**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** Расстояния Левенштейна и Дамерау-Левенштейна

**Студент** Топорков П. А.

**Группа** ИУ7-53Б

**Оценка (баллы)** ______________________

**Преподаватели** Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

## Введение

Редакционные расстояния позволяют понять, насколько близки две строки. Поисковая система все равно находит то, что надо, даже если вы совершили пару опечаток в запросе, потому что для неправильно введённого слова в словаре ищется наиболее похожее на него слово. Для определения сходвстав между словами или, в обсщем случае, строками, используется расстояние Левенштейна, или редакционное расстояние: для пары строк рассчитывается, сколько действий по редактированию одной строки нужно совершить, чтобы получить другую строку.

Алгоритмы нечеткого поиска являются основой систем проверки орфографии и полноценных поисковых систем. Например, такие алгоритмы используются в тех же поисковых системах, сравнения текстовых файлов утилитой diff и ей подобными, в биоинформатике для сравнения генов.

# 1 Аналитическая часть

## 1.1 Детализация задачи

В данной работе рассматриваются алгоритмы:

- поиска расстояния Левенштейна;
- поиска расстояния Дамерау-Левенштейна.

Целью данной лабораторной работы является изучить и реализовать алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштнейна и сравнить разные варианты их реализации.

Задачами данной лабораторной являются:

- изучение алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна;
- Реализация алгоритма поиска расстояния Левенштейна с заполнением таблицы по формуле, рекурсивным заполнением таблицы, с использованием рекурсии;
- реализация алгоритма поиска расстояния Дамерау–Левенштейна без использования рекурсии.
- сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

## 1.2 Расстояние Левенштейна

Расстояние Левенштейна [1] - это минимальное необходимое количество редакторский операций(вставки, удаления, замены) для преобразования одной строки в другую.

Возможные действия и их обозначения:

- R (replace) — заменить,
- I (англ. insert) — вставить,
- D (англ. delete) — удалить,
- M(match) - совпадение.

Штраф за совпадение 0, за остальные действия 1.

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1, где $|a|$ означает длину строки $a$; $a[i]$ — i-ый символ строки $a$ , функция $D(i, j)$, равная расстоянию между подстроками длины i и j, определена как:

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

В формуле 1 функция m выражается как:

$$m(a,b) = \begin{cases} 0 & \text{если a = b,} \\ 1 & \text{иначе} \end{cases}. \tag{2}$$

### Рекурсивный алгоритм поиска расстояния Левенштейна

Рекурсивный алгоритм реализует формулу 1.

Для перевода из строки $a$ в строку $b$ требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. 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 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.

### Матричный алгоритм поиска расстояния Левенштейна

Прямая реализация формулы 1 может быть малоэффективна по времени исполнения при больших $i, j$, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$.

На рисунке 1 изображена матрица, которой можно представить весь процесс:

| | | А | Р | Е | С | Т | А | Н | Т |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Д | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| А | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 7 |
| Г | 3 | 2 | 2 | 3 | 4 | 5 | 6 | 6 | 7 |
| Е | 4 | 3 | 3 | 2 | 3 | 4 | 5 | 6 | 7 |
| С | 5 | 4 | 4 | 3 | 2 | 3 | 4 | 5 | 6 |
| Т | 6 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 |
| А | 7 | 6 | 6 | 5 | 4 | 3 | 2 | 3 | 4 |
| Н | 8 | 7 | 7 | 6 | 5 | 4 | 3 | 2 | 3 |

**Рисунок 1** – Матрица поиска расстояния Левенштейна.

На рисунке 2 представлена работа с ячейками матрицы.Если посмотреть на процесс работы алгоритма, несложно заметить, что на каждом шаге используются только две последние строки матрицы, следовательно, потребление памяти можно уменьшить до O(min(m, n)) [2].

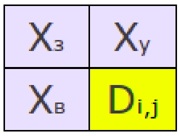



**Рисунок 2** – Используемые за одну итерацию ячейки матрицы при поиске расстояния Левенштейна.

### Рекурсивный алгоритм поиска расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода 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.

## 1.3 Расстояние Дамерау–Левенштейна

Эта вариация вносит в определение расстояния Левенштейна еще одно правило — транспозиция (перестановка) двух соседних букв также учитывается как одна операция, наряду со вставками, удалениями и заменами [2]. Используемые на каждой итерации ячейки изображены на рисунке 3.

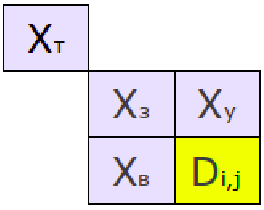



**Рисунок 3** – Используемые за одну итерацию ячейки матрицы при поиске расстояния Дамерау–Левенштейна.

Расстояние Дамерау — Левенштейна может быть найдено по формуле 3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j)=0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1, & \\ \quad d_{a,b}(i-1,j)+1, & \text{иначе} \\ \quad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \\ \quad \left[ \begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \tag{3}$$

## Вывод

Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно(с помощью матрицы).

# 2 Конструкторская часть

## 2.1 Требования к вводу

В разрабатываемом ПО предъявляются следующие требования ко вводу.

1. На вход подаются две строки.
2. Строчные и прописные буквы считаются разными.

## 2.2 Требования к выводу

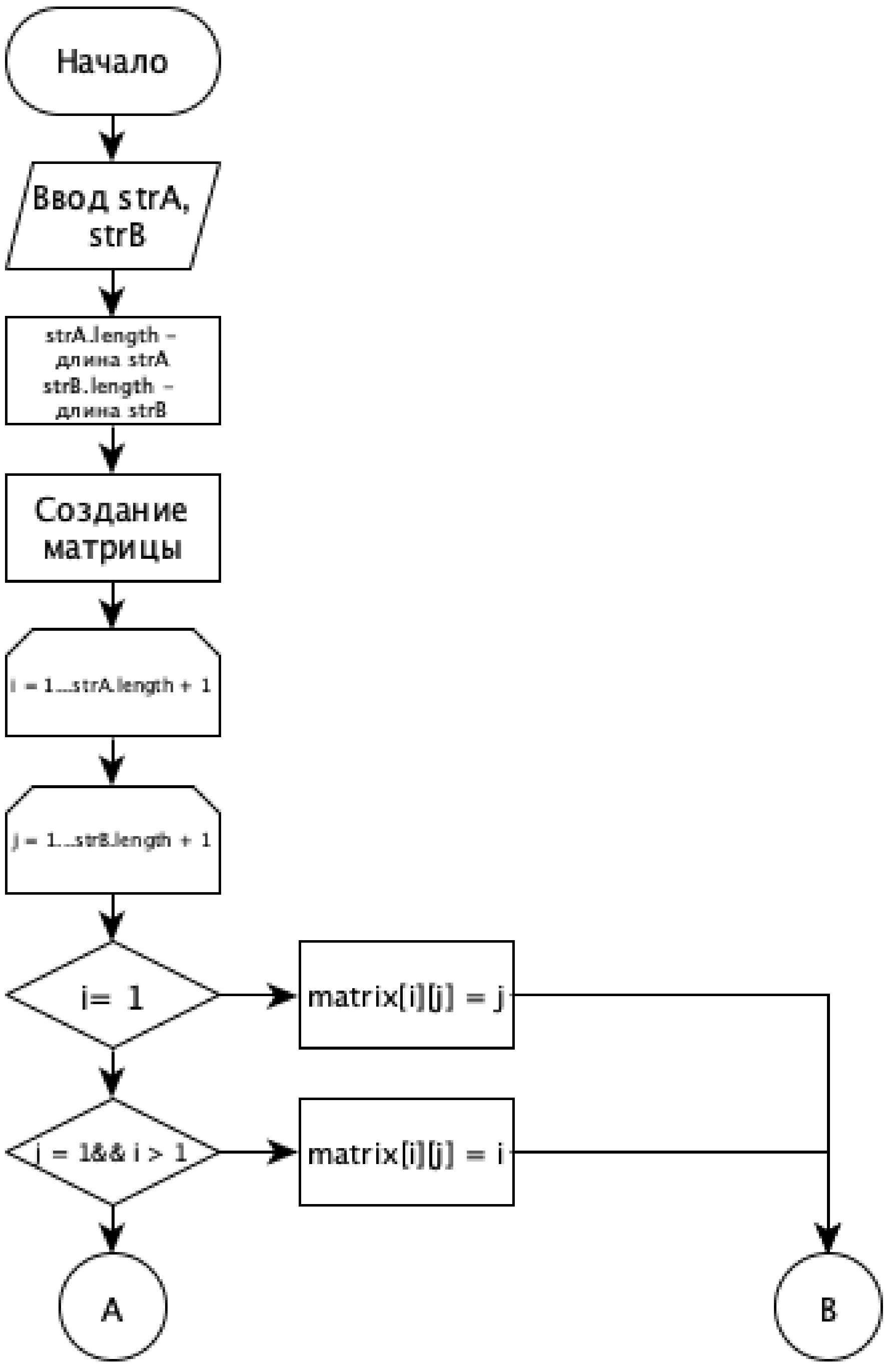
Программа выводит расстояние и матрицу (если она использовалась).

## 2.3 Требования к программе

Две пустые строки – корректный ввод, программа не должна аварийно завершаться.

## 2.4 Схемы алгоритмов

На рисунках 4, 5, 6 показаны схемы алгоритмов поиска расстояния Левенштейна. На рисунке 7 показан алгоритм поиска расстояния Дамерау–Левенштейна без рекурсии.



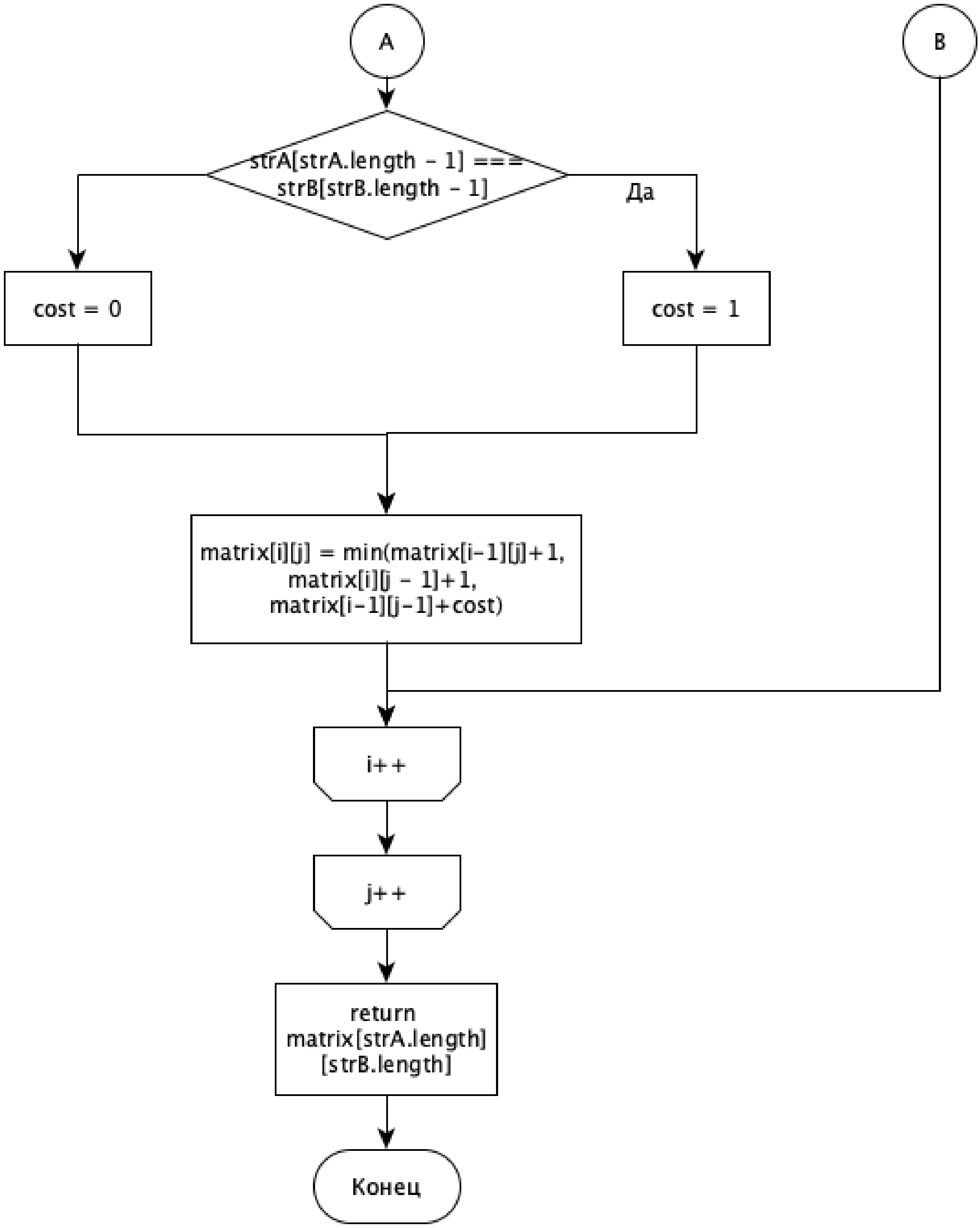



**Рисунок 4** – Схема алгоритма поиска расстояния Левенштейна с помощью матрицы(без рекурсии).

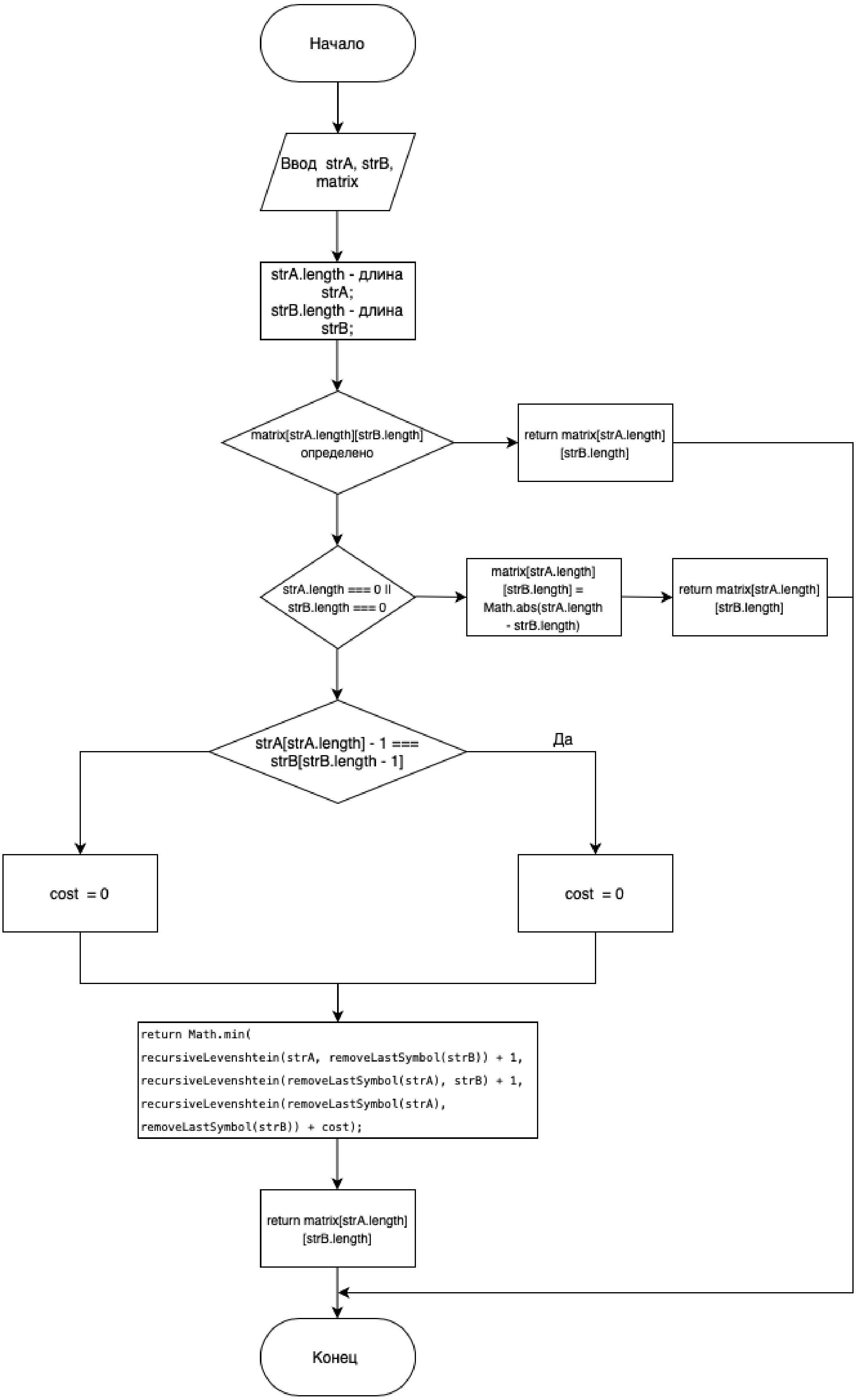



**Рисунок 5** – Схема алгоритма поиска расстояния Левенштейна с помощью матрицы(с использованием рекурсии).

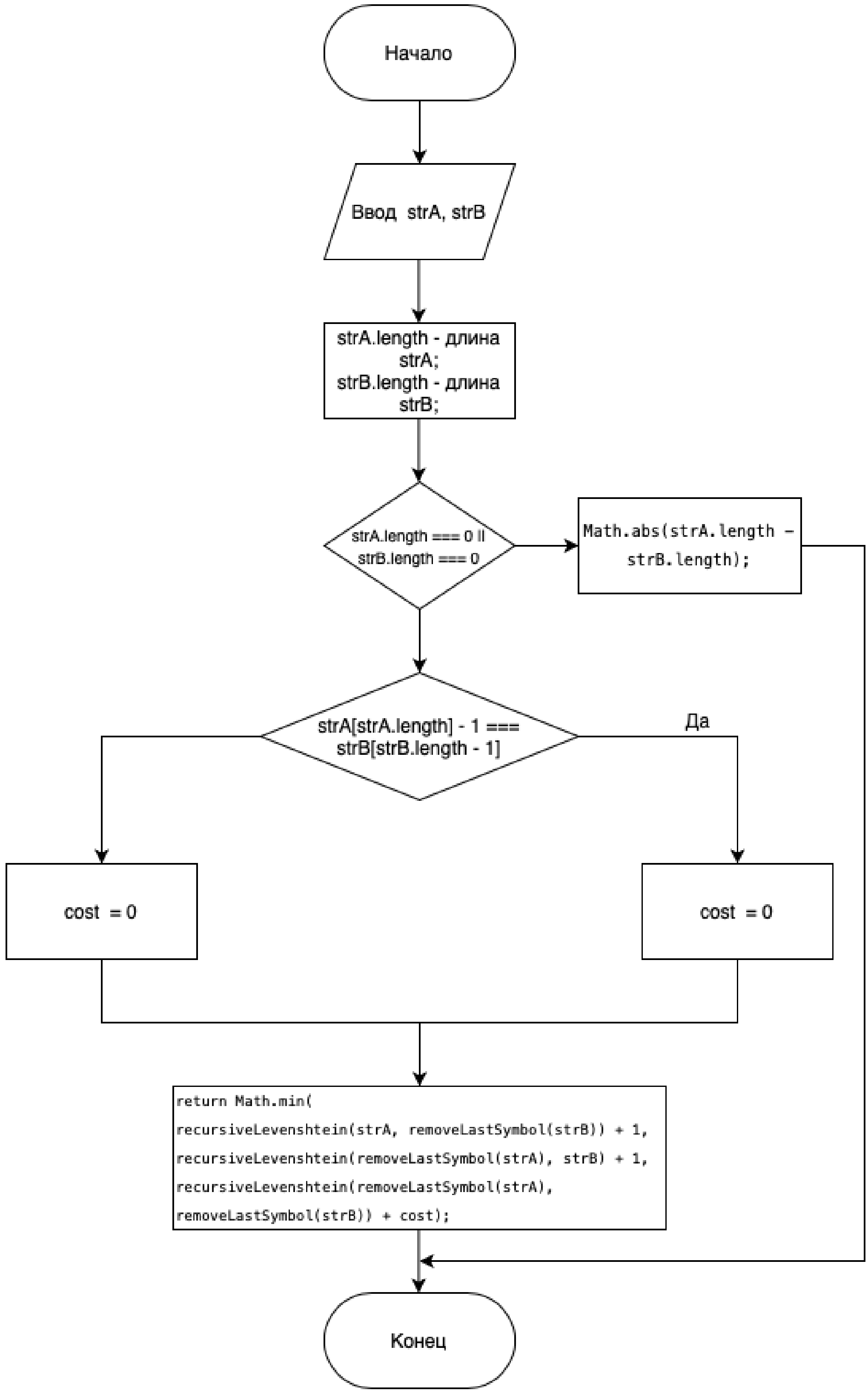
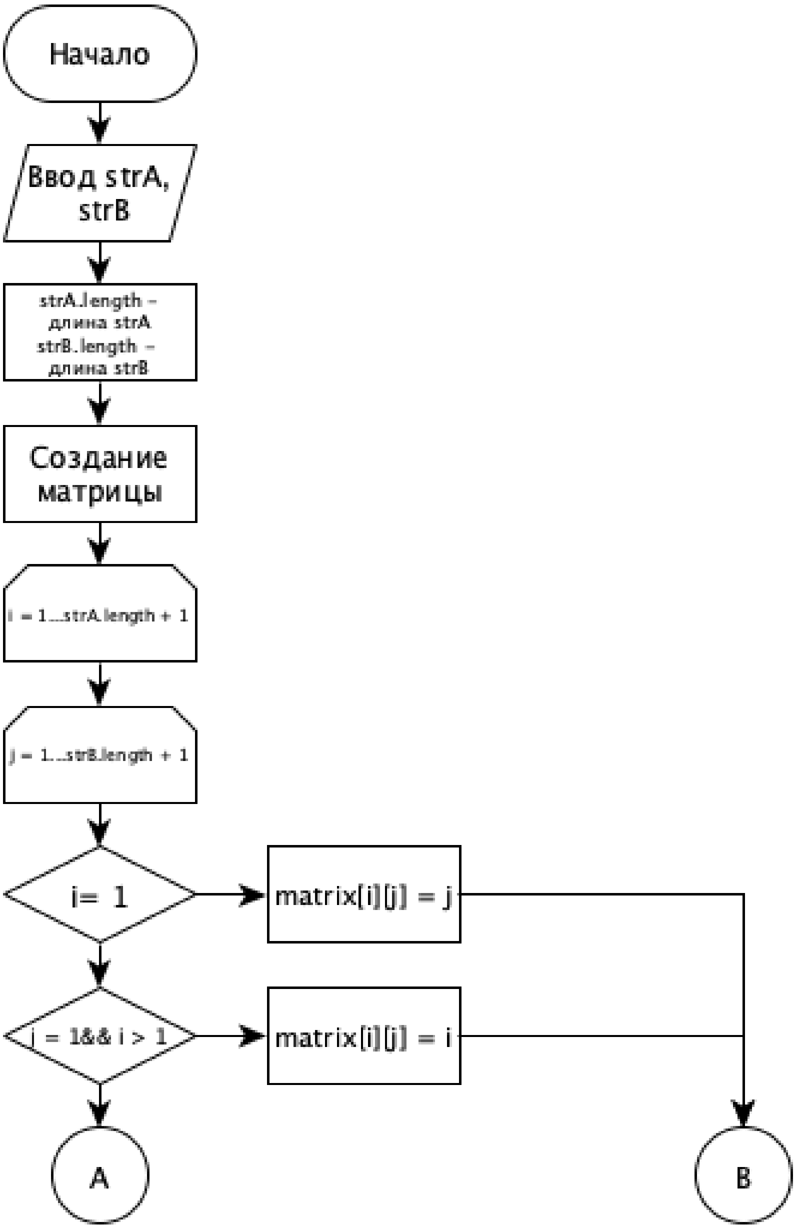



**Рисунок 6** – Схема алгоритма поиска расстояния Левенштейна с использованием рекурсии.


Начало
Ввод strA, strB
strA.length – длина strA
strB.length – длина strB
Создание матрицы
i = 1...strA.length + 1
j = 1...strB.length + 1
i= 1
matrix[i][j] = j
j = 1&& i > 1
matrix[i][j] = i
A
B

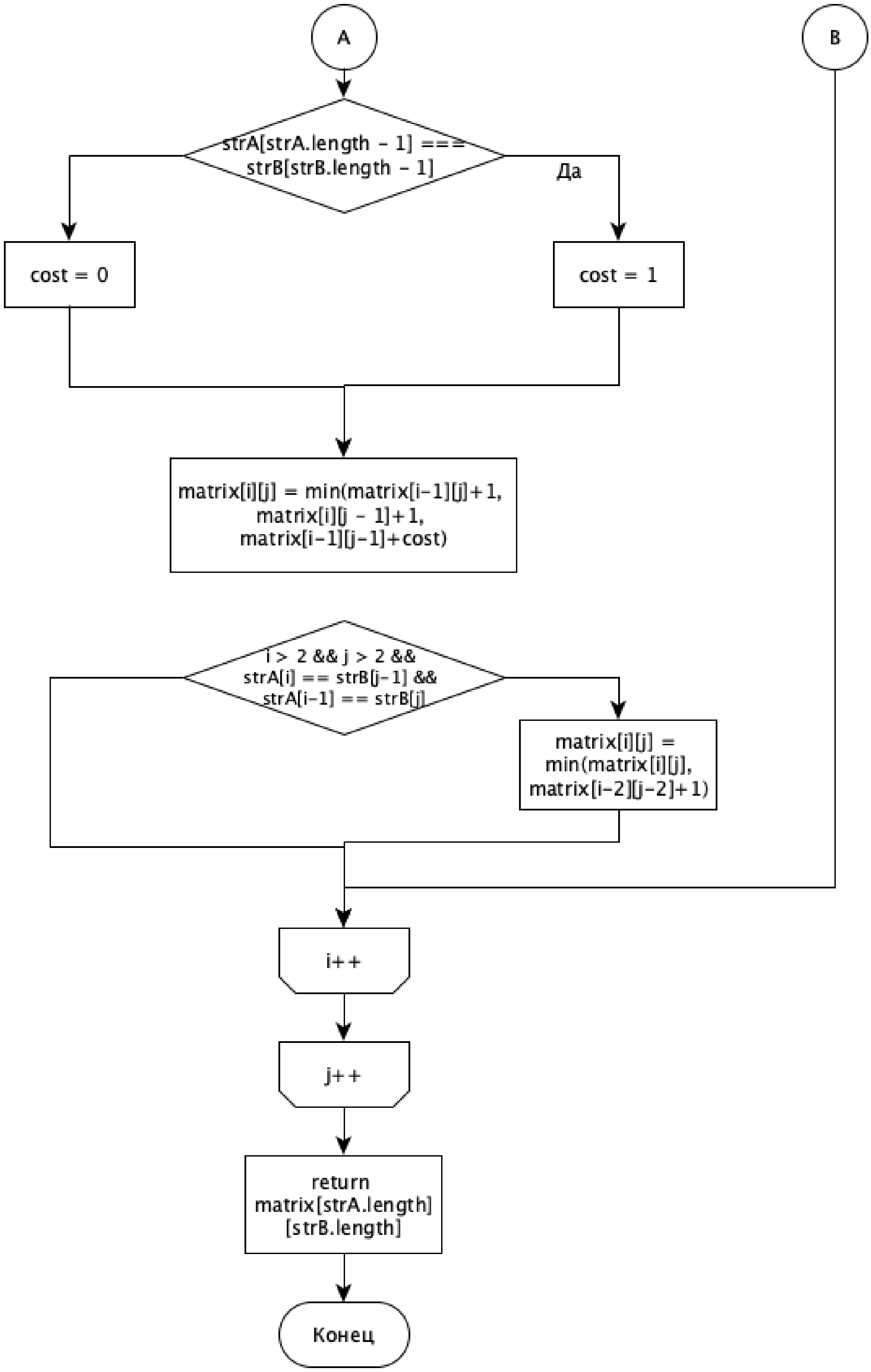



**Рисунок 7** – Схема алгоритма поиска расстояния Дамерау–Левенштейна с помощью матрицы(без рекурсии).

## Вывод

Были проанализированы схемы алгоритмов, а также требования к конечному ПО.

# 3 Технологическая часть

## 3.1 Выбор языка программирования

Для большей практики был выбран язык программирования Python.

Время работы алгоритмов было замерено с помощью функции time() из библиотеки time.

## 3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- main.py - главный файл программы, в котором располагаются алгоритмы, меню и замеры времени выполнения.

В листингах 1, 2, 3, 4 представлен код используемых алгоритмов.

**Листинг 1** – Функция нахождения расстояния Левенштейна с использованием рекурсии.

```
def recursiveLevenshtein(s1, s2):
    if not(len(s1)) or not(len(s2)):
        return abs(len(s1) - len(s2))

    cost = 0 if s1[-1] == s2[-1] else 1
    return min(
        recursiveLevenshtein(s1, s2[:-1]) + 1,
        recursiveLevenshtein(s1[:-1], s2) + 1,
        recursiveLevenshtein(s1[:-1], s2[:-1]) + cost
    )
```

**Листинг 2** – Функция нахождения расстояния Левенштейна с использованием матрицы(не рекурсивно)

```
def levenshtein(s1, s2):
    matrix = [[i + j if not(i * j) else 0 for j in range(len(s2) + 1) ] for i in range(len(s1) + 1)]

    for i in range(1, len(s1) + 1):
        for j in range(1, len(s2) + 1):
```

```
            if s1[i - 1] == s2[j - 1]:
                matrix[i][j] = matrix[i - 1][j - 1]
                continue
            matrix[i][j] = min(matrix[i - 1][j], matrix[i][j - 1], matrix[i - 1][j - 1]) + 1

    return matrix[len(s1)][len(s2)]
```

**Листинг 3** – Функция нахождения расстояния Левенштейна с использованием матрицы(рекурсивно)

```
def matrixRecursiveLevenshtein(s1, s2, matrix):
    if matrix[len(s1)][len(s2)] != -1:
        return matrix[len(s1)][len(s2)]
    if not(len(s1)) or not(len(s2)):
        matrix[len(s1)][len(s2)] = abs(len(s1) - len(s2))
        return matrix[len(s1)][len(s2)]

    cost = 0 if s1[-1] == s2[-1] else 1
    matrix[len(s1)][len(s2)] = min(
        matrixRecursiveLevenshtein(s1, s2[:-1], matrix) + 1,
        matrixRecursiveLevenshtein(s1[:-1], s2, matrix) + 1,
        matrixRecursiveLevenshtein(s1[:-1], s2[:-1], matrix) + cost
    )

    return matrix[len(s1)][len(s2)]
```

**Листинг 4** – Функция нахождения расстояния Дамерау–Левенштейна с использованием матрицы(не рекурсивно)

```
def damerauLevenshtein(s1, s2):
    matrix = [[i + j if not(i * j) else 0 for j in range(len(s2) + 1) ] for i in range(len(s1) + 1)]

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

            if (i > 1 and j > 1
```

```
                and s1[i - 1] == s2[j - 2]
                and s1[i - 2] == s2[j - 1]):
                matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1)

    return matrix[len(s1)][len(s2)]
```

## 3.3 Тесты для проверки корректности программы

В таблицах 1, 2 представлены функциональные тесты для проверки работы программы. Все тесты пройдены успешно.

Таблица 1 – Функциональные тесты(Ожидаемый результат результат)

| Входные данные | Ожидаемый результат |
|---|---|
| Пустая строка, пустая строка | 0 |
| Пустая строка, "переезд" | 7 |
| "переезд" "переезд" | 0 |
| "переезд" "перекур" | 3 |
| "переезд" "lololo" | 7 |
| "12345" "21354" | 4 |

Таблица 2 – Функциональные тесты(Реальный результат)

| Входные данные | Расстояние Левенштейна | Расстояние Дамерау–Левенштейна |
|---|---|---|
| Пустая строка, пустая строка | 0 | 0 |
| Пустая строка, "переезд" | 7 | 7 |
| "переезд" "переезд" | 0 | 0 |
| "переезд" "перекур" | 3 | 3 |
| "переезд" "lololo" | 7 | 7 |
| "12345" "21354" | 4 | 2 |

## 3.4 Сравнительный анализ памяти, затрачиваемой алгоритмами

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти. Рассмотрим разницу между рекурсивной и матричной реализациями.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти (4)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 3 \cdot \mathcal{C}(\text{int})), \tag{4}$$

где $\mathcal{C}$ — оператор вычисления размера, $S_1$, $S_2$ — строки, int — целочисленный тип, string — строковый тип.

Использование памяти при итеративной реализации теоретически равно

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 7 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string}). \tag{5}$$

## Вывод

Были разработаны и протестированы спроектированные алгоритмы: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау – Левенштейна с заполнением матрицы.

# 4 Исследовательская часть

## 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система MacOS Mojave 10.14.5
- Память 8 ГБ 1600 MHz DDR3
- Процессор 2,6 GHz Intel Core i5

## 4.2 Демонстрация работы программы

На рисунке 8 представлен результат работы программы.

```
Введите первую строку: стол
Введите вторую строку: стул

Матрица расстояния Левенштейна
0 0 с т у л
0 0 1 2 3 4
с 1 0 1 2 3
т 2 1 0 1 2
о 3 2 1 1 2
л 4 3 2 2 1
Расстояние Левенштейна =  1

Матрица заполненная рекурсивно:
0 0 с т у л
0 0 1 2 3 4
с 1 0 1 2 3
т 2 1 0 1 2
о 3 2 1 1 2
л 4 3 2 2 1
Расстояние Левенштейна =  1

Расстроение Левенштейна, полученное с использованием рекурсии:  1

Матрица расстояния Дамерау-Левенштнейна
0 0 с т у л
0 0 1 2 3 4
с 1 0 1 2 3
т 2 1 0 1 2
о 3 2 1 1 2
л 4 3 2 2 1
Расстояние Дамерау-Левенштейна =  1
```

**Рисунок 8** – Результат работы программы.

## 4.3 Время работы алгоритмов.

Тестирование алгоритмов производилось при помощи функции time() из библиотеки time языка Python. time.time() всегда возвращает текущее значение времени, прошедшего с начала эпохи, типа float. Не включает время, прошедшее во время сна. Контрольная точка возвращаемого значения не определена, поэтому допустима только разница между результатами последовательных вызовов. Замеры времени для каждой длины слов проводились 100 раз. В качетсве результата взято среднее время работы алгоритма на данной длине слова.

Результат замера времени представлен в таблице 3(время в секундах).

## 4.4 Функциональные тесты

Таблица 3 – Замеры времени

| Длина | Л.(матр.) | Л.(рек с матр.) | Л.(рек) | Д.-Л.(матр.) |
|---|---|---|---|---|
| 0 | 0.000006 | 0.000005 | 0.000006 | 0.000006 |
| 1 | 0.000011 | 0.000013 | 0.000009 | 0.000011 |
| 2 | 0.000021 | 0.000030 | 0.000026 | 0.000018 |
| 3 | 0.000022 | 0.000042 | 0.000088 | 0.000023 |
| 4 | 0.000038 | 0.000065 | 0.000367 | 0.000035 |
| 5 | 0.000048 | 0.000095 | 0.001734 | 0.000064 |
| 6 | 0.000062 | 0.000131 | 0.008931 | 0.000095 |
| 7 | 0.000076 | 0.000188 | 0.053638 | 0.000100 |
| 8 | 0.000140 | 0.000231 | 0.264763 | 0.000133 |
| 9 | 0.000152 | 0.000303 | 1.530243 | 0.000171 |

## Вывод

Рекурсивный алгоритм Левенштейна работает дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии.При увеличении длины строк становится очевидна выигрышность по времени матричного варианта. Уже при длине в 7 символов матричная реализация в 950 раз быстрее. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный на аналогичных данных в 9876 раз. Алгоритм Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна. В нём добавлены дополнительные проверки, и по сути он является алгоритмом другого смыслового уровня.

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 работы решены следующие задачи.

- Были изучены алгоритмы поиска расстояни Левенштейна и Дамерау–Левенштейна.
- Реализованны алгоритмы поиска расстояния Левенштейна с заполнением таблицы по формуле, рекурсивным заполнением таблицы, с использованием рекурсии.
- Реализован алгоритм поиска расстояния Дамерау–Левенштейна без использования рекурсии.
- Проведён сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти).
- Проведено экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.
- В ходе работы было установленно, что рекурсивный алгоритм Левенштейна работает дольше итеративных реализаций (время его работы увеличивается в геометрической прогрессии), но по затратам памяти остальные алгоритмы проигрывают ему.
- Были описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

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

В результате исследований я пришла к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, но проигрывает по количеству затрачиваемой памяти.

## Список литературы


[1] В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.

[2] Нечёткий поиск в тексте и словаре [Электронный ресурс].Режим доступа:

https://habr.com/ru/post/114997/ (Дата обращения: 12.09.20)