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 курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Ивахненко Д.А.

Группа ИУ7-56Б

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна.

Расстояние Левенштейна (редакционное расстояние) — 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. Впервые задачу нахождения редакционного расстояния поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0–1[1].

Расстояние Дамерау—Левенштейна вляется модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Левенштейна и его обобщения активно применяются:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста);
- для сравнения текстовых файлов утилитой `diff` и ей подобными;
- в биоинформатике для сравнения генов, хромосом и белков.

Задачи данной лабораторной работы:

1. изучение алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна;
2. применение методов динамического программирования для реализации алгоритмов;
3. получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна;
4. сравнительный анализ алгоритмов на основе экспериментальных данных;
5. подготовка отчета по лабораторной работе.

1 Аналитическая часть

Расстояние Левенштейна между двумя строками - это минимальное количество операций, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) 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 S_1 и S_2 может быть вычислено по реккурентной формуле 1.1, где $|S_1|$ означает длину строки S_1, $S_1[i]$ — i-ый символ строки S_1, функция $D_{S_1,S_2}(i, j)$ определена как:

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

а функция 1.2 определена как:

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

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

1) для перевода пустой строки в пустую требуется ноль операций;

2) для перевода пустой строки в непустую строку S_1 требуется $|S_1|$ операций;

3) для перевода непустой строки S_1 в пустую требуется $|S_1|$ операций;

Для перевода из строки S_1 в строку S_2 требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена). Полагая, что S_1', S_2' — строки S_1 и S_2 без последнего символа соответственно, цена преобразования из строки S_1 в строку S_2 может быть выражена как:

1) сумма цены преобразования строки S_1 в S_2' и цены проведения операции вставки, которая необходима для преобразования S_2' в S_2;

2) сумма цены преобразования строки S_1' в S_2 и цены проведения операции удаления, которая необходима для преобразования S_1 в S_1';

3) сумма цены преобразования из S_1' в S_2' и операции замены, предполагая, что S_1 и S_2 оканчиваются на разные символы;

4) цена преобразования из S_1' в S_2', предполагая, что S_1 и S_2 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, так как множество промежуточных значений $D(i, j)$ вычисляются заново несколько раз.

Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $M_{(|S_1|+1),(|S_2|+1)}$ 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 подхода заключается в использовании матрицы в качестве кэша при выполнении рекурсии, что позволяет снизить количество вычислений.

1.4 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{S_1,S_1}(i,j-1)+1, & \\ \quad d_{S_1,S_1}(i-1,j)+1, & \\ \quad d_{S_1,S_1}(i-1,j-1)+\theta(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{S_1,S_1}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.3}$$

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, а также Дамерау-Левенштейна, который учитывает возможность перестановки соседних символов. Формулы Левенштейна и Дамерау—Левенштейна для рассчета расстояния между строками задаются рекуррентно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

2.1 Схема алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна. На рисунках 2.2 - 2.3 приведены схемы рекурсивного алгоритма нахождения расстояния Левенштейна с кэшированием.

2.2 Схема алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунках 2.4 - 2.5 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна. На рисунках 2.6 - 2.7 приведена схема итерационного алгоритма нахождения расстояния Дамерау-Левенштейна.

+

2.3 Вывод

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

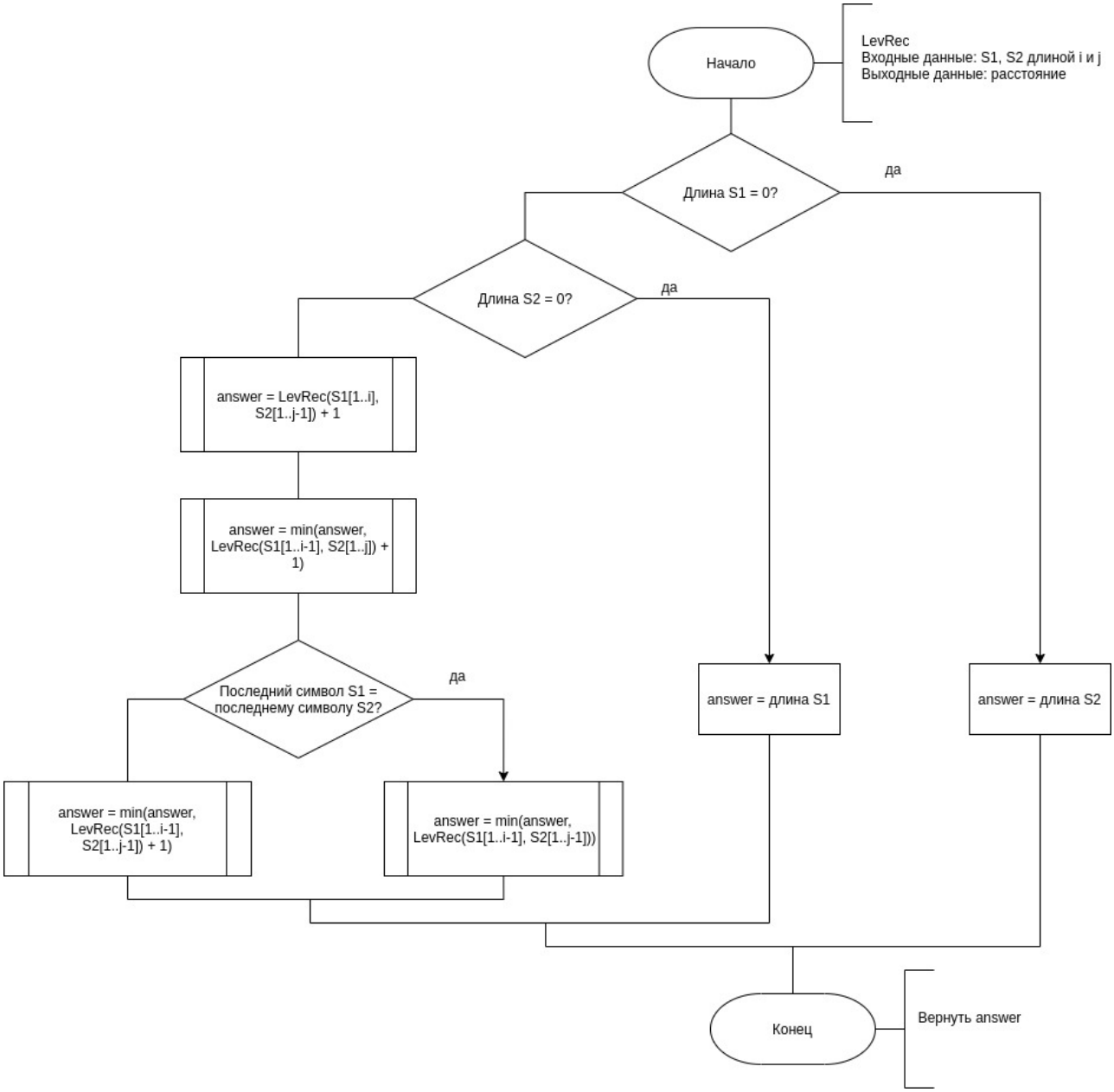

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

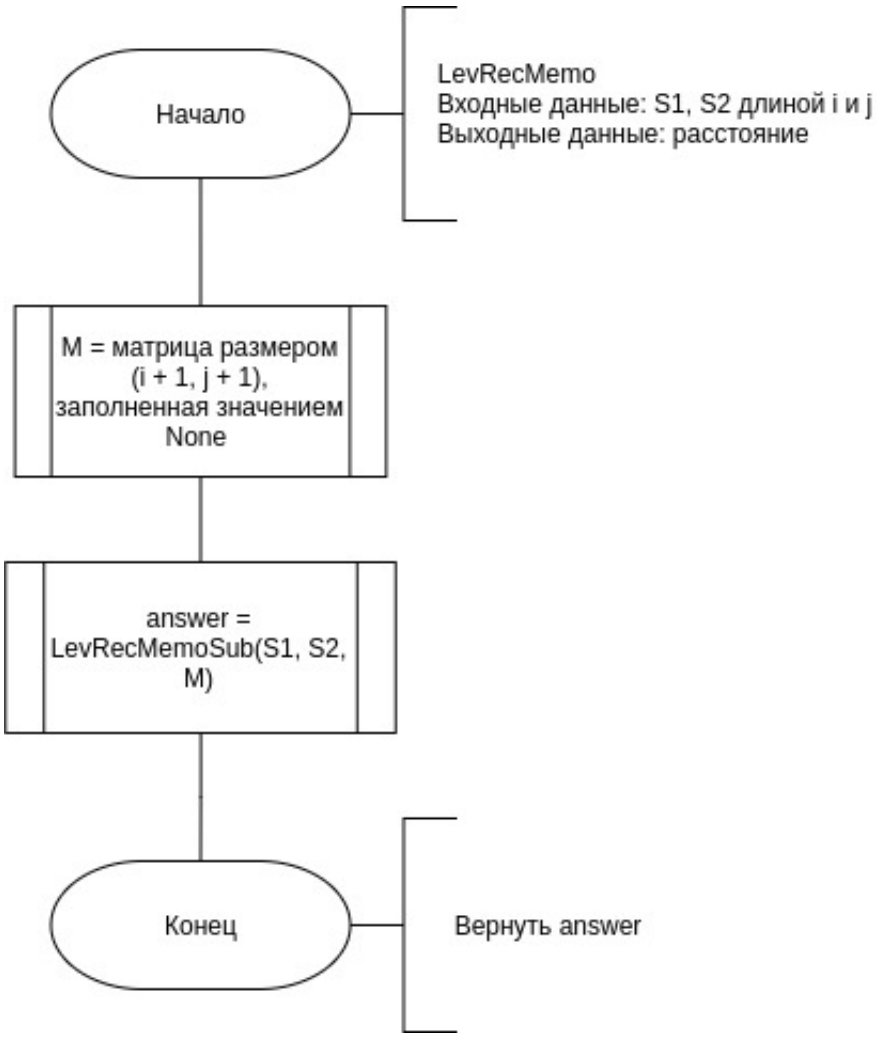

Рисунок 2.2 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшированием

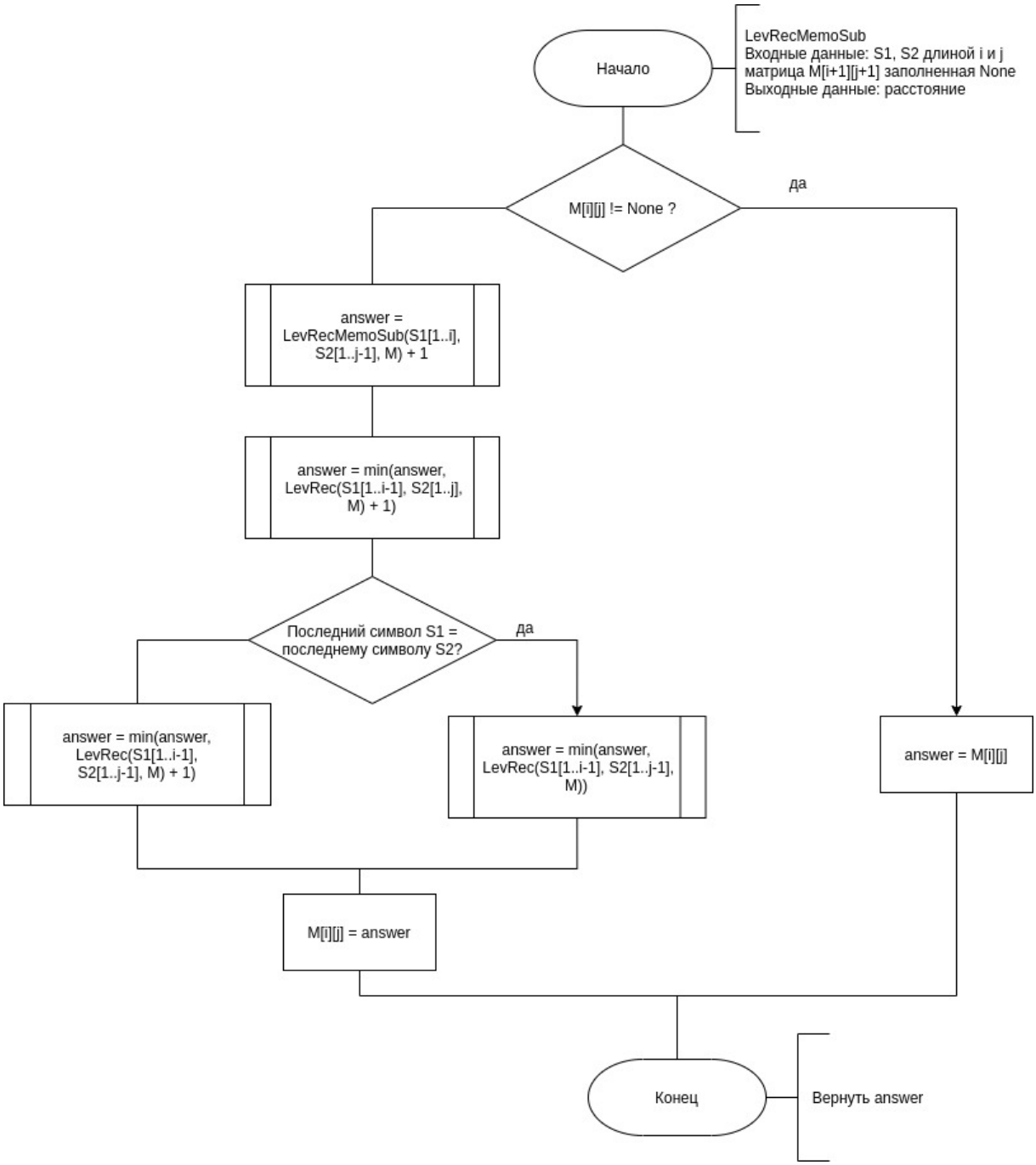

Рисунок 2.3 – Схема процедуры рекурсивного алгоритма нахождения расстояния Левенштейна с кэшированием

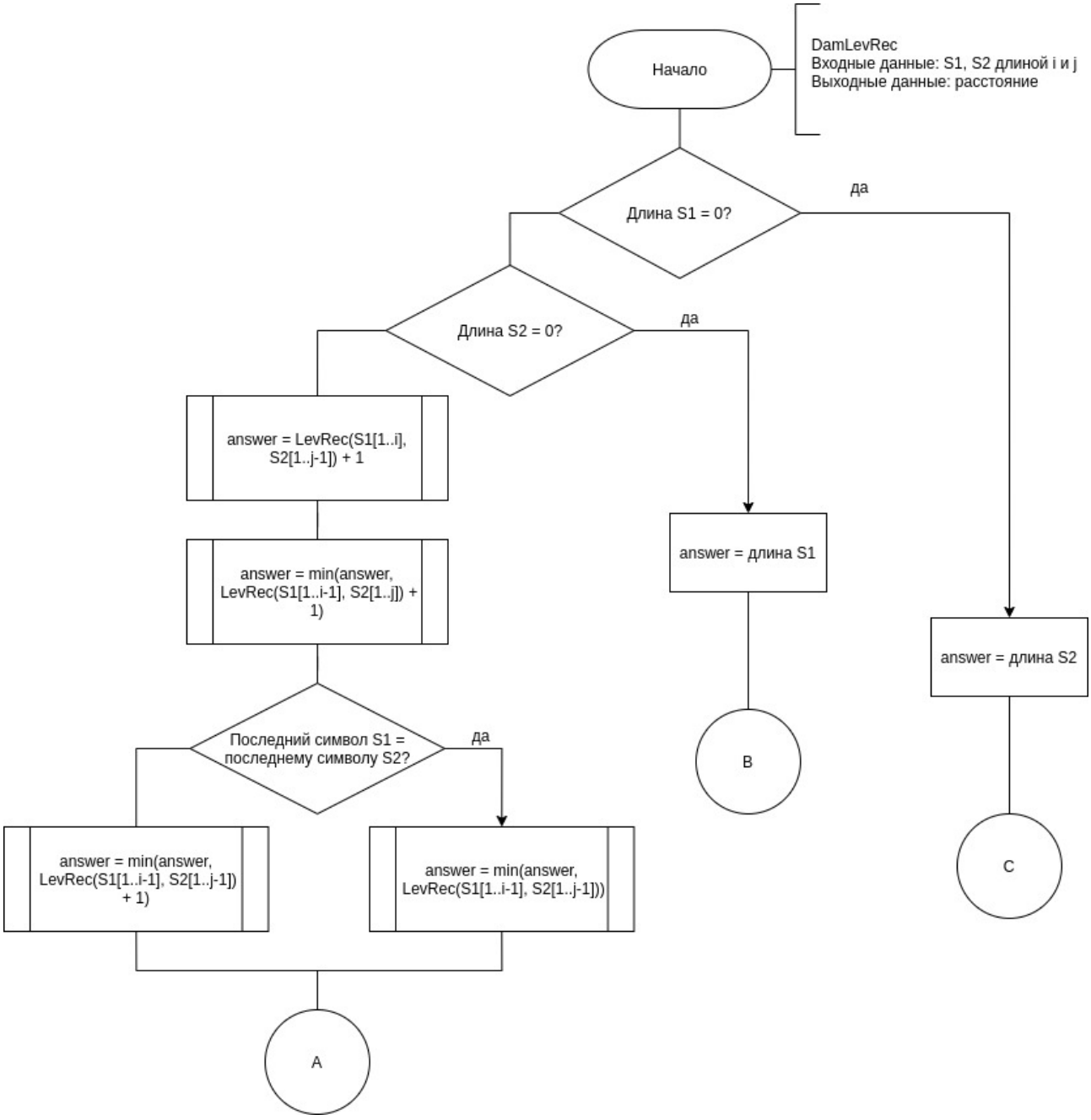

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

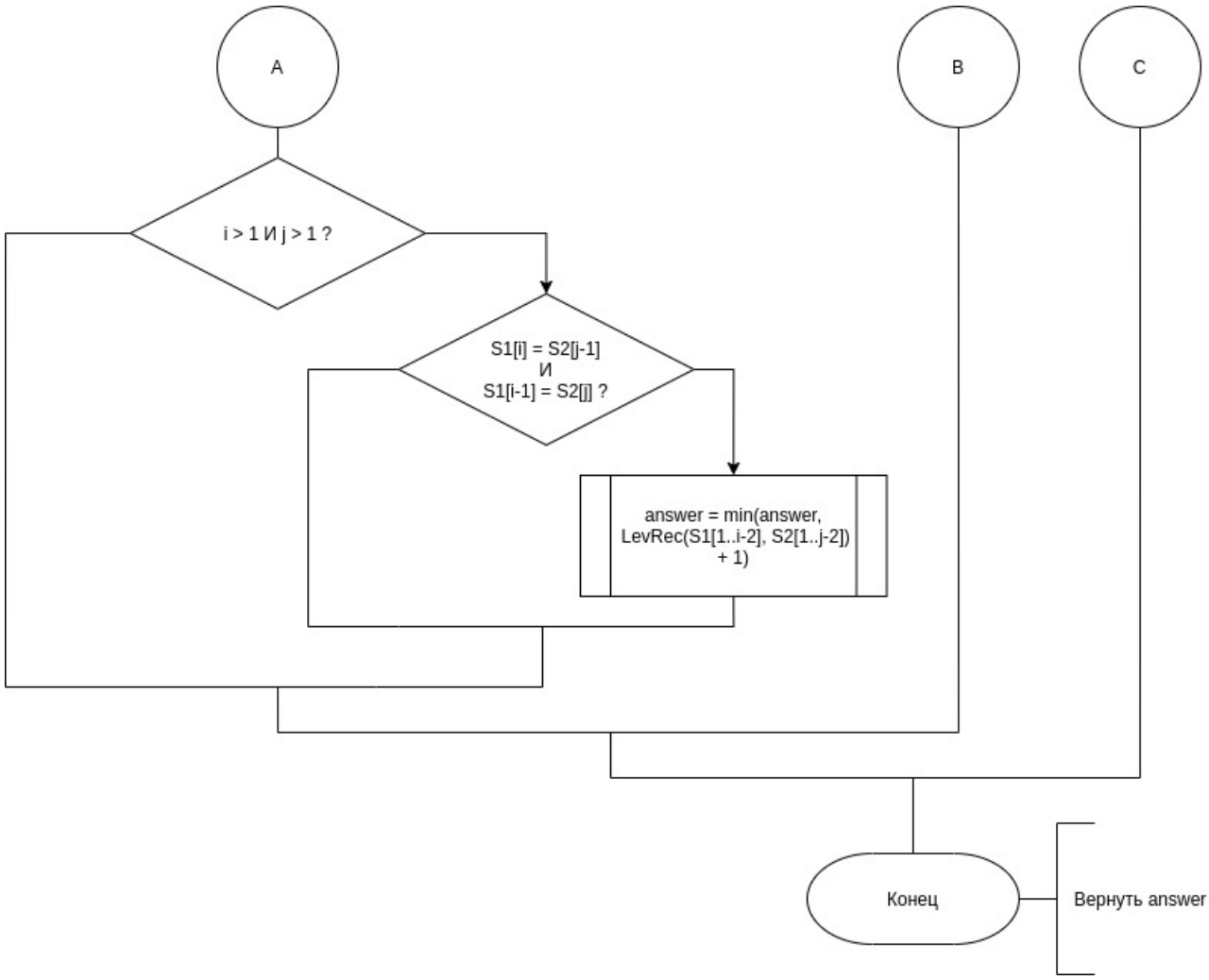

Рисунок 2.5 – Продолжение схемы рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

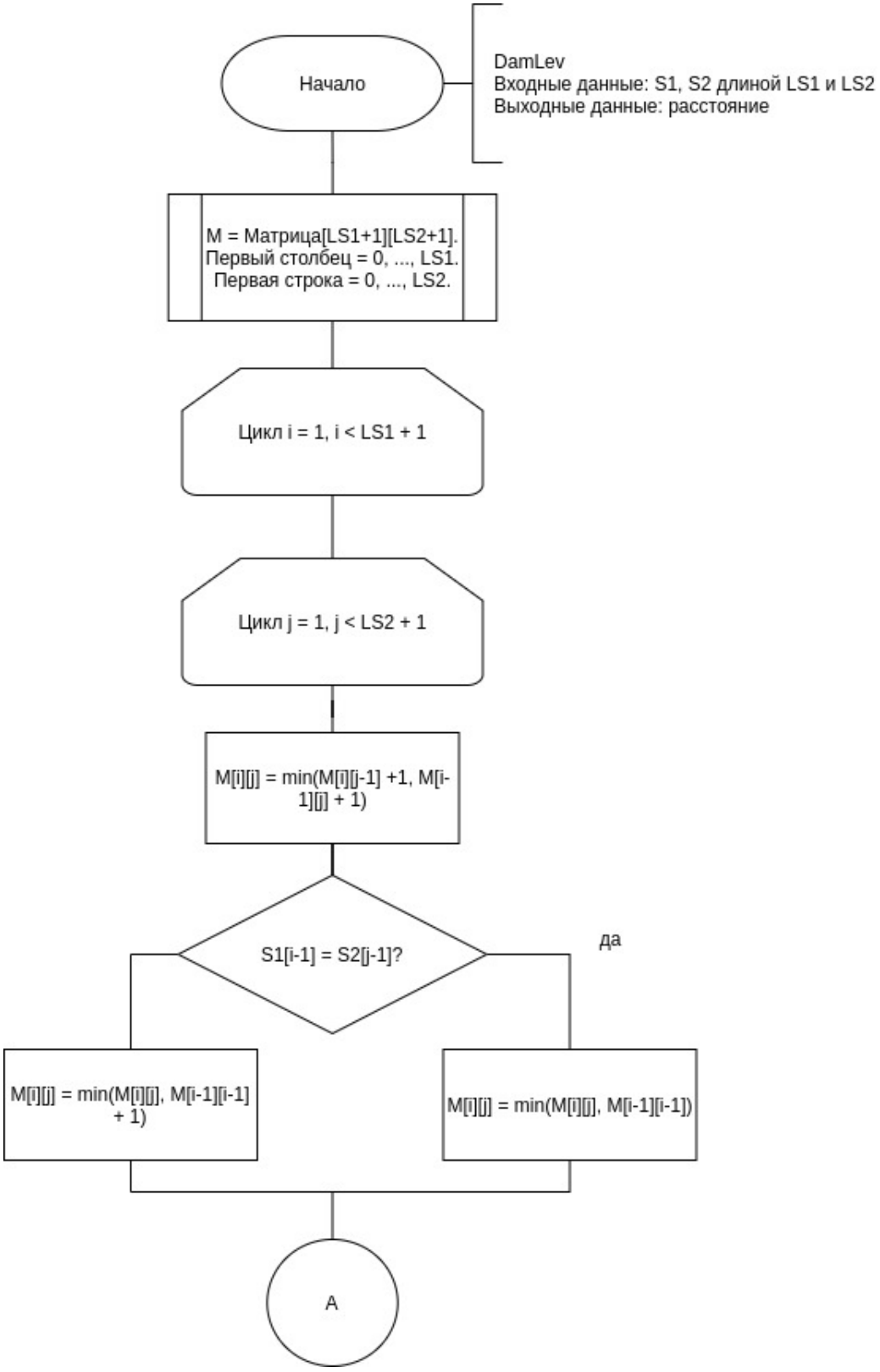

Рисунок 2.6 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

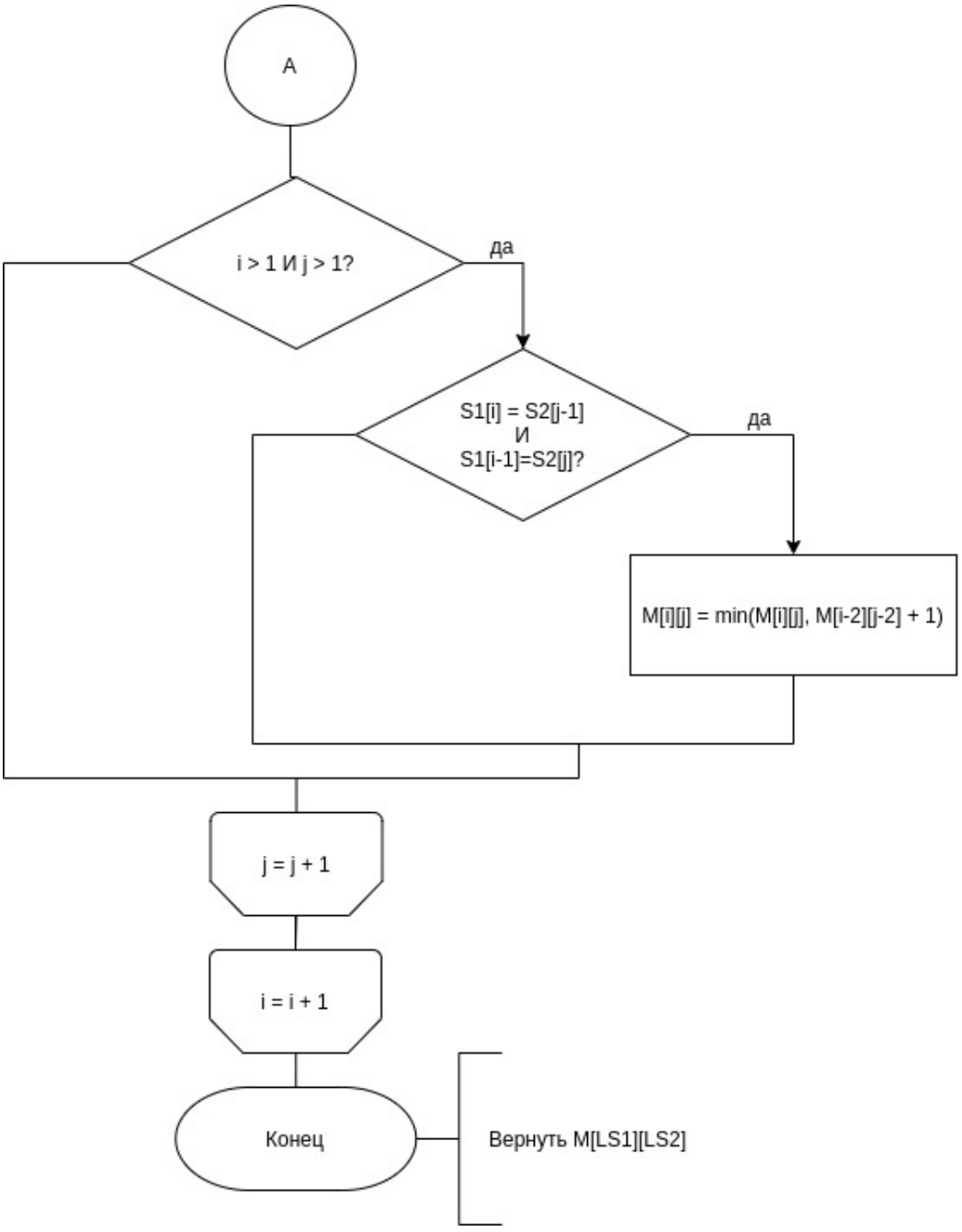

Рисунок 2.7 – Продолжение схемы матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

3.1 Требование к ПО

К программе предъявляется ряд требований:

1. предоставление возможность выбора алгоритма:
2. возможность ввода строк в любой раскладке и регистре;
3. печать на экран полученного расстояние и дополнительной информации (вспомогательные матрицы, затраченное время и объем памяти);

3.2 Средства реализации

Для реализации программы нахождения расстояние Левенштейна был выбран язык программирования Python3[2]. Данный выбор обусловлен простотой и скоростью написания программ, а также наличием встроенных библиотек для построения графиков функций и тестирования. В качестве среды разработки был выбран `PyCharm`[3], как наиболее популярная IDE для языка Python3.

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

```
1 def levenshtein_recurs(s1: str, s2: str, depth: int = 0) -> tuple[
    int, int]:
2     if not (s1 and s2):
3         return (len(s1) + len(s2)), depth
4
5     if s1[-1] == s2[-1]:
6         return levenshtein_recurs(s1[:-1], s2[:-1], depth + 1)
7
```

```
    dist, depth = min(
        levenshtein_recurs(s1[:-1], s2, depth + 1),
        levenshtein_recurs(s1, s2[:-1], depth + 1),
        levenshtein_recurs(s1[:-1], s2[:-1], depth + 1),
    )
    return dist + 1, depth
```

Листинг 3.2 – Функция нахождения расстояние Левенштейна рекурсивно с матрицей в качестве кэша

```
def levenshtein_recurs_mem(s1: str, s2: str, depth: int = 0) ->
    tuple[int, int, CacheMatrix]:
    _CACHE: CacheMatrix = [[None for _ in range(len(s2) + 1)] for _
        in range(len(s1) + 1)]
    result = _lev_rec_mem_inner(s1, s2, _CACHE, depth)
    return result[0], result[1], _CACHE


def _lev_rec_mem_inner(s1: str, s2: str, m: CacheMatrix, depth: int
    = 0) -> tuple[int, int]:
    ls1, ls2 = len(s1), len(s2)

    if m[ls1][ls2] is None:
        if not (s1 and s2):
            m[ls1][ls2] = ls1 + ls2
        elif s1[-1] == s2[-1]:
            m[ls1][ls2], depth = _lev_rec_mem_inner(s1[:-1], s2
                [:-1], m, depth + 1)
        else:
            dist, depth = min(_lev_rec_mem_inner(s1[:-1], s2, m,
                depth + 1),
                              _lev_rec_mem_inner(s1, s2[:-1], m,
                                  depth + 1),
                              _lev_rec_mem_inner(s1[:-1], s2[:-1], m
                                  , depth + 1))
            m[ls1][ls2] = dist + 1

    return m[ls1][ls2], depth
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна матрично

```python
def damerau_levenshtein(s1: str, s2: str) -> tuple[int, CacheMatrix]:
    ls1, ls2 = len(s1), len(s2)
    m = [[(i + j) if i * j == 0 else 0 for j in range(ls2 + 1)] for i in range(ls1 + 1)]

    for i in range(1, ls1 + 1):
        for j in range(1, ls2 + 1):
            if s1[i - 1] == s2[j - 1]:
                m[i][j] = m[i - 1][j - 1]
            else:
                m[i][j] = 1 + min(
                    m[i - 1][j],
                    m[i][j - 1],
                    m[i - 1][j - 1],
                    m[i - 2][j - 2]
                    if all((i >= 2, j >= 2, s1[i - 1] == s2[j - 2], s1[i - 2] == s2[j - 1])) else inf
                )
    return m[-1][-1], m
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```python
def damerau_levenshtein_recursive(s1: str, s2: str, depth: int = 0) -> tuple[int, int]:
    if not (s1 and s2):
        return len(s1) + len(s2), depth
    if s1[-1] == s2[-1]:
        return damerau_levenshtein_recursive(s1[:-1], s2[:-1], depth + 1)
    ls1, ls2 = len(s1), len(s2)
    dist, depth = min(
        damerau_levenshtein_recursive(s1[:-1], s2, depth + 1),
        damerau_levenshtein_recursive(s1, s2[:-1], depth + 1),
        damerau_levenshtein_recursive(s1[:-1], s2[:-1], depth + 1),
        damerau_levenshtein_recursive(s1[:-2], s2[:-2], depth + 1)
        if all((ls1 >= 2, ls2 >= 2, s1[-1] == s2[-2], s1[-2] == s2[-1])) else inf,
    )
    return dist + 1, depth
```

3.4 Тестовые данные

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. В колонке "Ожидаемый результат" указаны два числа - ожидаемые результаты работы алгоритмов Левенштейна и Дамерау-Левенштейна соответственно. Тестирование проводилось при помощи модуля `pytest`[4] Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Класс	**Строка 1**	**Строка 2**	**Ожидаемый результат**
Пустые строки			0 0
Одна из строк пустая		Бауманка	8 8
Одинаковые строки	Бауманка	Бауманка	0 0
Строки одинаковой длины	Бауманка	Бвуманкк	2 2
Транспозиция	Бауманка	Бауманак	2 1
Двойная тразпозиция	Александр	Аелксанрд	4 2
Полностью разные строки	МГТУ	армия	5 5
Добавить один символ	МГТ	МГТУ	1 1
Удалить один символ	МГТУ	МГТ	1 1
Заменить один символ	МГТУ	МВТУ	1 1
Латиница	Hello	Hi	4 4

3.5 Вывод

В данном разделе были разработаны исходные коды четырех алгоритмов:

1. вычисление расстояния Левенштейна рекурсивно;
2. вычисление расстояния Левенштейна рекурсивно с использованием матрицы в качестве кэша;
3. вычисление расстояния Дамерау—Левенштейна рекурсивно;
4. вычисление расстояния Дамерау—Левенштейна с заполнением матрицы;

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Ubuntu[5] Linux x86_64.
- Память: 16 GiB.
- Процессор: AMD Ryzen™ 7 4700U[6].

4.2 Время выполнения алгоритмов

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи встроенного встроенного модуля `timeit`[7]. Специальная функция делает 50 замеров и в качестве результата возвращает среднее значение.

Результаты замеров приведены в таблице 4.1 В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN. На рисунках 4.1 и 4.2 приведены графики зависимостей времени работы алгоритмов от длины строк.

Таблица 4.1 – Таблица времени выполнения алгоритмов (мс)

Длина строк	LevRec	DamLevRec	LevRecMem	DamLevMatrix
3	3.591	3.784	NaN	NaN
5	573.876	428.998	NaN	NaN
7	16357.543	20189.442	NaN	NaN
10	1434103.206	3197884.986	95.161	59.505
50	NaN	NaN	2285.094	1374.323
100	NaN	NaN	9420.838	5464.0038
200	NaN	NaN	37634.303	21797.016
300	NaN	NaN	87196.739	50530.042
400	NaN	NaN	166520.663	94606.463
500	NaN	NaN	258407.907	149814.719

4.4 Использование памяти

Алгоритмы нахождения расстояний Левенштейна 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, при этом для каждого вызова рекурсии в моей реализации требуется:

- 2 аргумента типа строка: $2 \cdot size(str) = 2 \cdot 44 = 88$ байт (примерно);
- 2 локальные переменные типа int, в моем случае: $2 \cdot 4 = 8$ байт;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт.

Таким образом получается, что при обычной рекурсии на один вызов требуется (4.1):

$$M_{percall} = 88 + 8 + 8 + 8 = 112 \text{ байта} \tag{4.1}$$

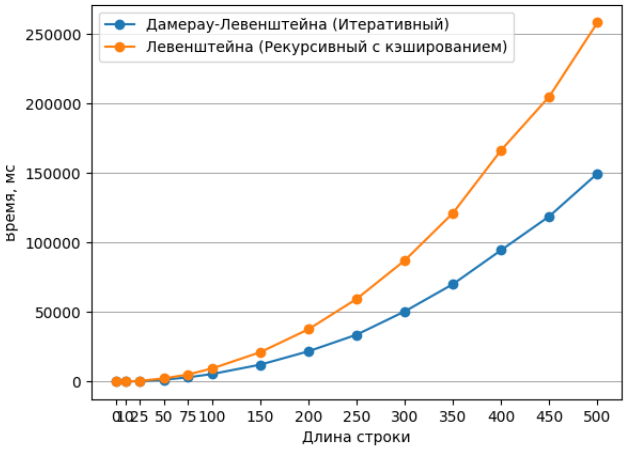

Рисунок 4.1 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна (рекурсивный с кэшированием) и Дамерау-Левенштейна (итеративный) от длины строк

Следовательно память, расходуемая в момент, когда стек вызовов максимален, равна (4.2):

$$M_{recursive} = 112 \cdot depth \tag{4.2}$$

где $depth$ - максимальная глубина стека вызовов, которая равна (4.3):

$$depth = |S_1| + |S_2| \tag{4.3}$$

где S_1, S_2 - строки.

Если мы используем рекурсивный алгоритм с заполнением матрицы матрицы, то для каждого вызова рекурсии добавляется новый аргумент - ссылка на матрицу размером 8 байт. Также в данном алгоритме требуется память на саму матрицу, размеры которой: $m = |S_1| + 1, n = |S_2| + 1$. Размер элемента матрицы равен размеру int, используемого в моей реализации, то есть 4 байт. Отсюда выходит, что память, которая тратится на хранение матрицы (4.4):

$$M_{Matrix} = (|S_1| + 1) \cdot (|S_2| + 1) \cdot 4 + 8 \tag{4.4}$$

Таким образом, при рекурсивной реализации требуемая память равна (4.5):

$$M_{recursive} = 112 \cdot depth + M_{Matrix} \tag{4.5}$$

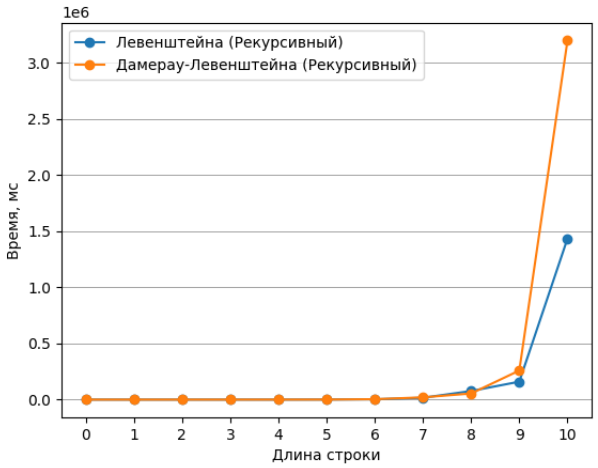

Рисунок 4.2 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна (рекурсивный) и Дамерау-Левенштейна (рекурсивный) от длины строк

где M_{Matrix} взято из соотношения 4.4.

Память, требуемая для при итеративной реализации, состоит из следующего:

- 2 локальные переменные типа int, в моем случае: $2 \cdot 4 = 8$ байт;
- 2 аргумента типа строка: $2 \cdot 44 = 88$ байта;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 4 байт;
- матрица: M_{Matrix} из соотношения 4.4.

Таким образом общая расходуемая память итеративных алгоритмов (4.6):

$$M_{iter} = M_{Matrix} + 108 \tag{4.6}$$

где M_{Matrix} определяется из соотношения 4.4.

4.5 Вывод

Рекурсивный алгоритм нахождения расстояния Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается

в геометрической прогрессии. На словах длиной 10 символов, матричная реализация алгоритма нахождения расстояния Левенштейна превосходит по времени работы рекурсивную на несколько порядков.

Алгоритм нахождения расстояния Дамерау—Левенштейна по времени выполнения сопоставим с алгоритмом нахождения расстояния Левенштейна. В нём добавлена дополнительная проверка, позволяющая находить ошибки пользователя, связанные с неверным порядком букв, в связи с чем он работает незначительно дольше, чем алгоритм нахождения расстояния Левенштейна.

В то же время по расходу памяти итеративный алгоритмы проигрывает рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк (необходимость хранить матрицу), в то время как у рекурсивного алгоритма — как сумма длин строк. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный и сравним по времени работы с матричными алгоритмами

Заключение

В ходе выполнения лабораторной работы была проделана следующая работа:

- были теоретически изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- для некоторых реализаций применены методы динамического программирования, что позволило повысить их эффективность;
- практически реализованы алгоритмы в 2 вариантах: рекурсивном и итеративном;
- на основе полученных в ходе экспериментов данных сделаны выводы по поводу эффективности всех реализованных алгоритмов;

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] Python 3.9.7 documentation [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/` (дата обращения: 08.09.2021).

[3] PyCharm: IDE для профессиональной разработки на Python [Электронный ресурс]. Режим доступа: `https://www.jetbrains.com/ru-ru/pycharm/` (дата обращения: 08.09.2021).

[4] pytest: helps you write better programs [Электронный ресурс]. Режим доступа: `https://docs.pytest.org/en/6.2.x/` (дата обращения: 08.09.2021).

[5] Ubuntu 20.04.3 LTS [Электронный ресурс]. Режим доступа: `https://ubuntu.com/download/desktop` (дата обращения: 08.09.2020).

[6] Проессор AMD Ryzen™ 7 4700U [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-7-4700u` (дата обращения: 08.09.2021).

[7] timeit — Measure execution time of small code snippets [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/timeit.html` (дата обращения: 08.09.2021).