



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"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Динь Вьет Ань

Группа ИУ7И-54Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

Москва — 2022 г.

Оглавление

Введение	3
1 Аналитическая часть	5
1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна	5
1.2 Матричный алгоритм нахождения расстояния Левенштейна	7
1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша	7
1.4 Расстояние Дамерау — Левенштейна	8
2 Конструкторская часть	10
2.1 Требования к вводу	10
2.2 Требования к программе	10
2.3 Схема алгоритма нахождения расстояния Левенштейна . . .	10
2.4 Схема алгоритма нахождения расстояния Дамерау — Левенштейна	11
3 Технологическая часть	16
3.1 Требования к ПО	16
3.2 Средства реализации	16
3.3 Сведения о модулях программы	16
3.4 Листинг кода	17
3.5 Функциональные тесты	20
4 Исследовательская часть	22
4.1 Технические характеристики	22
4.2 Демонстрация работы программы	22
4.3 Время выполнения алгоритмов	22
4.4 Использование памяти	25
Заключение	28

Введение

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

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной строки в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены. Широко используется в теории информации и компьютерной лингвистике.

Расстояние Левенштейна и его обобщения активно применяются для:

- исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);
- сравнения генов, хромосом и белков в биоинформатике.

Расстояние Дameraу — Левенштейна (названо в честь учёных Фредерика Дameraу и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна, так как к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для реализации алгоритмов;
3. получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна;
4. сравнительный анализ алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. экспериментальное подтверждение различий во временной эффективности алгоритмов 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

В этом разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дameraу-Левенштейна и их практическое применение.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

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

Каждая операция имеет свою цену (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначения операций:

- D (англ. delete) — удаление ($w(a, \lambda) = 1$);
- I (англ. insert) — вставка ($w(\lambda, b) = 1$);
- R (англ. replace) — замена ($w(a, b) = 1, a \neq b$);
- M (англ. match) - совпадение ($w(a, a) = 0$).

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:

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

Функция 1.2 определена как:

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

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

- 1) сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a ;
- 2) сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b ;
- 3) сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;
- 4) 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

При больших i, j прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения, так как множество промежуточных значения $D(i, j)$ вычисляются не по одному разу. Для оптимизации нахождения можно использовать матрицу для хранения соответствующих промежуточных значений.

Матрица размером $(length(S1) + 1) \times (length(S2) + 1)$, где $length(S)$ — длина строки S . Значение в ячейке $[i, j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой 1.3.

$$A[i][j] = \min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j]) \end{cases} \quad (1.3)$$

Функция 1.4 определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases} \quad (1.4)$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием кеша [?]. В качестве кеша используется мат-

рица. Суть данной оптимизации 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 Расстояние Дамерау — Левенштейна

Расстояние Дамерау-Левенштейна [?] - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.5, которая задана как

$$d_{a,b}(i, j) = \begin{cases} \max(i, j), & \text{если } \min(i, j) = 0, \\ \min\{ \\ \quad d_{a,b}(i, j - 1) + 1, \\ \quad d_{a,b}(i - 1, j) + 1, \\ \quad d_{a,b}(i - 1, j - 1) + m(a[i], b[j]), & \text{иначе} \\ \quad \left[\begin{array}{ll} d_{a,b}(i - 2, j - 2) + 1, & \text{если } i, j > 1; \\ & a[i] = b[j - 1]; \\ & b[j] = a[i - 1] \\ & \infty, & \text{иначе} \end{array} \right. \\ \} \end{cases}, \quad (1.5)$$

Формула выводится по тем же соображениям, что и формула (1.1).

Вывод

Формулы Левенштейна и Дамерау — Левенштейна для расчета расстояния между строками задаются рекуррентно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

1. На вход подаются две строки.
2. Буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться.
2. На выход программа должна вывести число - расстояние Левенштейна (Дамерау-Левенштейна), матрицу при необходимости.

2.3 Схема алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

На рисунке 2.2 приведена схема алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

На рисунке 2.3 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием кеша в виде матрицы.

2.4 Схема алгоритма нахождения расстояния Дамерау — Левенштейна

На рисунке 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна.

Вывод

Перечислены требования к вводу и программе, а также на основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

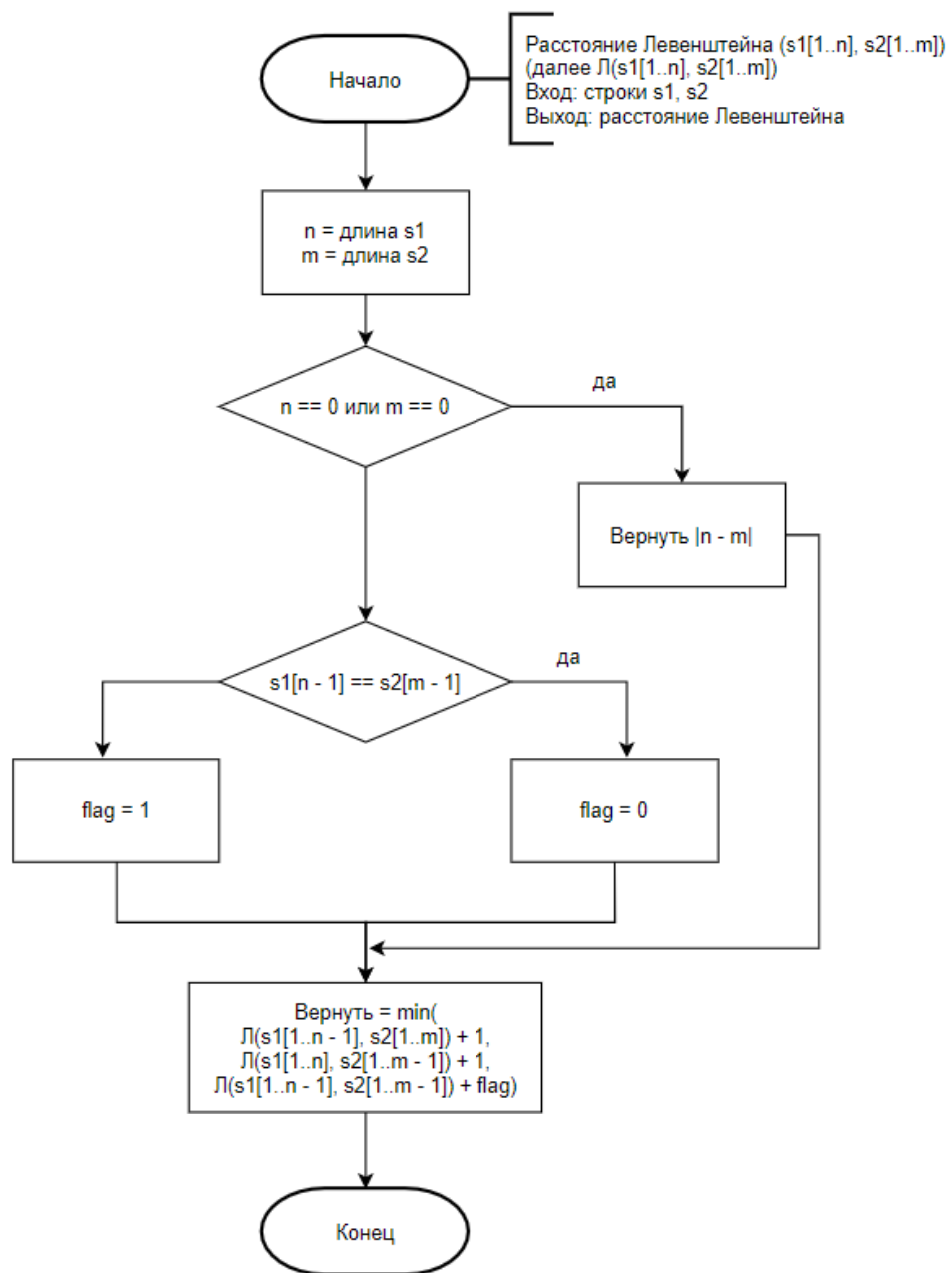


Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

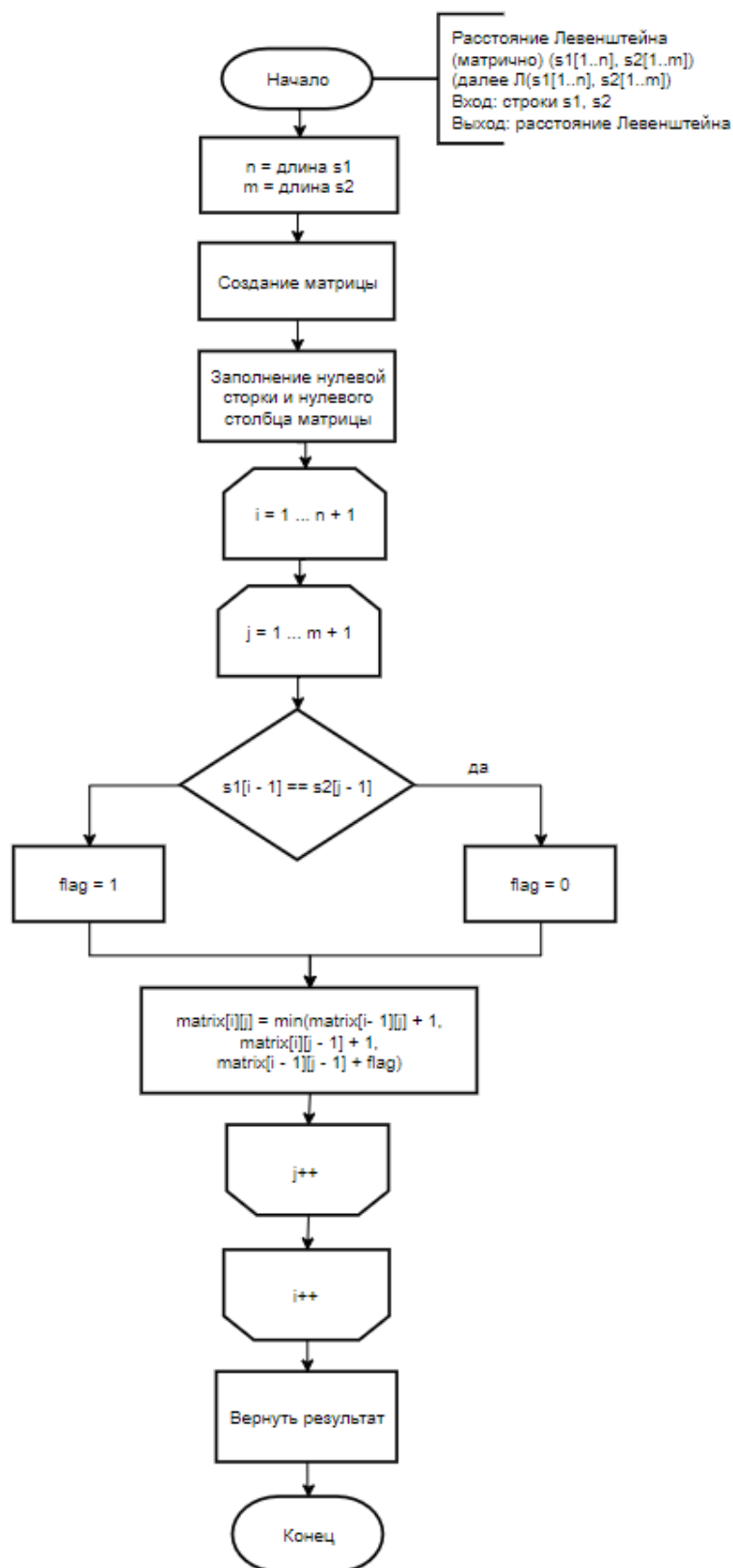


Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Левенштейна

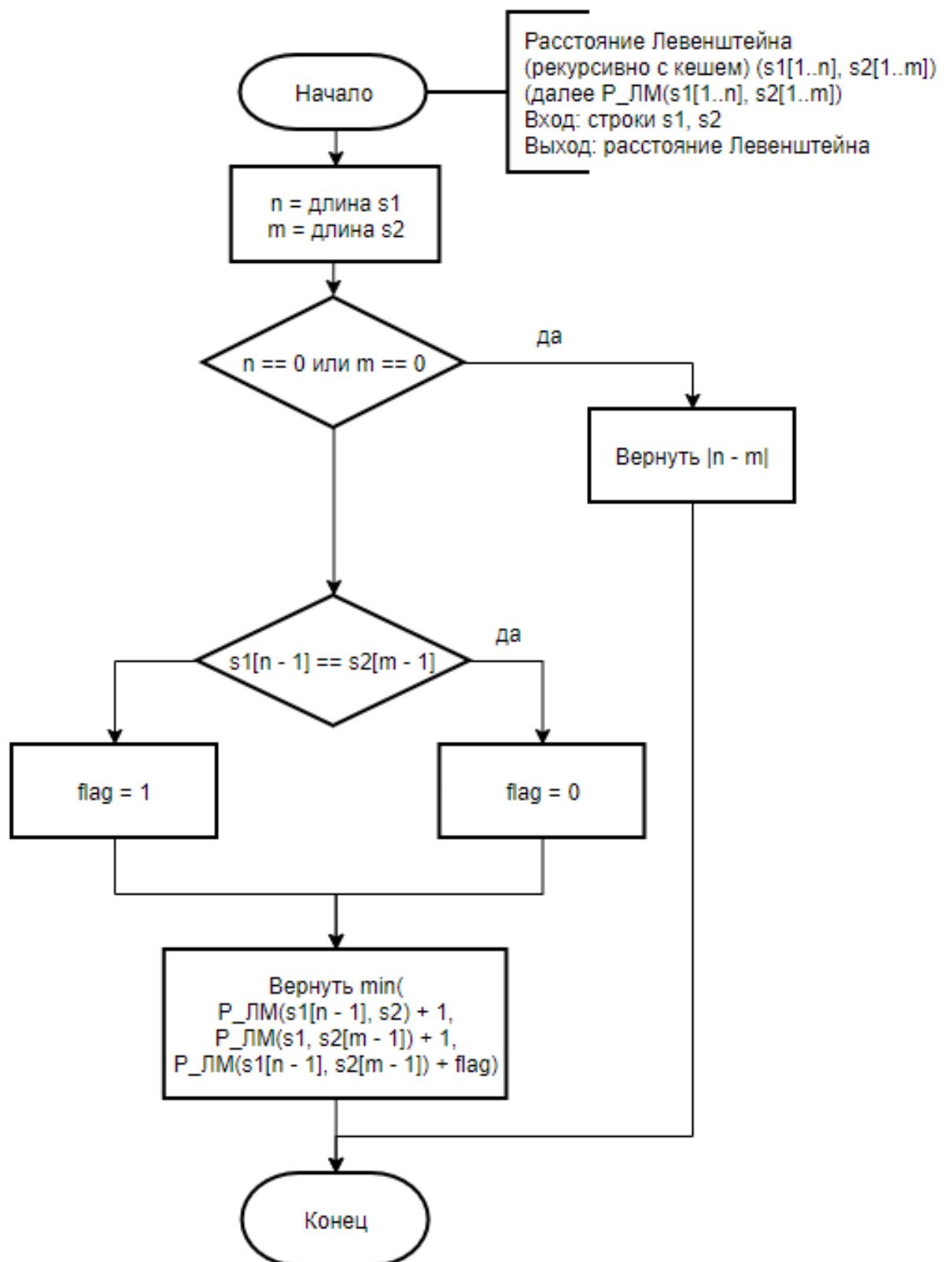


Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием кеша (матрицы)

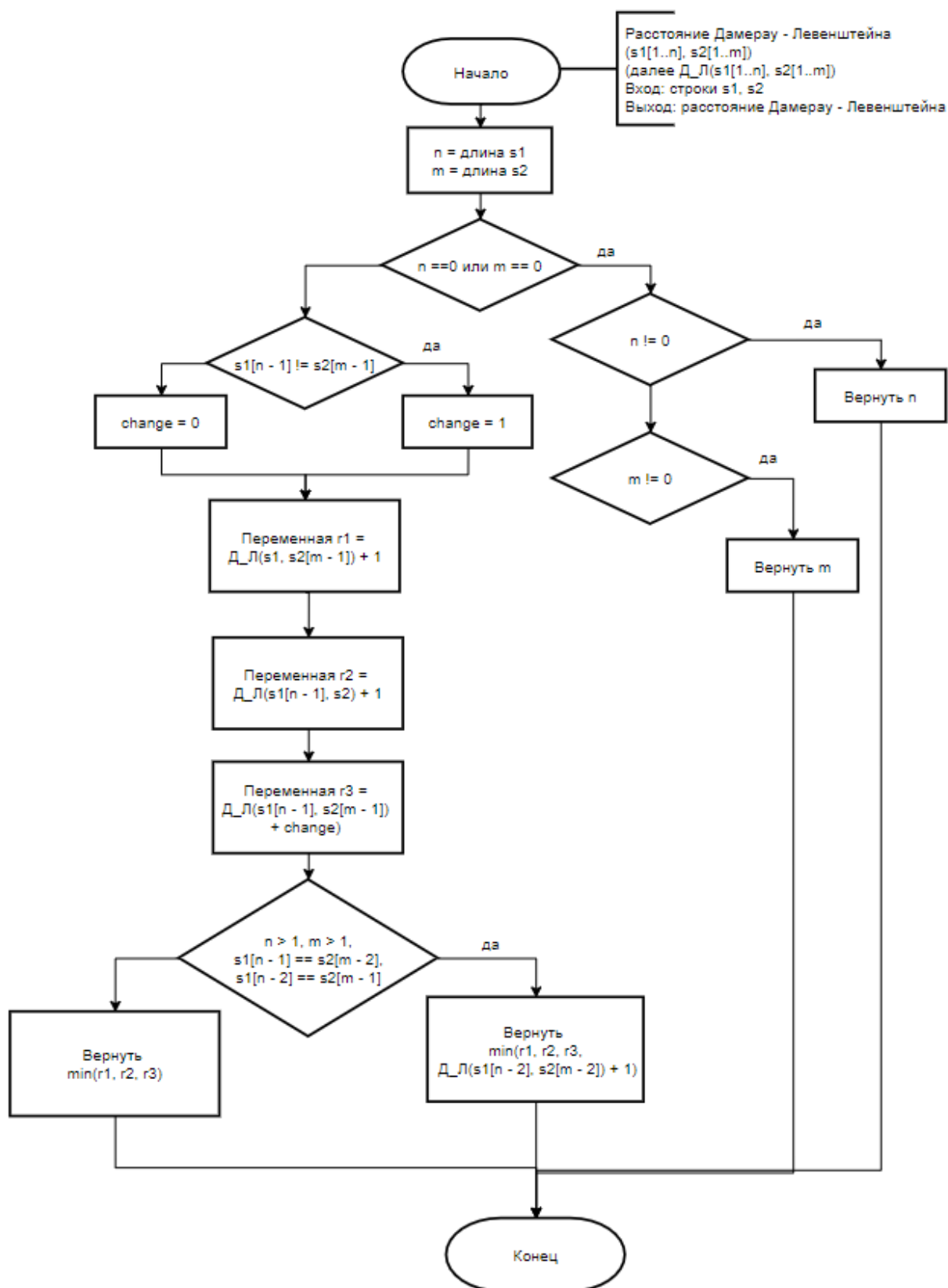


Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- у пользователя есть выбор алгоритма, или какой-то один, или все сразу, а также есть выбор тестирования времени;
- на вход подаются две строки на русском или английском языке в любом регистре;
- на выходе — искомое расстояние для выбранного метода (выбранных методов) и матрицы расстояний для матричных реализаций.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран ЯП Python [?].

Данный язык достаточно удобен и гибок в использовании.

Время работы алгоритмов было замерено с помощью функции `process_time()` из библиотеки `time` [?]

3.3 Сведения о модулях программы

Программа состоит из двух модулей:

1. `main.py` - главный файл программы, в котором располагаются коды всех алгоритмов и меню;
2. `test.py` - файл с замерами времени для графического изображения результата.

3.4 Листинг кода

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дamerau–Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием рекурсии.

```
1 def levenshtein_recursive(str1, str2, out_put = False):
2     n = len(str1)
3     m = len(str2)
4
5     if n == 0 or m == 0:
6         return abs(n - m)
7
8     flag = 0
9     if str1[-1] != str2[-1]:
10         flag = 1
11
12     min_lev_rec = min(levenshtein_recursive(str1[:-1], str2) + 1,
13                       levenshtein_recursive(str1, str2[:-1]) + 1,
14                       levenshtein_recursive(str1[:-1], str2[:-1])
15                           + flag)
16
17     return min_lev_rec
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна с использованием рекурсии.

```
1 def levenshtein_matrix(str1, str2, out_put = True):
2     n = len(str1)
3     m = len(str2)
4
5     matrix = create_matrix(n + 1, m + 1)
6
7     for i in range(1, n + 1):
8         for j in range(1, m + 1):
9             add, delete, change = matrix[i - 1][j] + 1, \
10                                     matrix[i][j - 1] + 1, \
11                                     matrix[i - 1][j - 1]
12             if str2[j - 1] != str1[i - 1]:
13                 change += 1
```

```

14         else :
15             change += 0
16
17             matrix[i][j] = min(add, delete, change)
18
19     if out_put:
20         print_matrix(str1, str2, matrix)
21
22     return matrix[n][m]

```

Листинг 3.3 – Функция нахождения расстояния Левенштейна с использованием рекурсии.

```

1 def levenshtein_matrix_recursive(str1, str2, out_put = True):
2     n = len(str1)
3     m = len(str2)
4
5     def recursive(str1, str2, n, m, matrix):
6         if (matrix[n][m] != -1):
7             return matrix[n][m]
8
9         if (n == 0):
10             matrix[n][m] = m
11             return matrix[n][m]
12
13         if (n > 0 and m == 0):
14             matrix[n][m] = n
15             return matrix[n][m]
16
17         delete = recursive(str1, str2, n - 1, m, matrix) + 1
18         add = recursive(str1, str2, n, m - 1, matrix) + 1
19
20         flag = 0
21
22         if (str1[n - 1] != str2[m - 1]):
23             flag = 1
24
25         change = recursive(str1, str2, n - 1, m - 1, matrix) +
26             flag
27
28         matrix[n][m] = min(add, delete, change)

```

```

29         return matrix[n][m]
30
31     matrix = create_matrix(n + 1, m + 1)
32
33     for i in range(n + 1):
34         for j in range(m + 1):
35             matrix[i][j] = -1
36
37     recursive(str1, str2, n, m, matrix)
38
39     if out_put:
40         print_matrix(str1, str2, matrix)
41
42     return matrix[n][m]

```

Листинг 3.4 – Функция нахождения расстояния Левенштейна с использованием рекурсии.

```

1 def damerau_levenshtein_recursive(str1, str2, out_put = False):
2     n = len(str1)
3     m = len(str2)
4
5     if n == 0 or m == 0:
6         if n != 0:
7             return n
8         if m != 0:
9             return m
10        return 0
11
12    change = 0
13    if str1[-1] != str2[-1]:
14        change += 1
15
16    if n > 1 and m > 1 and str1[-1] == str2[-2] \
17        and str1[-2] == str2[-1]:
18        min_ret = min(damerau_levenshtein_recursive(str1[:n - 1],
19                                                    str2) + 1,
20                    damerau_levenshtein_recursive(str1, str2[:m
21                                                    - 1]) + 1,
22                    damerau_levenshtein_recursive(str1[:n - 1],
23                                                    str2[:m - 1]) + change,
24                    damerau_levenshtein_recursive(str1[:n - 2],

```

```

22         str2[:m - 2]) + 1)
23     else:
24         min_ret = min(damerau_levenshtein_recursive(str1[:n - 1],
25             str2) + 1,
26             damerau_levenshtein_recursive(str1, str2[:m
27         - 1]) + 1,
28             damerau_levenshtein_recursive(str1[:n - 1],
29             str2[:m - 1]) + change)
30     return min_ret

```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дameraу — Левенштейна (в таблице - "Дameraу-Л."). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

№	Строка 1	Строка 2	Ожидаемый результат	
			Левенштейн	Дameraу-Л.
1	скат	кот	2	2
2	машина	малина	1	1
3	дворик	доврик	2	1
4	"пустая строка"	университет	11	11
5	сентябрь	"пустая строка"	8	8
8	тело	телодвижение	8	8
9	ноутбук	планшет	7	7
10	глина	малина	2	2
11	рекурсия	ркерусия	3	2
12	браузер	баурзер	2	2
13	bring	brought	4	4
14	moment	minute	4	4
15	person	eye	5	5
16	week	weekend	3	3
17	city	town	4	4

Вывод

Были разработаны и протестированы алгоритмы: нахождения расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также нахождения расстояния Дамерау — Левенштейна рекурсивно.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Ubuntu 20.04.3 [?] Linux [?] x86_64.
- Память: 8 GiB.
- Процессор: 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz [?].

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

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи функции `process_time()` из библиотеки `time` языка Python. Данная функция всегда возвращает значения времени, а именно сумму системного и пользовательского процессорного времени текущего процессора, типа `float` в секундах.

Контрольная точка возвращаемого значения не определена, поэтому допустима только разница между результатами последовательных вызовов.

```

МЕНЮ:
0. Выход
1. Расстояние Левенштейна рекурсивно
2. Расстояние Левенштейна матрица
3. Расстояние Левенштейна рекурсивно (кеш)
4. Расстояние Дамерау-Левенштейна рекурсивно
5. Все алгоритмы вместе
6. Замер времени (длина слов от 1 до 10)
Выбор: 5
Введите первую строку 1: скат
Введите первую строку 2: котик

Расстояние Левенштейна, полученное с использованием рекурсии: 4
Расстояние, вычисленное с помощью матрицы Левенштейна:
0 0 к о т и к
0 0 1 2 3 4 5
с 1 1 2 3 4 5
к 2 1 2 3 4 4
а 3 2 2 3 4 5
т 4 3 3 2 3 4
Расстояние равно 4
Расстояние, вычисленное с помощью матрицы Левенштейна (рекурсивно)
0 0 к о т и к
0 0 1 2 3 4 5
с 1 1 2 3 4 5
к 2 1 2 3 4 4
а 3 2 2 3 4 5
т 4 3 3 2 3 4
Расстояние равно 4

Расстояние Дамерау-Левенштейна, полученное с использованием рекурсии: 4

```

Рисунок 4.1 – Пример работы программы

Замеры времени для каждой длины слов проводились 100 раз. В качестве результата взято среднее время работы алгоритма на данной длине слова.

Результаты замеров приведены в таблице 4.1 (время в мкс).

На рисунке ?? представлен график сравнения рекурсивных реализаций алгоритмов поиска расстояния Левенштейна с использованием кеша и без. На графике видно, что полученные результаты частично накладываются друг на друга (до длины равной 6).

На рисунке ?? представлен график сравнения рекурсивных реализаций алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна. На графике видно, что полученные результаты частично накладываются друг на друга (до длины равной 7).

На рисунке 4.4 представлен график сравнения матричной реализаций

Таблица 4.1 – Результаты замеров времени.

Длина	Л.(матр.)	Л.(рек с matr.)	Л.(рек)	Д.-Л.(рек.)
0	1.43534	1.80707	0.43232	0.41829
1	2.26853	2.91881	1.25935	1.23623
2	4.20428	6.41375	5.12573	5.43740
3	9.01130	14.53429	31.56933	33.42077
4	10.05873	16.91644	114.69468	127.88060
5	15.02883	25.31050	633.12262	692.21623
6	15.91587	27.53538	2650.31546	2918.78975
7	23.33747	37.99432	15230.66448	16730.88525
8	42.62168	74.88248	123855.96710	135893.39700
9	51.07370	87.96769	639529.53498	700352.48553

алгоритма поиска расстояния Левенштейна и рекурсивного алгоритма поиска расстояния Левенштейна с использованием кеша.

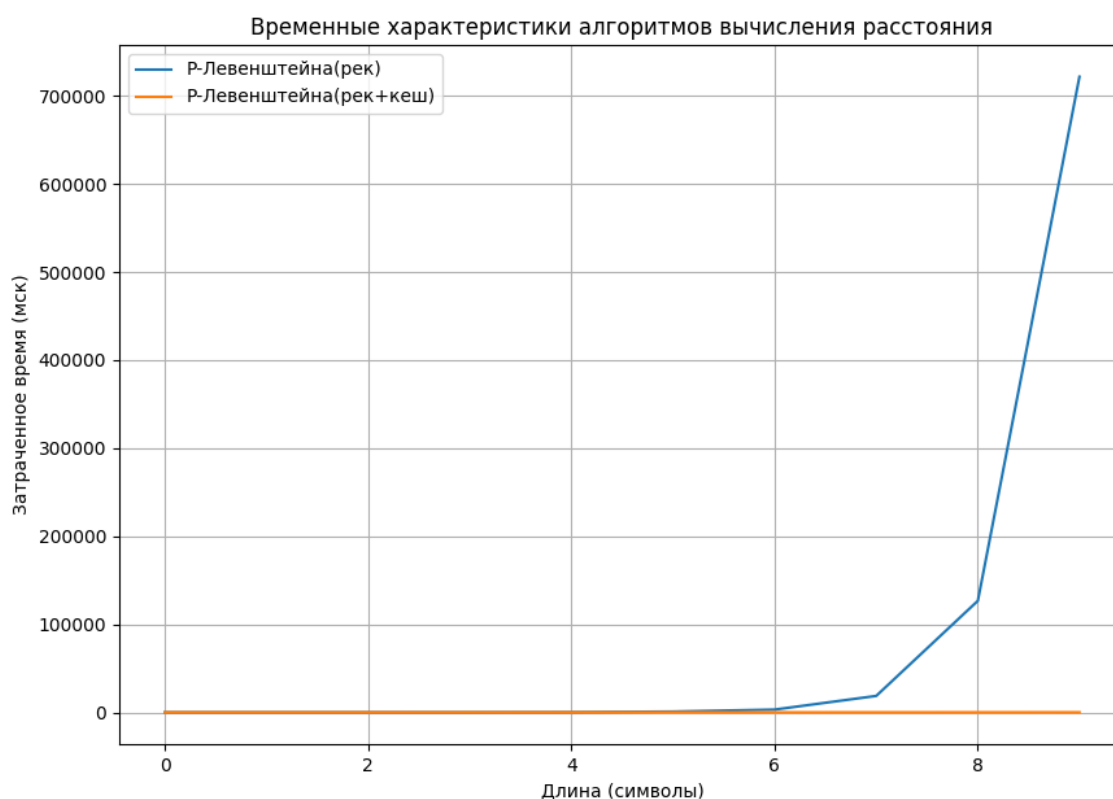


Рисунок 4.2 – Сравнения рекурсивных алгоритмов поиска расстояния Левенштейна с использованием кеша и без

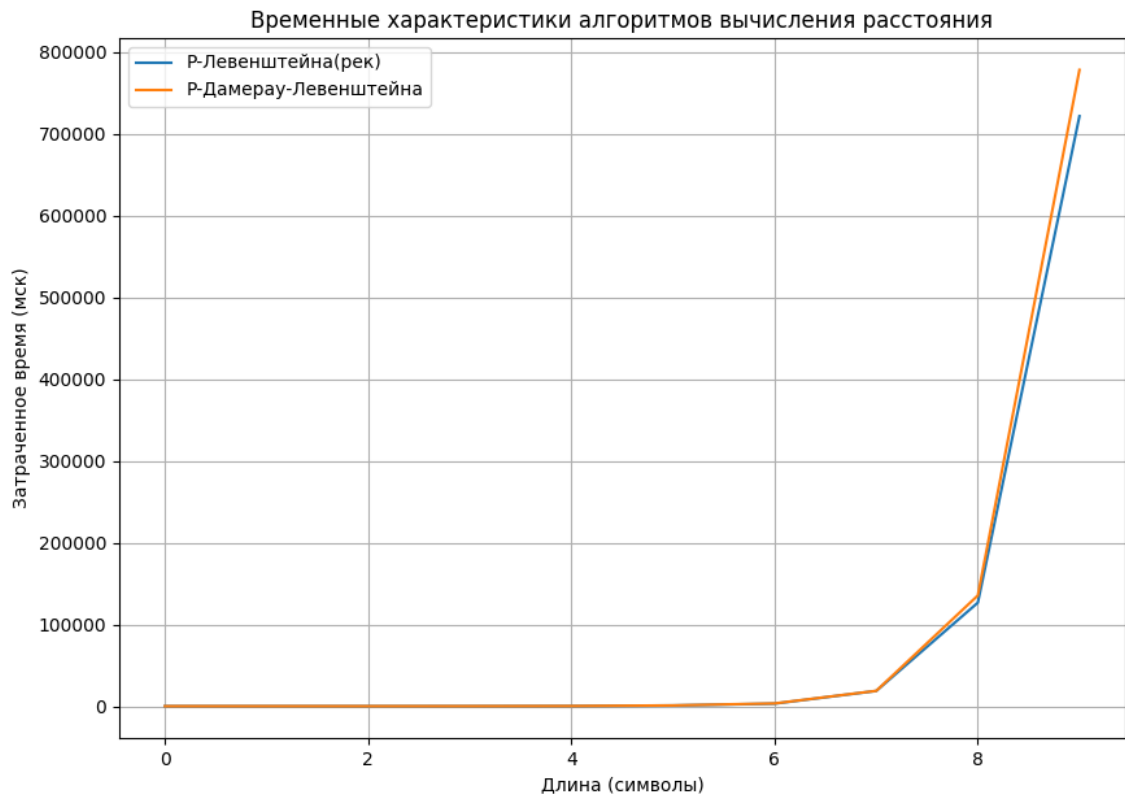


Рисунок 4.3 – Сравнения рекурсивных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

4.4 Использование памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау – Левенштейна не отличаются друг от друга с точки зрения использования памяти, поэтому достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций данных алгоритмов.

Пусть длина строки $S1$ - n , длина строки $S2$ - m , тогда затраты памяти на приведенные выше алгоритмы будут следующими:

- матричный алгоритм поиска расстояния Левенштейна:
 - строки $S1, S2$ - $(m + n) * \text{sizeof}(\text{char})$
 - матрица - $((m + 1) * (n + 1)) * \text{sizeof}(\text{int})$
 - текущая строка матрицы - $(n + 1) * \text{sizeof}(\text{int})$
 - длины строк - $2 * \text{sizeof}(\text{int})$

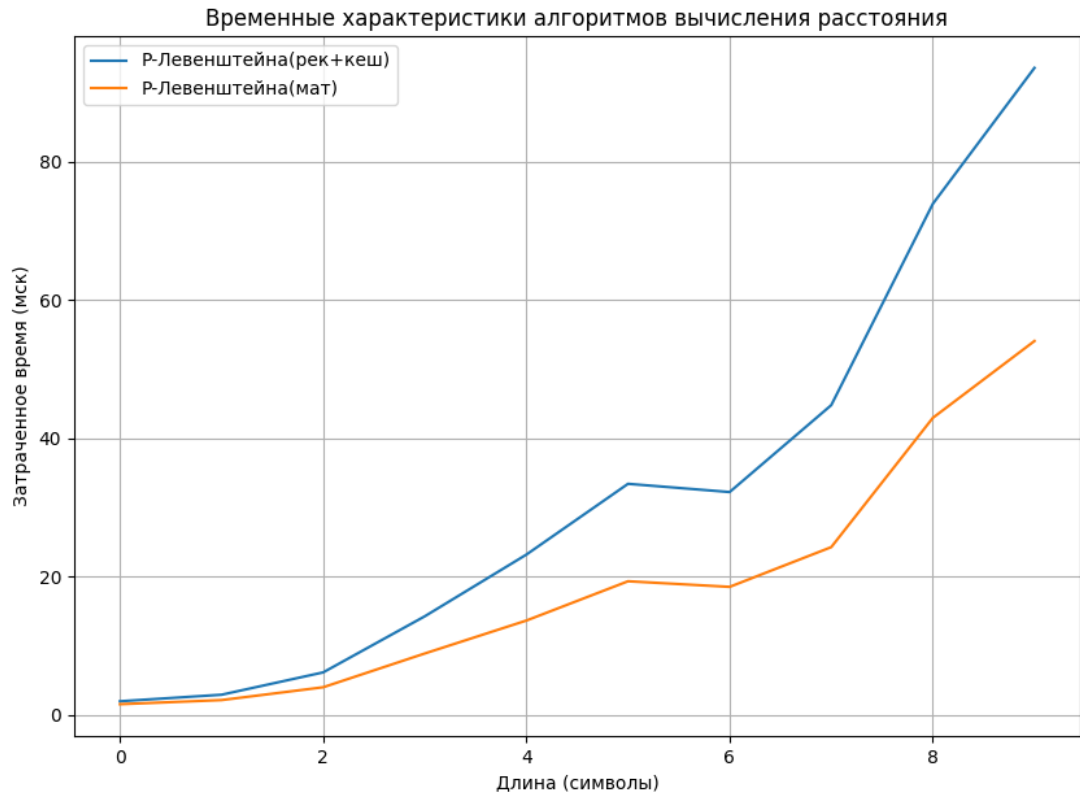


Рисунок 4.4 – Сравнения матричного алгоритма поиска расстояния Левенштейна и рекурсивного алгоритма поиска расстояния Левенштейна с использованием кеша

– вспомогательные переменные - $3 * \text{sizeof}(\text{int})$

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк.

- рекурсивный алгоритм поиска расстояния Левенштейна (для каждого вызова):
 - строки S1, S2 - $(m + n) * \text{sizeof}(\text{char})$
 - длины строк - $2 * \text{sizeof}(\text{int})$
 - вспомогательные переменные - $2 * \text{sizeof}(\text{int})$
 - адрес возврата
- рекурсивный алгоритм поиска расстояния Левенштейна с использованием кеша (для каждого вызова): Для всех вызовов еще память для хранения самой матрицы - $((m + 1) * (n + 1)) * \text{sizeof}(\text{int})$

- строки S1, S2 - $(m + n) * \text{sizeof}(\text{char})$
 - длины строк - $2 * \text{sizeof}(\text{int})$
 - вспомогательные переменные - $2 * \text{sizeof}(\text{int})$
 - ссылка на матрицу - 8 байт
 - адрес возврата
- рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна (для каждого вызова):
 - строки S1, S2 - $(m + n) * \text{sizeof}(\text{char})$
 - длины строк - $2 * \text{sizeof}(\text{int})$
 - вспомогательная переменная - $\text{sizeof}(\text{int})$
 - адрес возврата

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.

Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный и сравним по времени работы с матричными алгоритмами.

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.

Но по расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
- для некоторых реализаций применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- реализованы алгоритмы поиска расстояния Левенштейна с заполнением матрицы, с использованием рекурсии и с помощью рекурсивного заполнения матрицы (рекурсивный с использованием кеша);
- реализованы алгоритмы поиска расстояния Дамерау-Левенштейна с использованием рекурсии;
- проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- проведено экспериментальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций алгоритмов при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на различных длинах строк;
- подготовлен отчет о лабораторной работе.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длинах строк.

В результате исследований можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте строк, но проигрывает по количеству затрачиваемой памяти.