - VS Code [3].

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def LevRecursion(str1, str2, Dam=False):
    if str1 == '' or str2 == '':
        return abs(len(str1) - len(str2))
    forfeit = 0 if (str1[-1] == str2[-1]) else 1
    res = min(LevRecursion(str1, str2[:-1], Dam) + 1,
              LevRecursion(str1[:-1], str2, Dam) + 1,
              LevRecursion(str1[:-1], str2[:-1], Dam) +
                  forfeit
    return res
```

Листинг 3.2: Функция нахождения расстояния Левенштейна рекурсивно с матрицей

```
def LevRecursionMatrix(str1, str2, matrix):
    if matrix[len(str1)][len(str2)] is not None:
        return matrix[len(str1)][len(str2)]

    if len(str1) == 0 or len(str2) == 0:
        matrix[len(str1)][len(str2)] = len(str2) + len(str1
            )
        return matrix[len(str1)][len(str2)]

    forfeit = 0 if (str1[-1] == str2[-1]) else 1
    matrix[len(str1)][len(str2)] = min(LevRecursionMatrix(
        str1, str2[:-1], matrix) + 1,
    LevRecursionMatrix(str1[:-1], str2, matrix) + 1,
```

```
12      LevRecursionMatrix(str1[:-1], str2[:-1], matrix) +
            forfeit)
13      return matrix[len(str1)][len(str2)]
```

Листинг 3.3: Функция нахождения расстояния Левенштейна матрично

```
1  def LevTable(str1, str2, Dam=False):
2      len_i = len(str1) + 1
3      len_j = len(str2) + 1
4      table = [[i + j for j in range(len_j)] for i in range(
           len_i)]
5      for i in range(1, len_i):
6          for j in range(1, len_j):
7              forfeit = 0 if (str1[i - 1] == str2[j - 1])
                   else 1
8              table[i][j] = min(table[i - 1][j] + 1,
9                                table[i][j - 1] + 1,
10                               table[i - 1][j - 1] + forfeit
                                     )
11     return table[-1][-1]
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

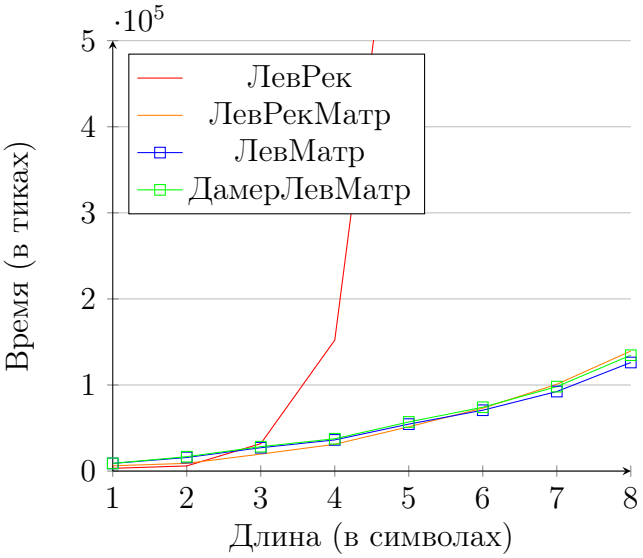
```
1  def LevTable(str1, str2, Dam=False):
2      len_i = len(str1) + 1
3      len_j = len(str2) + 1
4      table = [[i + j for j in range(len_j)] for i in range(
           len_i)]
5
6      for i in range(1, len_i):
7          for j in range(1, len_j):
8              forfeit = 0 if (str1[i - 1] == str2[j - 1])
                   else 1
9              table[i][j] = min(table[i - 1][j] + 1,
10                               table[i][j - 1] + 1,
11                               table[i - 1][j - 1] + forfeit
                                     )
12             if (i > 1 and j > 1) and str1[i - 1] == str2[j
                   - 2] and str1[i - 2] == str2[j - 1]:
```

```
13                 table[i][j] = min(table[i][j], table[i -
                       2][j - 2] + 1)
14     return table[-1][-1]
```

4. Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов на машине MacBook Pro 13 [4]. Для минимизации погрешности замеров времени каждый алгоритм исполнялся над одними и теми же строками 1000 раз и затраченное время делилось на 1000, для получения усредненного времени выполнения. Отмечу, что замеры рекурсивного алгоритма нахождения расстояния Левенштейна замерялся единожды, поскольку уже на словах, длиннее 10 символов алгоритм исполняется относительно долго.



Наиболее быстрым является алгоритм Левенштейна с матрицей; в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Рекурсивная версия алгоритма крайне сильно проигрывает матричному алгоритму уже на строках длиной 10 символов, из-за большой глубины рекурсии и многочисленных повторов. Рекурсивная версия с матрицей не теряет в своей производительности настолько сильно, поскольку в ней исключаются повторы благодаря кэшированию результатов вычисления ячеек матрицы. Заметим, что алгоритм Дамерау-Левенштейна выполняется относительного немного дольше алгоритма

Таблица 4.1: время исполнения в миллимекундах

len	Lev(R)	Lev(RM)	Lev(M)	DamLev(M)
1	0.025996	0.040448	0.022366	0.062549
2	0.048608	0.076069	0.033854	0.077381
3	0.106327	0.085223	0.028715	0.076817
4	0.502248	0.080871	0.026586	0.073056
5	0.862543	0.079230	0.030535	0.080112.
6	0.653954	0.048213	0.019119	0.046017
7	2.383116	0.074804	0.026301	0.066106
8	5.957557	0.064507	0.021543	0.002099
9	21.236497	0.084806	0.030479	0.072075
10	112.217936	0.088567	0.041352	0.088084

Левенштейна, т.к. в нем добавлена дополнительная проверка (проверка на транспозицию символов). В среднем алгоритм Дамерау-Левенштейна с матрицей работает на 10% дольше, чем матричный алгоритм Левенштейна.

4.2 Тесты

Проведем тестирование программы. В столбцах "Ожидание"и "Результат"4 числа соответсвуют рекурсивному алгоритму нахождения расстояния Левенштейна, рекурсивно-матричному алгоритму нахождения расстояния Левенштейна, матричному алгоритму расстояния Левенштейна, матричному алгоритму нахождения расстояния Дамерау-Левенштейна.

Вывод

Закодировав исследованные алгоритмы, была получена итоговая программа. Протестированная программа выдала результаты, соответсвующие теоретическим данным.

Таблица 4.2: Таблица тестовых данных

№	Слово №1	Слово №2	Ожидание	Результат
1			0 0 0 0	0 0 0 0
2	1234	2143	3 3 3 2	3 3 3 2
3	amm	add	2 2 2 2	2 2 2 2
4	error	dog	4 4 4 4	4 4 4 4
5		submission	10 10 10 10	10 10 10 10
6	mochombo	0	8 8 8 8	8 8 8 8
7	dfufdfd	fddfdffdd	5 5 5 4	5 5 5 4

Заключение

В ходе лабораторной работы были изучены алгоритмы Левенштейна и Дамерау-Левенштейна - алгоритмы нахождения редакционного расстояния между строками, получены практические навыки реализации указанных алгоритмов в матричной и рекурсивных версиях. Также был изучен метод динамического программирования на материале данных алгоритмов.

Экспериментально было подтверждено различие по временным затратам рекурсивной и линейной реализаций алгоритма Левенштейна при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на строках различной длины.

Самым быстродейственным является алгоритм Левенштейна с матрицей. Рекурсивная версия алгоритма крайне сильно проигрывает матричному алгоритму уже на строках длиной 10 символов, из-за большой глубины рекурсии и многочисленных повторов. Рекурсивная версия с матрицей не теряет в своей производительности настолько сильно, поскольку в ней исключаются повторы благодаря кэшированию результатов вычисления ячеек матрицы. Линейный алгоритм Дамерау-Левенштейна выполняется относительного незначительно дольше линейного алгоритма Левенштейна, поскольку в нем добавлена дополнительная проверка (проверка на транспозицию символов): в среднем матричный алгоритм Левенштейна работает на 10% быстрее, чем алгоритм Дамерау-Левенштейна с матрицей.

Подытожив, матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах и задачах.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] JavaScript Programming Language [Электронный ресурс]. Режим доступа: https://https://developer.mozilla.org/ru/docs/Web/JavaScript/ (дата обращения: 20.09.2020).

[3] Редактор кода VS Code. Режим доступа: https://code.visualstudio.com/ (дата обращения: 10.10.2020).

[4] Процессор Intel® Core™ i5 10 gen. [Электронный ресурс]. Режим доступа: https://https://www.intel.ru/content/www/ru/ru/products/docs/processors/core/ 10th-gen-processors.html (дата обращения: 20.09.2020).

[5] Absolute time. Режим доступа: https://https://developer.apple.com/documentation/kernel/1462446-machabsolutetime (дата обращения: 20.09.2020).

[6] MacOs Catalina [Электронный ресурс]. Режим доступа: https://https://www.apple.com/ru/macos/ catalina/ (дата обращения: 20.09.2020).

[7] Технические характеристики ноутбука Apple MacBook Pro 13. Режим доступа: https://support.hp.com/ru-ru/document/c05739572 (дата обращения: 10.10.2020).