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»

Лабораторная работа №1
по дисциплине "Анализ алгоритмов"

Тема Алгоритмы Левенштейна и Дамерау-Левенштейна

Студент Тузов Даниил Александрович

Группа ИУ7-52Б

Преподаватель Строганов Дмитрий Владимирович

Москва, 2024 г.

Содержание

Введение

В первой лабораторной работе по анализу алгоритмов рассматривается понятие расстояния Левенштейна. Это самое расстояние показывает минимальное количество операций вставки, удаления и замены, которое необходимо для преобразования первой строки во вторую.

Целью лабораторной работы является изучение алгоритмов нахождения расстояния Левенштейна. Для достижения поставленной цели небходимо решить следующие задачи:

- изучить понятие расстояния Левенштейна и расстояние Дамерау-Левенштейна
- реализовать алгоритмы для нахождения искомого расстояния:
  - рекурсивный алгоритм
  - рекурсивный алгоритм с кэшированием
  - нерекурсивный алгоритм, основанный на динамическом программировании
  - алгоритм нахождения расстояния Дамерау-Левенштейна
- сравнить алгоритмы по затраченному процессорному времени и памяти
- обосновать полученные результаты

1 Аналитическая часть

В этой части рассматриваются теоретические аспекты понятия расстояния между строками на примере алгоритмов Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [4] – минимальное количество операций вставки, удаления и замены, необходимых для преобразования одной строки в другую.

Расстояние Левенштейна $\mathrm{D}(S_1,\, S_2)$ для строк S_1 и S_2 можно найти по формуле 1:

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

где i и j – индексы в строках S_1 и S_2.

1.2 Возможные реализации

Поскольку алгоритм Левенштейна задается рекурентной формулой, то одной из возможных реализаций является рекурсия. Однако при рекурсивной реализации возможны множественные рекурсивные вызовы с одинаковыми параметрами. Это можно предотвратить, если ввести кэш-матрицу, в которую будут заносится уже вычисленные значения. Затем они используются при повторяющихся рекурсивных вызовах. Возможна также нерекурсивная реализация с помощью динамического программирования. Все алгоритмы подробнее будут описаны в следующей части.

1.3 Расстояние Дамерау-Левенштейна

Для вычисления расстояния Дамерау-Левенштейна [4] необходимо дополнительно учитывать операцию обмена двух соседних символов.

Это расстояние D(S_1, S_2) для строк S_1 и S_2 можно найти по формуле 2:

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

где i и j – индексы в строках S_1 и S_2.

1.4 Вывод

В этой части были рассмотрены теоретические аспекты алгоритмов нахождения расстояния между строками – алгоритмов Левенштейна и Дамерау Левенштейна. Алгоритмы в этом разделе представлены рекурентными формулами. В последующих разделах будет рассматриваться 4 реализации алгоритмов: рекурсивный алгоритм Левенштейна, рекурсивный алгоритм Левенштейна с кэшированием данных, алгоритм Левенштейна, основанный на динамическом программировании и алгоритм Дамерау-Левенштейна, основанный на динамическом программировании.

2 Конструкторская часть

В этой части представлены схемы алгоритмов.

2.1 Схемы алгоритмов

На вход алгоритмам подаются строки S_1 и S_2 (в реализации с кэширование подается еще и кэш-матрица), на выходе единственное число – искомое расстояние.

На рисунках 1 - 5 приведены схемы алгоритмов.

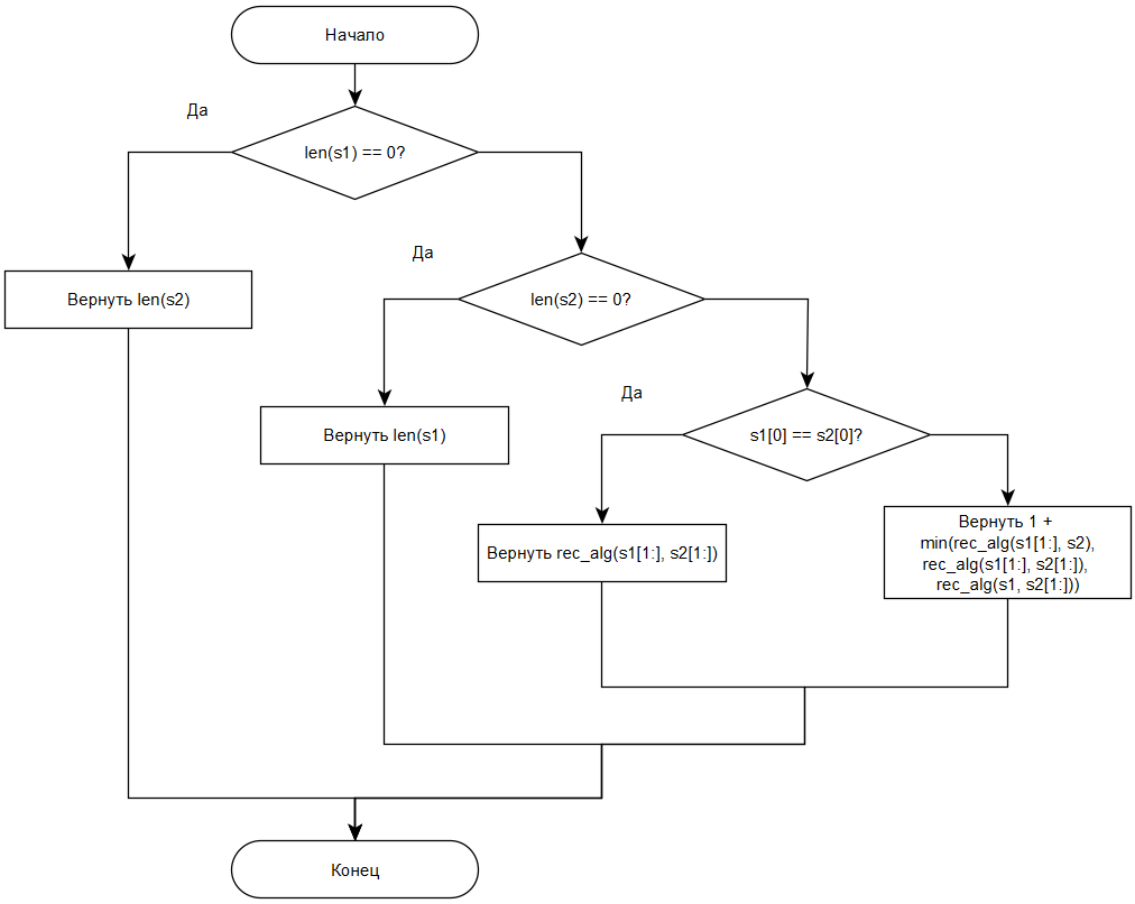

Рис. 1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

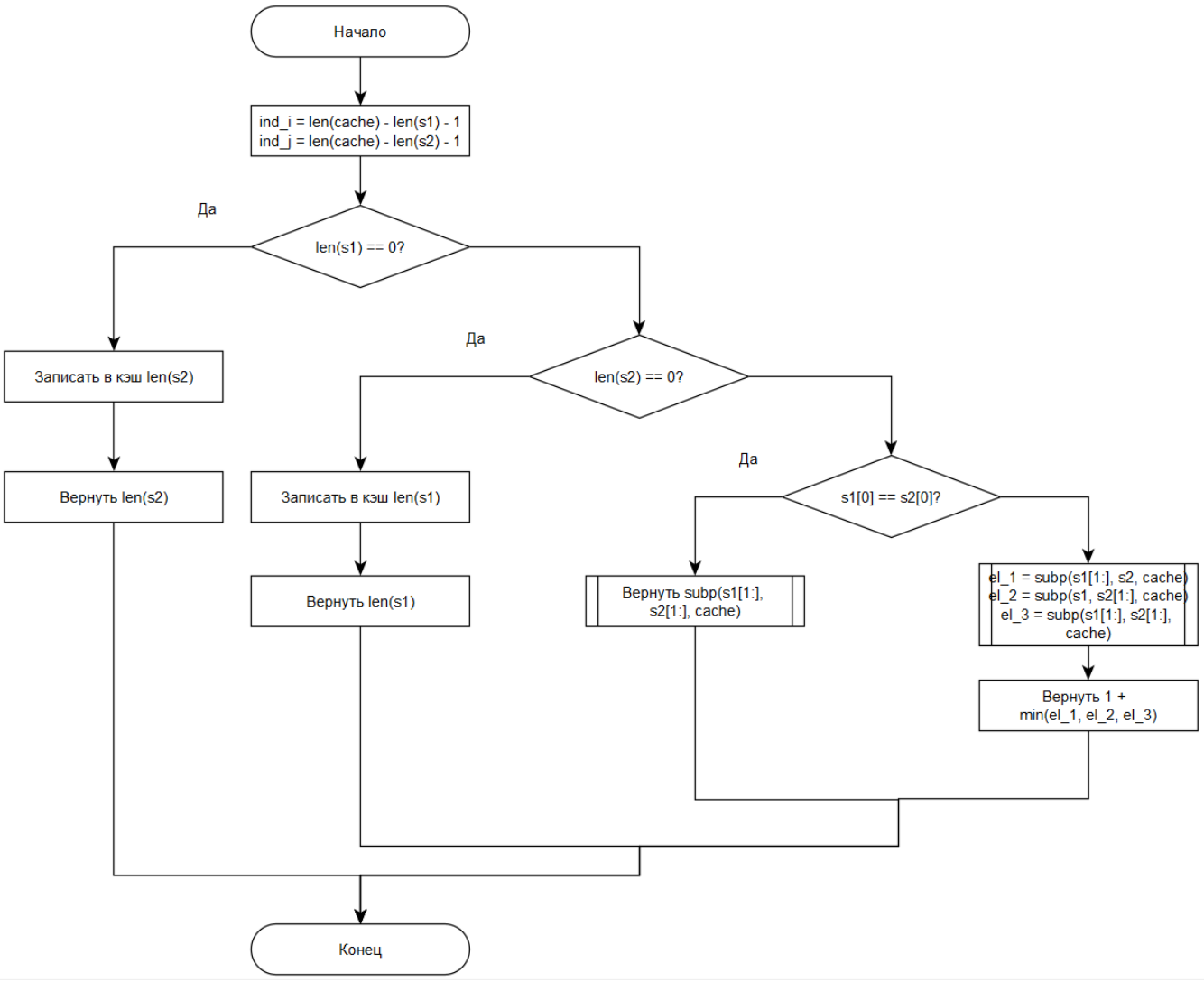

Рис. 2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшированием

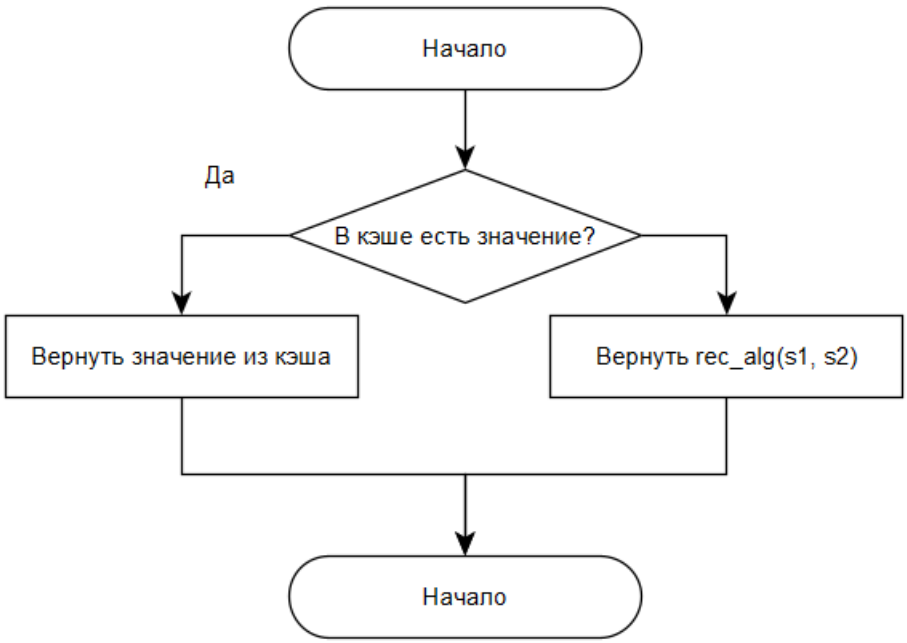

Рис. 3: Схема подпрограммы subp, вызываемой в алгоритме с кэшированием

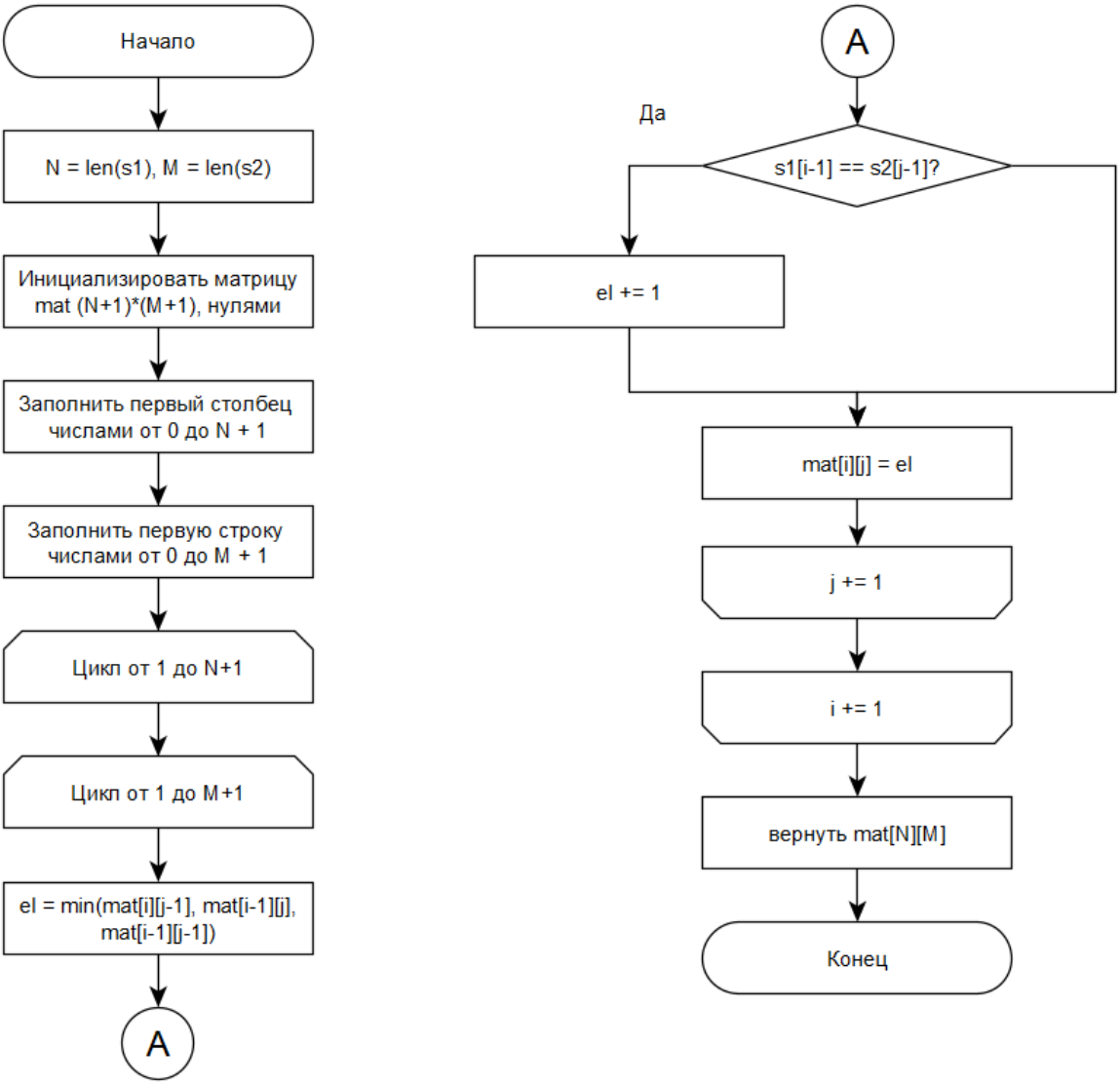

Рис. 4: Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

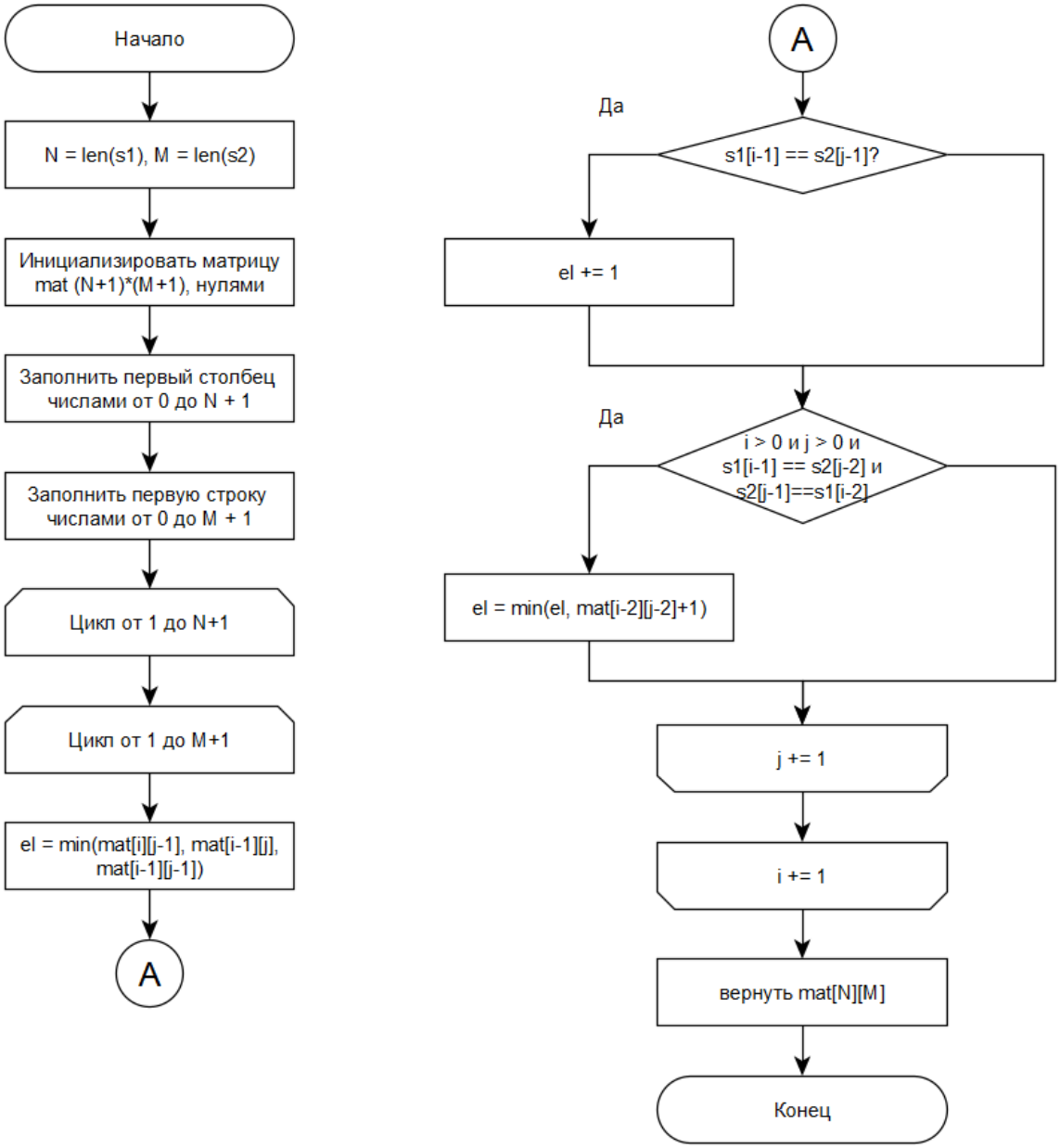

Рис. 5: Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

2.2 Вывод

В этом разделе на основе теоретических аспектов были представлены схемы рекурсивного алгоритма, рекурсивного алгоритма с кэшированием, нерекурсивного алгоритма, основанного на динамическом программировании, а также схема алгоритма Дамерау-Левенштейна.

3 Технологическая часть

В этом разделе обоснованы средства реализации, а так же представлены листинги кодов алгоритмов и функциональные тесты.

3.1 Средства реализации

Для реализации алгоритмов в этой лабораторной работы был выбран язык *Python* [2], потому что он прост в работе, а также в нем нет автоматического сборщика мусора. Время работы было замерено с помощью функции *process_time* [1] из библиотеки *time*. [3]

3.2 Листинги кодов алгоритмов

В листингах 1 - 4 представлены коды написанных алгоритмов.

Листинг 1: Рекурсивный алгоритм

```
1  def lev_recursion(s1, s2, output = False):
2      len1 = len(s1)
3      len2 = len(s2)
4
5      if len1 == 0 or len2 == 0:
6          return abs(len1 - len2)
7
8      m = 0 if s1[-1] == s2[-1] else 1
9
10     return min(lev_recursion(s1,      s2[:-1]) + 1,
11                lev_recursion(s1[:-1], s2     ) + 1,
12                lev_recursion(s1[:-1], s2[:-1]) + m)
```

Листинг 2: Рекурсивный алгоритм с кэшированием

```
def cache_alg(s1, s2, cache_mat):
    ind_i = len(cache_mat) - len(s1) - 1
    ind_j = len(cache_mat[0]) - len(s2) - 1
    if len(s1) == 0:
        cache_mat[ind_i][ind_j] = len(s2)
        return len(s2)
    elif len(s2) == 0:
        cache_mat[ind_i][ind_j] = len(s1)
        return len(s1)
    elif s1[0] == s2[0]:
        el = cache_mat[ind_i + 1][ind_j + 1]
        return el if el != -1 else cache_alg(s1[1:], s2[1:], cache_mat)

    el_1 = cache_mat[ind_i + 1][ind_j]
    el_2 = cache_mat[ind_i][ind_j + 1]
    el_3 = cache_mat[ind_i + 1][ind_j + 1]
    return 1 + min(el_1 if el_1 != -1 else cache_alg(s1[1:], s2, cache_mat),
                   el_2 if el_2 != -1 else cache_alg(s1, s2[1:], cache_mat),
                   el_3 if el_3 != -1 else cache_alg(s1[1:], s2[1:], cache_mat)
                      )
```

Листинг 3: Нерекурсивный алгоритм

```
def DP_alg(s1, s2):
    n = len(s1)
    m = len(s2)
    matrix = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(m + 1):
        matrix[0][i] = i

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

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            el = min(matrix[i - 1][j],
                     matrix[i - 1][j - 1],
                     matrix[i][j - 1])
            if s1[i - 1] != s2[j - 1]:
                el += 1
            matrix[i][j] = el

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

Листинг 4: Нерекурсивный алгоритм Дамерау-Левенштейна

```
def damerau_alg(s1, s2):
    n = len(s1)
    m = len(s2)
    matrix = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(m + 1):
        matrix[0][i] = i

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

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            el = min(matrix[i - 1][j],
                     matrix[i - 1][j - 1],
                     matrix[i][j - 1])
            if s1[i - 1] != s2[j - 1]:
                el += 1
            if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2] == s2[j
                  - 1]:
                el = min(el, matrix[i - 2][j - 2] + 1)
            matrix[i][j] = el

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

3.3 Функциональные тесты

В таблице 1 приведены функциональные тесты, на которых тестировалась программа.

Таблица 1: Функциональные тесты

S_1	S_2	Левенштейн	Дамерау-Левенштейн
а	[]	1	1
[]	[]	0	0
а	б	1	1
а	аб	1	1
река	мука	2	2
реак	мука	4	3
морковка	сосиска	5	5
река	мука	2	2
мамам	мамам	0	0

3.4 Вывод

В этом разделе на основе схем алгоритмов были написаны и представлены листинги кода. Помимо алгоритмов представлены функциональные тесты, на которых был протестирован каждый алгоритм.

4 Исследовательская часть

В этом разделе приведено сравнение по времени и по памяти написанных алгоритмов.

4.1 Технические характеристики ЭВМ

Все замеры проводились на ЭВМ, характеристики которой приведены ниже:

- Процессор – 12th Gen Intel(R) Core(TM) i5-12450H 2.00 GHz
- Оперативная память – 16,0 ГБ
- Тип системы – 64-разрядная операционная система, процессор x64
- Операционная система – Windows 11
- Версия ОС – 23H2

4.2 Сравнение по времени

Время выполнения работы алгоритмов представлено в секундах. Проводилось 150 замеров.

Таблица 2: Сравнение алгоритмов по времени выполнения

Размер строк	Рекурсивный	С кэшем	Нерекурсивный	Дамерау
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0.001	0	0
6	0.02	0.001	0	0
7	0.01	0.005	0	0
8	0.03	0.03	0	0
9	0.145	0.113	0	0

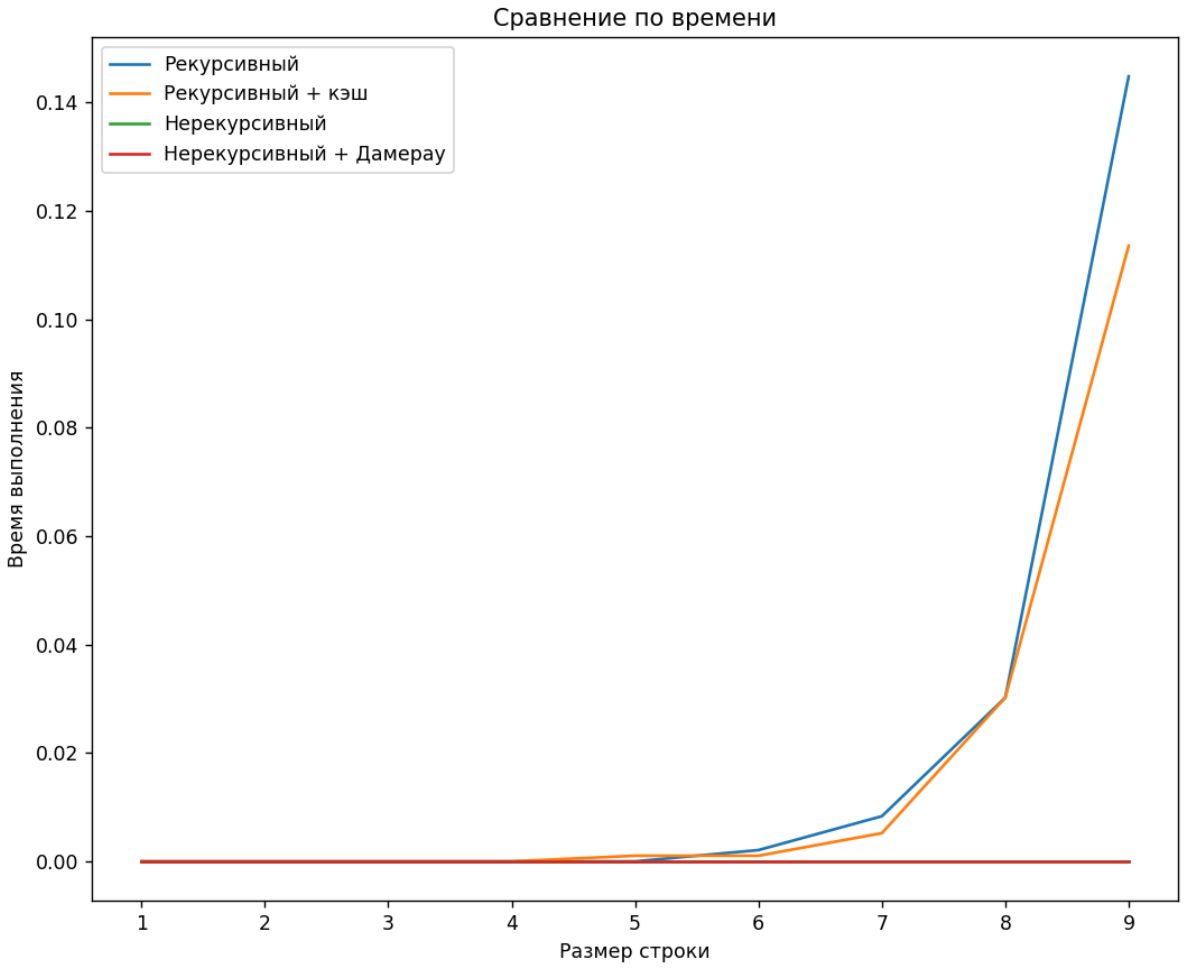

Рис. 6: Сравнение алгоритмов по времени выполнения

Таблица 3: Сравнение нерекурсивных алгоритмов по времени выполнения

Размер строк	Левенштейн	Дамерау-Левенштейн
50	0	0
100	0.001	0.001
150	0.002	0.001
200	0.002	0.003
250	0.005	0.007
300	0.008	0.009
350	0.015	0.016
400	0.016	0.022

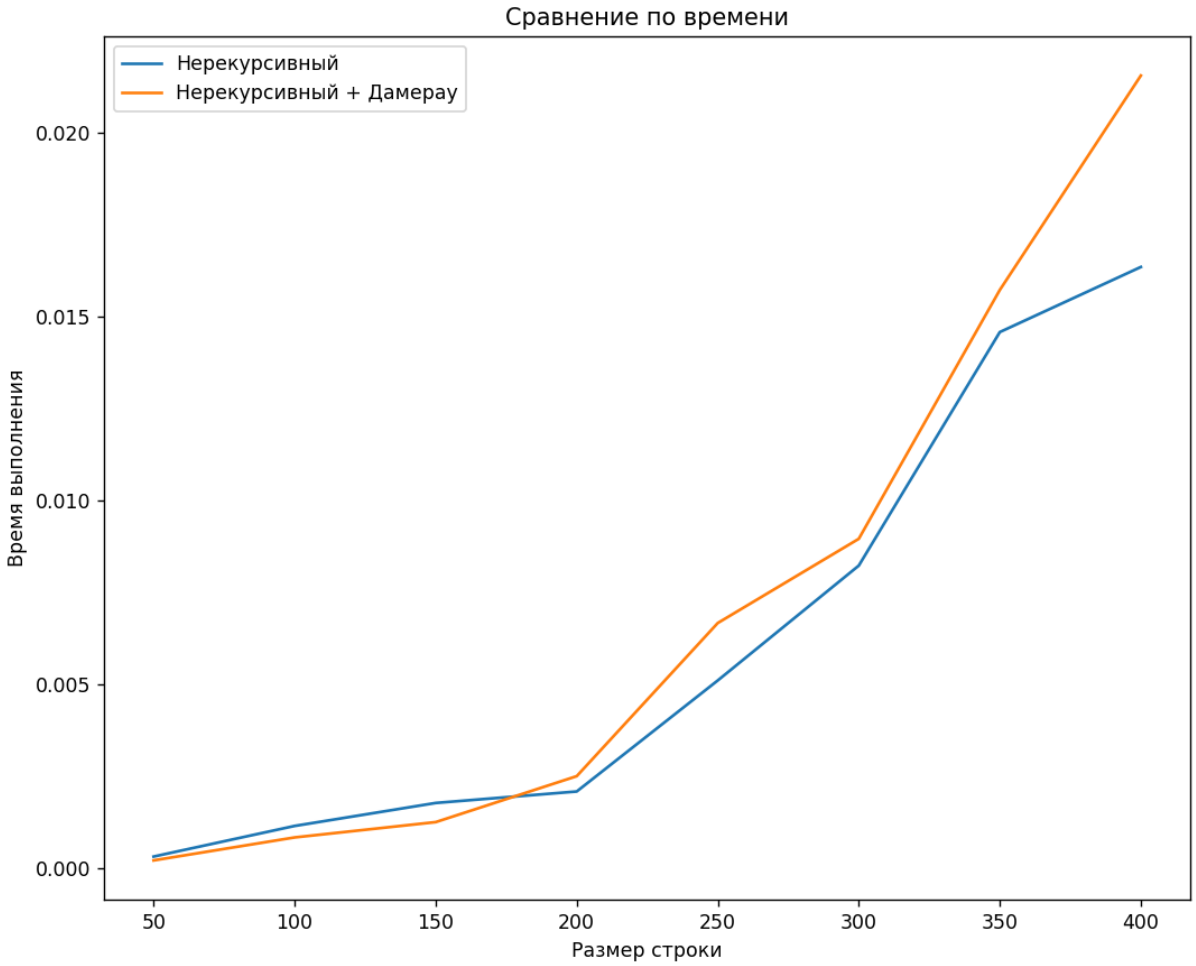

Рис. 7: Сравнение нерекурсивных алгоритмов по времени выполнения

4.3 Сравнение по памяти

В таблице 4 приведено сравнение различных реализаций по памяти. Нерекурсивные алгоритмы Левенштейна и Дамерау-Левенштейна эквивалентны по расходу памяти.

Таблица 4: Сравнение по памяти двух строк длины N и M соответственно

N	M	Рекурсивно, байт	Рекурсивно + кэш, байт	Нерекурсивно, байт
10	10	20	141	141
100	100	200	10401	10401
200	200	400	40801	40801
500	500	1000	252001	252001
1000	1000	2000	1004001	1004001
n	m	n + m	n * m + n + m	n * m + n + m

4.4 Вывод

Рекурсивные алгоритмы работают намного дольше, чем нерекурсивные алгоритмы, в силу большей сложности. Алгоритм с кэшем дает небольшое преимущество перед простым рекурсивным алгоритмом, так как выполняет меньше рекурсивных вызовов. Простой рекурсивный алгоритм расходует меньше памяти, чем все остальные, потому что не хранит дополнительную матрицу. Нерекурсивный алгоритм Дамерау-Левенштейна медленнее, чем нерекурсивный алгоритм Левенштейна, в силу появления дополнительного сравнения на возможность обмена двух символов строки.

Заключение

В ходе выполнения лабораторной работы поставленная цель была достигнута, а также были решены следующие задачи:

- изучены алгоритмы Левенштейна и Дамерау-Левенштейна
- реализованы алгоритмы на языке *Python*
- проведено сравнение алгоритмов по времени и памяти
- описаны и обоснованы полученные результаты

Список литературы

[1] process_time – https://docs.python.org/3/library/time.html#functions

[2] Welcome to Python – https://www.python.org

[3] Библиотека time – https://docs.python.org/3/library/time.html

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