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		И.В. Кобаренков
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Содержание

Введение

Редакционное расстояние или же расстояние Левенштейна - это минимальное кол-во редакторских операций, которое необходимо для превращения одной строки в другую.

Применяются данные алгоритмы в поисковых строках браузеров, а также в биоинформатике.
Задания данной лабораторной работы:
1) изучить алгоритмы Левенштейна и Дамерау-Левенштейна;
2) реализовать четыре алгоритма: матричный, рекурсивный, рекурсивный с заполнением матрицы и Дамерау-Левенштейна;
3) провести сравнительный анализ данных алгоритмов определения расстояния между строками;
4) предоставить эксперементальное подтверждение различий во временной эффективности алгоритмов при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирущихся длинах строк;

1 Аналитическая часть

Задача по нахождению расстояния Левенштайна заключается в поиске минимального количества операций необходимых для преобразования одной строки в другую.
Для выполнения этой задачи используются следующие редакционные операции:
Вставка (I - Insert) - штраф 1
Удаление (D - Delete) - штраф 1
Замена (R - Replace) - штраф 1
Совпадение (M - Match) - штраф 0

В алгоритме Далмерау-Левенштайна допускается ещё одна редакционная операция:
Транспозиция(T) - штраф 1

Все операции обладают штрафом.
Штраф - это условная стоимость совершения данной операции. Цель алгоритмов подобрать набор операций, суммарный штраф которых будет минимален.

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

2.1 Схемы алгоритмов

В данном разделе представлены схемы реализуемых алгоритмов.

2.1.1 Схема рекурсивной реализации алгоритма Левенштейна

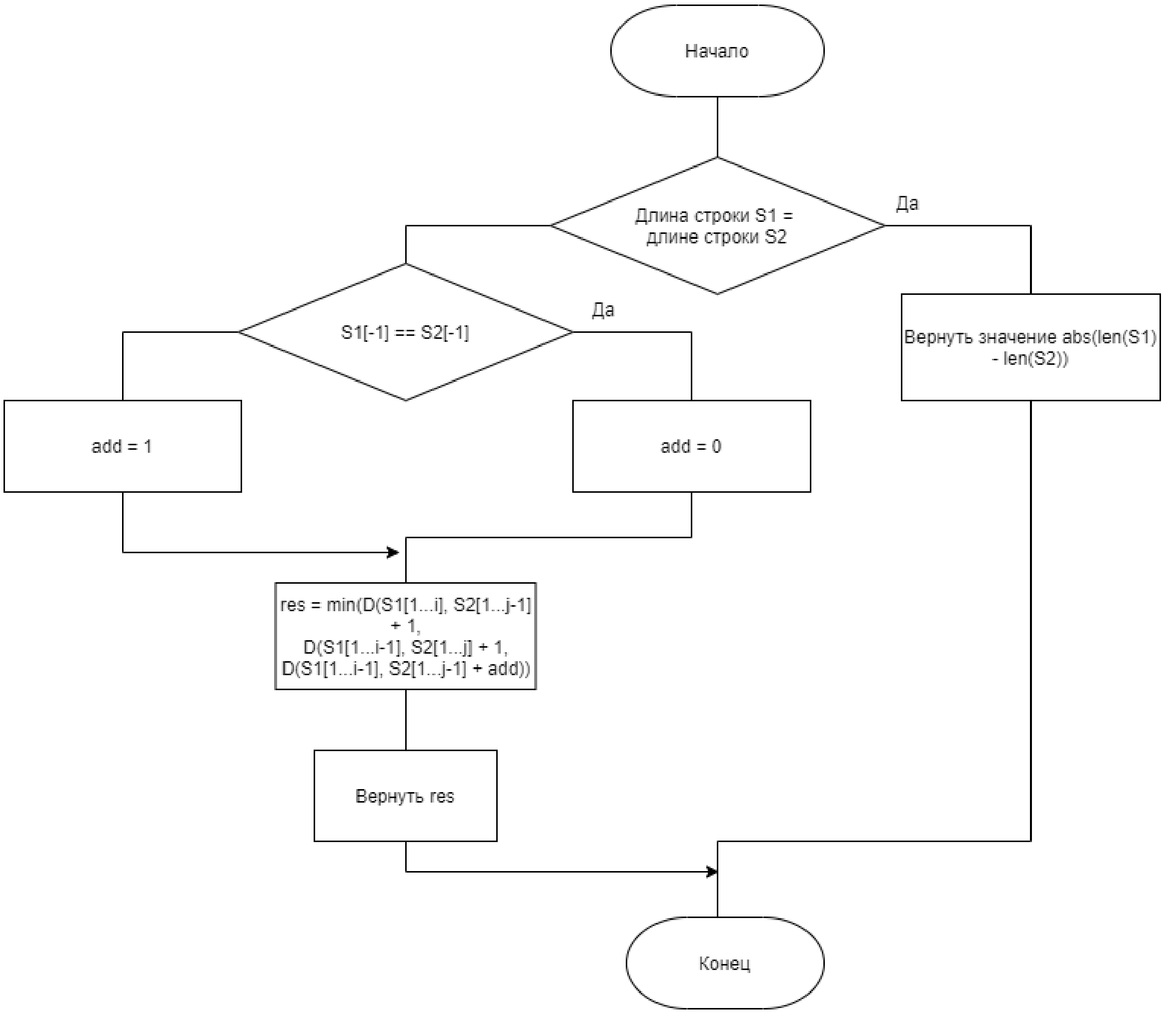


Рис. 1: Схема рекурсивного алгоритма Левенштейна

2.1.2 Схема матричной реализации алгоритма Левенштейна

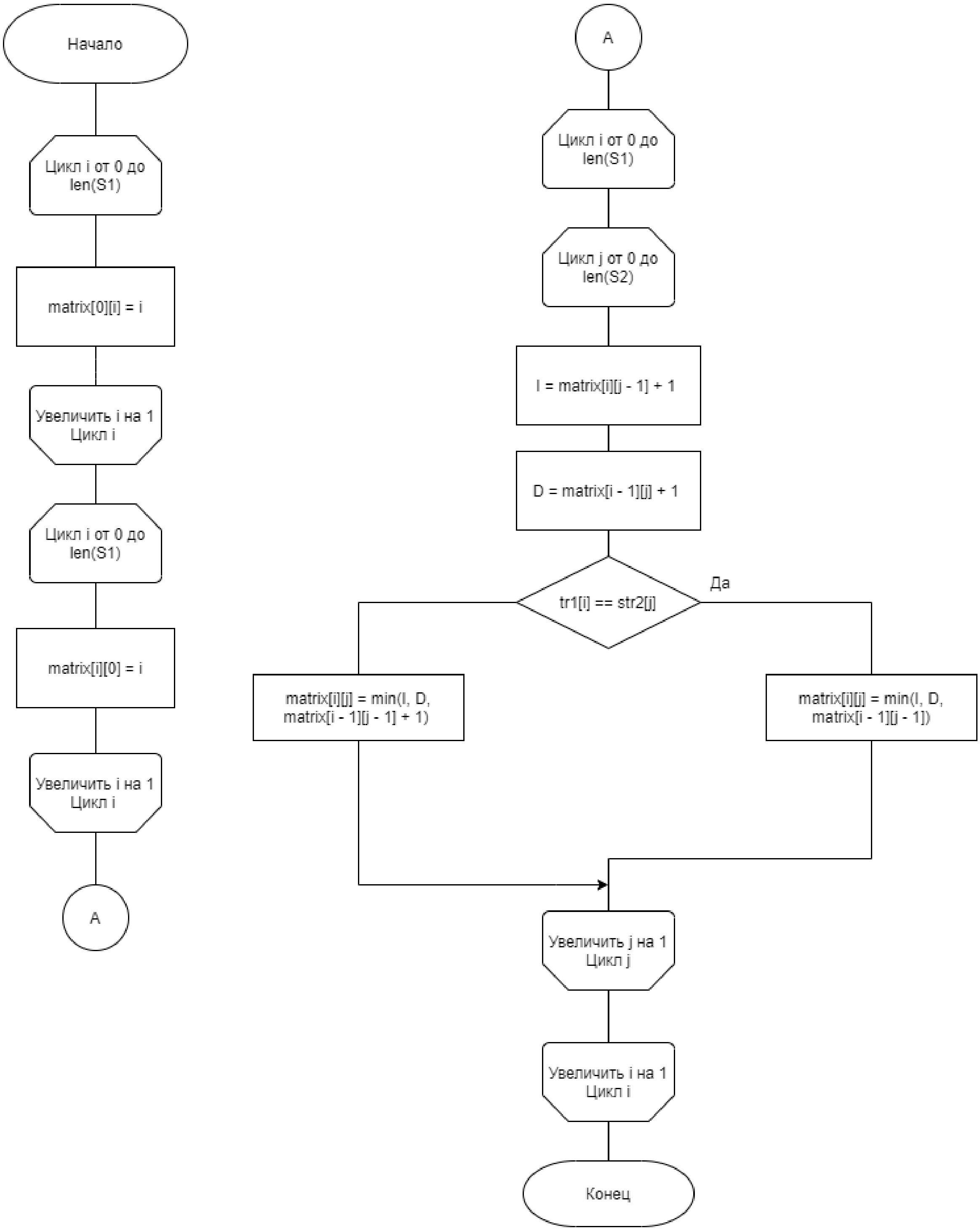


Рис. 2: Схема матричной реализации алгоритма Левенштейна

2.1.3 Схема рекурсивного матричного алгоритма Левенштейна

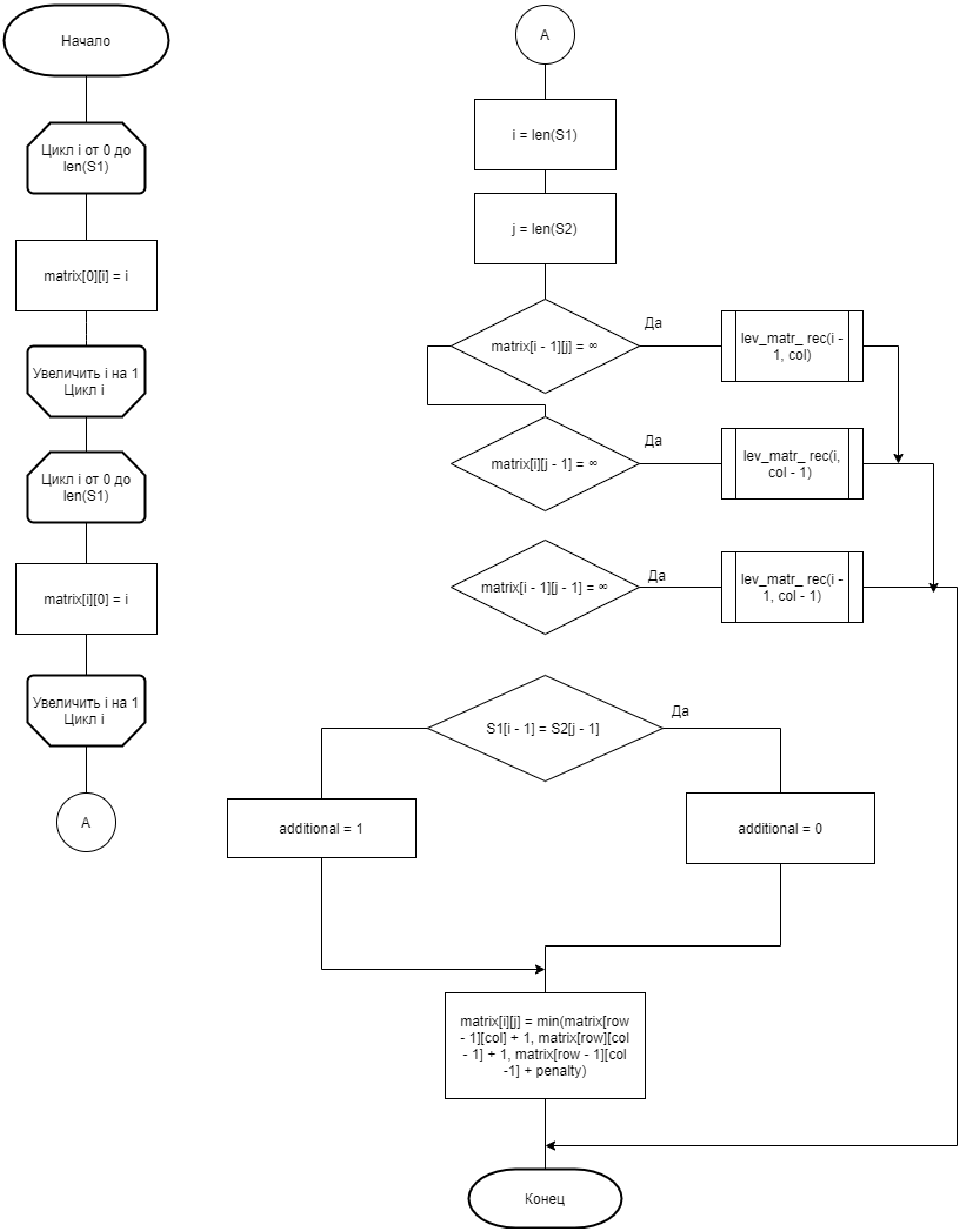


Рис. 3: Схема рекурсивной матричной реализации алгоритма Левенштейна

2.1.4 Схема алгоритма Дамерау-Левенштейна

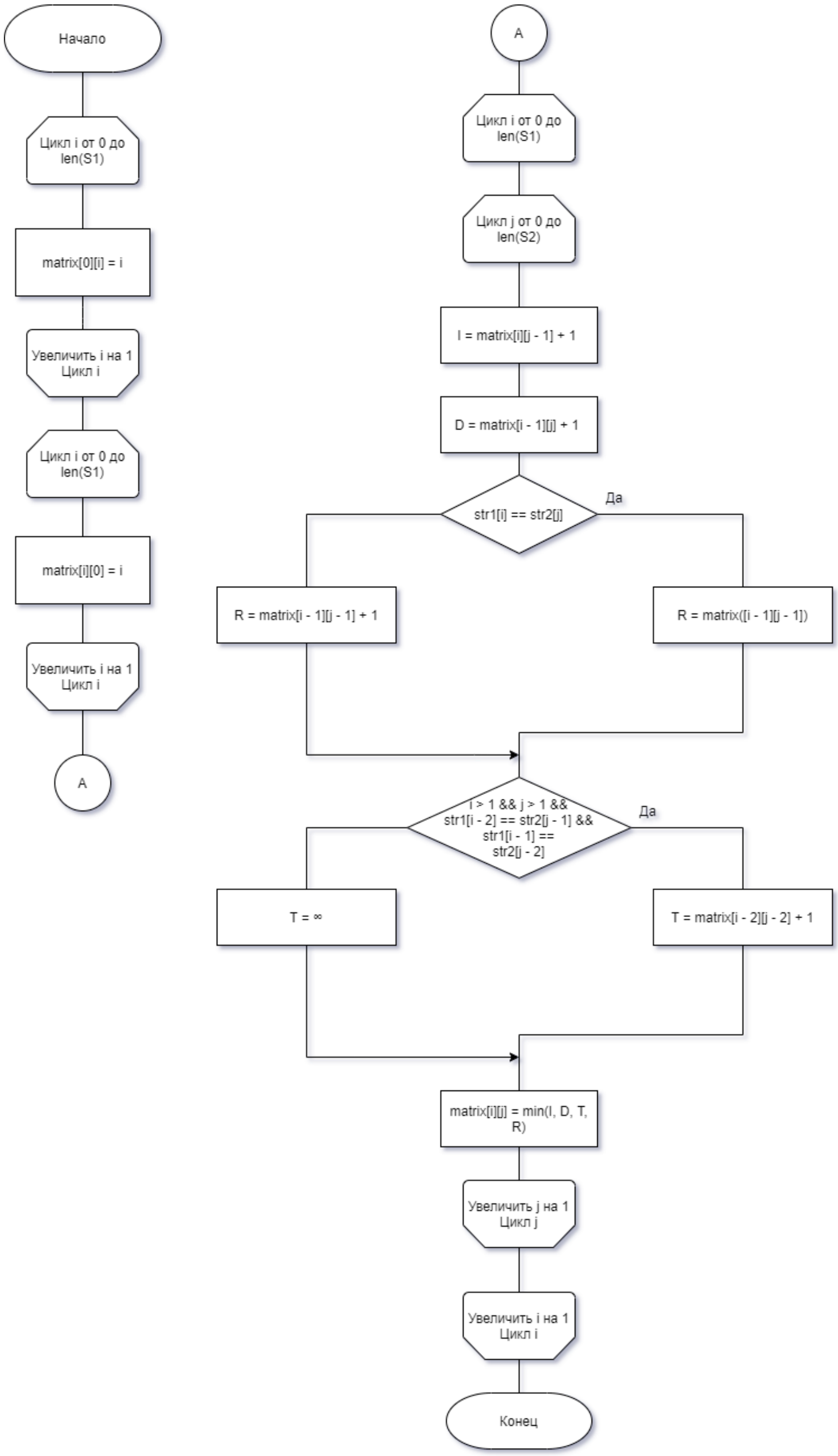


Рис. 4: Схема алгоритма Дамерау-Левенштейна

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: две строки
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 3.8.0

3.3 Листинг кода

В данном разделе представлен листинг кода разработанных алгоритмов.

3.3.1 Рекурсивный алгоритм Левенштейна

```
def levenshtein_recursive(s1, s2, i, j):
    if min(i, j) == 0:
        return max(i, j)
    else:
        m = 0 if s1[i-1] == s2[j-1] else 1
        return min(levenshtein_recursive(s1, s2, i-1, j) + 1,
                   levenshtein_recursive(s1, s2, i, j-1) + 1,
                   levenshtein_recursive(s1, s2, i-1, j-1) + m)
```

3.3.2 Матричный алгоритм Левенштейна

```python
def levenshtein_matrix(s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    matrix = [[0 for j in range(n2+1)] for i in range(n1+1)]

    for i in range(n1+1):
        matrix[i][0] = i

    for j in range(n2+1):
        matrix[0][j] = j

    for i in range(1, n1+1):
        for j in range(1, n2+1):
            m = 0 if s1[i-1] == s2[j-1] else 1
             matrix[i][j] = min(matrix[i-1][j] + 1,
                                matrix[i][j-1] + 1,
                                matrix[i-1][j-1] + m)

    return matrix[-1][-1]
```

3.3.3 Рекурсивный матричный алгоритм Левенштейна

```python
def levenshtein_recursive_matrix(s1, s2, i, j, matrix):
    if min(i, j) == 0:
        matrix[i][j] = max(i, j)
    else:
        if matrix[i][j] == -1:
            m = 0 if s1[i-1] == s2[j-1] else 1
             matrix[i][j] = min(levenshtein_recursive_matrix(s1, s2, i-1,
                                levenshtein_recursive_matrix(s1, s2, i, j-
                                levenshtein_recursive_matrix(s1, s2, i-1,
    return matrix[i][j]
```

3.3.4 Алгоритм Дамерау-Левенштейна

```
def dameray_levenshtein(s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    matrix = [[0 for j in range(n2+1)] for i in range(n1+1)]

    for i in range(n1+1):
        matrix[i][0] = i

    for j in range(n2+1):
        matrix[0][j] = j

    for i in range(1, n1+1):
        for j in range(1, n2+1):
            m = 0 if s1[i-1] == s2[j-1] else 1
            if i > 1 and j > 1 and s1[i-2] == s2[j-1] and s1[i-1] == s2[j
                matrix[i][j] = min(matrix[i-1][j] + 1,
                                   matrix[i][j-1] + 1,
                                   matrix[i-1][j-1] + m,
                                   matrix[i-2][j-2] + 1)
            else:
                matrix[i][j] = min(matrix[i-1][j] + 1,
                                   matrix[i][j-1] + 1,
                                   matrix[i-1][j-1] + m)

    return matrix[-1][-1]
```

3.4 Сравнительный анализ матричной и рекурсивной реализаций

Рекурсивная версия алгоритма работает значительно медленне матричной реализации из-за многократного вызова функции. На каждый вызов необходимо производить соответствующие операции со стеком. Главным недостатком является повторное вычисление тех значений, которые были посчитаны на более ранних этапах рекурсии. В матричных же реализация будет затрачена дополнительная память на хранение матриц и дополнительных переменных, однако скорость работы алгоритма будет значительно быстрее чем у рекурсивного.

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:

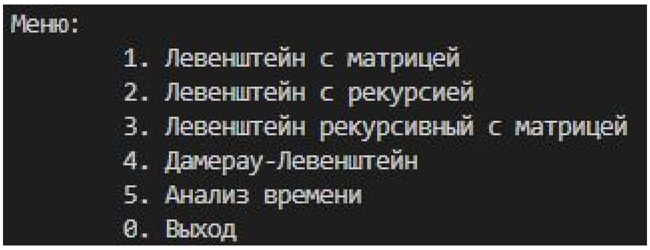


Рис.5: Меню программы

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

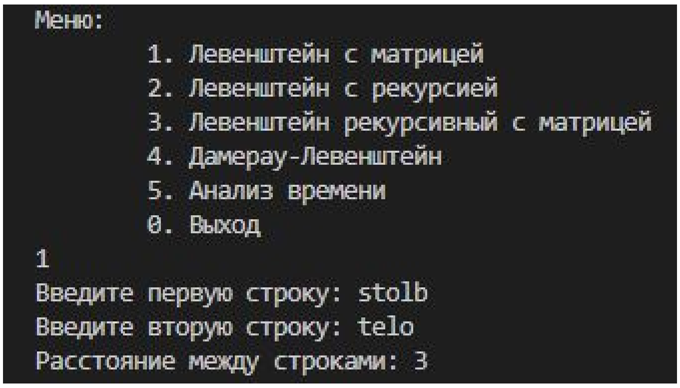


Рис.6: Ввод строк и результат работы программы

3.5.2 Тесты

Тестирование было организовано с помощью библиотеки unittest.

4 Исследовательская часть

В данной части отчета приведены примеры работы программ, а также анализ алгоритмов на основе эксперементальных данных.

4.1 Примеры работы

Пустые строки:

```
string_1 = ""
string_2 = ""
Recursive Levenshtein: 0
Table Levenshtein: 0
Recursive-Table Levenshtein: 0
Damerau-Levenshtein: 0
```

Равенство строк:

```
string_1 = "abc"
string_2 = "abc"
Recursive Levenshtein: 0
Table Levenshtein: 0
Recursive-Table Levenshtein: 0
Damerau-Levenshtein: 0
```

Удаление:

```
string_1 = "aac"
target = "aa"
Recursive Levenshtein: 1
Table Levenshtein: 1
Recursive-Table Levenshtein: 1
Damerau-Levenshtein: 1
```

Замена:

```
string_1 = "abc"
string_2 = "abd"
Recursive Levenshtein: 1
Table Levenshtein: 1
Recursive-Table Levenshtein: 1
Damerau-Levenshtein: 1
```

Вставка:

```
string_1 = "ab"
string_2 = "abc"
Recursive Levenshtein: 1
Table Levenshtein: 1
Recursive-Table Levenshtein: 1
Damerau-Levenshtein: 1
```

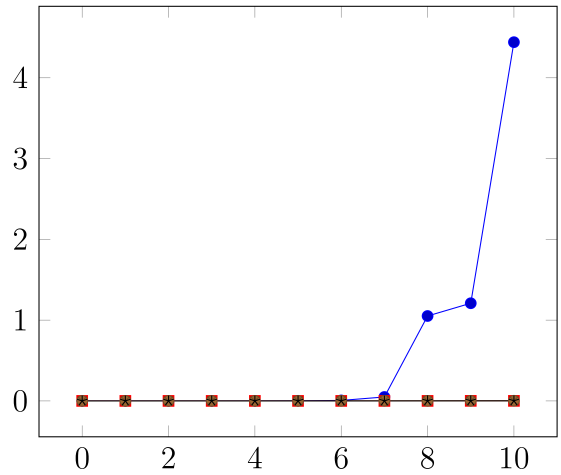
Перестановка:

```
string_1 = "abc"
string_2 = "acb"
Recursive Levenshtein: 2
Table Levenshtein: 2
Recursive-Table Levenshtein: 2
Damerau-Levenshtein: 1
```

4.2 Замер времени

Были взяты строки 4 размерностей. 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:

Длина строки	Lev(M)	Lev(R)	Lev(RM)	DamLev
3	2.0103	3.9951	2.0020	2.0096
5	1.9924	106.0962	4.0040	3.0026
7	4.0032	3012.1695	8.0068	4.9967
10	7.9977	500312.1714	16.0050	8.0165



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 из-за многократного вызова функции и пересчёта уже известных значений выполняется очень долго, рекурсивная матричная реализация выполняется быстрее, но всё равно из-за затрат времени на вызов самой себя уступает по времени обычной матричной реализации. Алгоритм Дамерау-Левенштейна незначитлеьно уступает обычной матричной реализации ввиду дополнительной проверки.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна, реализованы указанные алгоритмы, а также произведён сравнительный анализ алгоритмов. Было установлено, что обычный рекурсивный алгоритм занимает меньше памяти по сравнению с матричными реализациями, однако сильно уступает им по времени.