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: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-51Б _____ М.С.Матвиенко
(Группа) (Подпись, дата) (И.О. Фамилия)

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

Москва, 2020

Содержание

Введение

Расстояние Левенштейна (редакционное расстояние) - это минимальное кол-во редакторских операций, которое необходимо для превращения одной строки в другую.

Применение.
1) Поисковики (Google), автоисправления
2) Биоинформатика

Задания для данной ЛР:
1) изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2) получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
3) сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
4) эксперементальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирущихся длинах строк;
5) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

Задача по нахождению расстояния Левенштайна заключается в поиске минимального количества операций:
Вставка (I - Insert)
Удаление (D - Delete)
Замена (R - Replace)
Совпадение (M - Match)
Для превращения одной строки в другую.
В алгоритме Далмерау-Левенштайна добавляется ещё одна операция:
Транспозиция(T)
Все операции, кроме "Совпадения"имею штраф 1. Операция "Совпадения"имеет штраф 0.

1.1 Описание алгоритмов

1.1.1 Рекурсивный алгоритм Левенштейна

Введём понятие D(s1, s2) = минимальному количеству редакторских операций, с помощью которых строка s1 преобразуется в строку s2. Тогда рекурсивный алгоритм Левенштейна можно записать следующим образом:

$$D(i,j) = \begin{cases} 0, i = 0, j = 0 \\ i, j = 0, i > 0 \\ j, i = 0, j > 0 \\ min(D(S_1[1,...,i], S_2[1,...,j-1]) + 1, \\ D(S_1[1,...,i-1], S_2[1,...,j]) + 1, \\ D(S_1[1,...,i-1], S_2[1,...,j-1]) + \\ \begin{bmatrix} 0, if S_1[i] = S_2[j], \\ 1, else \end{bmatrix} \end{cases}$$

1.1.2 Матричный алгоритм Левенштейна

Вводится матрица, размерностью $[Len(S_1) + 1 \mathbf{X} Len(S_2) + 1]$
Первая строки и столбец матрицы заполняются от 0 до Len(S) (первые 3 пункта системы из предыдущего пункта).

$$A = \begin{pmatrix} & \emptyset & С & Т & О & Л & Б \\ \emptyset & 0 & 1 & 2 & 3 & 4 & 5 \\ Т & 1 & & & & & \\ Е & 2 & & & & & \\ Л & 3 & & & & & \\ О & 4 & & & & & \end{pmatrix}$$

Далее для нахождения ответа применяется последняя формула из системы, описанной в предыдущем пункте.

$$A = \begin{pmatrix} & \emptyset & С & Т & О & Л & Б \\ \emptyset & 0 & 1 & 2 & 3 & 4 & 5 \\ Т & 1 & 1 & 1 & 2 & 3 & 4 \\ Е & 2 & 2 & 2 & 2 & 3 & 4 \\ Л & 3 & 3 & 3 & 3 & 2 & 3 \\ О & 4 & 4 & 4 & 3 & 3 & \mathbf{3} \end{pmatrix}$$

Ответ в правом нижнем углу.

Чтобы определить, какая именно цепочка преобразований привела к ответу представим матрицу как карту высот: нужно спуститься на санках из клетки с ответом в левый верхний угол. В нашем случае:
I: ТЕЛО → СТЕЛО
M: Т = Т
R: Е → О
M: Л = Л
R: О → Б

1.1.3 Рекурсивно-матричный алгоритм Левенштейна

Аналогичен алогритму из предыдущего пункта с той лишь разницей, что матрица начинает заполнение "с конца". Вычисляем значение ячейки матрицы только в том случае, если значения там ещё нет (аналогично ∞ в алгоритме Дейкстры). Ответ всё так же в правом нижнем углу.

2 Конструкторская часть

Требования к вводу:
1) на вход подаются две строки;
2) одна и та же буква в разном регистре считается как разный символ.
Требования к программе::
1) Две пустые строки являются корректным вводом, который программа должна обработать.

2.1 Разработка алгоритмов

В данном разделе представлены схемы реализуемых алгоритмов.

2.1.1 Схема рекурсивной реализации алгоритма Левенштейна

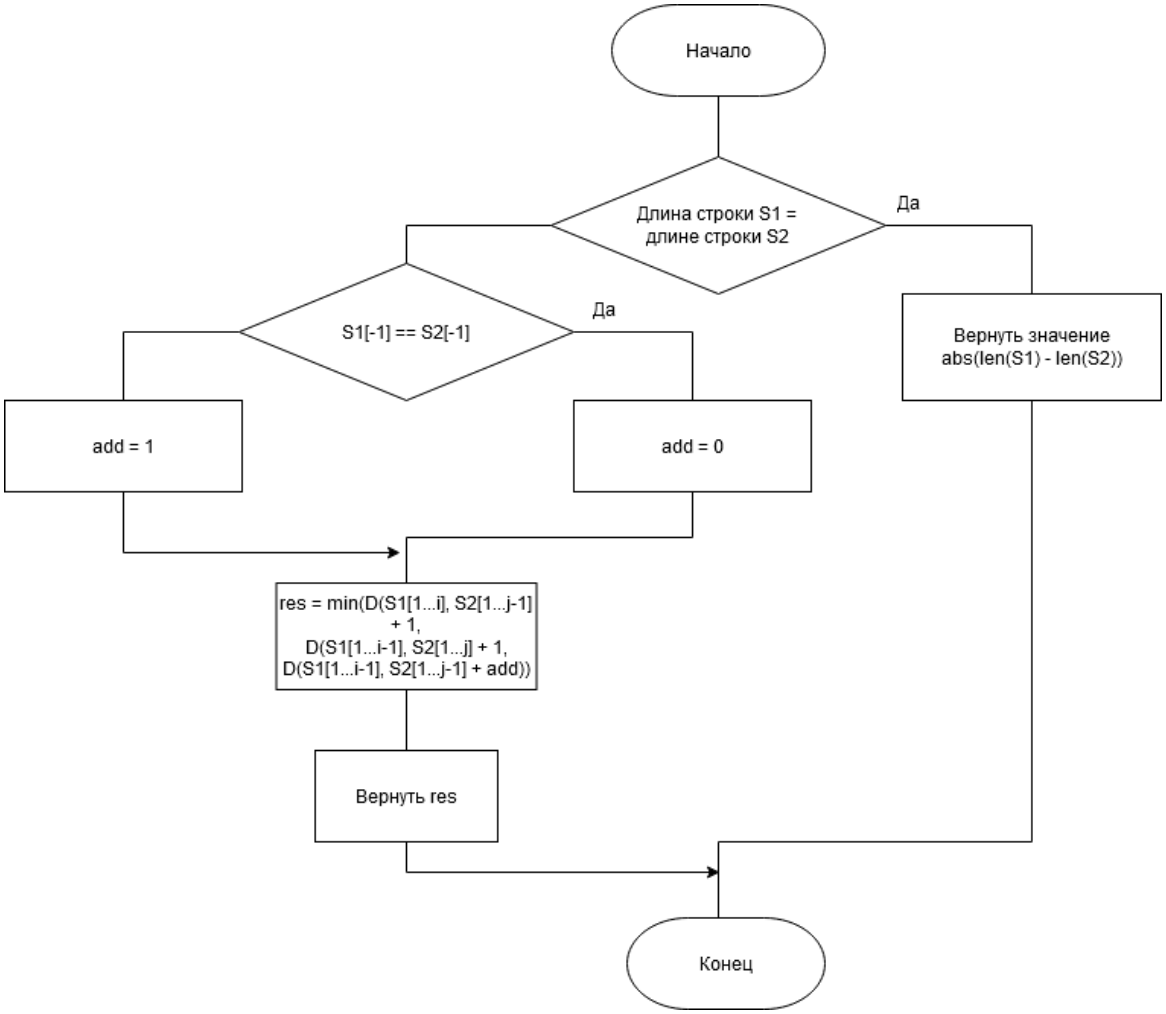


Рис. 1: Схема рекурсивного алгоритма Левенштейна

2.1.2 Схема матричной реализации алгоритма Левенштейна

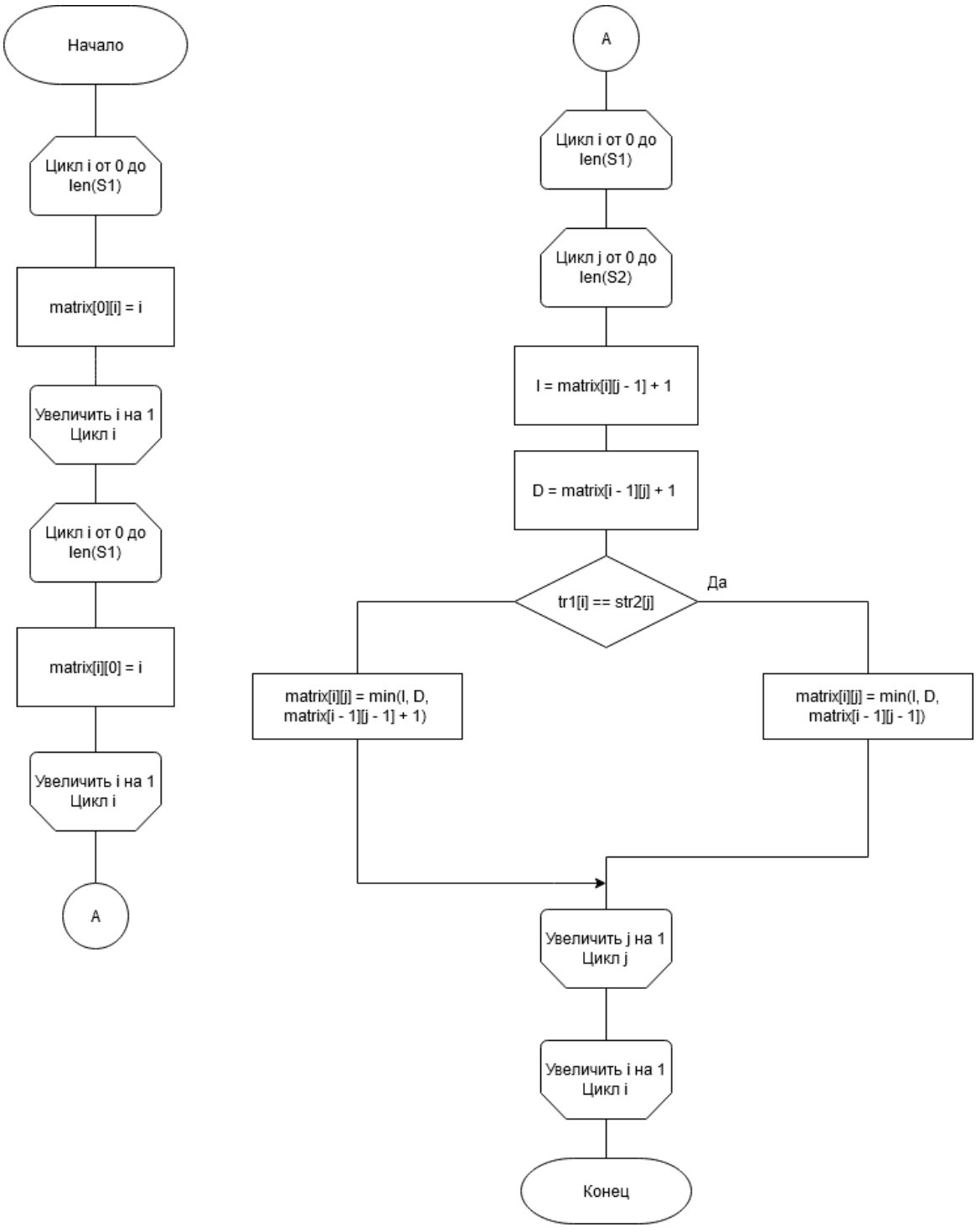


Рис. 2: Схема матричной реализации алгоритма Левенштейна

2.1.3 Схема рекурсивного матричного алгоритма Левенштейна

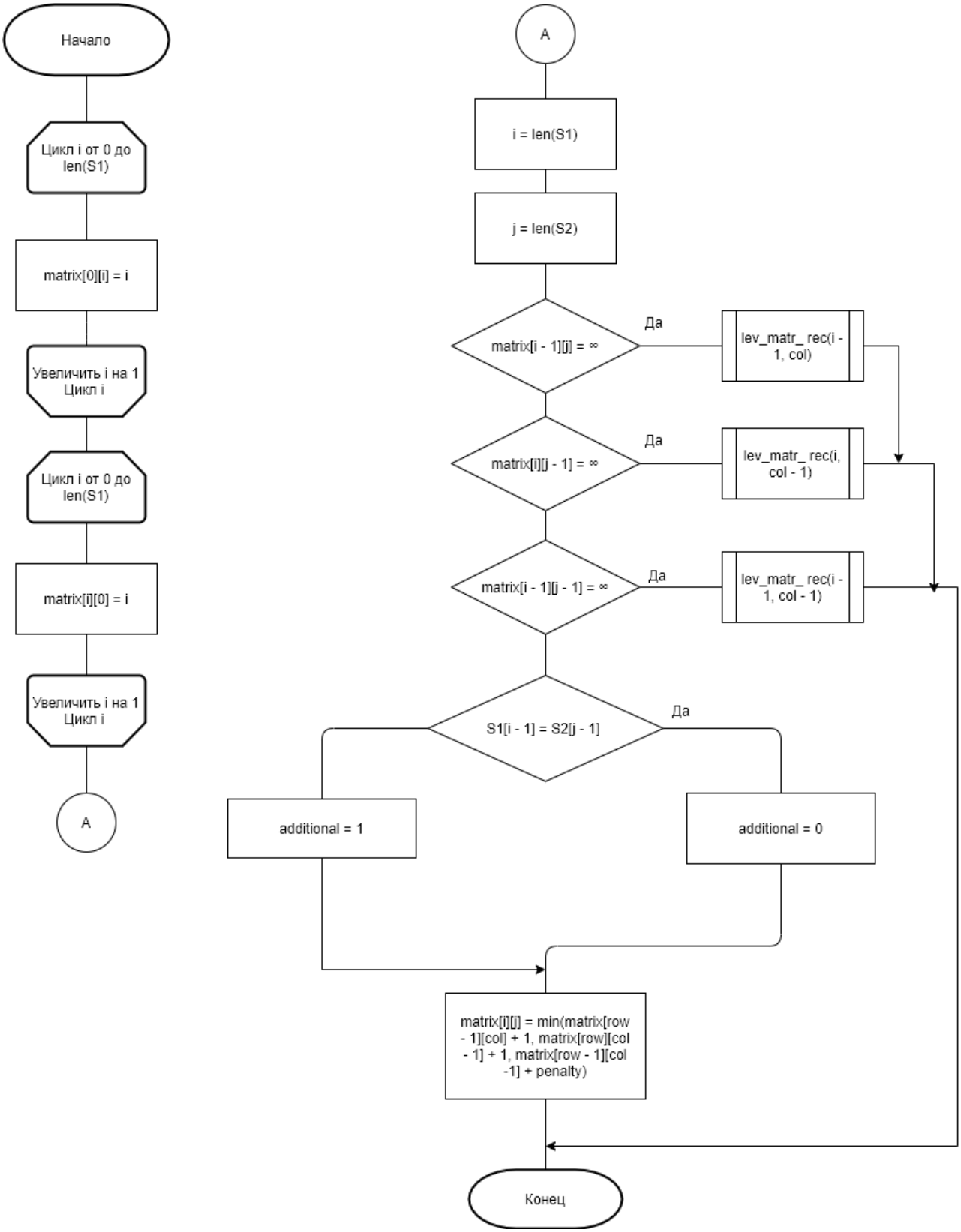


Рис. 3: Схема рекурсивной матричной реализации алгоритма Левенштейна

2.1.4 Схема алгоритма Дамерау-Левенштейна

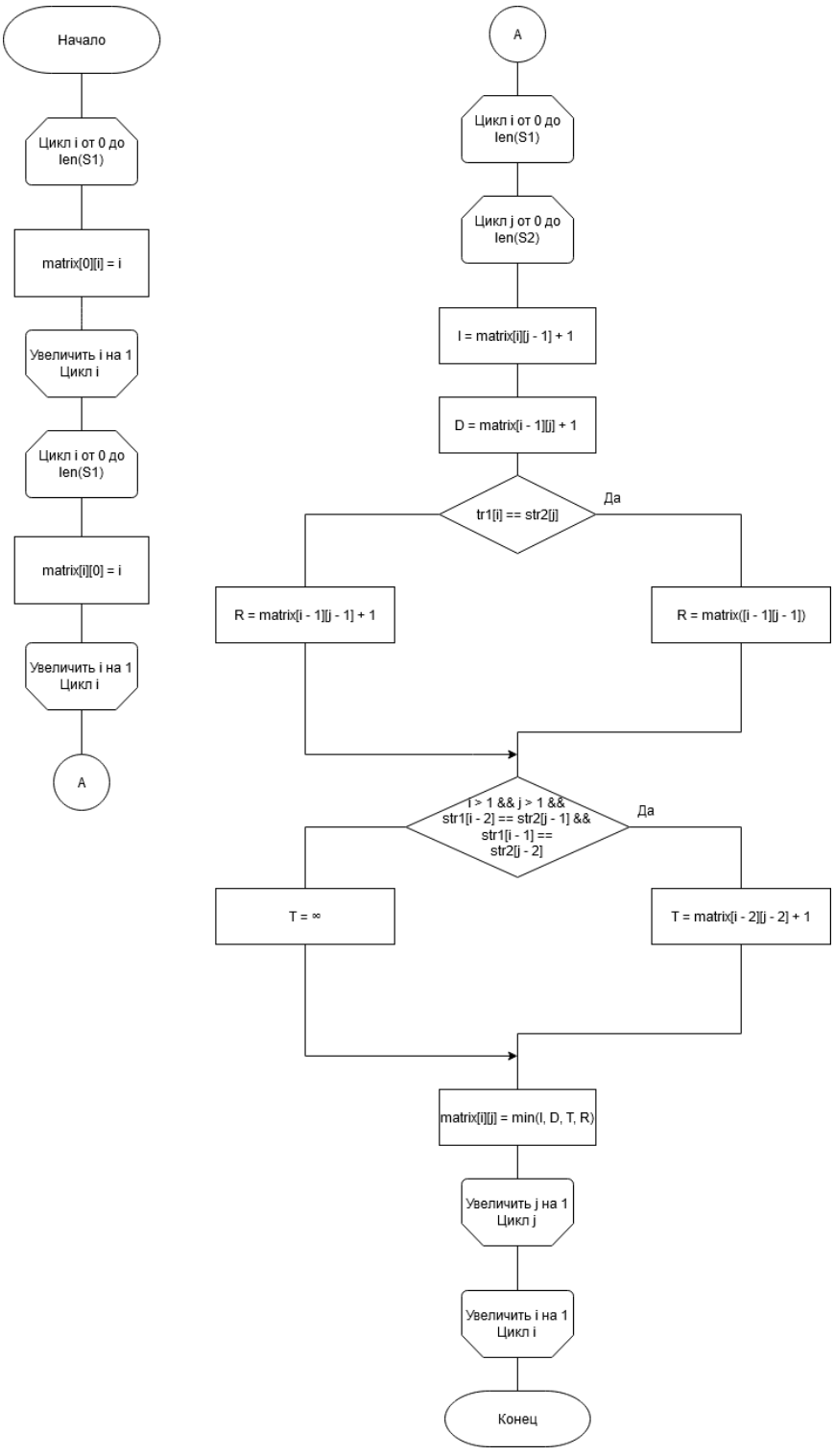


Рис. 4: Схема алгоритма Дамерау-Левенштейна

3 Технологическая часть

В данном разделе будет описана технологическая часть лабораторной работы: требования к ПО, листинг кода, сравнительный анализ всех алгоритмов.

3.1 Требования к программному обеспечению

Входные данные: два слова: str1, str2
Выходные данные: редакционное расстояние данных слов, а также матрица решения для матричных реализаций
Среда выполнения: Windows 10 x64

3.2 Средства реализации

Для выполнения данной лабораторной работы использовался ЯП Python 3.9.0

3.3 Листинг кода

В данном разделе будет представлен листинг кода разработанных алгоритмов.

3.3.1 Рекурсивный алгоритм Левенштейна

Листинг 1: Рекурсивный алгоритм Левенштейна

```
1  def lev_rec(source, target):
2      if len(source) == 0 or len(target) == 0:
3          return abs(len(source) - len(target))
4
5      if (source[-1] == target[-1]):
6          additional = 0
7      else:
8          additional = 1
9
10     return min(lev_rec(source, target[:-1]) + 1,
11                lev_rec(source[:-1], target) + 1,
12                lev_rec(source[:-1], target[:-1]) + additional)
```

3.3.2 Матричный алгоритм Левенштейна

Листинг 2: Матричный алгоритм Левенштейна

```
def lev_matrix(source, target)
    data = [[i + j for j in range(len(target) + 1)]
                for i in range(len(source) + 1)]

    for i in range(1, len(source) + 1)
        for j in range(1, len(target) + 1)
            if (source[i - 1] == target[j - 1])
                additional = 0
            else
                additional = 1

            data[i][j] = min(data[i - 1][j] + 1,
                             data[i][j - 1] + 1,
                             data[i - 1][j - 1] + additional)

    return data[-1][-1]
```

3.3.3 Рекурсивный матричный алгоритм Левенштейна

Листинг 3: Рекурсивный матричный алгоритм Левенштейна

```
def lev_matrix_rec(source, target):

    data = [[i + j for j in range(len(target) + 1)]
            for i in range(len(source) + 1)]

    for i in range(0, len(source) + 1):
        for j in range(0, len(target) + 1):
            data[i][j] = inf

    row = len(source)
    col = len(target)

    if row == 0 or col == 0:
       return abs(row - col)

    if data[row - 1][col] == inf:
        data[row - 1][col] = lev_matrix_rec(source[:-1], target)
    if data[row][col - 1] == inf:
        data[row][col - 1] = lev_matrix_rec(source, target[:-1])
    if data[row - 1][col - 1] == inf:
        data[row - 1][col - 1] = lev_matrix_rec(source[:-1],
                                 target[:-1])

    if source[row - 1] == target[col - 1]:
        additional = 0
    else:
        additional = 1
```

```
28
29      data[row][col] = min(data[row - 1][col] + 1,
30              data[row][col - 1] + 1,
31              data[row - 1][col - 1] + additional)
32
33      return data[-1][-1]
```

3.3.4 Алгоритм Дамерау-Левенштейна

Листинг 4: Алгоритм Дамерау-Левенштейна

```
def damer_lev(source, target):
    data = [[i + j for j in range(len(target) + 1)]
            for i in range(len(source) + 1)]

    for i in range(1, len(source) + 1):
        for j in range(1, len(target) + 1):
            if source[i - 1] == target[j - 1]:
                additional = 0
            else:
                additional = 1
            data[i][j] = min(data[i - 1][j] + 1,
                data[i][j - 1] + 1,
                data[i - 1][j - 1] + additional)

            if (i > 1 and j > 1 and
                source[i - 1] == target[i - 2] and
                source[i - 2] == target[i - 1]):
                data[i][j] = min(data[i][j], data[i - 2][j - 2] + 1)

    return data[-1][-1]
```

3.4 Сравнительный анализ матричной и рекурсивной реализаций

Рекурсивная версия алгоритма работает существенно медленне матричной реализации ввиду многократного вызова функции. На каждый вызов необходимо производить соответствующие операции со стеком. Для уменьшения затрат по времени (на копирование) и на дополнитульную память строки можно **передавать по ссылке** Тогда на каждую строку, независимо от её размера будет выделяться фиксированное число байт (количество зависит от разрядности системы). Однако даже такие ухищрения не способны избавить от главного недостатка - повторного вычисления тех значений, которые были посчитаны на более ранних этапах рекурсии. В матричных реализация будет затрачена дополнительная память на хранение матриц и дополнительных переменных в цикле, однако скорость работы подобной реализации будет значительно быстрее рекурсивной.

3.4.1 Теоретический анализ затрачиваемой памяти

Рекурсивная реализация алгоритма Левенштейна. Для получения конечной оценки затрачиваемой памяти необходимо память, затрачиваемую на единичный вызов функции умножить на максимальную глубину рекурсии, то есть на n + m, где n и m - длины сравниваемых строк s1 и s2 соответственно.

1. ссылки на строки s1, s2: (m + n) * sizeof(reference),
2. длины строк: 2 * sizeof(int),
3. дополнительная переменная внутри алгоритма: sizeof(int)
4. адрес возврата

Матричная реализация алгоритма Левенштейна

1. строки: sizeof(str) * (n + m)
2. матрица: sizeof(int) * (n + 1) * (m + 1)
3. дополнительная переменная внутри алгоритма: sizeof(int)

Рекурсивный матричный алгоритм Левенштейна. Аналогично обычному рекурсивному алгоритму для получения конечной оценки затрачиваемой памяти необходимо память, затрачиваемую на каждом рекурсивном вызове умножить на максимальную глубину рекурсии.

1. строки: sizeof(str) * (n + m)
2. матрица: sizeof(int) * (n + 1) * (m + 1)

При каждой необходимости предварительного подсчёта значения (рек. вызова)

1. передача строки и столбца: 2 * sizeof(int)
2. дополнительная переменная: sizeof(int)
3. адрес возврата

Матричная реализация алгоритма Дамера-Левенштейна

1. строки: sizeof(str) * (n + m)
2. матрица: sizeof(int) * (n + 1) * (m + 1)
3. дополнительная переменная внутри алгоритма: sizeof(int)

3.5 Описание тестирования

3.5.1 Интерфейс программы

При запуске программы пользователя встречает меню выбора реализаций алгоритма:

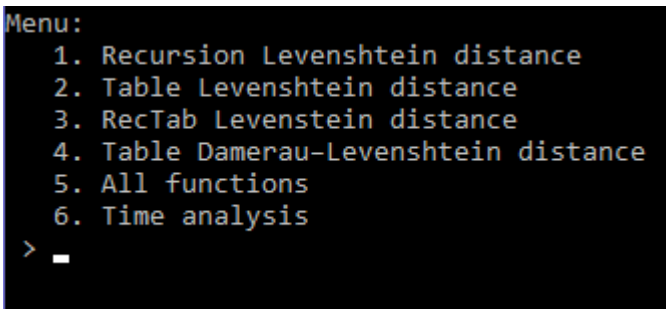


Рис.5: Меню программы

После выбора необходимой реализации пользователю предлагают ввести строки s1 и s2. После ввода программа выдаёт результат:

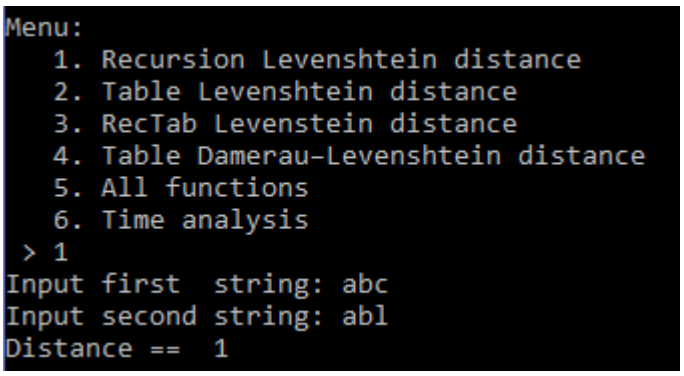


Рис.6: Ввод строк и результат работы программы

3.5.2 Тесты

Тестирование было организовано с помощью библиотеки **unittest**. Было создано две вариации тестов:

В первой сравнивались результаты функции с реальным результатом.

Во второй сравнивались результаты двух функций(рекурсивной и табличной). При сравнении результатов двух функций использовалась функция random string, которая генерирует случайную строку нужной длины.

Листинг 5: Функция random string

```
1 def RandomString(strLength = 5):
2     letters = string.ascii_lowercase
3     return ''.join(random.choice(letters) for i in range(strLength))
```

4 Исследовательская часть

4.1 Примеры работы

Проверка на пустые строки:

```
source = ""
target = ""
Recursive Levenshtein: 0
Table Levenshtein: 0
Recursive-Table Levenshtein: 0
Table Damerau-Levenshtein: 0
```

Проверка на на равенство строк:

```
source = "abc"
target = "abc"
Recursive Levenshtein: 0
Table Levenshtein: 0
Recursive-Table Levenshtein: 0
Table Damerau-Levenshtein: 0
```

Операция удаления:

```
source = "abc"
target = "ab"
Recursive Levenshtein: 1
Table Levenshtein: 1
Recursive-Table Levenshtein: 1
Table Damerau-Levenshtein: 1
```

Операция замены:

```
source = "abf"
target = "abc"
Recursive Levenshtein: 1
Table Levenshtein: 1
Recursive-Table Levenshtein: 1
Table Damerau-Levenshtein: 1
```

Операция вставки:

```
source = "ab"
targer = "abc"
Recursive Levenshtein: 1
Table Levenshtein: 1
Recursive-Table Levenshtein: 1
Table Damerau-Levenshtein: 1
```

Операция перестановки:

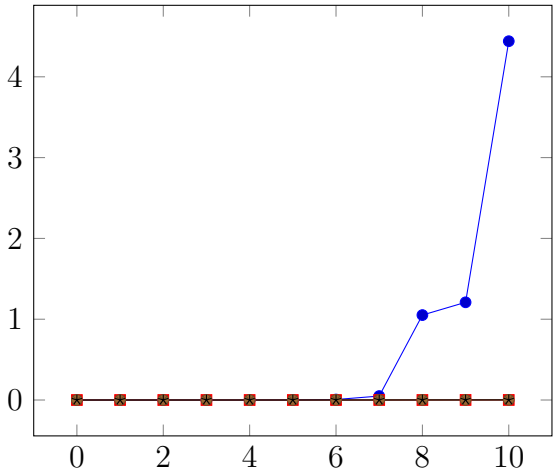
```
source = "abc"
target = "acb"
Recursive Levenshtein: 2
Table Levenshtein: 2
Recursive-Table Levenshtein: 2
Table Damerau-Levenshtein: 1
```

4.2 Постановка эксперимента по замеру времени

Для замера времени было использовано средство **QueryPerformanceCounter**
Были произведены замеры для строк длиной от 0 до 1000.
Для каждой размерности было проведено 100 вызовов функции. После чего получившееся время было поделено на 100. Таким образом было получено аппроксимированное значение времени выполнения функции
Результаты замеров процессорного времени:

Был проведен замер времени работы каждого из алгоритмов.

Длина строки	Lev(R)	Lev(MR)	Lev(M)	DamLev(M)
3	506	500.7	180.3	170.1
5	2027.3	1820.2	200.3	223.3
7	28388.8	24681.7	292.3	285.1
9	645759	538134	311.7	294.5
11	12511100	11792800	331	391.6



Легенда:
Синий цвет - Рекурсивная реализация
Коричныевый цвеи - Рекурсивная матричная реализация
Чёрный цвет - Алгоритм Дамерау-Левенштейна
Красный цвет - Обычная матричная реализцаия

4.3 Сравнительный анализ на материале экспериментальных данных

Теоретические расчёты подтвердились результатами, полученными на пратике: рекурсивный алгоритм ввиду многократного вызова функции и пересчёта уже известных значений

выполняется крайне долго, рекурсивная матричная реализация выполняется быстрее, но всё равно из-за операций со стеком и вызовом самой себя уступает по времени обычной матричной реализации. Алгоритм Дамерау-Левенштейна уступает по времени обычной матричной реализации ввиду дополнительной проверки на перестановку символов.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна, применена методика динамического программирования для реализации указанных алгоритмов, а также произведён сравнительный анализ линейной и рекурсивной реализации алгоритмов. Было установлено, что обычный рекурсивный алгоритм занимает меньше памяти по сравнению с матричными реализациями, однако за быстродействие матричных алгоритмов приходится расплачиваться памятью.