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 курсу «Анализ алгоритмов»

«Расстояние Левенштейна и Дамерау-Левенштейна»

Студент Маслова Марина Дмитриевна

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватель Волкова Лилия Леонидовна

2021 г.

Оглавление

Введение

Расстояние Левенштейна — минимальное количество операций вставки, удаления и замены символа, необходимых для превращения одной строки в другую. Если к указанным операциям добавить перестановку двух соседних символов, получим определение *расстояния Дамерау-Левенштейна* [1]. Поиск каждой из этих характеристик основан на рекуррентных вычислениях, то есть на вычислениях, которые используют уже вычисленные значения для вычисления новых.

Таким образом, задача поиска данных расстояний основывается на методе динамического программирования — разбиении задач на более мелкие и простые подзадачи такого же вида, решение которых проводится один раз и далее используется при решении других задач и подзадач [2]. Поэтому изучение, разработка и реализация алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна позволит получить навыки использования данного метода.

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

Для достижения поставленной цели необходимо выполнить следующие **задачи**:

- изучить алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- разработать алгоритмы поиска расстояния между строками;
- оценить объем используемой алгоритмами памяти;
- определить средства программной реализации;
- реализовать указанные алгоритмы;
- провести тестирование реализованных алгоритмов;
- провести сравнительный анализ алгоритмов и процессорному времени работы реализации.

1 Аналитическая часть

В данном разделе представлено теоретическое описание алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна, а также рассмотрены области их применения.

1.1 Расстояние Левенштейна

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

Цена операции может зависеть от её вида и/или от участвующх в ней символов, что отражает разную вероятность различных ошибок при вводе текста и т. п. Для решения задачи поиска расстояния между двумя стракаами необходимо найти последовательность применяющихся операций, такую, что суммарная их цена будет минимальной. При вычислении расстояния Левенштейна используются следующие цены:

- $w(a, a) = 0$ — цена совпадения двух символов;
- $w(a, b) = 1,\ a \neq b$ — цена замены символа a на символ b;
- $w(\lambda, a) = 1$ — цена вставки символа a;
- $w(a, \lambda) = 1$ — цена удаления символа a.

1.1.1 Рекурсивный алгоритм

В рекурсивном алгоритме поиска расстояния Левенштейна между двумя строками искомая величина вычисляется через соответсвующие величины подстрок, а рекуррентная формула выводится из следующих рассуждений:

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

2) для перевода пустой строки в строку s требуется $|s|$ операций вставки (здесь и далее $|s|$ обозначает длину строки);

3) для перевода строки s в пустую строку требуется $|s|$ операций удаления;

4) для перевода строки s_1 в строку s_2 требуется выполнить некоторую последовательность операций удаления, вставки или замены, при этом операции в оптимальной последовательности можно произвольно менять места-

ми, так как две последовательные операции любых видов можно переставить, что доказывается простым перебором вариантов возможных пар, поэтому без ограничения общности можно считать, что операция над последним символом была произведена последней и цена преобразования строки s_1 в строку s_2 будет являться минимальной ценой из цен, полученных одним из следующих способов (пусть при этом s_1' и s_2' — строки s_1 и s_2 без последнего символа, соответственно):

– сумма цены преобразования строки s_1 в s_2' и цены проведения операции вставки, которая необходима для преобразования s_2' в s_2;

– сумма цены преобразования строки s_1' в s_2 и цены проведения операции удаления, которая необходима для преобразования s_1 в s_1';

– сумма цены преобразования из s_1' в s_2' и операции замены, предполагая, что s_1 и s_2 оканчиваются на разные символы;

– цена преобразования из s_1' в s_2', предполагая, что s_1 и s_2 оканчиваются на один и тот же символ.

Таким образом, для расчета расстояния Левенштейна между двумя строками s_1 и s_2 используется рекуррентная формула для расчета через подстроки:

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, & i = j = 0 \\ j, & i = 0,\ j > 0 \\ i, & j = 0,\ i > 0 \\ \min\{ & \\ \quad D(s_1[1..i], s_2[1..j-1]) + 1, & \\ \quad D(s_1[1..i-1], s_2[1..j]) + 1, & \\ \quad D(s_1[1..i-1], s_2[1..j-1]) + l(s_1[i], s_2[j]) & \\ \} & \end{cases}, \tag{1.1}$$

где величина $l(a, b)$ выражется формулой:

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

1.1.2 Матричный алгоритм

При явной реализации формулы 1.1 через рекурсию многие вызовы будут производиться при одних и тех же значениях параметров, то есть большое количество вычислений будет повторяться и не один раз. Данную проблему решает матричный алгоритм поиска расстояния Левенштейна, который представляет собой построчное заполнение матрицы с размерами $N + 1$ на $M + 1$, где N и M — размеры исходной s_1 и получаемой s_2 строк соответственно, а в ячейке с координатами (i, j) находится расстояние между подстрокой исходной строки длины i, идущей от начала строки, и подстрокой получаемой строки длины j, также идущей от начала строки.

1.1.3 Рекурсивный алгоритм с кэшем

Проблему повторяющихся вычислений можно решить и при использовании рекурсии. Для этого достаточно создать кэш в виде матрицы, где будут храниться уже вычисленные значения. Если при выполении рекурсии происходит вызов с теми данными, которые ещё не были обработаны, то необходимое значение вычисляется и заносится в соответствующую ячейку матрицы. Если же данные уже были обработаны в дальнейших вычислениях участвует значение из матрицы. Таким образом, повторных вычислений не происходит.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна[1] между двумя строками — это минимальное количество операций ставки, удаления, замены и транспозиции (перестановки двух соседних) символов, необходимых для превращения одной строки в другую.

1.2.1 Рекурсивный алгоритм

Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна полностью аналогичен алгоритму поиска расстояния Левенштейна, за исключением дополнительной операции транспозиции. Для её учета в формулу 1.1 добавляется рассмотрение случаев с возможной транспозицией последних символов. После добавления выражений, учитывающих дополнительную опера-

ции, получаем математическое представление поиска расстояния Дамерау-Левенштейна:

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, \quad i = j = 0 \\ j, \quad i = 0,\ j > 0 \\ i, \quad j = 0,\ i > 0 \\ \min\{ \\ \qquad D(s_1[1..i], s_2[1..j-1]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j-1]) + l(s_1[i], s_2[i]), \\ \qquad D(s_1[1..i-2], s_2[1..j-2]) + 1, \\ \}, \text{ если } i, j > 1 \text{ и } s_1[i] = s_2[j-1] \text{ и } s_1[i-1] = s_2[j] \\ \min\{ \\ \qquad D(s_1[1..i], s_2[1..j-1]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j-1]) + l(s_1[i], s_2[j]) \\ \}, \text{ иначе} \end{cases} . \tag{1.3}$$

1.2.2 Рекурсивный алгоритм с кешем

Также как и рекурсивный алгоритм полностью аналогичен соответсвующему алгоритму поиска расстояния Левенштейна. За счет дополнительной возможной операции в этом алгоритме для поиска расстояния от текущих подстрок анализируется дополнительная ячейка с индексами $i-2$ и $j-2$, где $i > 1$ и $j > 1$ — длины текущих подстрок.

1.3 Области применения алгоритмов

Поиск расстояний Левенштейна и Дамерау-Левенштейна широко используется в теории информации и компьютерной лингвистике. Так, он применяется:

— для автозамены, исправленя ошибок в словах (поисковые системы, базы данных, ввод текста, автоматическое распознавание отсканированного текса или речи);

– для сравнения текстовых файлов утилитой *diff*;
– в биоинформатике для сравнения генов, хромосом и белков.

1.4 Вывод

В данном разделе были описаны алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна. Для получения результирующего значения расстояния в обоих алгоритмах используется рекуррентная формула, поэтому каждый из алгоритмов может быть реализован или рекурсивно, или итерационно.

2 Конструкторская часть

В данном разделе разрабатываются алгоритмы, а также производится сравнительный анализ по используемой памяти рекурсивной и нерекурсивной их реализации на примере алгоритма поиска расстояния Левенштейна.

2.1 Разработка алгоритмов

В данном подразделе приводятся схемы разработанных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна. оло

2.1.1 Схемы алгоритмов поиска расстояния Левенштейна

Схема рекурсивного алгоритма поиска расстояния Левенштейна представлена на рисунке 2.1.

В двух следующих алгоритмах используется матрица для сохранения результатов вычислений, так как в обоих матрица инициализируется одним и тем же способом на рисунке 2.2 представен алгоритм инициализации матрицы.

Матричный алгоритм и рекурсивный алгоритм с кэшем для поиска расстояния Левенштейна приведены на рисунках 2.3 и 2.4-2.5 соответственно.

2.1.2 Схема алгоритма поиска расстояния Дамерау-Левенштейна

Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшем представлена на рисунке 2.6.

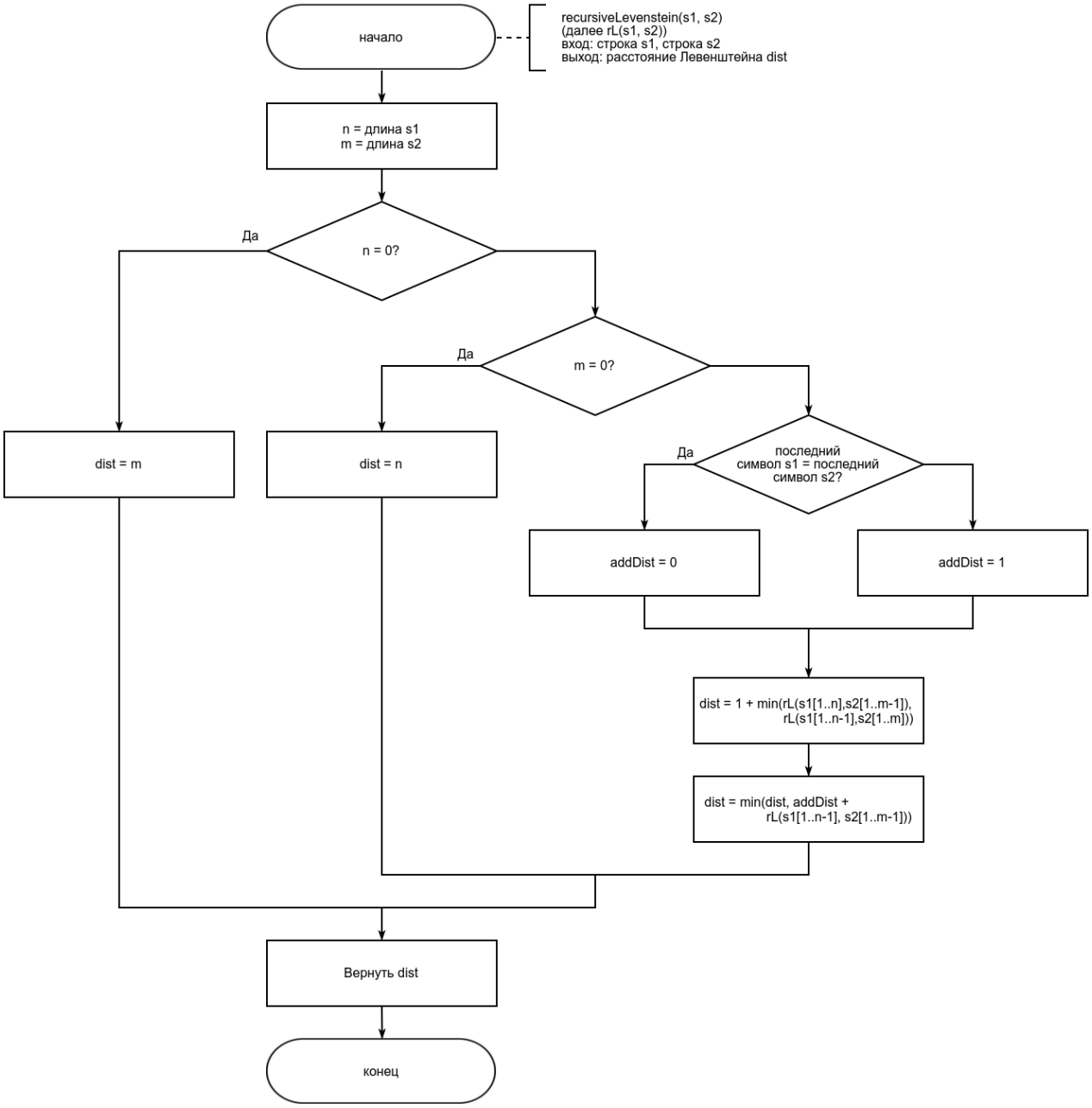

Рисунок 2.1 – Схема рекурсивного алгоритма поиска расстояния Левенштейна

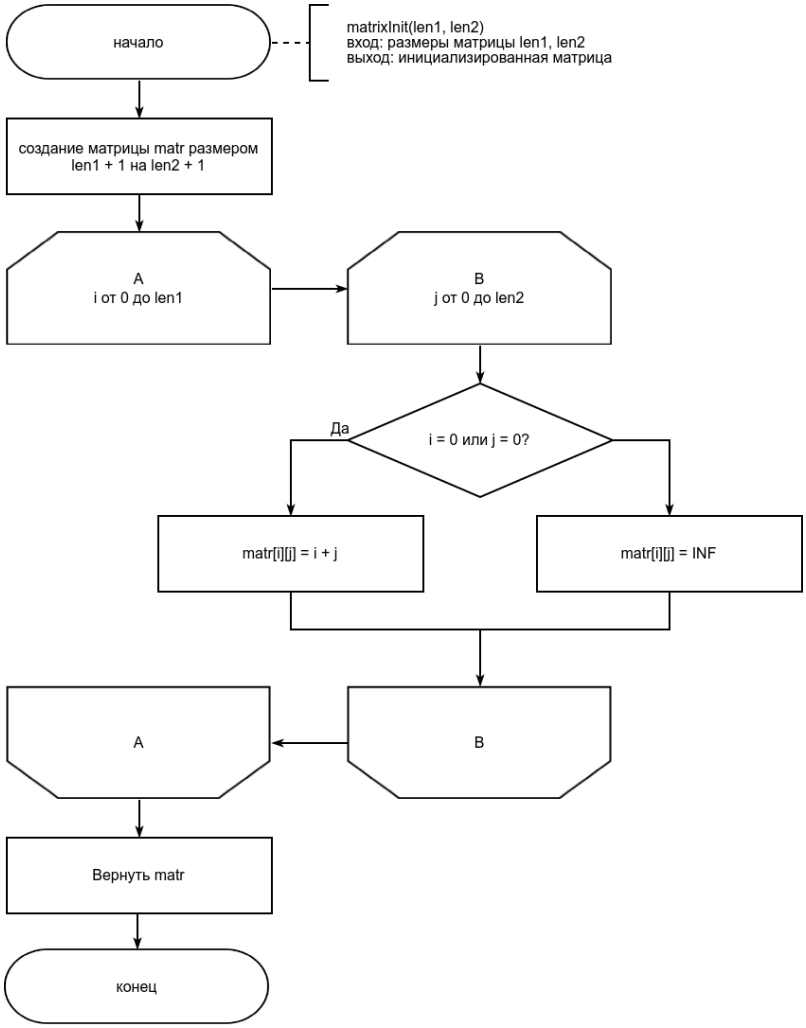

Рисунок 2.2 – Схема алгоритма инициализации матрицы

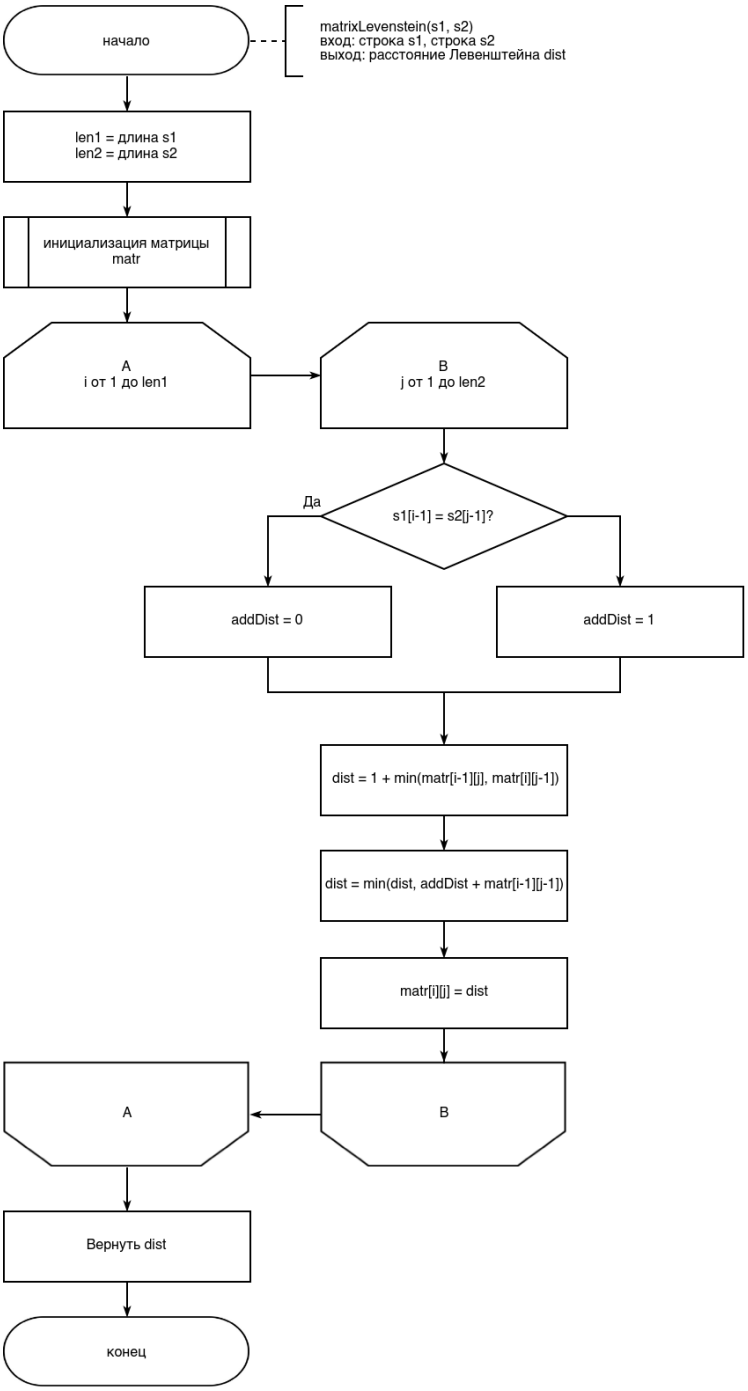

Рисунок 2.3 – Схема матричного алгоритма поиска расстояния Левенштейна

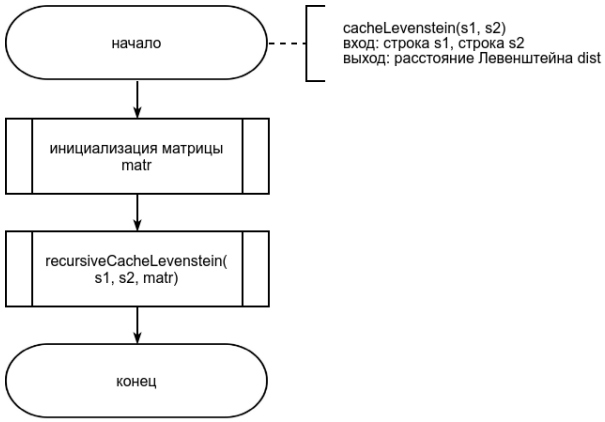

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Левенштейна с кэшем

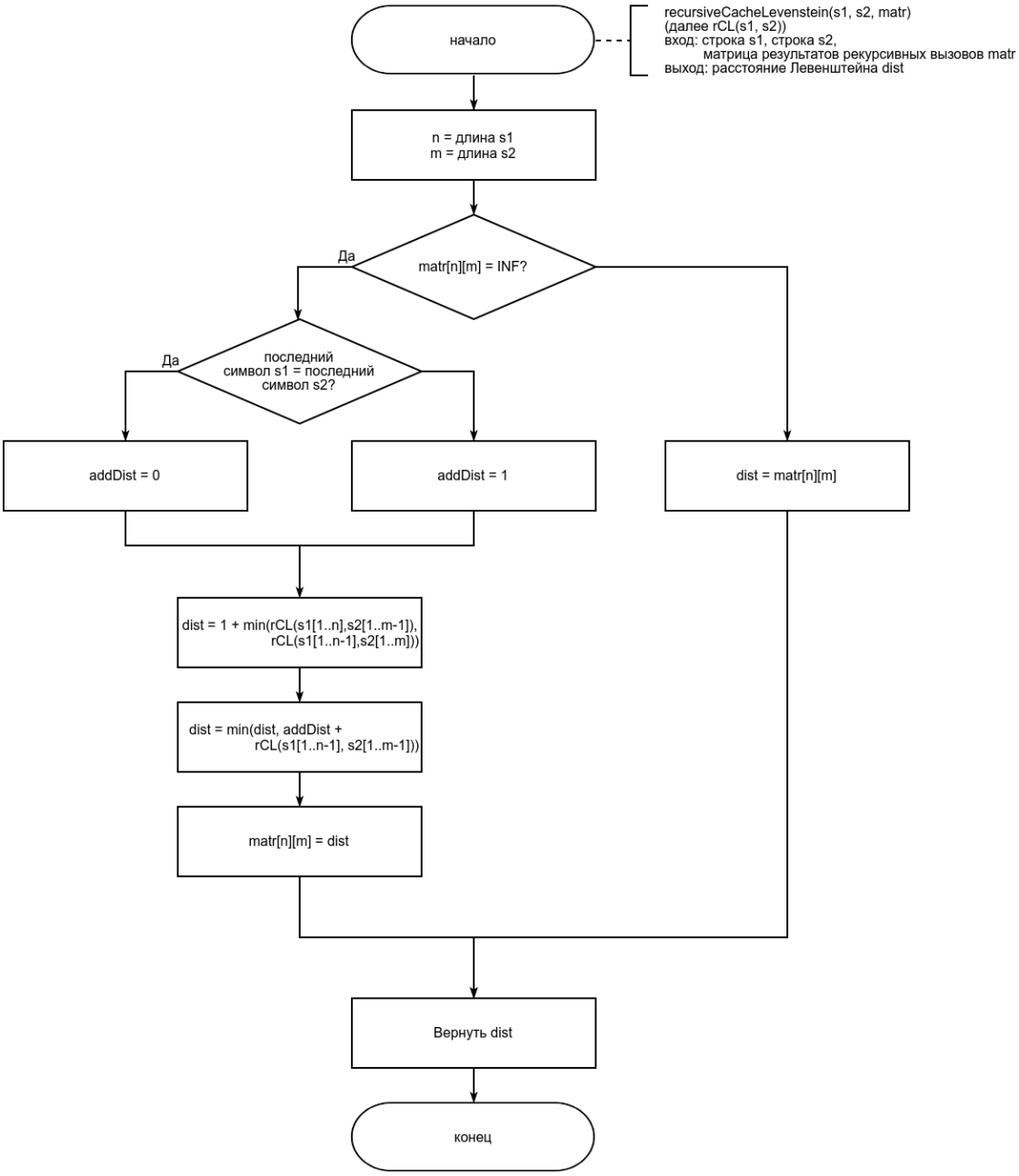

Рисунок 2.5 – Схема рекурсивной подпрограммы алгоритма поиска расстояния Левенштейна с кэшем

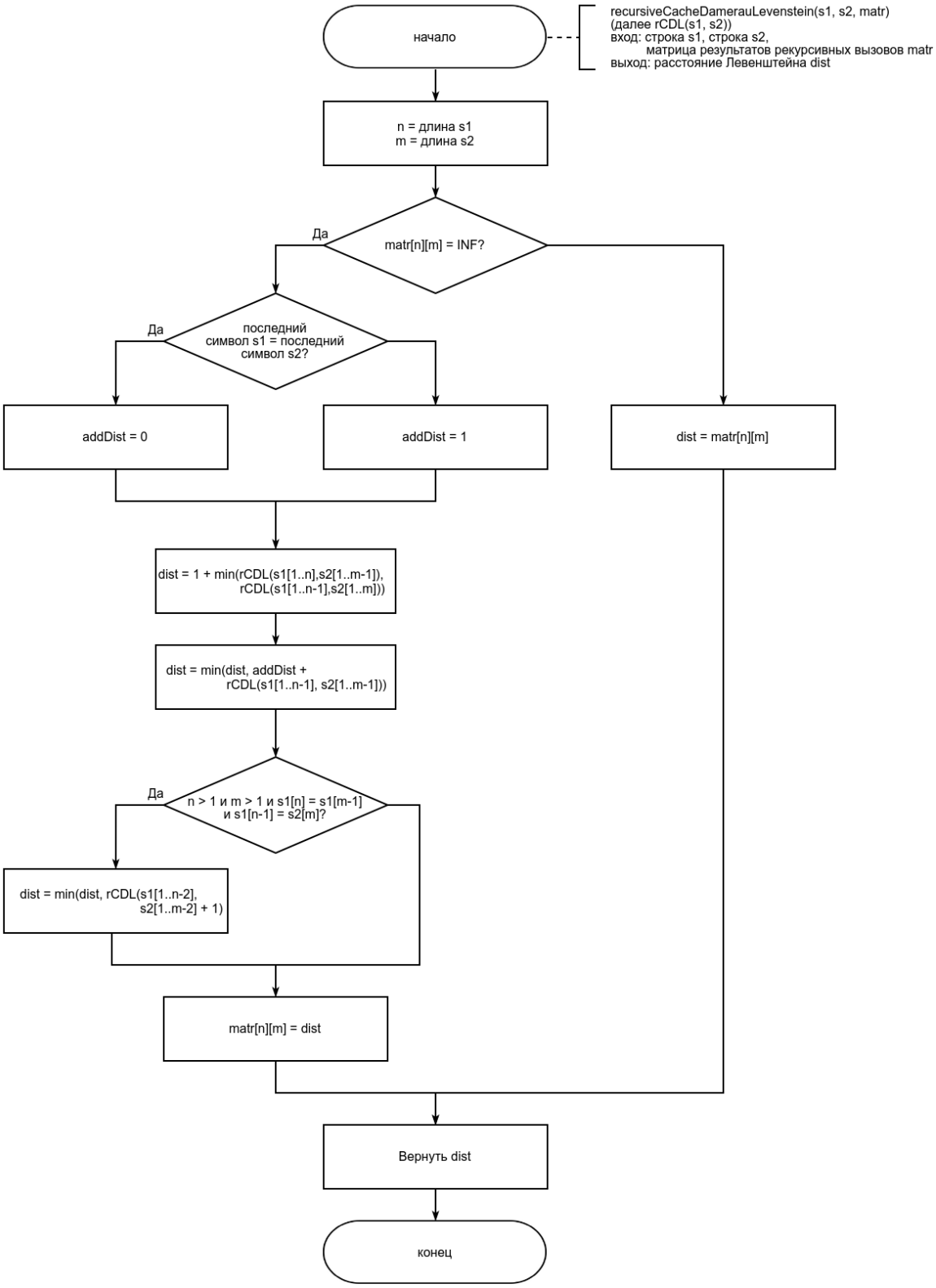

Рисунок 2.6 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшем

2.2 Сравнительный анализ рекурсивной и нерекурсивной реализации алгоритмов по используемой памяти

Схемы алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна практически индентичны за исключением нескольких блоков, включающихся при поиска расстояния Дамерау-Левенштейна из-за дополнительной операции транспозиции. Это говорит о том, что алгоритмы релизованные одним и тем же способом не будут отличаться друг от друга с точки зрения используемой памяти. Поэтому проведем анализ рекурсивной и нерекурсивной реализаций алгоритмов на примере поиска расстояния Левенштейна.

Проанализируем рекурсивную реализацию. При каждом вызове будет происходить выделения памяти под:

- 2 строки (string);
- длины строк (int);
- возвращаемое значение (int);
- адрес возврата (int);
- локальную перменную (int).

Таким образом, на каждом вызове используется память, выраженная формулой 2.1:

$$M_{call} = 5 \cdot Size(int) + 2 \cdot Size(string) \tag{2.1}$$

Количество вызовов в пике рекурсии соответствует сумме длин двух строк, то есть глубина рекурсии вычисляется по формуле 2.2:

$$depth = |S_1| + |S_2| \tag{2.2}$$

То есть память, используемая рекурсивной реализацией, выражается формулой 2.3:

$$M_{rec} = M_{call} \cdot depth \tag{2.3}$$

Теперь найдем память используемую нерекурсивной реализацией. Функция вызывается единожды, происходит выделение памяти под:

- 2 строки (string);

– длины строк (int);
– возвращаемое значение (int);
– адрес возврата (int);
– 5 локальных перменных (int);
– матрицу размерами, равными длинам строк, увеличенным на 1.

То есть память, используемая нерекурсивной реализацией, выражается формулой 2.4:

$$M_{notrec} = (9 + (|S_1| + 1)(|S_2| + 1)) \cdot Size(int) + 2 \cdot Size(string) \tag{2.4}$$

Таким образом, память, используемая рекурсивной реализацией, растет как сумма длин строк, в то же время память, используемая нерекурсивной реализацией, растет как произведение длин строк, то есть по расходу памяти итеративная реализация проигрывает рекурсивной. При рекурсивной реализации с кэшем при каждом вызове функции, как минимум, должна выделяться память под ссылку на матрицу, под которую также необходимо выделить память. То есть при рекурсивной реализации с кэшем выделяемая память будет также расти, как произведение длин строк.

2.3 Вывод

В данном разделе были разработаны и представлены в виде схем алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна в рекурсивной и итеративной форме. Также был проведен анализ используемой памяти каждой формой, в результате которого был сделан вывод о том, что итеративные алгоритмы проигрывают рекурсивным по используемой памяти, а использование кэша приравнивает рекурсивную реализации к итеративной по используемой памяти.

3 Технологическая часть

В данном разделе описаны требования к программному обеспечению, средства реализации, приведены листинги кода и данные, на которых будет проводиться тестирование.

3.1 Требования к программному обеспечению

Программа должна предоставлять следующие возможности:

- выбор режима работы: для единичного эксперимента и для массовых экспериментов;
- в режиме единичного эксперимента ввод двух строк на русском или английском языках и вывод полученных разными реализациями расстояний;
- в режиме массовых экспериментов измерение времени при различных длинах строк и построение графиков по полученным данным.

3.2 Средства реализации

Для реализации данной лабораторной работы выбран интерпретируемый язык программирования высокого уровня Python[3], так как он позволяет реализовывать сложные задачи за кратчайшие сроки за счет простоты синтаксиса и наличия большого количества подключаемых библиотек.

В качестве среды разработки выбран текстовый редактор Vim[4] с установленными плагинами автодополнения и поиска ошибок в процессе написания, так как он реализует быстрое перемещение по тексту программы и простое взаимодействие с командной строкой.

Замеры времени проводились при помощи функции process_time_ns из библиотеки time[5].

3.3 Листинги кода

В данном подразделе представлены листинги кода ранее описанных алгоритмов:

- рекурсивный алгоритм поиска расстояния Левенштейна (листинг 3.1);

– матричный алгоритм поиска расстояния Левенштейна (листинги 3.2-3.3);
– рекурсивный алгоритм поиска расстояния Левенштейна с кэшем (листинги 3.2, 3.4-3.5);
– рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем (листинг 3.6).

Листинг 3.1 – Реализация рекурсивного алгоритма поиска расстояния Левенштейна

```
1  def recursiveLevenstein(str1, str2):
2      len1 = len(str1)
3      len2 = len(str2)
4
5      dist = 0
6      if len1 == 0:
7          dist = len2
8      elif len2 == 0:
9          dist = len1
10     else:
11         addDist = 0 if str1[-1] == str2[-1] else 1
12         dist = 1 + min(recursiveLevenstein(str1, str2[:-1]),
13                        recursiveLevenstein(str1[:-1], str2))
14         dist = min(dist, addDist + recursiveLevenstein(str1[:-1], str2[:-1]))
15
16     return dist
```

Листинг 3.2 – Реализация инициализации матрицы

```
1  def matrixInit(len1, len2):
2      matrix = [[i + j if i * j == 0 else INF for j in range(len2 + 1)]
3                for i in range(len1 + 1)]
4
5      return matrix
```

Листинг 3.3 – Реализация матричного алгоритма поиска расстояния Левенштейна

```
def matrixLevenstein(str1, str2):
    len1 = len(str1)
    len2 = len(str2)

    matr = matrixInit(len1, len2)
    dist = 0

    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):
            addDist = 0 if str1[i - 1] == str2[j - 1] else 1
            dist = 1 + min(matr[i - 1][j], matr[i][j - 1])
            dist = min(dist, addDist + matr[i - 1][j - 1])
            matr[i][j] = dist

    return dist, matr
```

Листинг 3.4 – Реализация инициализации данных для рекурсивного алгоритма поиска расстояния Левенштейна с кэшем

```
def cacheLevenstein(str1, str2):
    matr = matrixInit(len(str1), len(str2))

    dist = recursiveCacheLevenstein(str1, str2, matr)

    return dist
```

Листинг 3.5 – Реализация рекурсивного алгоритма поиска расстояния Левенштейна с кэшем

```
def recursiveCacheLevenstein(str1, str2, matr):
    len1 = len(str1)
    len2 = len(str2)
    dist = 0

    if matr[len1][len2] == INF:
        addDist = 0 if str1[-1] == str2[-1] else 1
        dist = 1 + min(
                        recursiveCacheLevenstein(str1, str2[:-1], matr),
                        recursiveCacheLevenstein(str1[:-1], str2, matr),
                    )
        dist = min(
                    dist,
                    addDist
                    + recursiveCacheLevenstein(str1[:-1], str2[:-1], matr),
                )
        matr[len1][len2] = dist
    else:
        dist = matr[len1][len2]
```

Листинг 3.6 – Реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшем

```
def recursiveCacheDamerauLevenstein(str1, str2, matr):
    len1 = len(str1)
    len2 = len(str2)
    dist = 0

    if matr[len1][len2] == INF:
        addDist = 0 if str1[-1] == str2[-1] else 1
        dist = 1 + min(recursiveCacheDamerauLevenstein(str1, str2[:-1], matr),
                       recursiveCacheDamerauLevenstein(str1[:-1], str2, matr))
        dist = min(dist, addDist
                       + recursiveCacheDamerauLevenstein(str1[:-1], str2[:-1],
                                                         matr))

        if (len1 > 1 and len2 > 1
                and str1[-1] == str2[-2] and str1[-2] == str2[-1]):
            dist = min(
                    dist,
                    recursiveCacheDamerauLevenstein(str1[:-2], str2[:-2], matr)
                    + 1
                )
        matr[len1][len2] = dist
    else:
        dist = matr[len1][len2]

    return dist
```

3.4 Описание тестирования

В таблице 3.1 приведены функциональные тесты для алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Таблица 3.1 – Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат	
		Левенштейн	Дамерау-Левенштейн
’hello’	’hello’	0	0
”	”	0	0
”	’string’	6	6
’string’	”	6	6
”	’строка’	6	6
’строка’	”	6	6
’dif’	’str’	3	3
’разные’	’строки’	6	6
’len’	’diff’	4	4
’количество’	’разн’	10	10
’swap’	’sawp’	2	1
’wspa’	’swap’	3	2
’смена’	’мсена’	2	1
’мсеан’	’смена’	4	2
’add’	’adds’	1	1
’delete’	’dele’	2	2
’insrt’	’insert’	1	1
’insert’	’insrt’	1	1
’добав’	’добавить’	3	3
’удалитьош’	’удалить’	2	2
’вствить’	’вставить’	1	1
’внутери’	’внутри’	1	1
’роза’	’поза’	1	1
’bed’	’bad’	1	1
’лиса’	’сила’	2	2
’principal’	’principle’	2	2
’РеГисТР’	’регистр’	0	0
’LOW’	’low’	0	0

3.5 Вывод

В данном разделе были разработаны алгоритмы поиска расстояния Левенштейна: рекурсивный, матричный и рекурсивный с кэшем, — и рекурсивный алгоритм с кэшем поиска расстояния Дамерау-Левенштейна. Также были разработаны функциональные тесты для проверки корректности реализаций.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

– Операционная система: Manjaro [6] Linux x86_64.

– Память: 8 GiB.

– Процессор: Intel® Core™ i5-8265U[7].

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

4.2 Примеры работы программы

В данном подразделе представлены примеры работы программы. На рисунке 4.1 приведен пример работы программы при вводе строк в русской раскладке и равными расстояниями Левенштейна и Дамерау-Левенштейна. На рисунке 4.2 приведен пример работы программы при вводе строк в английской раскладке и разными полученными значениями расстояний.

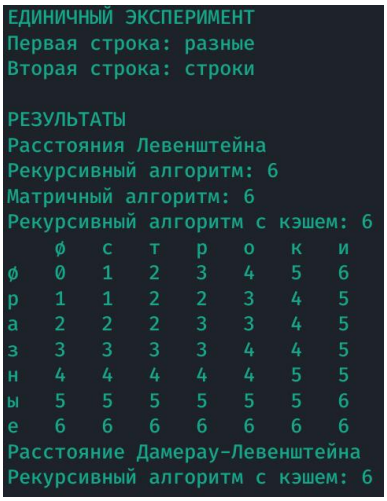

Рисунок 4.1 – Пример работы программы в русской раскладке

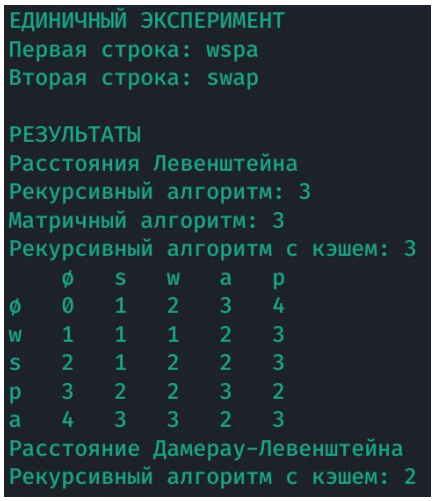

Рисунок 4.2 – Пример работы программы в английской раскладке

4.3 Результаты тестирования

В таблице 4.1 приведены результаты работы программы на тестах, описанных в таблице 3.1. В результате сравнения ожидаемого и полученного результата делаем вывод, что все тесты были пройдены.

Таблица 4.1 – Результаты тестирования

Строка 1	**Строка 2**	**Полученный результат**	
		Левенштейн	**Дамерау-Левенштейн**
’hello’	’hello’	0	0
”	”	0	0
”	’string’	6	6
’string’	”	6	6
”	’строка’	6	6
’строка’	”	6	6
’dif’	’str’	3	3
’разные’	’строки’	6	6
’len’	’diff’	4	4
’количество’	’разн’	10	10
’swap’	’sawp’	2	1
’wspa’	’swap’	3	2
’смена’	’мсена’	2	1
’мсеан’	’смена’	4	2
’add’	’adds’	1	1
’delete’	’dele’	2	2
’insrt’	’insert’	1	1
’insert’	’insrt’	1	1
’добав’	’добавить’	3	3
’удалитьош’	’удалить’	2	2
’вствить’	’вставить’	1	1
’внутери’	’внутри’	1	1
’роза’	’поза’	1	1
’bed’	’bad’	1	1
’лиса’	’сила’	2	2
’principal’	’principle’	2	2
’РеГисТР’	’регистр’	0	0
’LOW’	’low’	0	0

4.4 Постановка эксперимента по замеру времени

Был проведен эксперимент для определения влияния длины строк на время работы каждого вида алгоритмов и выявления более эффективной по времени реализации. Тестирование проводилось на строках длиной от 10 до 170 с шагом 20. Так как от запуска к запуску процессорное время, затрачиваемое на выполнение агоритма менялось в определенном промежутке времени, необходимо было усреднить вычисляемые значения. Для этого алгоритм на каждом значении длины строк запускался 10 раз, и для полученных 10 значений определялось среднее арифметическое, которое и заносилось в таблицу результатов.

Так как алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна подобны друг другу не имело особого практического значения делать замеры для каждого из алгоритмов, поэтому были проведены замеры времени для различных реализации поиска расстояния Левенштейна, а также для реализации с кэшем каждого из алгоритмов. Таким образом, были получены экспериментальные значения для сравнения разный реализаций одного алгоритма и сравнения одинаковых реализаций разных алгоритмов.

4.5 Результаты эксперимента

В таблице 4.2 представлены результаты измерения времени для разных реализаций алгоритма поиска расстояния Левенштейна. Так как даже при длине в 10 символов рекурсивный алгоритм работал в 20-50 тысяч раз медленнее реализаций с матрицей, вычисления при больших длинах не производились, что помечено в таблице как $None$. На основе табличных данных построены графики зависимости времени работы каждой из реализаций с использованием матрицы от длины строк 4.3.

В таблице 4.3 представлены результаты измерения времени для реализации с кэшем алгоримтов поиска расстояний Левенштейна и Дамерау-Левенштейна. На основе табличных данных построены графики зависимости времени работы каждого алгоритма от длины строк 4.4.

Таблица 4.2 – Время работы различных реализаций алгоритма поиска расстояния Левенштейна (время в наносекундах)

Длина	Рекурсивный	Матричный	Рекурсивный с кэшем
10	3108129819	60808	114971
30	None	487183	933851
50	None	1319384	2538327
70	None	2562695	4954486
90	None	4188278	8117175
110	None	6242140	12210075
130	None	8707854	17054196
150	None	11565148	22734575
170	None	14847137	29335572

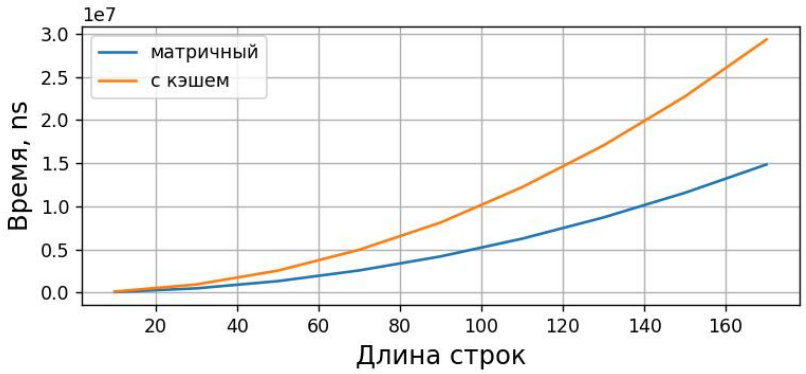

Рисунок 4.3 – Графики зависимости времени выполнения алгоримтов поиска расстояния Левенштейна от длины строк

Таблица 4.3 – Время работы алгоритмов с кэшем поиск расстояния Левенштейна и Дамерау-Левенштейна

Длина	Левенштейн	Дамерау-Левенштейн
10	106740	120003
30	905828	996809
50	2594842	2728086
70	4969278	5336874
90	8209447	8817052
110	12200319	13253322
130	17101509	18525322
150	22770071	24734911
170	29498979	31768124

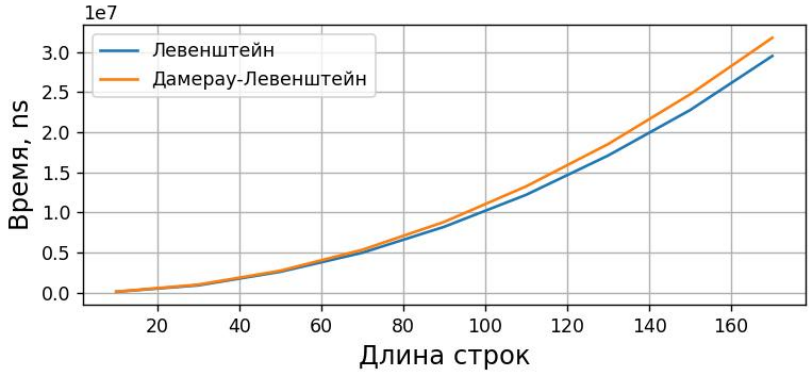

Рисунок 4.4 – Графики зависимости времени выполнения алгоритмов с кэшем поиска расстояния Левенштейна и Дамерау-Левенштейна от длины строк

4.6 Вывод

Рекурсивный алгоритм поиска расстояния Левенштейна работает в 20-50 тысяч раз дольше алгоритмов, использующих матрицу, уже при значении длин строк, равных 10, что свидельствует о неэффективности этого алгоритма по времени. Матричный алгоритм и алгоритм с кэшем работают в порядки раз быстрее рекурсивного алгоритма. Сравнение данных алгоритмов между друг другом приводит нас к выводу о том, что на значениях длин строк до 200 матричный алгоритм работает примерно в 1.9 раз быстрее рекурсивного алгоритма с кэшем.

Разница во времени выполнения рекурсивных алгоритмов с кэшем поиска расстояния Левенштейна и Дамерау-Левенштейна незначительна, а именно при поиске расстояния Дамерау-Левенштейна алгоримт работает примерно 1.1 раза дольше, что объясняется проверкой дополнительной операции, которая в свою очередь позволяет сократить число операций для исправления частой ошибки при наборе текста к клавитуры.

Заключение

В ходе выполения лабораторной работы:

- были описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна в том числе в математической форме;
- были применены методы динамического программирования для уменьшения времени работы рекурсивного алгоритма;
- была произведена оценка памяти, используемой каждым алгоритмом;
- были реализованы несколько вариантов алгоритмов: рекурсивный, матричный и рекурсивный с кэшем;
- по экспериментальным данным были сделаны выводы об эффективности по времени реализованых алгоритмов.

Литература

[1] Черненький В. М., Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным // Инженерный журнал: наука и инновации. 2012. № 3. С. 30–39.

[2] Окулов С. М., Пестов О. А. Динамическое программирование. М.: БИНОМ. Лаборатория знаний, 2012. с. 296.

[3] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 12.10.2021).

[4] welcome home : vim online [Электронный ресурс]. Режим доступа: `https://www.vim.org/` (дата обращения: 12.10.2021).

[5] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#time.process_time_ns` (дата обращения: 04.10.2021).

[6] Manjaro — enjoy the simplicity [Электронный ресурс]. Режим доступа: `https://manjaro.org/` (дата обращения: 17.10.2021).

[7] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/149088/intel-core-i5-8265u-processor-6m-cache-up-to-3-90-ghz.html` (дата обращения: 17.10.2021).