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 Динамическое программирование

Студент Малышев И.Н.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватель Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение и исследование особенностей задач динамического программирования.

В качестве исследуемых задач, взяты нахождение расстояний Левенштейна и Дамерау – Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной строки в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены. Широко используется в теории информации и компьютерной лингвистике.

Расстояние Левенштейна **Расстояние Левенштейна [1]** и его обобщения активно применяются для:

- исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);
- сравнения генов в биоинформатике.

Расстояние Дамерау – Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна, так как к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Задачами данной лабораторной являются:

1) выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;

2) создать ПО, реализующее нерекурсивный метод поиска расстояния Левенштейна, нерекурсивный метод поиска Дамерау – Левенштейна, рекурсивный метод поиска Дамерау – Левенштейна, рекурсивный с кешированием метод поиска Дамерау – Левенштейна;

3) провести анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики;

4) создать отчёт, содержащий:

   (a) схемы выбранных алгоритмов;

   (b) обоснование выбора технических средств;

   (c) результаты замеров времени и памяти реализации;

   (d) обобщающий вывод.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна и их практическое применение.

1.1 Нерекурсивный метод нахождения расстояния Левенштейна

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Каждая операция имеет свою цену (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначения операций:

- D (англ. delete) — удаление ($w(a, \lambda) = 1$);
- I (англ. insert) — вставка ($w(\lambda, b) = 1$);
- R (англ. replace) — замена ($w(a, b) = 1,\ a \neq b$);
- M (англ. match) - совпадение ($w(a, a) = 0$).

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0 & \text{, если i} = 0, \text{ j} = 0, \\ i & \text{, если j} = 0, \text{ i} > 0, \\ j & \text{, если i} = 0, \text{ j} > 0, \\ \min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \\ \quad D(i-1,j-1)+m(a[i],b[j]) & \\ \} & \text{, если i} > 0, \text{ j} > 0 \end{cases} \tag{1.1}$$

Функция 1.2 является вспомогательной для функции 1.1 и определена как:

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b}, \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

Нерекурсивный алгоритм реализует формулу 1.1. Функция D составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. 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 на 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.

При больших i, j прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения, так как множество промежуточных значения $D(i, j)$ вычисляются не по одному разу. Для оптимизации нахождения можно использовать матрицу для хранения соответствующих промежуточных значений.

Матрица размером $(length(S1) + 1)$x$((length(S2) + 1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i, j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой 1.3.

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j]) \end{cases} . \quad (1.3)$$

Функция 1.4 является вспомогательной для функции 1.3 и определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases} . \quad (1.4)$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.2 Нерекурсивный метод поиска Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна [2] - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Дамерау – Левенштейна может быть найдено по формуле

1.5, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{если } \min(i,j) = 0, \\ \min\{ \\ \quad d_{a,b}(i,j-1) + 1, \\ \quad d_{a,b}(i-1,j) + 1, \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, \text{если } i,j > 1; \\ a[i] = b[j-1]; \\ b[j] = a[i-1] \end{cases} \\ \quad \infty, \text{иначе} \\ \} \end{cases}, \tag{1.5}$$

Формула выводится по тем же соображениям, что и формула (1.1).

1.3 Рекурсивный метод поиска Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна может быть найдено по формуле 1.6, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{если } \min(i,j) = 0, \\ \min\{ \\ \quad d_{a,b}(i,j-1) + 1, \\ \quad d_{a,b}(i-1,j) + 1, \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, \text{если } i,j > 1; \\ \qquad a[i] = b[j-1]; \\ \qquad b[j] = a[i-1] \\ \qquad \infty, \text{иначе} \end{cases} \\ \} \end{cases}, \tag{1.6}$$

1.4 Рекурсивный с кешированием метод поиска Дамерау – Левенштейна

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием кеша. В качестве кеша используется матрица. 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.

Вывод

Формулы Левенштейна и Дамерау – Левенштейна для рассчета расстояния между строками задаются рекурентно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна.

2.1 Требования к вводу

Программа на взод получает две строки, символы нижнего и верхнего регистра в которых считаются различными.

2.2 Требования к программе

1. Программа не должна аварийно завершаться, если пользователь вводит некорректные входные данные.
2. Для одинаковых входных данных, программа должна выдавать одинаковый ответ.
3. На выход программа должна вывести число - расстояние Левенштейна (Дамерау – Левенштейна).

2.3 Разработка алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема нерекурсивного алгоритма нахождения расстояния Левенштейна.

2.4 Разработка алгоритма нахождения расстояния Дамерау – Левенштейна

На рисунке 2.2 приведена схема нерекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

На рисунке 2.3 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

На рисунке 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с использованием кеша в виде матрицы.

Вывод

Перечислены требования к вводу и программе, а также на основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

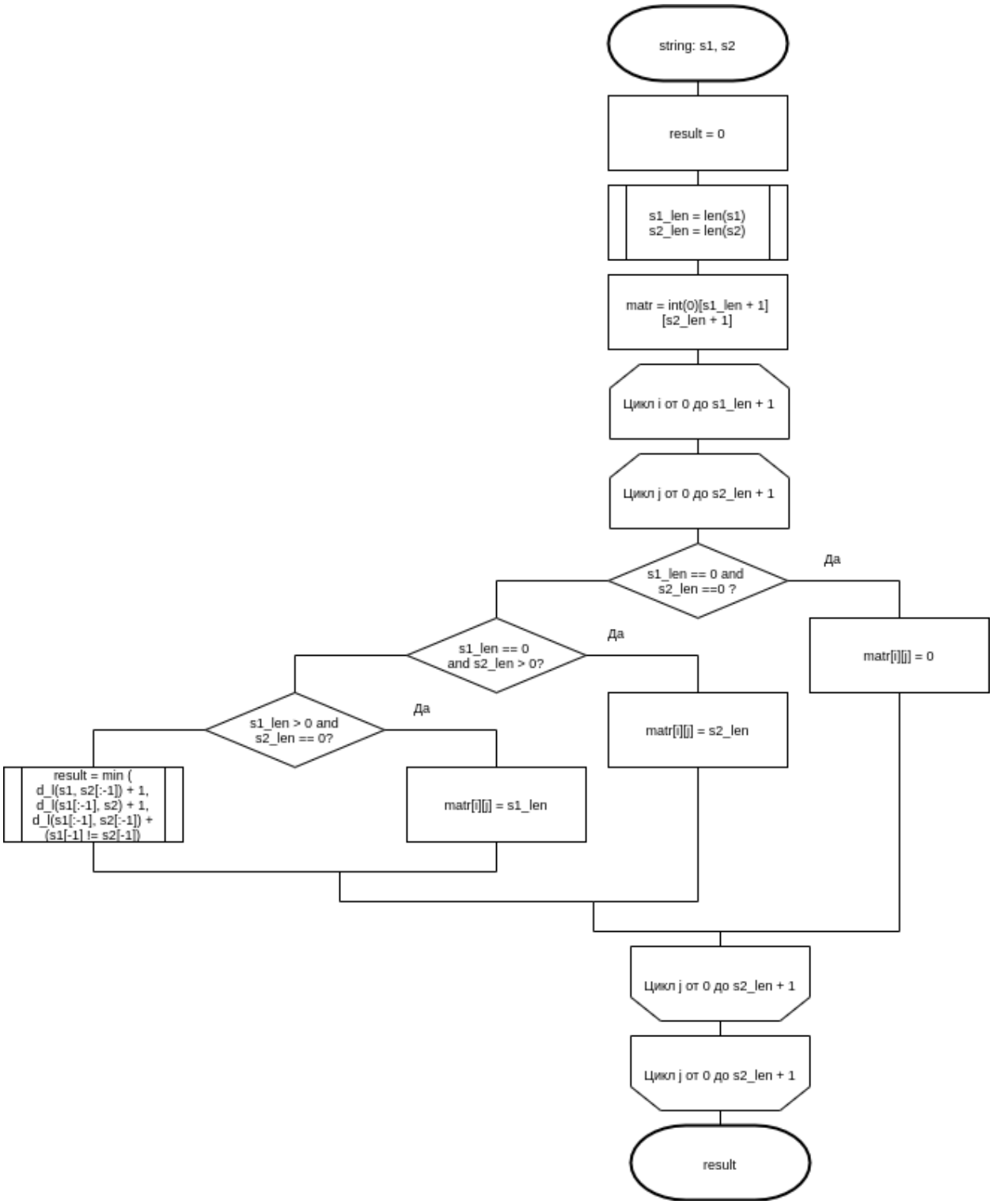

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

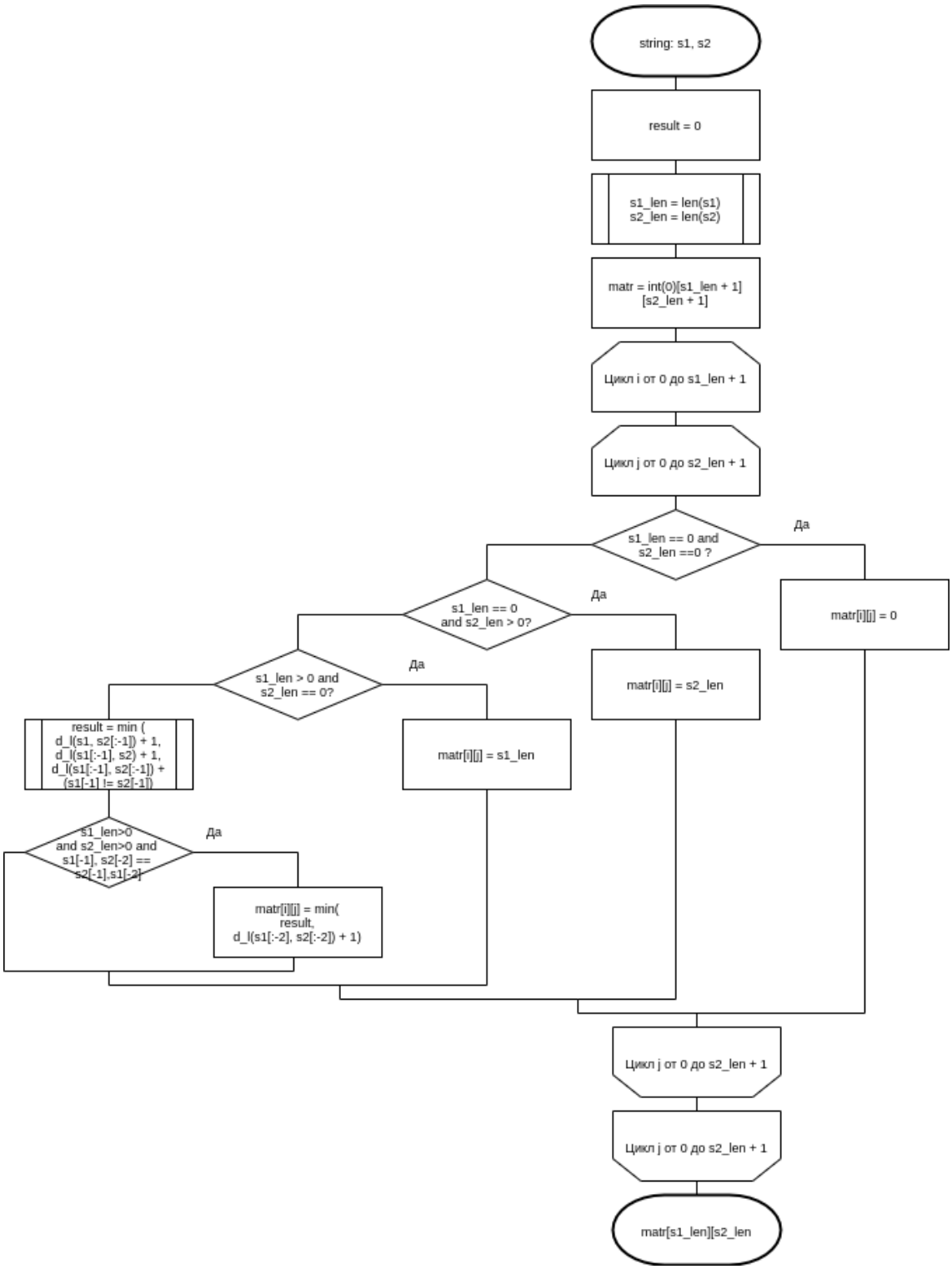

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна

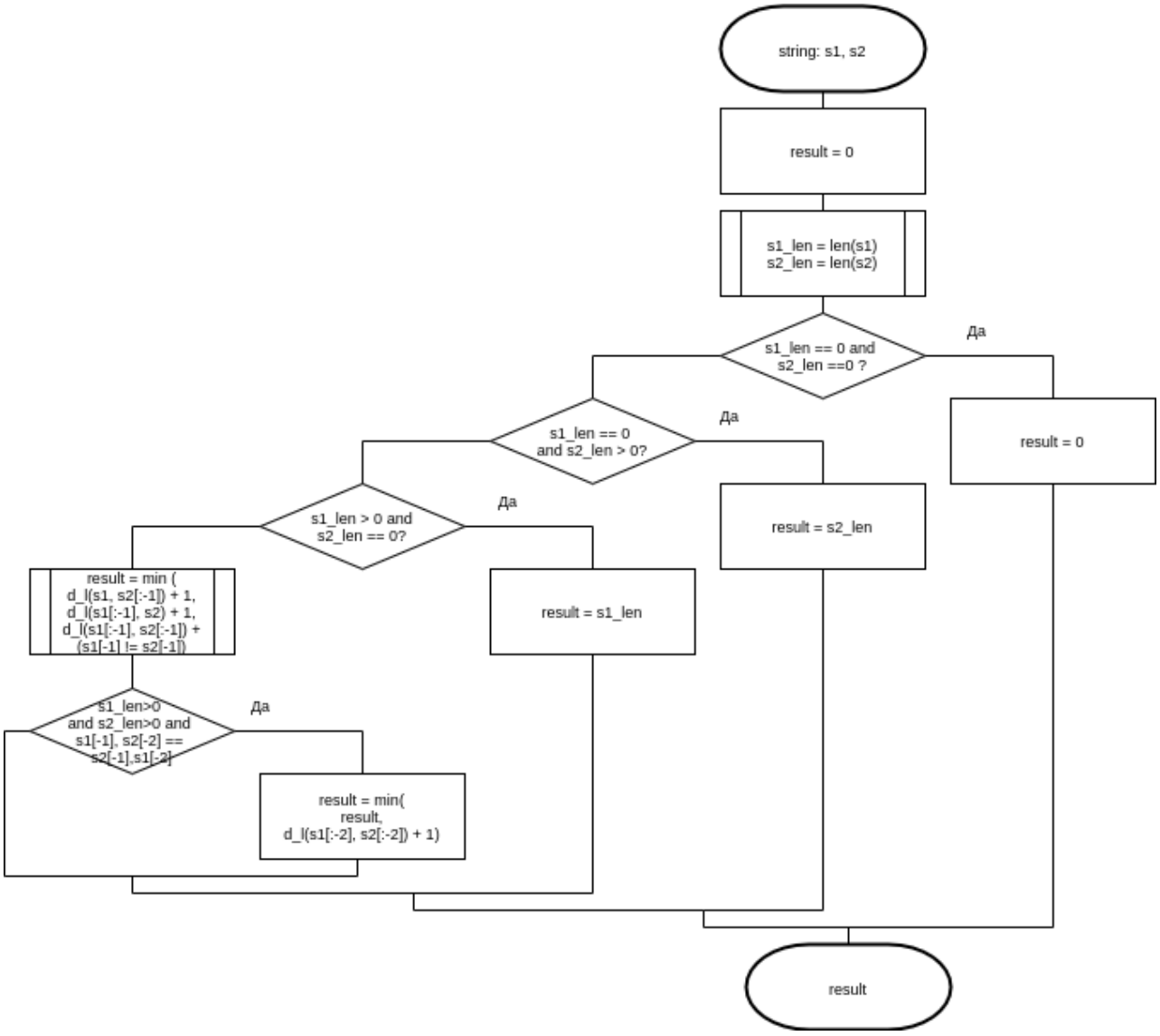

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна

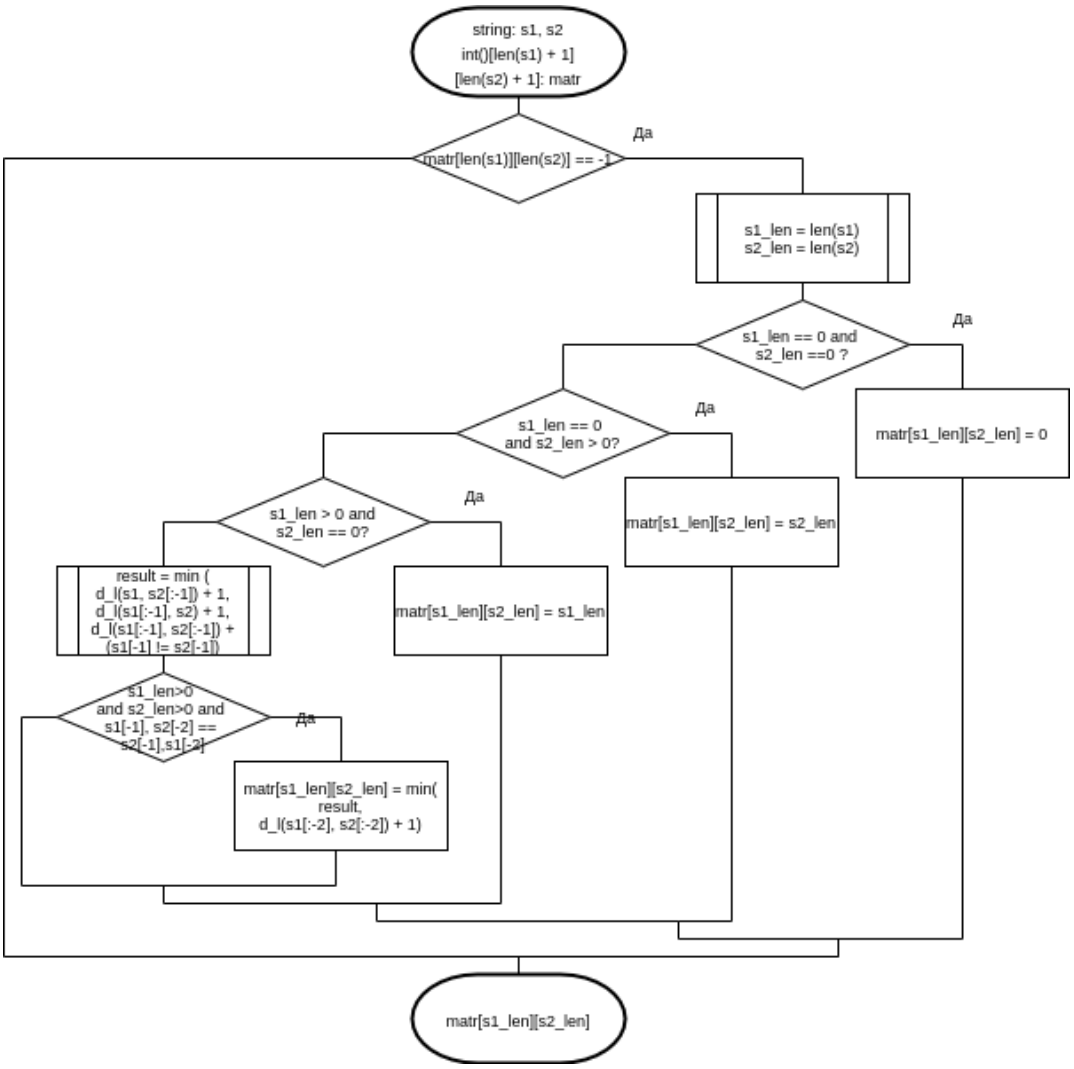

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с использованием кеша в виде матрицы

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

Программа принимает две строки (регистрозависимые). В качестве результата возвращается число, равное редакторскому расстоянию. Необходимо реализовать возможность подсчета процессорнго времени и пиковой использованной памяти для каждого из алгоритмов.

3.2 Средства реализации

Для реализации данной лабораторной работы был выбран язык программирования (ЯП) Python [3].

Данный язык достаточно удобен и гибок в использовании.

Время работы алгоритмов было замерено с помощью функции time() из библиотеки time [4]

3.3 Сведения о модулях программы

Программа состоит из пяти модулей:

1. algorithm.py - хранит реализацию алгоритмов;
2. test.py - хранит реализацию тестирующей системы и тесты;
3. memory.py - хранит реализацию системы замера памяти;
4. time.py - хранит реализацию системы замера времени;
5. utils.py - хранит реализацию вспомогательных функций.

3.4 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна нерекурсивным методом.

```
def non_recursive_levenshtein(s1, s2):
    distances = [[0 for _ in range(len(s1) + 1)] for __ in
       range(len(s2) + 1)]

    for i in range(len(s1) + 1):
        for j in range(len(s2) + 1):
            cur_dist = -1
            if i == 0 and j == 0:
                cur_dist = 0
            elif i == 0 and j > 0:
                cur_dist = j
            elif j == 0 and i > 0:
                cur_dist = i
            else:
                left = distances[j][i - 1] + 1
                up = distances[j - 1][i] + 1
                left_up = distances[j - 1][i - 1] + (0 if s1[i -
                   1] == s2[j - 1] else 1)
                cur_dist = min(left, up, left_up)

            distances[j][i] = cur_dist

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

Листинг 3.2 – Функция нахождения расстояния Дамерау – Левенштейна нерекурсивным методом.

```
def non_recursive_damerau_levenshtein(s1, s2):
    distances = [[0 for _ in range(len(s1) + 1)] for __ in
       range(len(s2) + 1)]

    for i in range(len(s1) + 1):
        for j in range(len(s2) + 1):
            cur_dist = -1
```

```
7               if i == 0 and j == 0:
8                   cur_dist = 0
9               elif i == 0 and j > 0:
10                  cur_dist = j
11              elif j == 0 and i > 0:
12                  cur_dist = i
13              else:
14                  left = distances[j][i - 1] + 1
15                  up = distances[j - 1][i] + 1
16                  left_up = distances[j - 1][i - 1] + (0 if s1[i -
                        1] == s2[j - 1] else 1)
17                  if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and
                        s1[i - 2] == s2[j - 1]:
18                      exchange = distances[j - 2][i - 2] + 1
19                      cur_dist = min(left, up, left_up, exchange)
20                  else:
21                      cur_dist = min(left, up, left_up)
22
23              distances[j][i] = cur_dist
24
25      return distances[len(s2)][len(s1)]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау – Левенштейна с использованием рекурсии.

```
1  def recursive_damerau_levenshtein(s1, s2, i, j):
2      if i == 0 and j == 0:
3          return 0
4      elif i == 0 and j > 0:
5          return j
6      elif j == 0 and i > 0:
7          return i
8      else:
9          left = recursive_damerau_levenshtein(s1, s2, i - 1, j) + 1
10         up = recursive_damerau_levenshtein(s1, s2, i, j - 1) + 1
11         left_up = recursive_damerau_levenshtein(s1, s2, i - 1, j
               - 1) + (0 if s1[i - 1] == s2[j - 1] else 1)
12         if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i -
               2] == s2[j - 1]:
13             exchange = recursive_damerau_levenshtein(s1, s2, i -
                   2, j - 2) + 1 if (i > 0 and j > 0 \
14                 and s1[i - 1] == s2[j - 2] and s1[i - 2] == s2[j
```

```
                    - 1]) else left_up
15              return min(left, up, left_up, exchange)
16          else:
17              return min(left, up, left_up)
```

Листинг 3.4 – Функция нахождения расстояния Дамерау – Левенштейна рекурсивным методом с использованием кеша.

```
1  def recursive_damerau_levenshtein_with_cache(s1, s2, i, j, cache):
2      if cache[j][i] != -1:
3          return cache[j][i]
4
5      if i == 0 and j == 0:
6          cache[j][i] = 0
7          return cache[j][i]
8      elif i == 0 and j > 0:
9          cache[j][i] = j
10         return cache[j][i]
11     elif j == 0 and i > 0:
12         cache[j][i] = i
13         return cache[j][i]
14     else:
15         left = recursive_damerau_levenshtein_with_cache(s1, s2, i
               - 1, j, cache) + 1
16         up = recursive_damerau_levenshtein_with_cache(s1, s2, i,
               j - 1, cache) + 1
17         left_up = recursive_damerau_levenshtein_with_cache(s1,
               s2, i - 1, j - 1, cache)\
18             + (0 if s1[i - 1] == s2[j - 1] else 1)
19         if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i -
               2] == s2[j - 1]:
20             exchange =
                   recursive_damerau_levenshtein_with_cache(s1, s2, i
                   - 2, j - 2, cache)\
21                 + 1 if (i > 0 and j > 0 and s1[i - 1] == s2[j -
                       2] and s1[i - 2] == s2[j - 1]) else left_up
22             cache[j][i] = min(left, up, left_up, exchange)
23         else:
24             cache[j][i] = min(left, up, left_up)
25
26         return cache[j][i]
```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дамерау – Левенштейна (в таблице - "Дамерау-Л."). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	"пустая строка"	"пустая строка"	0	0
2	aaa	aaa	0	0
3	"пустая строка"	asd	3	3
4	asd	пустая строка"	3	3
5	aaa	aaaaa	2	2
8	as	sa	2	1
9	asas	sasa	4	2
10	asas	saas	2	1
11	asas	saasa	3	2

Вывод

Были разработаны и протестированы алгоритмы: нахождения расстояния Левенштейна нерекурсивно, нахождения расстояния Дамерау – Левенштейна нерекурсивно, рекурсивно, а также рекурсивно с кешированием.

4 Исследовательская часть

В данном разделе приводятся результаты замеров алгоритмов по пиковой памяти и процессорному времени.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Ubuntu 22.04.1 [5] Linux x86_64.
- Память: 8 ГБ.
- Процессор: AMD® Ryzen 5 3500u.

Во время тестирования устройство было подключено к сети электропитания, нагружено приложениями окружения и самой системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи функции time() из библиотеки time языка Python. Данная функция возвращает количество секунд, прошедших с начала эпохи, типа float.

Контрольная точка возвращаемого значения не определна, поэтому допустима только разница между результатами последовательных вызовов.

Замеры времени для каждой длины слов проводились 1000 раз. В качестве результата взято среднее время работы алгоритма на данной длине слова.При каждом запуске алгоритма, на вход подавались случайно сгенерированные строки. Тестовые пакеты создавались до начала замера времени.

Результаты замеров приведены в таблице 4.1 (время в микросекундах).

Таблица 4.1 – Результаты замеров времени (в микросекундах).

Длина	Л.(не рек.)	Д.-Л.(не рек.)	Д.-Л.(рек.)	Д.-Л.(рек.кеш.)
0	1	2	0	0
1	3	4	1	2
3	11	11	27	16
5	20	23	685	38
10	64	79	3296002	144
25	359	477	-	946

4.3 Использование памяти

Алгоритмы тестировались при помощи функции get_traced_memory() из библиотеки tracemalloc языка Python, которая возвращает пиковый количество памяти, использованное процессором, на определенном этапе выполнения программы.

При каждом запуске алгоритма, на вход подавались случайно сгенерированные строки. Тестовые пакеты создавались до начала замера памяти.

Результаты замеров приведены в таблице 4.1 (в байтах).

Таблица 4.2 – Результаты замеров памяти (в байтах).

Длина	Л.(не рек.)	Д.-Л.(не рек.)	Д.-Л.(рек.)	Д.-Л.(рек.кеш.)
0	472	472	0	0
1	536	536	48	544
3	600	600	48	2856
5	872	872	411	3672
10	1960	1960	48	14096
25	7504	7336	-	64840

Вывод

В результате замеров можно сделать вывод о том, что чем меньше длина строки, тем эффективнее по памяти и времени работает алгоритм вычисления редакторского расстояния. Самым эффективным по памяти является рекурсивный алгоритм Дамерау – Левенштейна без кеширования,

самым неэффективным рекурсивный алгоритм Дамерау – Левенштейна с кешированием. Согласно результатам замеров процессорного времени самым быстрым оказался нерекурсивный алгоритм Дамерау – Левенштейна, самым медленным - рекурсивный алгоритм Дамерау – Левенштейна без кеширования.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены алгоритмы нахождения расстояний Левенштейна и Дамерау – Левенштейна;
- реализованы алгоритмы поиска расстояния Левенштейна с заполнением матрицы;
- реализованы алгоритмы поиска расстояния Дамерау – Левенштейна с заполнением матрицы, с использованием рекурсии и с помощью рекурсивного заполнения матрицы (рекурсивный с использованием кеша);
- проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- проведено экспериментальное подтвержение различий во временой эффективности рекурсивной и нерекурсивной реализаций алгоритмов при помощи разработанного программного обеспечения на материле замеров процессорного времени выполнения реализации на различных длинах строк;
- подготовлен отчет о лабораторной работе.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длин строк.

В результате исследований можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте строк, но проигрывает по количеству затрачиваемой памяти.

Список использованных источников

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

[2] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.

[3] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 04.10.2022).

[4] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 04.10.2022).

[5] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/20.04/` (дата обращения: 04.10.2022).