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

Студент Сабуров С.М.

Группа ИУ7-53Б

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

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 для:

- исправления ошибок в слове
- в биоинформатике для сравнения генов, хромосом и белков

Цели данной лабораторной работы:

1. Изучение метода динамического программирования на материале алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.
2. Оценка реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Задачи данной лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна;
2. Применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. Получение практических навыков реализации указанных алгоритмов: матричные и рекурсивные 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;
6. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

Расстояние Левенштейна 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:

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

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 может быть найдено по формуле 1.3, которая задана как

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

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.

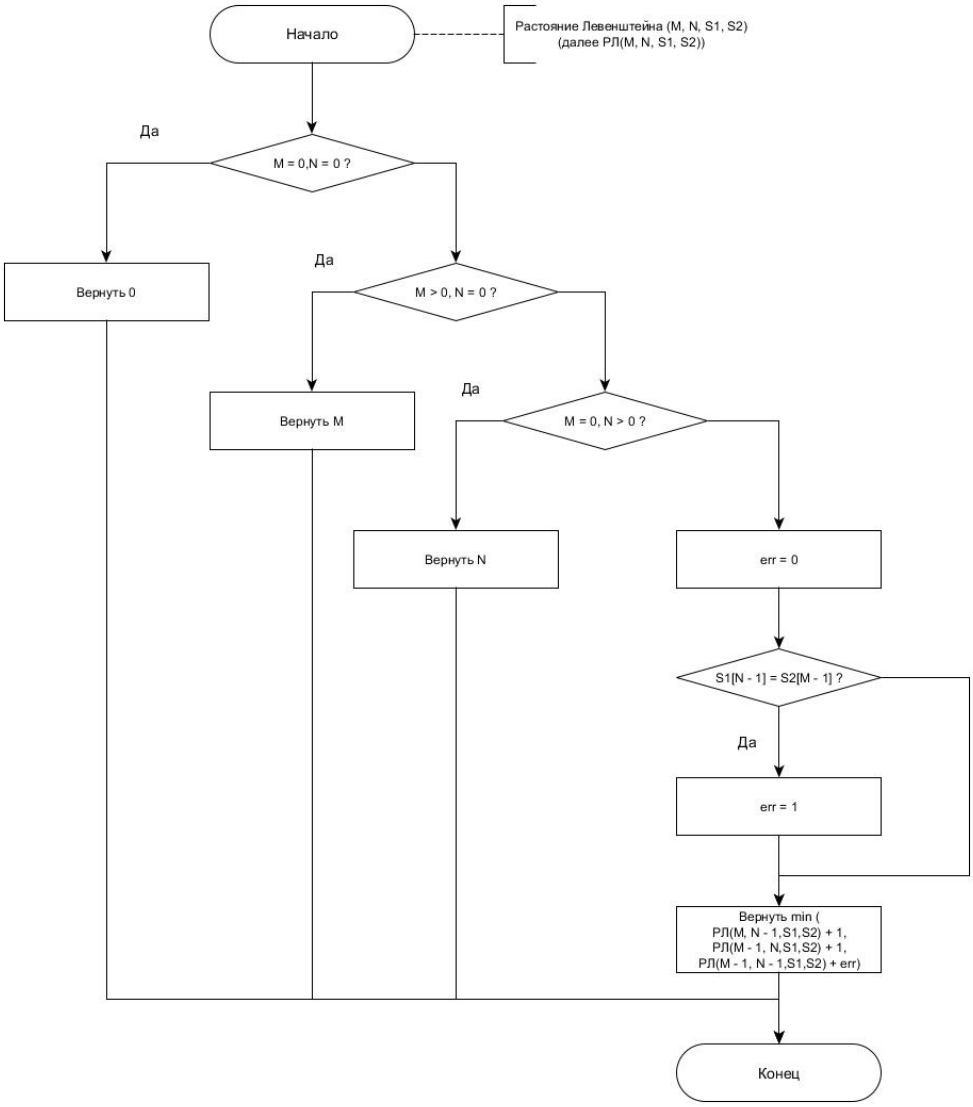


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

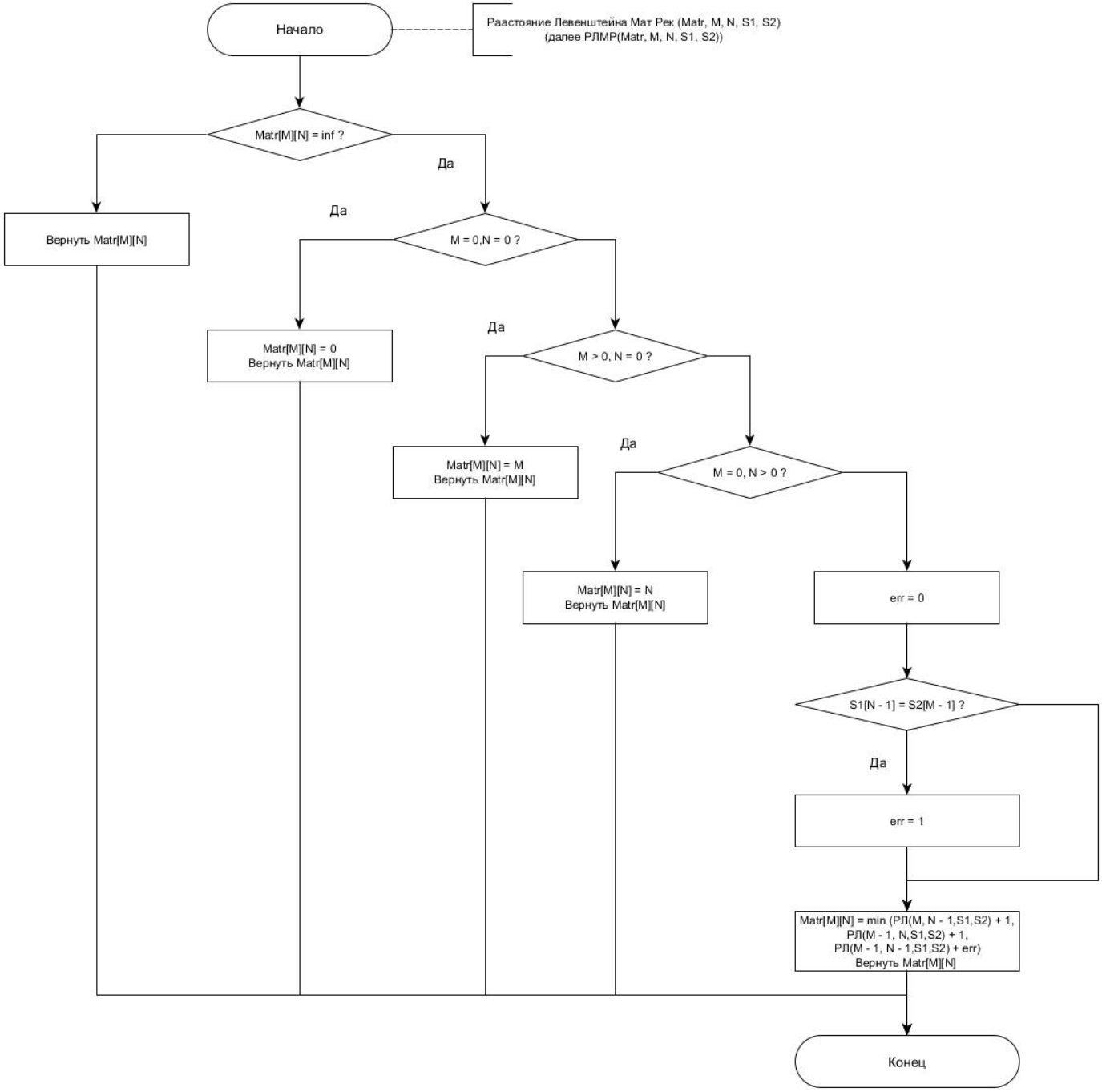
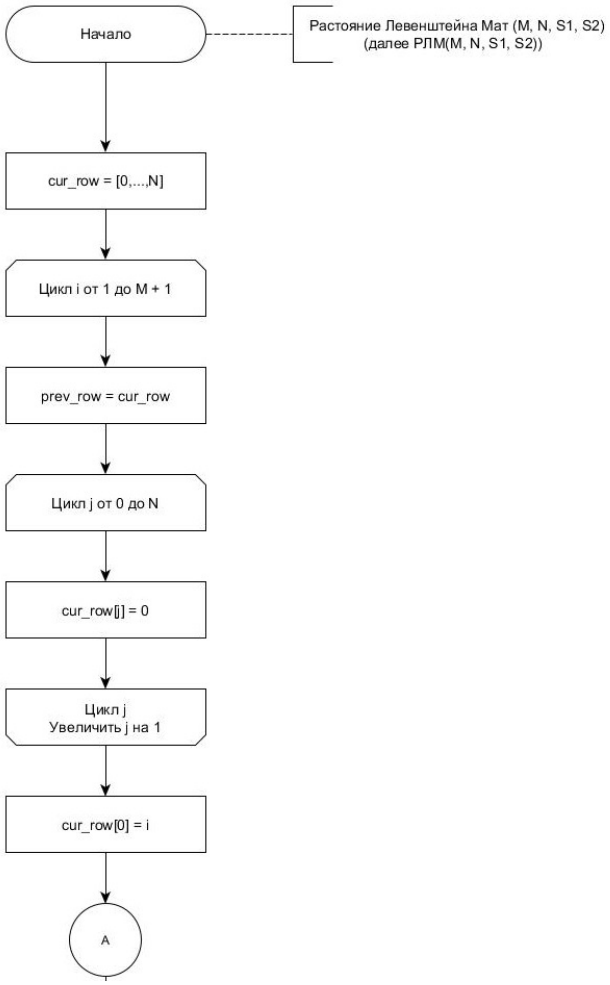


Рис. 2.2: Схема рекурсивного алгоритма с мемоизацией нахождения расстояния Левенштейна


Начало
Растояние Левенштейна Мат (M, N, S1, S2)
(далее РЛМ(M, N, S1, S2))
cur_row = [0,...,N]
Цикл i от 1 до M + 1
prev_row = cur_row
Цикл j от 0 до N
cur_row[j] = 0
Цикл j
Увеличить j на 1
cur_row[0] = i
A

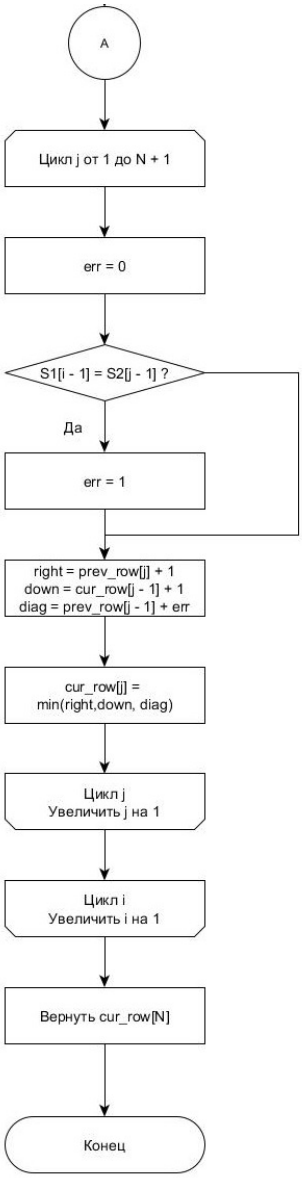
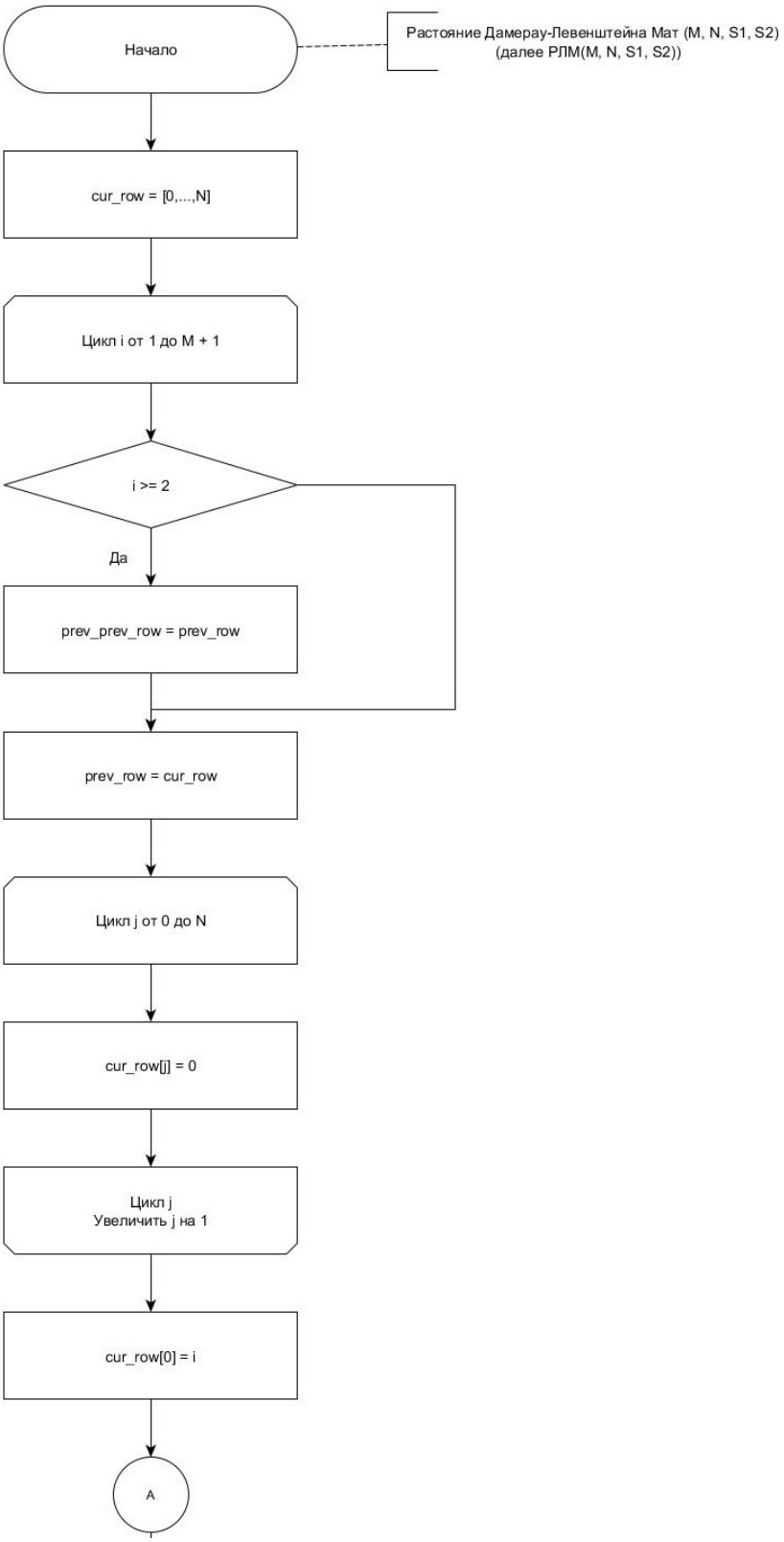


Рис. 2.3: Схема итеративного алгоритма нахождения расстояния Левенштейна


Начало
Растояние Дамерау-Левенштейна Мат (M, N, S1, S2)
(далее РЛМ(M, N, S1, S2))
cur_row = [0,...,N]
Цикл i от 1 до M + 1
i >= 2
Да
prev_prev_row = prev_row
prev_row = cur_row
Цикл j от 0 до N
cur_row[j] = 0
Цикл j
Увеличить j на 1
cur_row[0] = i
A

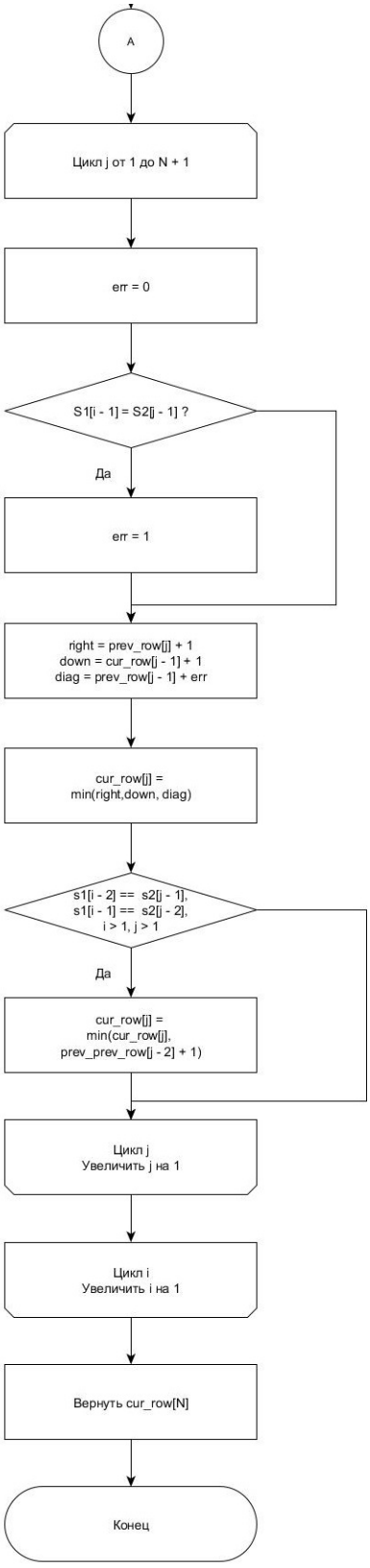


Рис. 2.4: Схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна

2.2 Вывод

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

3 Технологическая часть

3.1 Требование к ПО

Требования к вводу:

1. На вход подаются две строки в любой раскладке (в том числе и пустые);
2. ПО должно выводить полученное расстояние и вспомогательны матрицы;
3. ПО должно выводить потраченную память и время;

3.2 Средства реализации

Для реализации программы нахождения расстояние Левенштейна я выбрал язык программирования Python. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа программирования.

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_rec(M,N, s1,s2):
    if (M == 0 and N == 0):
        return 0
    elif (M > 0 and N == 0):
        return M
    elif (M == 0 and N > 0):
        return N
    else:
        return min(lev_rec(M, N - 1,s1,s2) + 1,\
                   lev_rec(M - 1, N,s1,s2) + 1,\
                   lev_rec(M - 1, N - 1,s1,s2) + bool(ord(s1[M - 1]) - ord(
                       s2[N - 1])))
```

Листинг 3.2: Функция нахождения расстояние Левенштейна рекурсивно с меморизацией

```
def rec_matr_lev(Matr, M, N, s1, s2):
    if Matr[M][N] == sys.maxsize:
        if (M == 0 and N == 0):
            Matr[M][N] = 0
            return int(0)
        elif (M > 0 and N == 0):
            Matr[M][N] = M
            return int(M)
        elif (M == 0 and N > 0):
            Matr[M][N] = N
            return int(N)
        else:
            err = 1
            if (s1[M - 1] == s2[N - 1]):
                err = 0
            Matr[M][N] =  min (rec_matr_lev(Matr,M, N - 1,s1,s2) + 1, \
                               rec_matr_lev(Matr,M - 1, N,s1,s2) + 1, \
                               rec_matr_lev(Matr,M - 1, N - 1,s1,s2) + err)
            return int(Matr[M][N])
    else:
        return Matr[M][N]
```

Листинг 3.3: Функция нахождения расстояния Левенштейна итеративно

```
def matr_lev(M, N, s1, s2):

    cur_row = range(N + 1)
    for i in range (1, M + 1):
        prev_row, cur_row = cur_row, [i] + [0] * N
        for j in range (1, N + 1):
            right, down, diag = prev_row[j] + 1, \
                                cur_row[j - 1] + 1, \
                                prev_row[j - 1] + bool(ord(s1[i - 1]) - ord(
                                    s2[j - 1]))
            cur_row[j] = min(right, down, diag)
    return cur_row[N]
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def damer_leven_opt(M, N, s1, s2):
    cur_row = range(N + 1)
    for i in range (1, M + 1):
        if i >= 2:
            prev_prev_row = prev_row
        prev_row, cur_row = cur_row, [i] + [0] * N
        for j in range (1, N + 1):
            right, down, diag = prev_row[j] + 1, cur_row[j - 1] + 1,
                prev_row[j - 1] + bool(ord(s1[i - 1]) - ord(s2[j - 1]))
            cur_row[j] = min(right, down, diag)
            if s1[i - 2] ==  s2[j - 1] and s1[i - 1] ==  s2[j - 2] and i > 1
                and j > 1:
                cur_row[j] = min(cur_row[j], prev_prev_row[j - 2] + 1)
    return cur_row[N]
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестированно разработанное ПО.

Таблица 3.1: Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0	0 0
2	kot	skat	2 2	2 2
3	kate	ktae	2 1	2 1
4	abacaba	aabcaab	4 2	4 2
5	sobaka	sboku	3 3	3 3
6	qwerty	queue	4 4	4 4
7	apple	aplpe	2 1	2 1
8		cat	3 3	3 3
9	parallels		9 9	9 9
10	русскиебуквы	русскиебукыв	2 1	2 1

3.5 Вывод

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

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

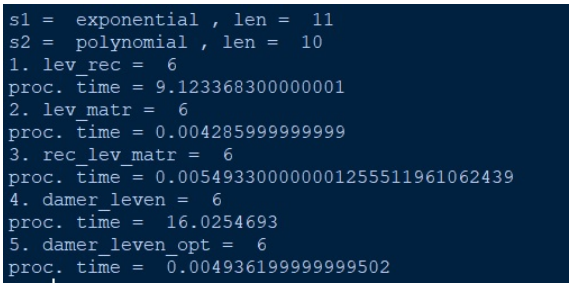


Рис. 4.1: Работа алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна.

4.2 Технические характеристики

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

- Операционная система: Ubuntu Linux 18.4 64-bit.
- Оперативная память: 8 GB.
- Процессор: Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz

4.3 Время выполнения алгоритмов

Время выполнения алгоритм замерялось с помощью метода clock() . Этот метод возвращает текущее время процессора как число с плавающей запятой, выраженное в секундах в Unix.

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

Таблица 4.1: Таблица времени выполнения алгоритмов (в наносекундах)

Длина строк	Rec	MatRec	Iter	DamIter
5	2143800	80000	60999	56999
10	10393456200	365599	188599	201799
15	None	673600	372799	454400
20	None	1398999	712599	819800
30	None	3075799	1553199	1917799
50	None	8534000	4737799	5200999
100	None	36561600	17005799	19993799
200	None	125299799	68700799	75188600

На рисунке 4.2. диаграмма работы для каждого из алгоритмов.

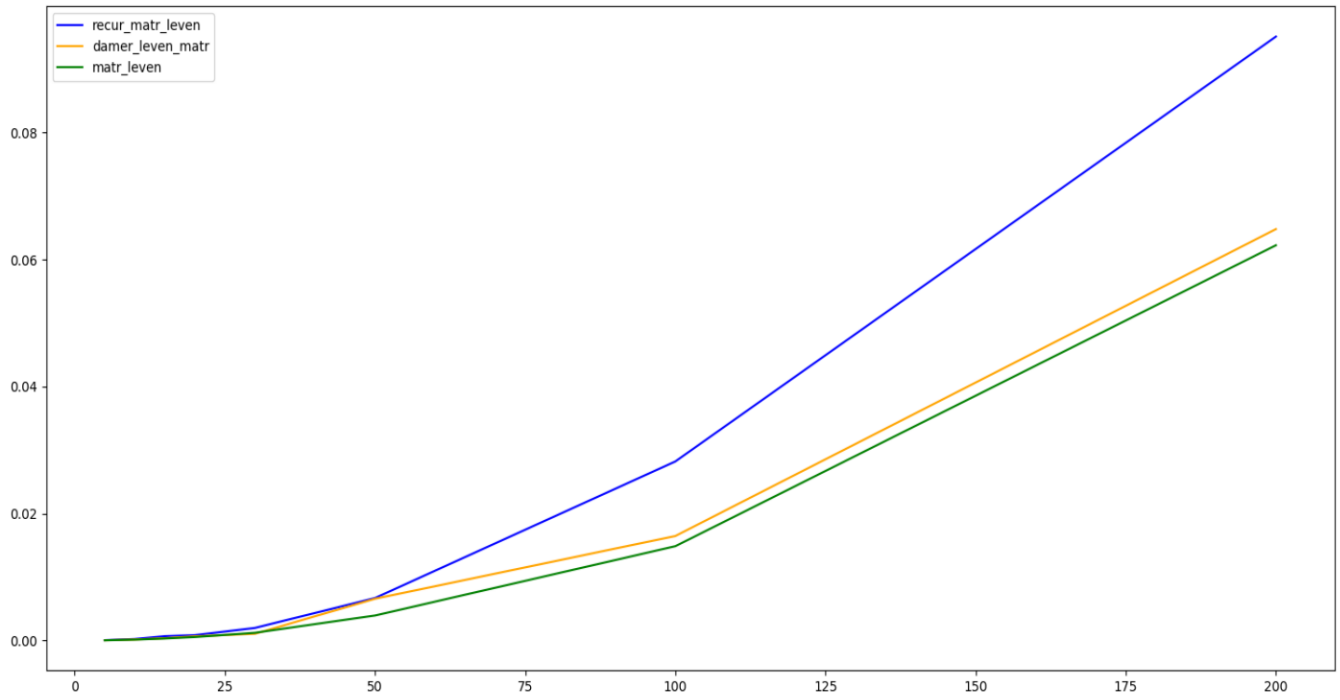


Рис. 4.2: Зависимость времени работы алгоритмов от длины строк.

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:

$$(\mathcal{S}(STR_1) + \mathcal{S}(STR_2)) \cdot (2 \cdot \mathcal{S}(\text{string}) + 3 \cdot \mathcal{S}(\text{integer})), \tag{4.1}$$

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

Рекурсивная реализация алгоритма нахождения расстояния Левенштейна работает дольше итеративных реализаций, время работы этой реализации увеличивается в геометрической прогрессии.

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 использования памяти в каждой из реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау - Левенштейна. Рекурсивный алгоритм с мемоизацией в несколько десятков раз больше памяти, чем итеративная реализация алгоритма нахождения расстояния Левенштейна, из-за рекурсивного копирования вспомогательной матрицы.