«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)
ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа №1

«Анализ алгоритмов»

Расстояние Левенштейна и Дамерау-Левенштейна

Студент: Нгуен Фыок Санг
Группа ИУ7-56Б
Преподаватель: Волокова Л. Л.
Оценка:

Москва, 2020 г.

Оглавление

Введение

В современном мире почти каждый человек пользуются компьютером и Интернетом в частности. Люди пишут текст в документах, выполняют поиск в поисковых системах, ищут переводы слов и текстов в онлайн-словарях. В таких ситуациях человек часто делает орфографические ошибки или опечатки, и на их исправление он тратит своё время. Чтобы этого избежать, в подобных системах есть опции поиска ошибок и автоисправления. Для такой опции необходим поиск расстояния между строками по алгоритмам Левенштейна и Дамерау-Левенштейна. Также эта задача необходима и в программировании (например, для сравнения текстовых файлов или файлов кода в системах контроля версий) и в биоинформатике (например, для сравнения белков, генов и хромосом).

1. Аналитическая часть

1.1 Задачи

Цель лабораторной работы: Разработать и сравнить алгоритмы поиска расстоянии Левенштейна и Дамерау-Левенштейна.

Задачи работы

1. Дать математическое описание расстояний
2. Описать алгоритмы
3. Реализовать алгоритмы
4. Провести тестирование
5. Осуществить замеры процессорного времени работы алгоритмов

1.2 Описание алгоритмов

1.2.1 Расстояние Левенштейна

Расстояние Левенштейна определяет минимальное количество операций, необходимых для превращения одной строки в другую, среди которых:

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

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

1.2.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояние Левенштейна. К исходному набору возможных операций добавляется операция транспозиции (T - transpose), или перестановка двух соседних символов.

При вычислении расстояния Дамерау-Левенштейна в рекурретную формулу вносится дополнительное соотношение в минимум:

$$D(S_1[i-2], S_2[j-2]) + 1 \tag{1.2}$$

Соотношение (1.2) вносится в выражение только при выполнении следующих условий:

$$\begin{cases} i > 1,\ \ j > 1 \\ S_1[i] = S_2[j-1] \\ S_1[i-1] = S_2[j] \end{cases} \tag{1.3}$$

Таким образом получаем следующую рекурретную формулу:

$$D(S_1[i], S_2[j]) = \begin{cases} i\ if\ \ j = 0 \\ j\ if\ \ i = 0 \\ min\begin{cases} D(S_1[i-1], S_2[j]+1) \\ D(S_1[i], S_2[j-1]+1) \\ D(S_1[i-1], S_2[j-1]) + \begin{cases} 1, if\ \ S_1[i] \neq S_2[j] \\ 0, else \end{cases} \\ D(S_1[i-2], S_2[j-2]) + 1 \end{cases} & \text{if (1.3)} \\ min\begin{cases} D(S_1[i-1], S_2[j]+1) \\ D(S_1[i], S_2[j-1]+1) \\ D(S_1[i-1], S_2[j-1] + \begin{cases} 1, if\ \ S_1[i] \neq S_2[j] \\ 0, else \end{cases} \end{cases} & \text{else} \end{cases} \tag{1.4}$$

2. Конструкторская часть

2.1 Схемы алгоритмов

На рисунках 2.1 - 2.5 представлены схемы алгоритмов реализаций алгоритмов поиск расстояния между строками.

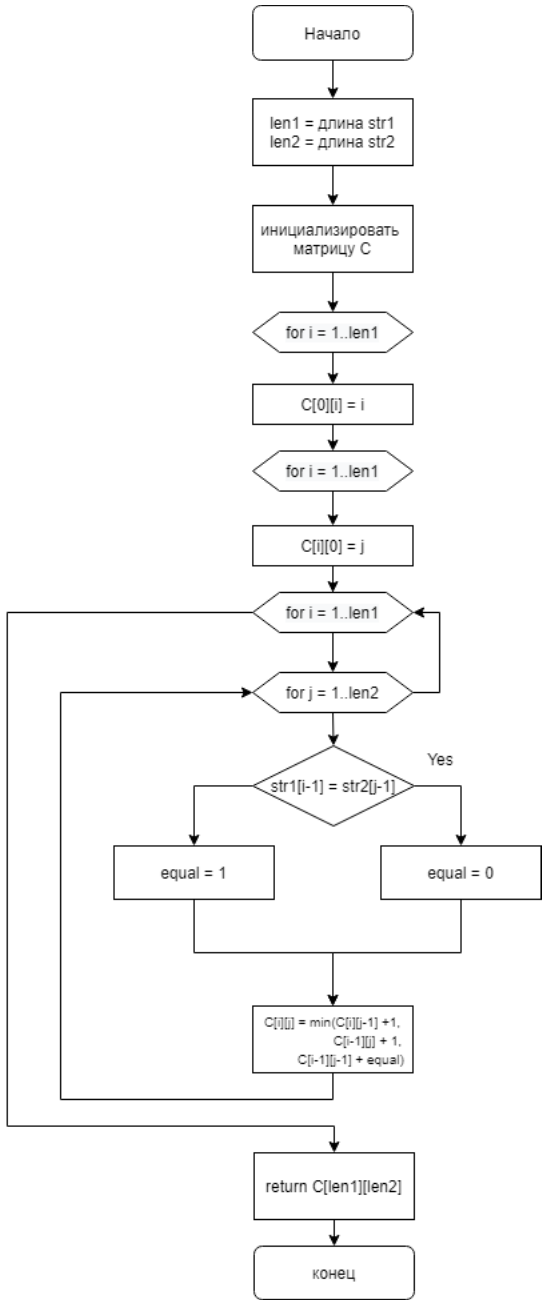

Рис. 2.1: Матричная реализация алгоритма Левенштейна

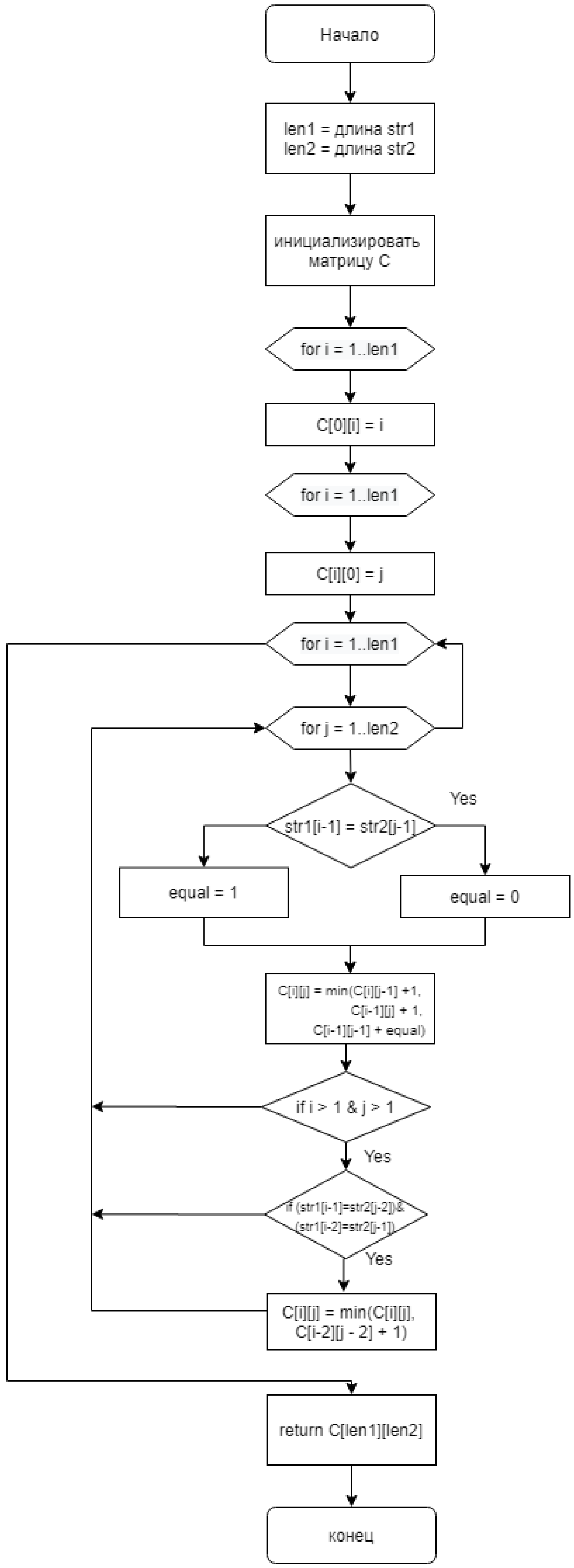

Рис. 2.2: Матричная реализация алгоритма Дамерау-Левенштейна

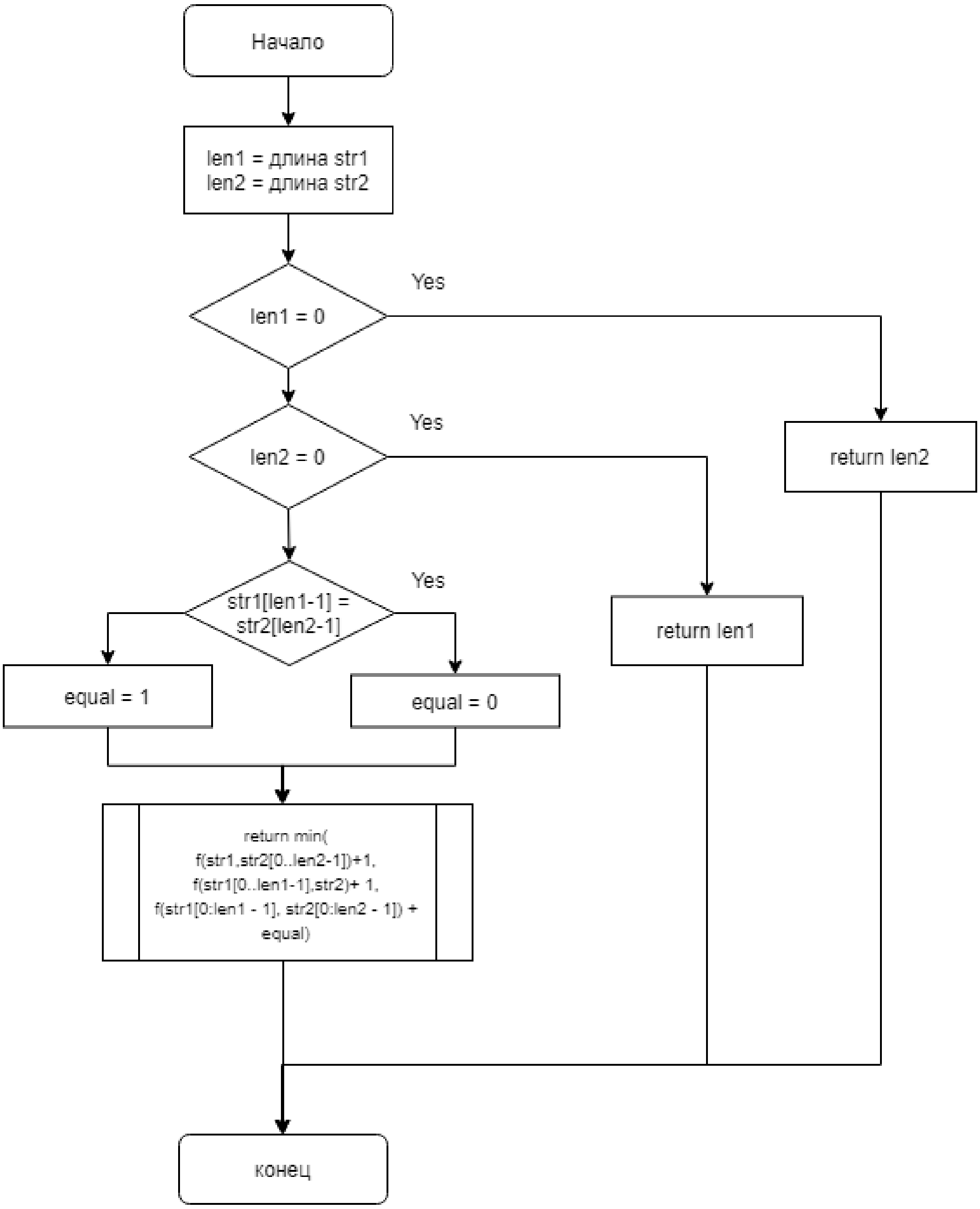

Рис. 2.3: Рекурсивная реализация алгоритма Левенштейна

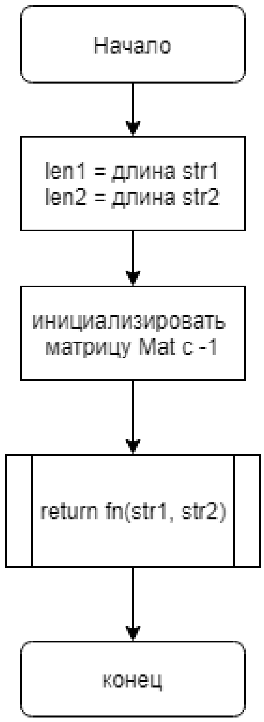

Рис. 2.4: Рекурсивная реализация алгоритма Левенштейна с заполнением матрицу

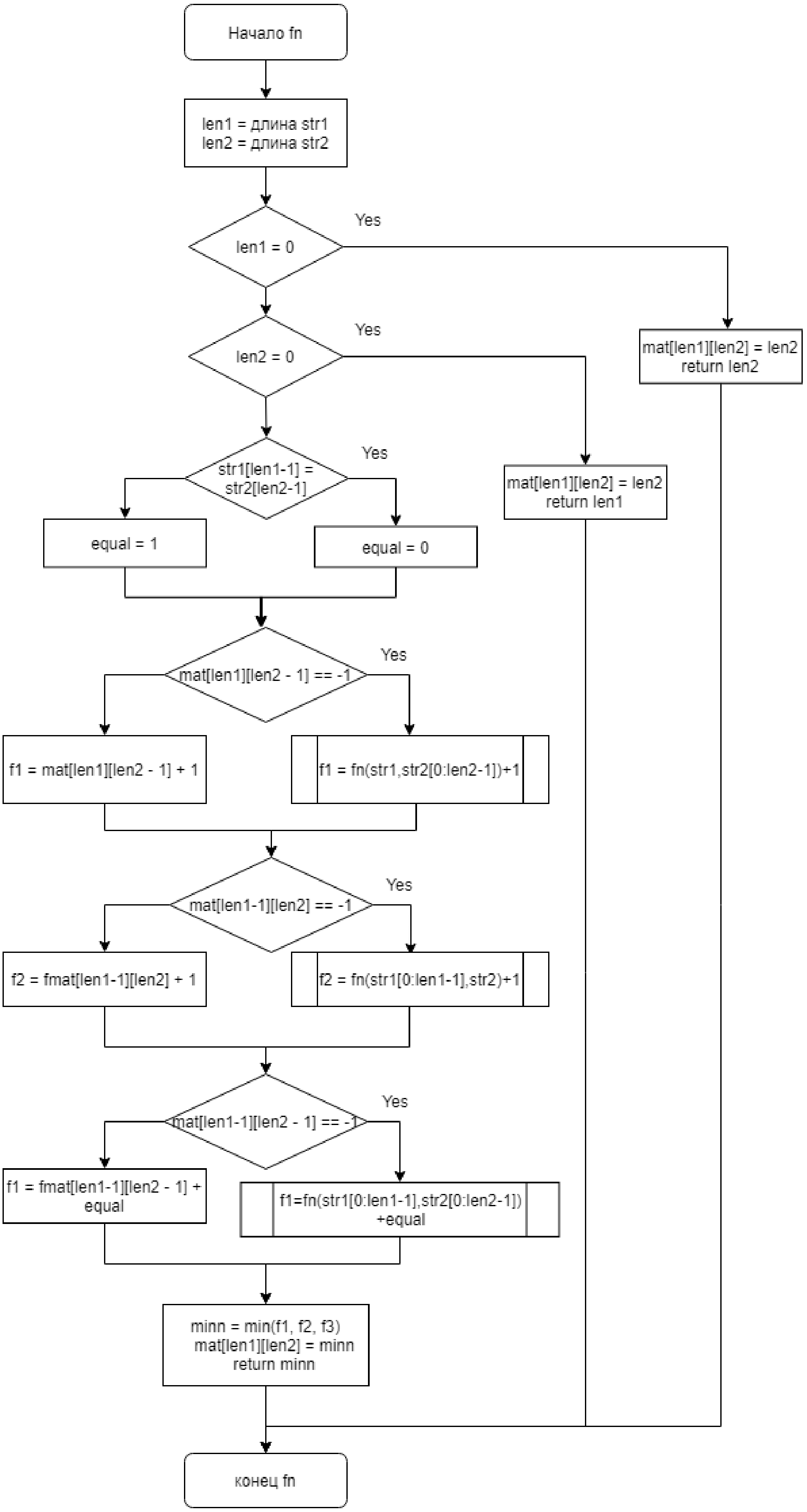

Рис. 2.5: Рекурсивная реализация алгоритма Левенштейна с заполнением матрицу

3. Технологическая часть

3.1 Средства реализации

Для реализации программы был использован язык Python. Для замера процессорного времени была использована функция time() из библиотеки time.

3.2 Реализации алгоритмов

На листингах 3.1 - 3.4 представлены коды реализации алгоритмов поиска расстояния.

Листинг 3.1: Матричная реализация алгоритма Левенштейна

```

def levenshtein(str1, str2):

    len1 = len(str1)
    len2 = len(str2)

    C = [[0 for i in range(len2 + 1)] for j in range(len1 + 1)]

    for i in range(0, len2 + 1):
        C[0][i] = i

    for i in range(0, len1 + 1):
        C[i][0] = i

    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):
            if (str1[i - 1] == str2[j - 1]):
                equal = 0
            else:
                equal = 1

            C[i][j] = min(C[i][j-1] + 1,
                          C[i-1][j] + 1,
                          C[i-1][j-1] + equal)
    #matrix_print(str1, str2, C)
    return C[len1][len2]
```

Листинг 3.2: Матричная реализация алгоритма Дамерау-Левенштейна

```

def damerau_levenshtein(str1, str2):
    len1 = len(str1)
    len2 = len(str2)
```

```

    C = [[0 for i in range(len2 + 1)] for j in range(len1 + 1)]

    for i in range(0, len2 + 1):
        C[0][i] = i

    for i in range(0, len1 + 1):
        C[i][0] = i

    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):
            if (str1[i - 1] == str2[j - 1]):
                equal = 0
            else:
                equal = 1

            C[i][j] = min(C[i][j-1] + 1,
                          C[i-1][j] + 1,
                          C[i-1][j-1] + equal)

            if (i > 1 and j > 1 and str1[i - 1] == str2[j - 2] and str1[i - 2] == str2[j - 1]):
                C[i][j] = min(C[i][j], C[i-2][j - 2] + 1)

    #matrix_print(str1, str2, C)
    return C[len1][len2]
```

Листинг 3.3: Рекурсивная реализация алгоритма Левенштейна

```

def levenshtein_recursive(str1, str2):
    len1 = len(str1)
    len2 = len(str2)

    if (len1 == 0):
        return len2
    if (len2 == 0):
        return len1

    is_equal = 1
    if (str1[len1 - 1] == str2[len2 - 1]):
        is_equal = 0

    return min(levenshtein_recursive(str1, str2[0 : len2 - 1]) + 1,
               levenshtein_recursive(str1[0 : len1 - 1], str2) + 1,
               levenshtein_recursive(str1[0:len1 - 1], str2[0:len2 - 1]) + is_equal)
```

Листинг 3.4: Рекурсивная реализация с заполнением матрицу алгоритма Левенштейна

```
    def levenshtein_recursive_table(str1, str2):
        res = levenshtein_recursive_tab(str1, str2)
        delattr(levenshtein_recursive_tab, "mat")
    return res

    def levenshtein_recursive_tab(str1, str2):
```

```
    fn = levenshtein_recursive_tab
    len1 = len(str1)
    len2 = len(str2)
    if not hasattr(fn, "mat"):
      fn.mat = [[-1 for i in range(len2 + 1)] for i in range(len1 + 1)]

    if (len1 == 0):
      fn.mat[len1][len2] = len2
      return len2

    if (len2 == 0):
      fn.mat[len1][len2] = len1
      return len1

    f1, f2, f3 = 0, 0, 0
    is_equal = 1

    if (str1[len1 - 1] == str2[len2 - 1]):
      is_equal = 0

    if (fn.mat[len1][len2 - 1] == -1):
      f1 = fn(str1, str2[0 : len2 - 1]) + 1
    else:
      f1 = fn.mat[len1][len2 - 1] + 1

    if (fn.mat[len1 - 1][len2] == -1):
      f2 = fn(str1[0 : len1 - 1], str2) + 1
    else:
      f2 = fn.mat[len1 - 1][len2] + 1

    if (fn.mat[len1 - 1][len2 - 1] == -1):
      f3 = fn(str1[0:len1 - 1], str2[0:len2 - 1]) + is_equal
    else:
      f3 = fn.mat[len1 - 1][len2 - 1] + is_equal

    minn = min(f1, f2, f3)
    fn.mat[len1][len2] = minn
    return minn
```

3.3 Тесты

Для проверки корректности работы были подготовлены функциональные тесты, представленные в таблице 3.1. В данной таблице λ означает пустую строку, а числа в столбцах "Ожидание"и "Результат"соответствуют результатам работы алгоритмов в следующем порядке:

1. Расстояние Левенштейна.
2. Расстояние Дамерау-Левенштейна.

Таблица 3.1: Функциональные тесты

Строка 1	**Строка 2**	**Ожидание**	**Результат**
λ	λ	0 0	0 0
λ	a	1 1	1 1
a	λ	1 1	1 1
a	a	0 0	0 0
a	b	1 1	1 1
abc	acb	2 1	2 1
1234	567	4 4	4 4
human	cat	4 4	4 4

В результате проверки все реализации алгоритмов прошли все поставленные функциональные тесты.

4. Исследованая часть

4.1 Сравнение работы алгоритмов

Таблица 4.1: Время работы матричных реализаций алгоритмов (ms) процессора

Длина слова	Алг. Лев-на	Алг. Дамерау Лев-на	Алг. Лев-на (Рекурсивной с мат)
0	0.0	0.0	0.0
50	2.8449297	3.4764767	7.5498819
100	10.2889538	15.2704477	27.7799129
200	51.2362719	60.3877068	111.9863033
300	104.4904947	124.054718	239.5556688

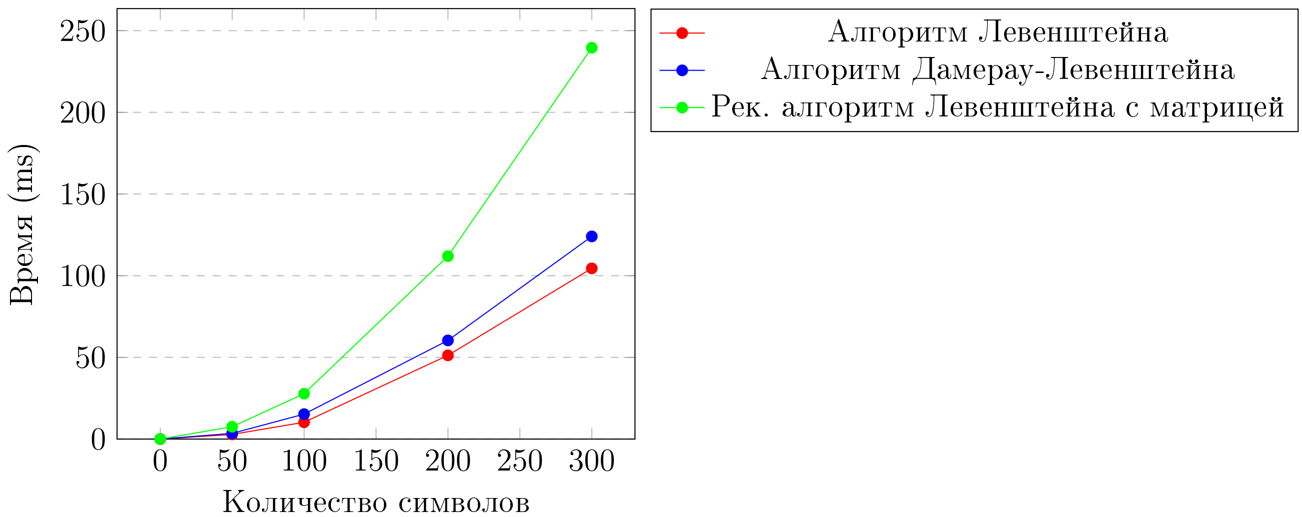

Рис. 4.1: График времени работы матричных реализаций алгоритмов Левенштейна и Дамерау-Левенштейна

Алгоритм Левенштейна выигрывает по времени. Алгоритм Дамерау-Левенштейна выполняется дольше за счёт добавления небольшого количества операций. Рекурсивный алгоритм Левенштейна с заполнением матрицу самый долгии.

4.2 Сравнение работы реализаций алгоритма Левенштейна

Время выполнения рекурсивной реализации алгоритма резко возрастает с увеличением длины слов. Можно сделать вывод о том, что матричная реализация алгоритма значи-

Таблица 4.2: Время (ms) работы реализаций алгоритма Левенштейна процессора

Длина слова	Матричная реализация	Рекурсивная реализация
1	0.00947	0.00219
2	0.01944	0.01216
3	0.02293	0.06726
4	0.03590	0.34368
5	0.04488	2.12764
6	0.06332	9.37488
7	0.09275	52.40877
8	0.10023	295.03857

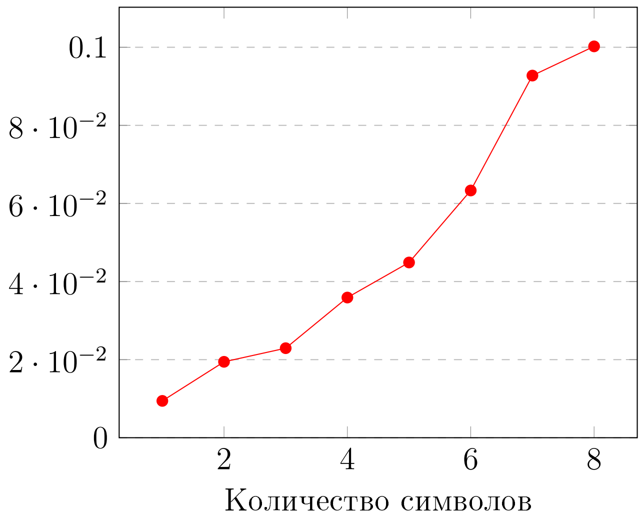

Рис. 4.2: График времени работы матричной реализации алгоритма Левенштейна

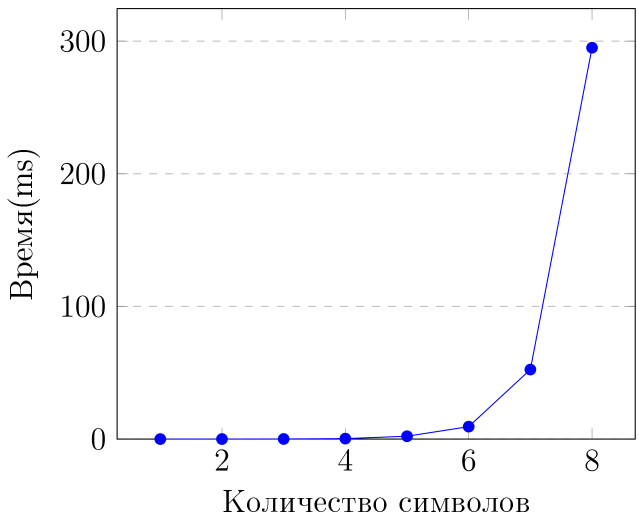

Рис. 4.3: График времени работы рекурсивной реализации алгоритма Левенштейна

тельно эффективнее рекурсивной при любой длине слова.

4.3 Вывод

Заключение

1. Дано математическое описание расстояний
2. Описаны алгоритмы
3. Реализованы алгоритмы
4. Провести тестирование
5. Осущественны замеры процессорного времени работы алгоритмов