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 (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

ОТЧЕТ
по лабораторной работе №1

Название: Расстояние Левенштейна

Дисциплина: Анализ алгоритмов

Студент: ИУ7-55Б _____ М. Н. Серова
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель: _____ Л. Л. Волкова
(Подпись, дата) (И. О. Фамилия)

Москва, 2021

Содержание

Введение

Цель: сравнить алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Задачи.

1. Изучить алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.
2. Реализовать алгоритмы:
   (a) рекурсивный алгоритм нахождения расстояния Левенштейна;
   (b) матричный алгоритм нахождения расстояния Левенштейна;
   (c) рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы;
   (d) расстояние Дамерау-Левенштейна.
3. Применить методы динамического программирования для реализации алгоритмов.
4. Провести сравнительный анализ алгоритмов по затрачиваемым ресурсам (времени и памяти).

1 Аналитическая часть

Расстояние Левенштейна [1] - минимальное количество редакторских операций, необходимых для преобразования одной строки в другую. Редакторскими операциями являются:

- вставка (I - insert);
- удаление (D - delete);
- замена (R - replace);
- совпадение (M - match).

В расстоянии Дамерау-Левенштейна дополнительно рассматривается операция перестановки соседних символов (X - exchange). Для каждой редакторской операции задается своя цена (штраф). Следовательно, решение задачи нахождения редакционного расстояния сводится к нахождению минимальной суммарной цены всех проведенных для слов операций. Заметим, что для всех редакторских операций, кроме совпадения (M), штраф составляет 1.

Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна применяются нескольких областях:

- в автокоррекции и автолингвистике (для автоматического исправления ошибок в слове, сравнении введенных слов со словарями в поисковых системах);
- в биоинформатике (для сравнения генов, хромосом, белков).

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Пусть даны две строки S_1 и S_2 длиной M и N соответственно. Тогда расстояние Левенштейна можно вычислить по рекуррентной формуле 1.1, где D(i, j) - расстояние между строками $S_1[1..i]$ и $S_2[1..j]$, $S_k[i]$ - i-ый символ

строки S_k.

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

При этом очевидно, что верны следующие утверждения:

- D(S_1, S_2) $\geq$ ||S_1| - |S_2||;
- D(S_1, S_2) $\leq$ max(|S_1|, |S_2|);
- D(S_1, S_2) = 0 $\Leftrightarrow$ $S_1 = S_2$.

Рекурсивный алгоритм реализует формулу 1.1.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 нахождения расстояния Левенштейна может быть малоэффективной при больших i, j, поскольку промежуточные значения D(i, j) вычисляются повторно много раз. Для снижения затрат можно ввести матрицу для хранения соответствующих промежуточных значений - матрицу расстояний. В таком случае алгоритм представляет собой построчное заполнение матрицы A_{mxn}, элемент A[n][m] которой будет содержать итоговое расстояние Левенштейна.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Важный недостаток рекурсивной реализации алгоритма нахождения расстояния Левенштейна заключается в многократном вычислении промежуточных значении D(i, j). Для оптимизации введем матрицу расстояний. Если рекурсивный алгоритм обрабатывает данные первый раз, полученное

значение заносится в соответствующую ячейку матрицы, иначе используется уже вычисленное значение, и алгоритм переходит к следующему шагу.

1.4 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна вычисляется по формуле, аналогичной 1.1, но с добавлением дополнительной операции. Таким образом, получим формулу 1.2.

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

Рекурсивная реализация данной формулы при больших значениях i, j будет работать долго по причинам, аналогичным рекурсивному алгоритму поиска расстояния Левенштейна, в связи с чем удобно ввести матрицу, аккумулирующую вычисленные значения.

1.5 Вывод

Формулы Левенштейна и Дамерау-Левенштейна для вычисления расстояния между строками задаются рекуррентно, что позволяет реализовать алгоритмы как рекурсивно, так и итерационно. Для оптимизации рекурсивных алгоритмов вводится вспомогательная матрица расстояний, хранящая промежуточные вычисления, она же применяется в итеративных алгоритмах.

2 Конструкторская часть

Данный раздел содержит схемы алгоритмов, реализуемых в работе: Левенштейна и Дамерау-Левенштейна.

2.1 Схемы алгоритмов нахождения расстояния Левенштейна

На рисунке 2.1 представлена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

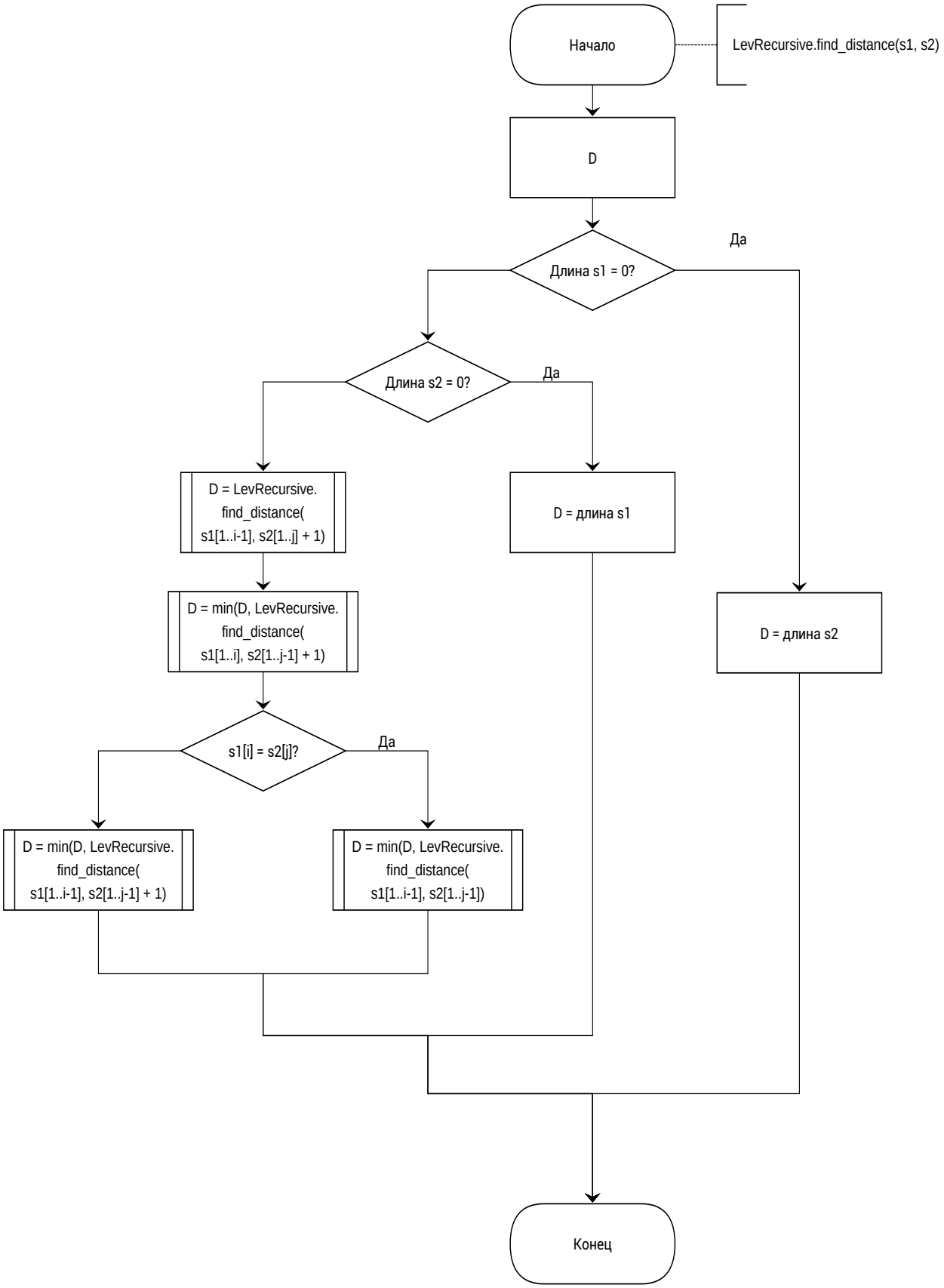

Рис. 2.1: Схема рекурсивного алгоритма Левенштейна

На рисунке 2.2 представлена схема итеративного алгоритма нахождения редакционного расстояния между словами.

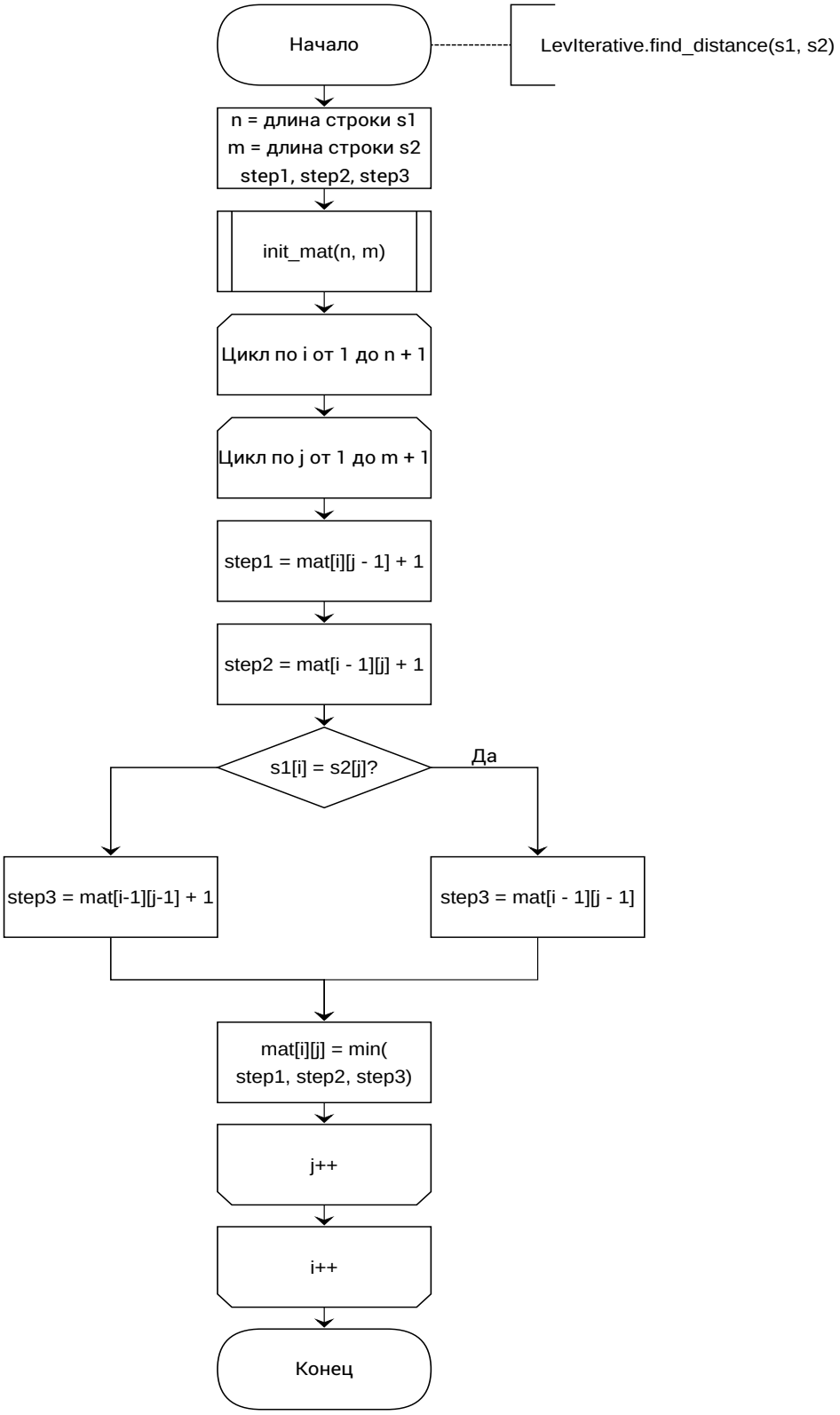

Рис. 2.2: Схема матричного алгоритма Левенштейна

Схема функции заполнения матрицы, вызываемой в итеративном алгоритме нахождения расстояния Левенштейна, представлена на рисунке 2.3.

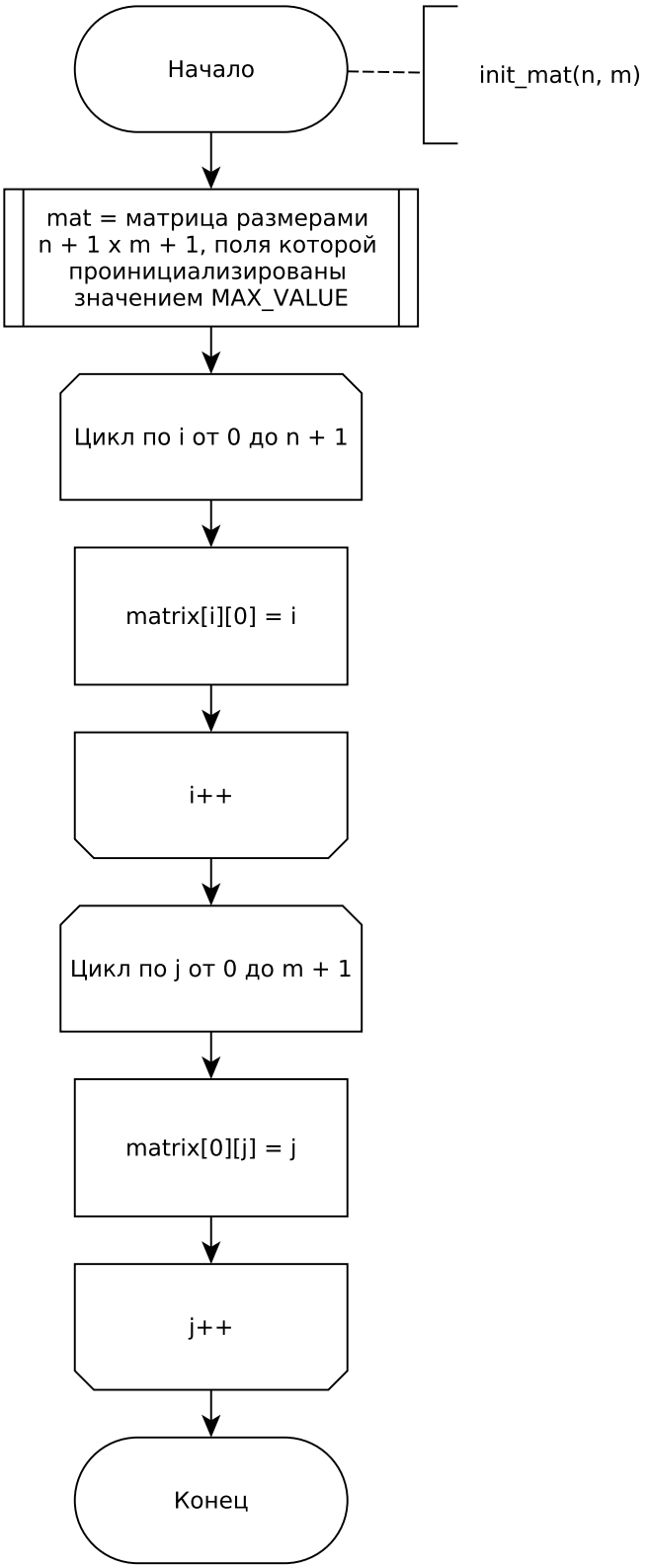

Рис. 2.3: Схема функции, инициализирующей матрицу расстояний

На рисунке 2.4 представлена схема рекурсивного алгоритма нахождения

расстояния Левенштейна с заполнением матрицы.

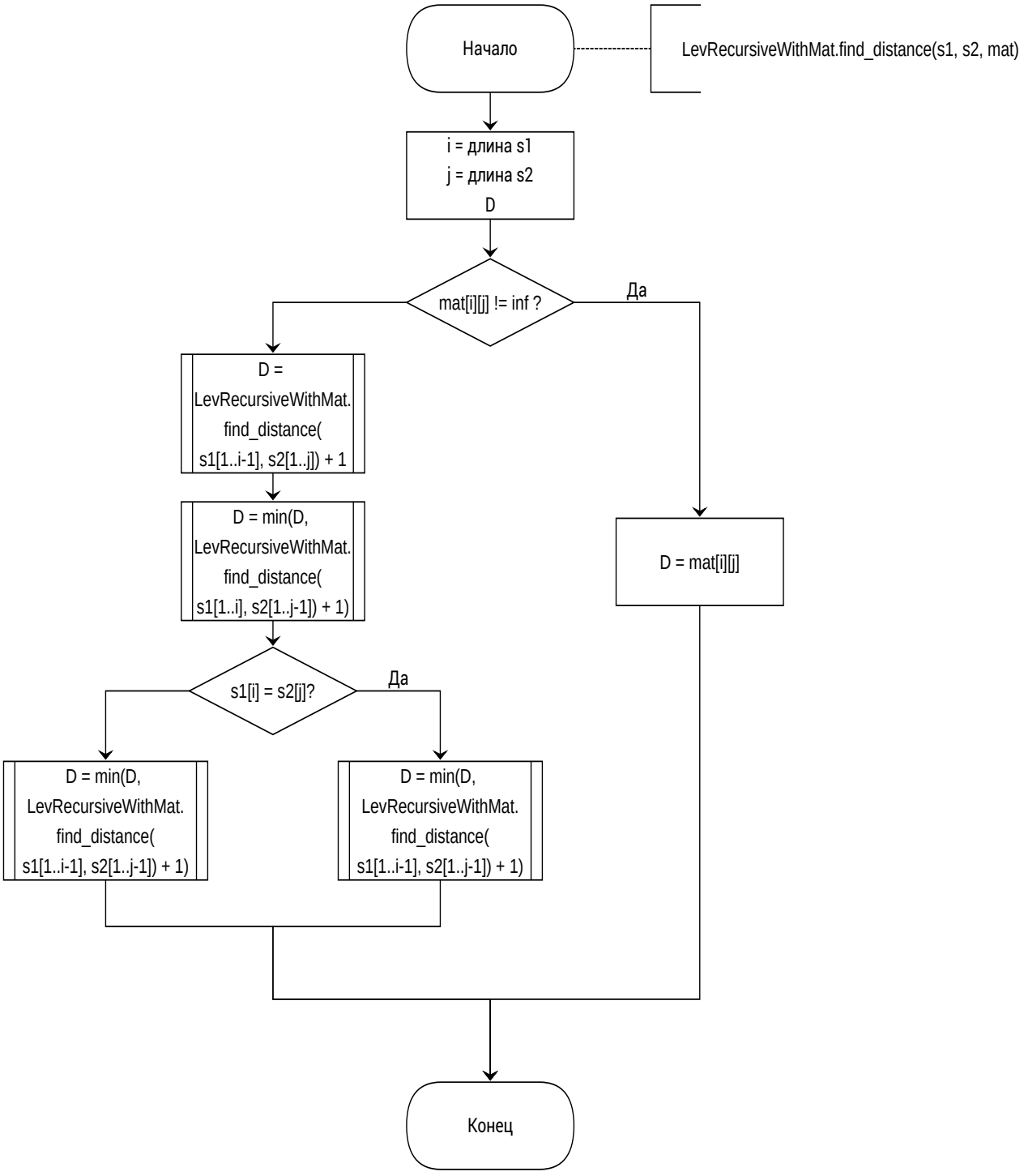

Рис. 2.4: Схема рекурсивного алгоритма Левенштейна с кэшем

2.2 Схема алгоритма нахождения расстояния Дамерау-Левенштейна

Схема алгоритма поиска расстояния Дамерау-Левенштейна представлена на рисунке 2.5.

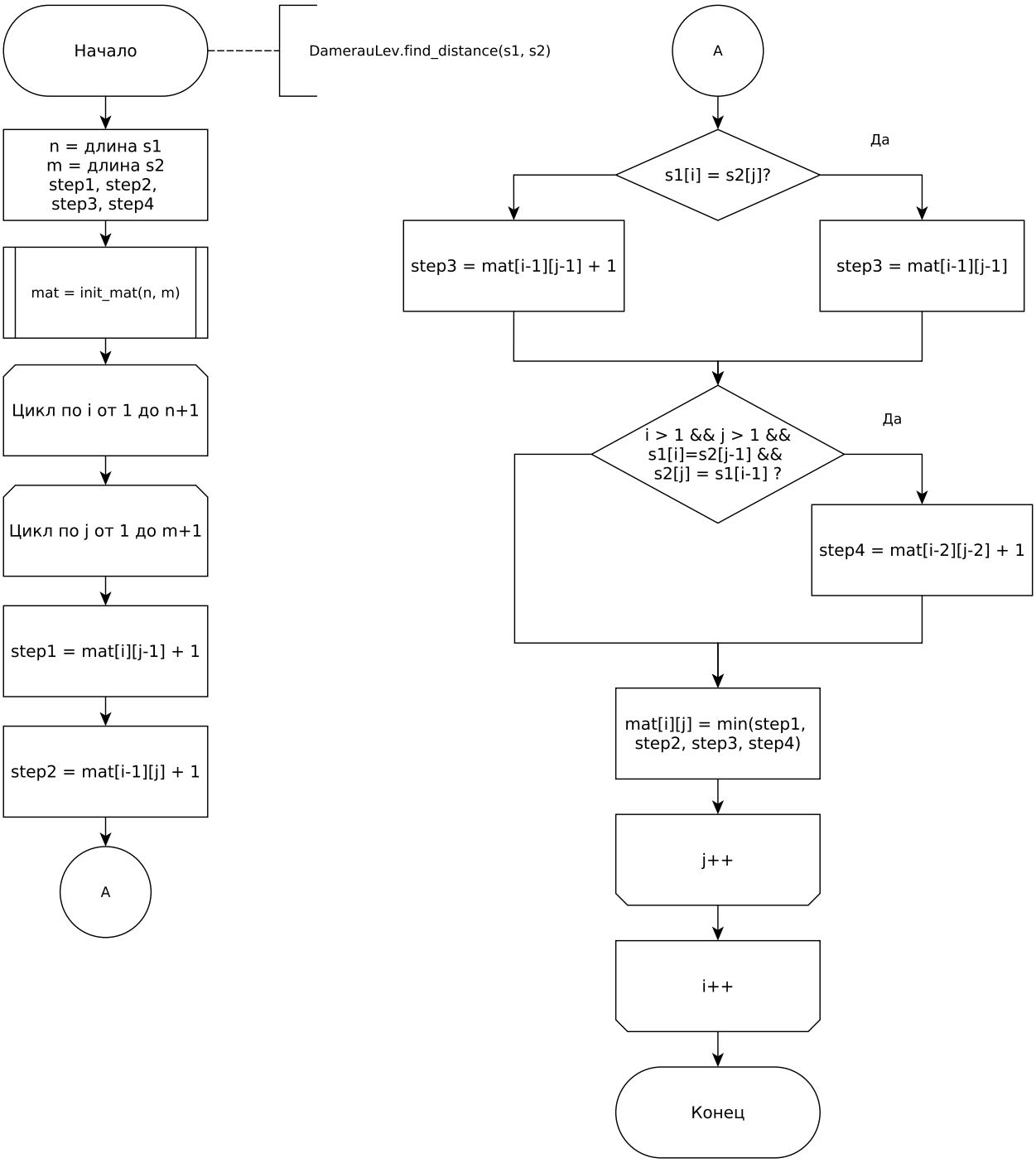

Рис. 2.5: Схема итерационного алгоритма Дамерау-Левенштейна

2.3 Вывод

В данном разделе на основе приведенных в аналитическом разделе теоретических данных были составлены схемы алгоритмов для реализации в технологической части.

3 Технологическая часть

Данный раздел содержит обоснование выбора языка и среды разработки, реализацию алгоритмов.

3.1 Требования к программному обеспечению

Требования, выдвигаемые к разрабатываемому ПО:

- входные данные - две строки S_1 и S_2 на английском или русском языках в любом регистре;
- выходные данные - искомое расстояние для всех алгоритмов.

3.2 Средства реализации

Для реализации программы был выбран язык программирования Python [2]. Такой выбор обусловлен следующими причинами:

- данный язык программирования был освоен на практических занятиях;
- имеет большое количество расширений и библиотек, что облегчает работу с некоторыми типами данных и математическими формулами;
- обладает информативной документацией.

3.3 Реализация алгоритмов

В листингах 3.2 - 3.5 представлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Рекурсивный алгоритм вычисления расстояния Левенштейна (часть 1)

```
1 class LevRecursive(StringDistanceFinder):
2   def __init__(self):
3     super().__init__()
4
5   def find_distance(self, s1, s2):
6     return self.__find_distance(s1, s2)
```

Листинг 3.2: Рекурсивный алгоритм вычисления расстояния Левенштейна (часть 2)

```
  def __find_distance(self, s1, s2):
    if len(s1) == 0:
      return len(s2)
    elif len(s2) == 0:
      return len(s1)
    D = self.__find_distance(s1[:-1], s2) + 1
    D = min(D, self.__find_distance(s1, s2[:-1]) + 1)
    m = 0 if s1[-1] == s2[-1] else 1

    D = min(D, self.__find_distance(s1[:-1], s2[:-1]) + m)

    return D
```

Листинг 3.3: Матричный алгоритм вычисления расстояния Левенштейна

```
class LevIterative(StringDistanceFinder):
  def __init__(self):
    super().__init__()

  def find_distance(self, s1, s2):
    return self.__find_distance(s1, s2)

  def __find_distance(self, s1, s2):
    n = len(s1)
    m = len(s2)
    mat = self.init_matrix(n, m)
    for i in range(1, n + 1):
      for j in range(1, m + 1):
        step1 = mat[i][j - 1] + 1
        step2 = mat[i - 1][j] + 1
        elem = 0 if s1[i-1] == s2[j-1] else 1
        step3 = mat[i - 1][j - 1] + elem
        mat[i][j] = min(step1, step2, step3)
    return mat[-1][-1]
```

Листинг 3.4: Рекурсивный алгоритм вычисления расстония с заполнением матрицы

```
class LevRecursiveWithMat(StringDistanceFinder):
  def __init__(self):
    super().__init__()

  def find_distance(self, s1, s2):
    mat = self.init_matrix(len(s1), len(s2))
    return self.__find_distance(s1, s2, mat)

  def __find_distance(self, s1, s2, mat):
    i = len(s1)
    j = len(s2)
    if mat[i][j] != np.inf:
      return mat[i][j]
    if i == 0:
      return j
    if j == 0:
      return i
    D = self.__find_distance(s1[:-1], s2, mat) + 1
    D = min(D, self.__find_distance(s1, s2[:-1], mat) + 1)
    m = 0 if s1[-1] == s2[-1] else 1
    D = min(D, self.__find_distance(s1[:-1], s2[:-1], mat) + m)
    mat[i][j] = D
    return D
```

Листинг 3.5: Алгоритм Дамерау-Левенштейна

```
class DamerauLev(StringDistanceFinder):
  def __init__(self):
    super().__init__()

  def find_distance(self, s1, s2):
    return self.__find_distance(s1, s2)

  def __find_distance(self, s1, s2):
    n = len(s1)
    m = len(s2)
    mat = self.init_matrix(n, m)
    for i in range(1, n + 1):
      for j in range(1, m + 1):
        step1 = mat[i][j - 1] + 1
        step2 = mat[i - 1][j] + 1
        elem = 0 if s1[i - 1] == s2[j - 1] else 1
        step3 = mat[i - 1][j - 1] + elem
        step4 = np.inf
        if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s2[j -
           1] == s1[i - 2]:
          step4 = mat[i - 2][j - 2] + 1
        mat[i][j] = min(step1, step2, step3, step4)
    return mat[-1][-1]
```

3.4 Тестирование

В таблице 3.1 представлены использованные для тестирования методом "черного ящика" данные, были рассмотрены все возможные тестовые случаи. Все тесты пройдены успешно. Здесь и далее "Рек.Лев." - рекурсивный алгоритм Левенштейна, "Ит.Лев." - матричный алгоритм Левенштейна, "Рек.Лев.(мат.)" - рекурсивный алгоритм Левенштейна с кэшем, "Д.-Лев." - итерационный алгоритм Дамерау-Левенштейна.

Таблица 3.1: Проведенные тесты

		Ожидаемый результат			
Строка 1	Строка 2	Рек.Лев.	Ит.Лев.	Рек.Лев.(мат.)	Д.-Лев.
окно	компот	4	4	4	4
monarchy	democracy	6	6	6	5
	text	4	4	4	4
ура		3	3	3	3
		0	0	0	0
яичница	ячиинца	3	3	3	2
adventure	time	7	7	7	7

3.5 Выводы

В данном разделе были реализованы и протестированы различные варианты алгоритма нахождения расстояния Левенштейна и итерационный алгоритм Дамерау-Левенштейна.

4 Экспериментальная часть

В данном разделе сравниваются реализованные алгоритмы, дается сравнительная оценка затрат на память и время.

4.1 Пример работы программы

Пример работы программы представлен на рисунке 4.1.

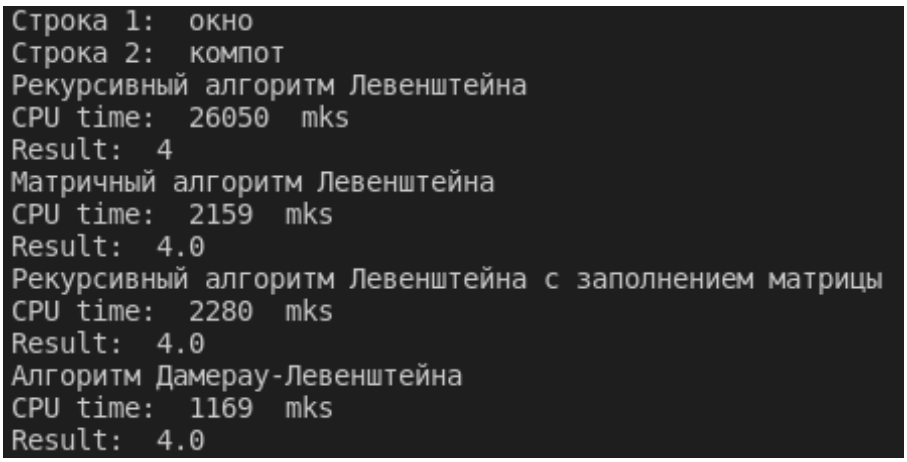

Рис. 4.1: Демонстрация работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Ubuntu 20.01 Linux x86_64 [3];
- оперативная память: 8 Гб;
- процессор: AMD Ryzen5 3500U [4].

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов замерялось на автоматически генерируемых строках необходимой длины с использованием функции getrusage библиотеки resources [5]. Усредненные результаты замеров процессорного времени приведены в таблице. Прочерк означает, что время ожидания значения результата превысило минуту, далее замеры не проводились.

Таблица 4.1: Время обработки строк разной длины в микросекундах

Длина	Рек.Лев.	Ит.Лев.	Рек.Лев.(мат.)	Д.-Лев.
5	56193	1127	3171	1282
7	1238515	1961	5723	2227
10	-	2260	6884	2605
20	-	8449	27139	9911
50	-	51052	167683	61132
100	-	203366	668480	244068

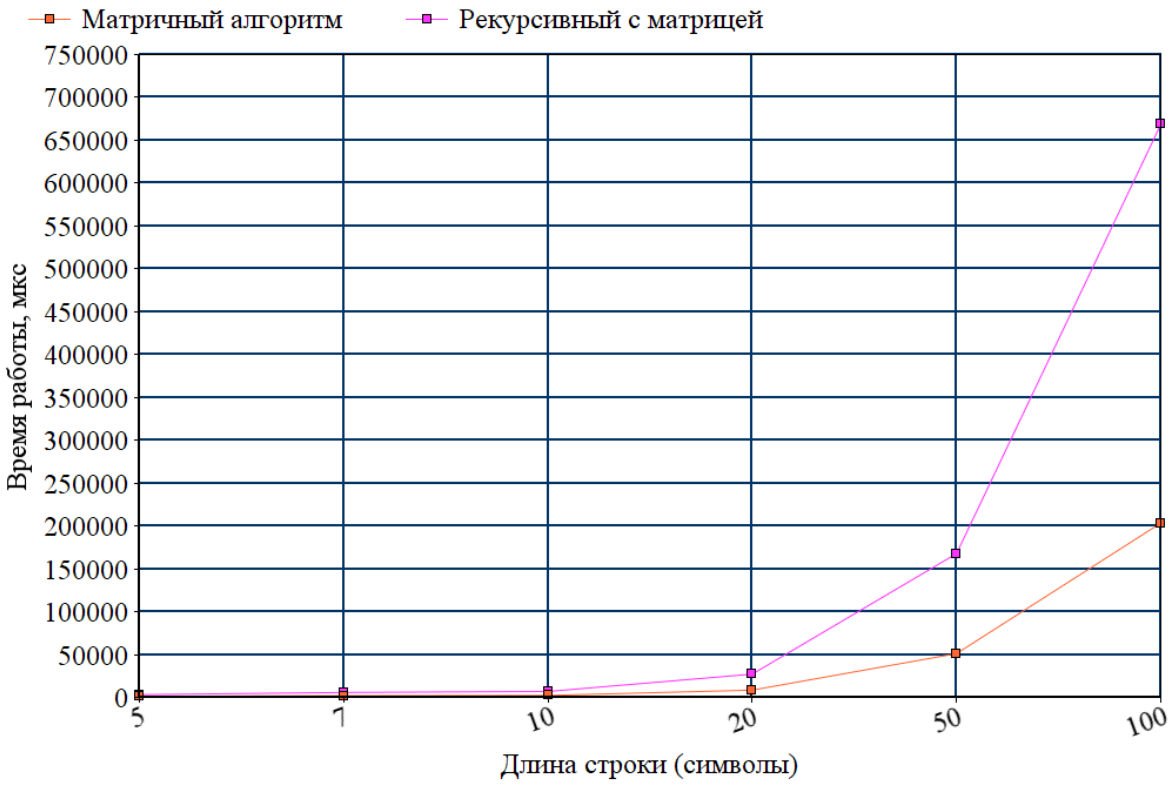

Рис. 4.2: Сравнение времени работы алгоритмов Левенштейна

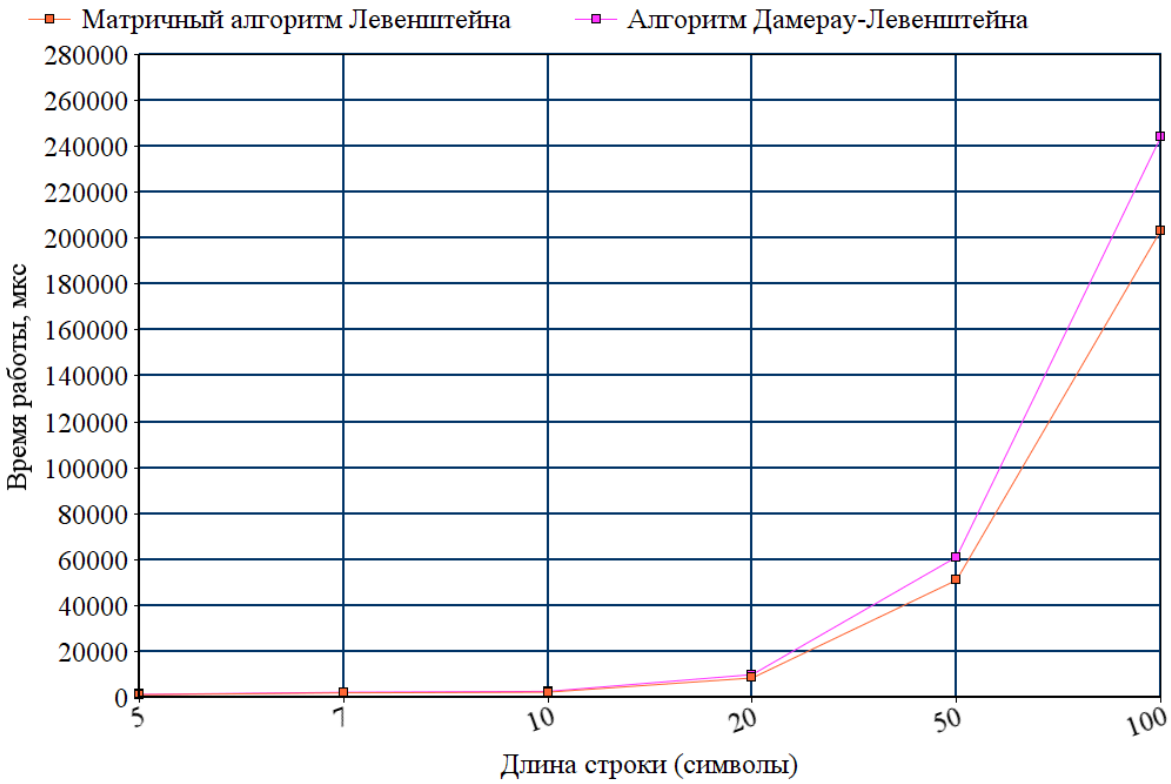

Рис. 4.3: Сравнение времени работы алгоритмов Левенштейна

4.4 Использование памяти

С точки зрения использования памяти алгоритмы Левенштейна и Дамерау-Левенштейна отличаются незначительно. Поэтому для оценки использования памяти достаточно рассмотреть разницу рекурсивной и матричной реализаций алгоритмов.

Нагрузка функции на память состоит из следующих частей:

- аргументы - две строки - $2 * sizeof(string)$;
- локальные переменные -целочисленная переменная - sizeof(int);
- переменная с ответом - целочисленная переменная - sizeof(int);
- адрес возврата - addr.

Важно учесть, что максимальная глубина стека будет соответствовать сумме длин входных строк: sizeof(S_1) + sizeof(S_2).

Таким образом, общее количество памяти, затрачиваемое рекурсивным алгоритмом, можно вычислить по формуле 4.1.

$$M_{recursive} = (sizeof(S_1) + sizeof(S_2)) \cdot (2 \cdot sizeof(string) + \\ +2 \cdot sizeof(int) + addr) \tag{4.1}$$

Для вычисления объема используемой памяти при матричной реализации алгоритма используется формула 4.2.

$$M_{matrix} = (len(S_1) + 1) \cdot (len(S_2) + 1) \cdot \\ \cdot sizeof(int) + 2 \cdot sizeof(string) + 9 \cdot sizeof(int) + addr \tag{4.2}$$

Для рекурсивного алгоритма с заполнением матрицы необходимо учесть дополнительные затраты на хранение матрицы и передачу ее в качестве аргумента. Таким образом получим формулу 4.3 для вычисления затрат по памяти на выполнение рекурсивного алгоритма с матрицей.

$$m_{mat} = (len(S_1) + 1) \cdot (len(S_2) + 1) \cdot sizeof(int) \\ M_{recurs-with-matrix} = (sizeof(S_1) + sizeof(S_2)) \cdot \\ \cdot (2 \cdot sizeof(string) + 4 \cdot sizeof(int) + addr + m_{mat}) + m_{mat} \tag{4.3}$$

4.5 Выводы

Рекурсивная реализация алгоритма нахождения расстояния Левенштейна работает дольше других более чем в 40 раз за счет большого количества повторяющихся вычислений, причем время работы возрастает в 2 раза при увеличении длины входных строк на 2. Итерационные реализации алгоритмов Левенштейна и Дамерау-Левенштейна сопоставимы по времени выполнения. Рекурсивный алгоритм Левенштейна с заполнением матрицы работает быстрее обычного рекурсивного, но в три раза медленнее обеих матричных реализаций.

Однако итеративные алгоритмы менее эффективны по памяти за счет необходимости хранения матрицы, размерность которой превышает на 1 длину строк. Таким образом, если для рекурсии количество памяти возрастает пропорционально сумме длин строк, то для матричного алгоритма - пропорционально их произведению.

Заключение

В процессе выполнения лабораторной работы были изучены алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна. Получен практический навык реализации этих алгоритмов в итерационной версии, алгоритма Левенштейна в рекурсивной и рекурсивной с заполнением матрицы версиях.

Были исследованы процессорное время выполнения и затраты на память выше обозначенных алгоритмов. В результате было выявлено, что дольше всего выполняется рекурсивный алгоритм Левенштейна (более чем в 40 раз дольше других), быстрее всего - итерационный алгоритм Левенштейна, при этом время выполнения итерационного алгоритма Дамерау-Левенштейна сопоставимо с аналогичной версией алгоритма Левенштейна.

На основе кода были получены формулы для вычисления использованной памяти для каждой версии алгоритмов. Согласно полученным выражениям, матричные алгоритмы значительно проигрывают по памяти рекурсивным за счет необходимости хранения вспомогательной матрицы.

Список литературы

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

[2] About Python [Электронный ресурс]. Режим доступа: `https://www.python.org/about/`. Дата обращения: 23.09.2021.

[3] Ubuntu по-русски [Электронный ресурс]. Режим доступа: `https://ubuntu.ru/`. Дата обращения: 23.09.2021.

[4] Мобильный процессор AMD Ryzen™ 5 3500U с графикой Radeon™ Vega 8 [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-5-3500u`. Дата обращения: 23.09.2021.

[5] resource — Resource usage information - Python 3.11.0a0 documentation. Режим доступа: `https://docs.python.org/dev/library/resource.html?highlight=resources`. Дата обращения: 23.09.2021.