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 Расстояние Левенштейна

Студент Шахнович Дмитрий Сергеевич

Группа ИУ7-52Б

Преподаватели Волкова Л.Л., Строганов Д.М.

Москва, 2024

Содержание

ВВЕДЕНИЕ

Расстояние Левенштейна [1] – минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

— исправления ошибок в слове

— сравнения текстовых файлов утилитой diff

— в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы является изучение методов динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

1) Рассмотрение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;

2) Разработка рекурсивного алгоритма расстояния Левенштейна, а также применение методов динамического программирования для разработки алгоритмов расстояний Левенштейна и Дамерау-Левенштейна с использованием кеширования;

3) Реализация разработанных алгоритмов;

4) Исследование различий рекурсивной и итерационной реализаций алгоритма определения расстояния Левенштейна во временной и ёмкостной характеристиках с помощью замеров процессорного времени и пиковой памяти на строках варьирующейся длины.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна – метрика, измеряющая редакторское расстояние между двумя последовательностями символов. Расстояние Левенштейна – минимальное количество операций вставки, замены и удаления символов, необходимых для преобразования одной последовательности в другую. Стоимости каждого вида операций могут быть различны, однако в данной работе примем их всех равными единице, то есть будем считать количество операций, требуемых для преобразования.

1.1.1 Рекурсивная формула расстояния Левенштейна

Пусть существует две последовательности символов S_1 и S_2 длины N и M соответственно. Введём функцию $D(i, j)$, равную расстоянию Левенштейна между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Тогда расстояние Левенштейна для двух строк может быть выражено следующей рекурентной формулой:

$$D(i,j) = \begin{cases} 0, & \text{i = 0 и j = 0,} \\ i, & \text{i > 0 и j = 0,} \\ j, & \text{i = 0 и j > 0,} \\ min \begin{cases} D(i,j-1)+1, & \text{Вставка} \\ D(i-1,j)+1, & \text{Удаление} \\ D(i-1,j-1)+f_e(S_1[i],S_2[j]), & \text{Замена} \end{cases} & \text{i > 0 и j > 0,} \end{cases} \tag{1.1}$$

где функция сравнения строк описывается как:

$$f_e(a1,a2) = \begin{cases} 0, & \text{a1 == a2,} \\ 1, & \text{a1 != a2.} \end{cases} \tag{1.2}$$

Функция 1.2 просчитывает минимальное расстояние Левенштейна к текущему моменту(индексу в первой строке), пытаясь привести первую строку к виду второй с конца. В рекурсивной части функции первая строка символизирует вставку символа в первую строку из второй, вторая строка функции символизирует удаление символа из первой, а последняя замену символа первой на вторую, при этом если символы уже равны, то замену проводить не нужно и операция стоит 0.

1.1.2 Расстояние Левенштейна с кешированием

Рекурсивная форма может оказаться малоэффективной при больших значениях N и M, так как в ней могут постоянно пересчитываться значения $D(i, j)$ для одних и тех же аргументов. Для устранения потенциального недостатка используем итерационный алгоритм, который сохраняет промежуточные значения в виде матрицы размерности $(N + 1)x(M + 1)$.

Значения в ячейке [i, j] описанной выше матрицы содержат значение $D(i, j)$, то есть расстояние Левенштейна между подстроками $S_1[1...i]$ и $S_2[1...j]$. При этом нулевая строка и нулевой столбец матрицы заполняются слева–направо и сверху–вниз от 0 до соответствующей размерности, так как приведение пустой строки к любой строке длины, скажем, i требует i операций вставки.

Далее алгоритм построчно заполняет матрицу по формуле (1.1), при этом вместо просчёта предыдущих значений D используются значения из матрицы в соответствующих ячейках.

Данный алгоритм может работать эффективнее по времени, однако за счёт хранения дополнительной матрицы может тратить больше памяти. Можно оптимизировать это, если на каждой итерации хранить только текущую строку матрицы и предыдущую, так как остальные не нужны для просчёта.

1.2 Расстояние Дамерау–Левенштейна

Расстояние Дамерау–Левенштейна – метрика, расстояние между двумя последовательностями символов, определяемая минимальным количеством операций вставки, замены, удаления и транспозиции символов, необходимых для преобразования одной последовательности в другую. Под транспозицией понимается перестановка двух соседних символов. От расстояния Левенштейна отличается тем, что вводится операция транспозиции, которая может существенно уменьшить значение расстояния.

1.2.1 Рекурсивная формула расстояния Дамерау–Левенштейна

Введём также как и для расстояния Левенштейна функцию $D(i, j)$, при этом с учётом транспозиции формула будет выглядеть так:

$$D(i,j) = \begin{cases} 0, & \text{i = 0 и j = 0,} \\ i, & \text{i > 0 и j = 0,} \\ j, & \text{i = 0 и j > 0,} \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+f_e(S_1[i],S_2[j]), \\ D(i-2,j-2)+1, \end{cases} & i>1, j>1, \\ & S_1[i] == S_2[j-1], \\ & S_1[i-1] == S_2[j], \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+f_e(S_1[i],S_2[j]), \end{cases} & \text{иначе.} \end{cases} \quad (1.3)$$

Фактическое отличие от алгоритма расстояния Левенштейна в том, что при j > 1 и i > 1, а также при условии что соседние символы в строках равны "накрест может быть проведена операция транспозиции, которая может сократить расстояние.

1.2.2 Расстояние Дамерау–Левенштейна с кешированием

Также как и в случае расстояния Левенштейна, рекурсивный алгоритм может быть малоэффективным, поэтому имеет место быть итерационный алгоритм с кешированием, используя матрицу, либо текущую и последние две строки матрицы. Данный алгоритм аналогичен алгоритму поиска расстояния Левенштейна, за исключением того, что при равенстве накрест двух соседних символов строк и значению i>1 и j>1 при расчёте очередной ячейки нужно проверить ещё и вариант с транспозицией.

Вывод

В результате аналитического раздела были рассмотрены алгоритмы поиска расстояния Левенштейна и Дамерау–Левенштейна, а также их рекурсивные формулы, итерационные алгоритмы с кешированием матрицами и отдельными строками матриц.

2 Конструкторская часть

2.1 Требования к программному обеспечению

К разрабатываемой программе предъявлен ряд требований:

Входные данные: Две строки

Выходные данные: Целое число

— Заглавные и строчные символы считаются разными;

— Возможность обработки строк как на кириллице так и на латинице;

— Должна быть возможность замера процессорного времени программы;

— Должна быть возможность вывода графиков и таблиц замеров процессорного времени и памяти.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема алгоритма нахождения расстояния Левенштейна. На рисунках 2.2 – 2.3 представлена итерационная реализация алгоритма поиска расстояния Левенштейна. В первой части алгоритма, на рисунке 2.2 представлены создание и начальное заполнение матрицы кеша.На рисунке 2.3, во второй части алгоритм, изображён цикл с заполнением матрицы и поиском расстояния. На рисунках 2.4 – 2.5 представлен алгоритм поиска расстояния Дамерау–Левенштейна. Алгоритм представлен только в итерационной вариации с использованием кеширования. Первая часть алгоритма совпадает с первой частью на рисунке 2.2. Вторая часть отличается от 2.3, тем, что в заполнение матрицы добавлена операция транспозиции соседних элементов строки.

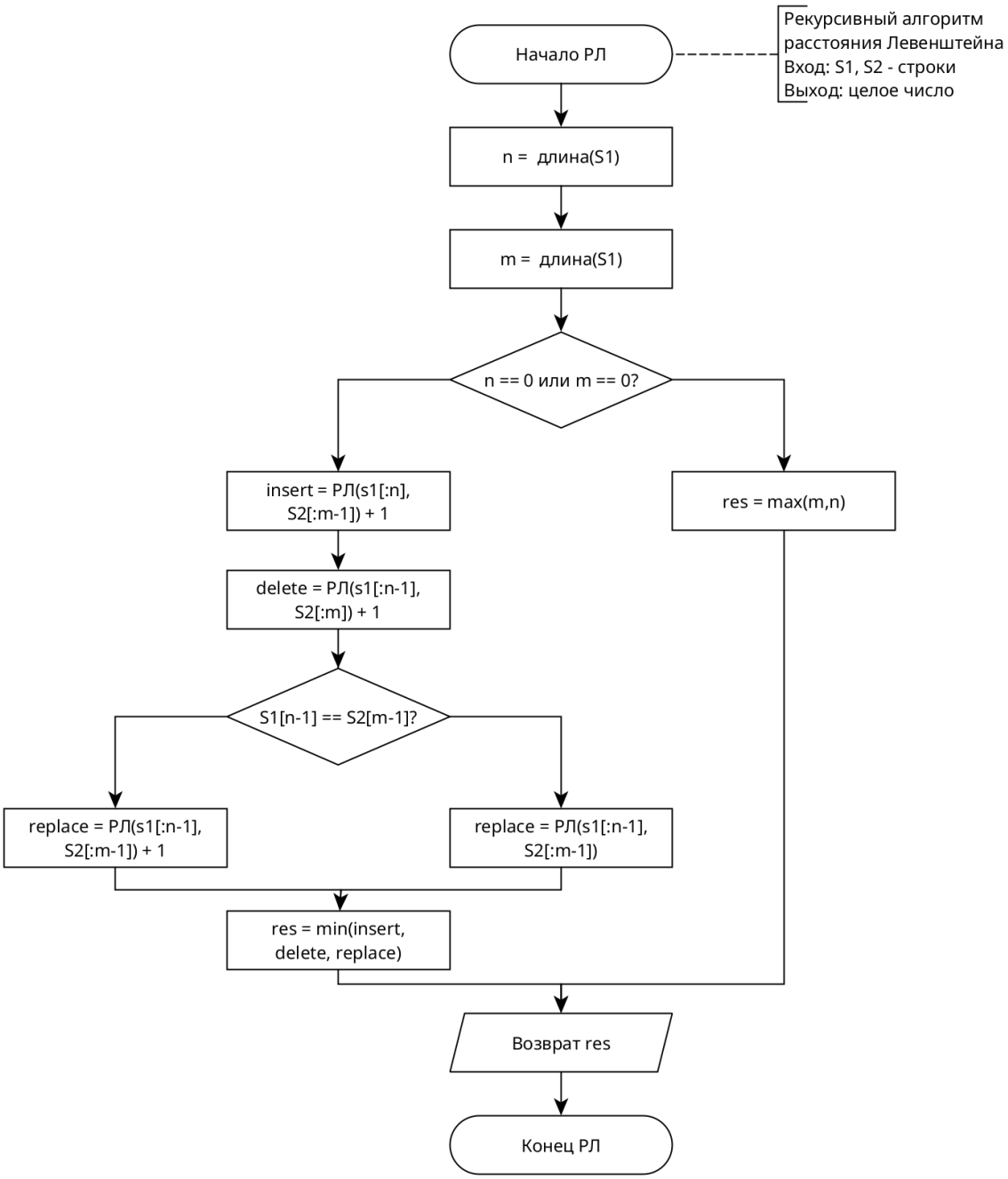

Рисунок 2.1 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна

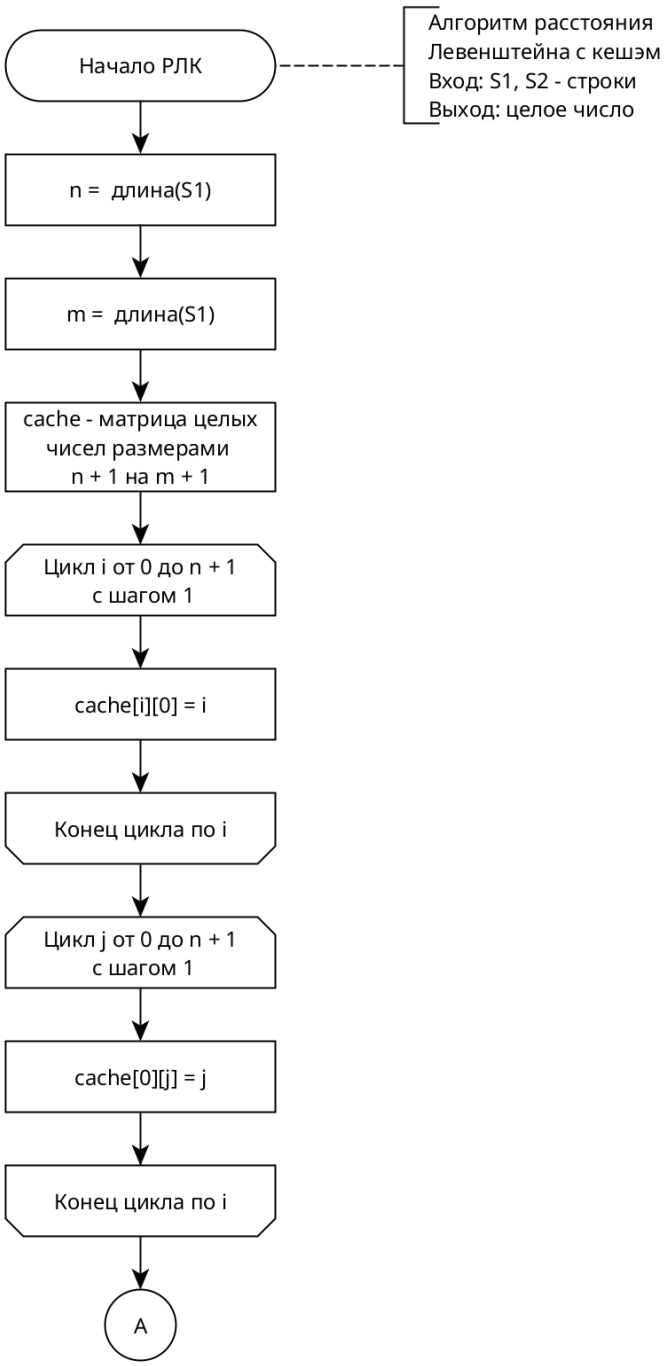

Рисунок 2.2 — Схема алгоритма нахождения расстояния Левенштейна с кешем, часть 1

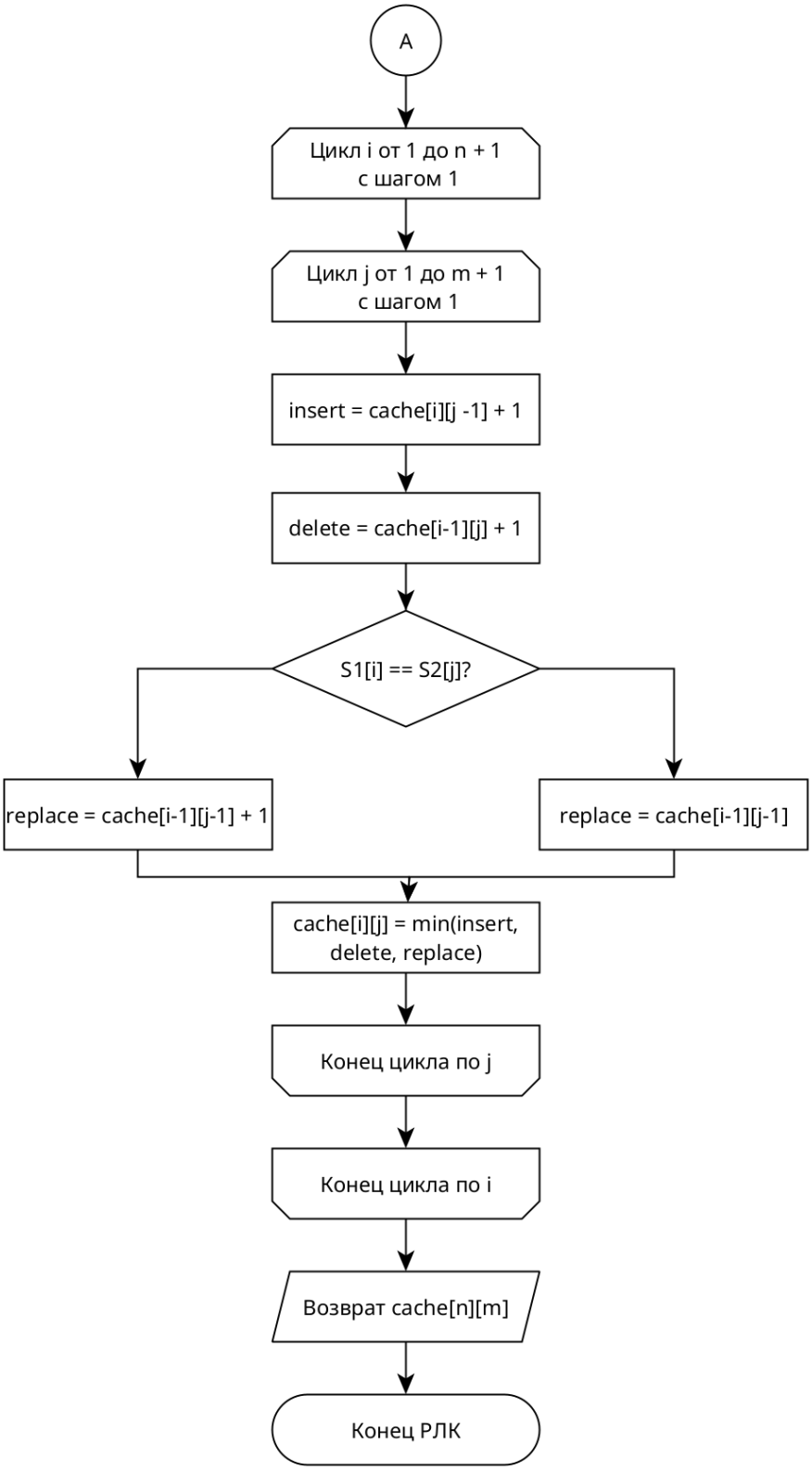

Рисунок 2.3 — Схема алгоритма нахождения расстояния Левенштейна с кешем, часть 2

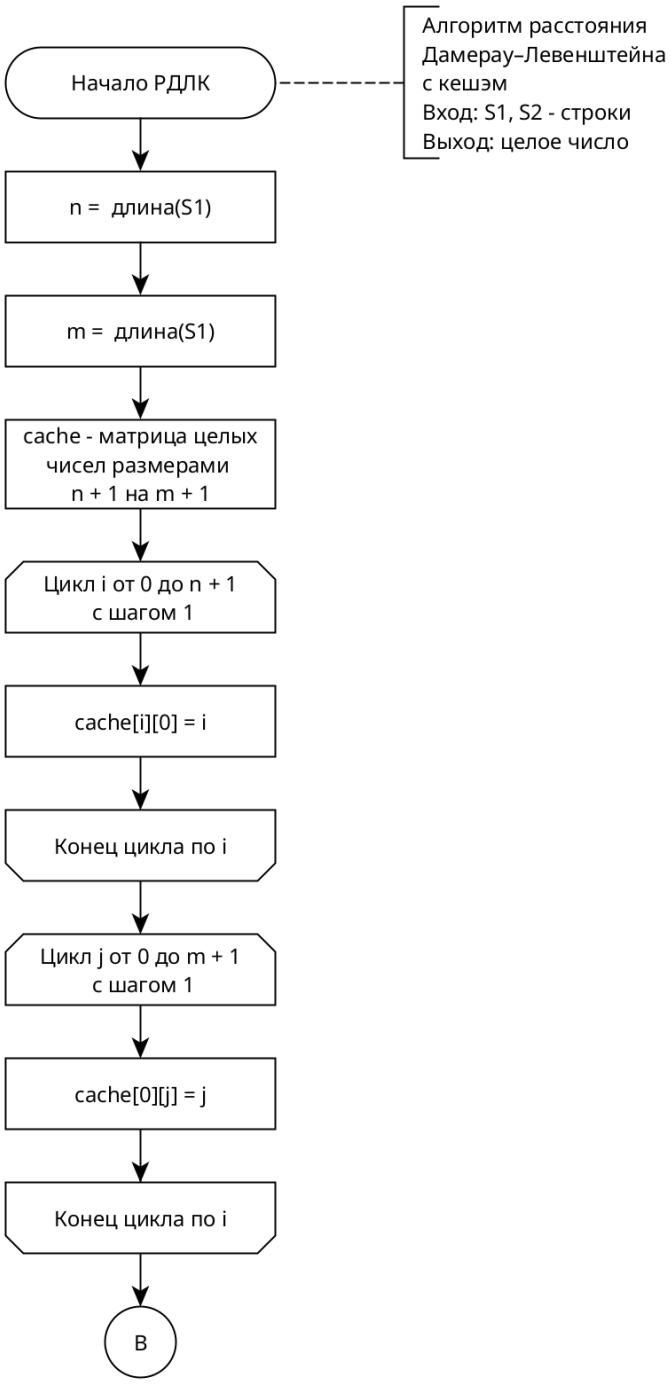

Рисунок 2.4 — Схема алгоритма нахождения расстояния Дамерау–Левенштейна с кешем, часть 1

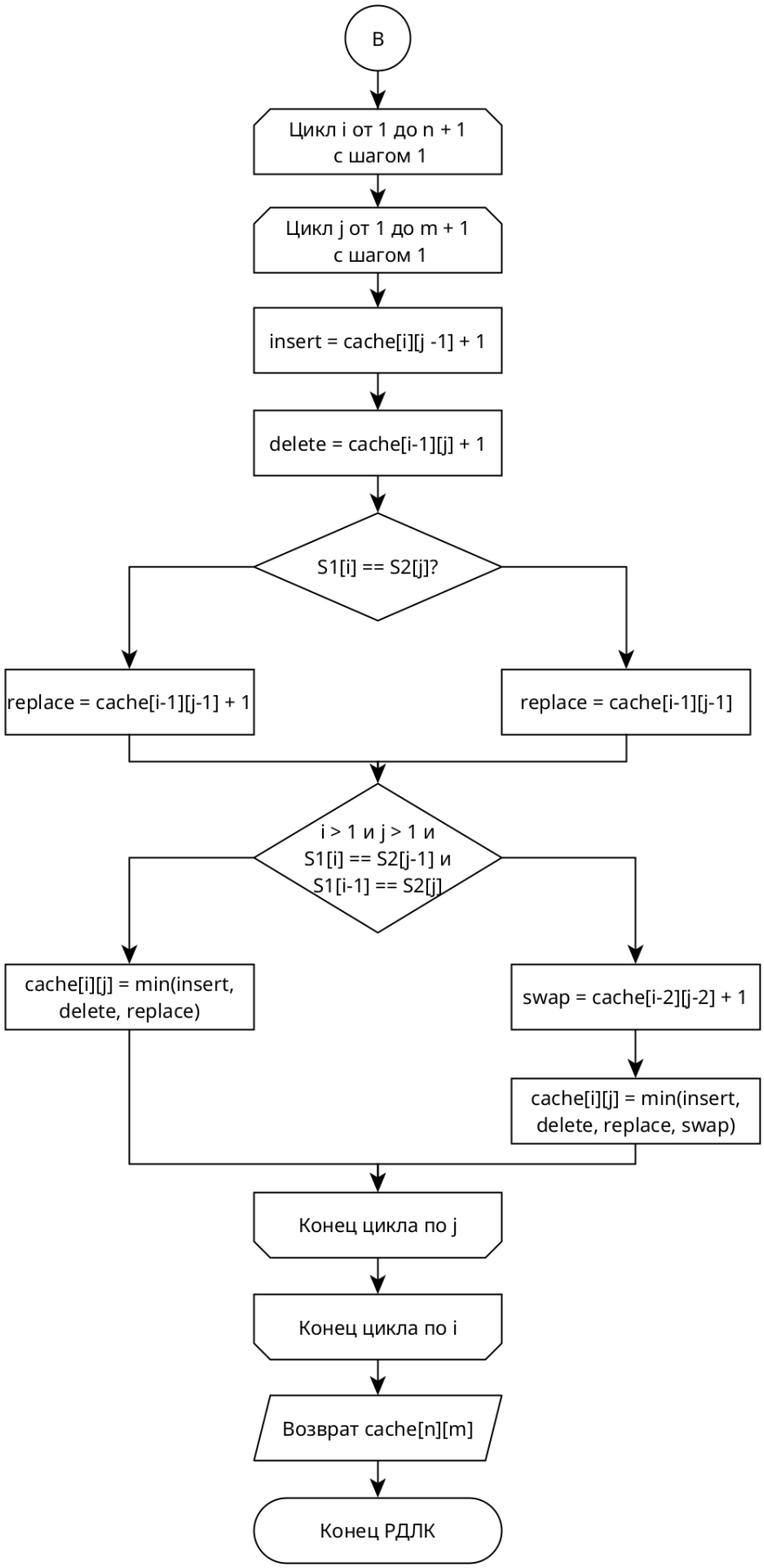

Рисунок 2.5 — Схема алгоритма нахождения расстояния Дамерау–Левенштейна с кешем, часть 2

2.3 Вывод

В результате конструкторской части были определены требования к ПО, а также разработаны схемы алгоритмов рекурсивного поиска расстояния Левенштейна, итерационного поиска с кешем и итерационного алгоритма поиска расстояния Дамерау–Левенштейна с кешем.

3 Технологическая часть

3.1 Средства разработки

В качестве языка программирования был выбран python3 [2], так как в его стандартной библиотеке присутсвуют функции замера процессорного времени, которые требуются в условиях, а также данный язык обладающий множеством инструментов для визуализации и работы с данными и таблицами.

Для основного файла был выбран инструмент jupyter notebook [3], так как он позволяет организовать код в удобные блоки, а также выводить данные и графики прямо в нём, что позволяет легко продемонстрировать все замеры.

Для построения графиков использовалась библиотека plotly [5].

Для замера времени использовалась функция process_time_ns из стандартного модуля time [4].

Для замеров памяти использовалась функция get_traced_memory стандартного модуля tracemalloc [6], которая получает текущую и пиковую выделенную память относительно начала замера.

3.2 Реализация алгоритмов

В листингах 3.1–3.3 приведены реализации разработанных в конструкторской части алгоритмов(рисунки 2.1–2.5).

Листинг 3.1 — Рекурсивный алгоритм нахождения расстояния Левенштейна

```
def RecursiveLevenshtein(s1, s2):
  if len(s1) == 0:
    return len(s2)
  if len(s2) == 0:
    return len(s1)
  if s1[0] == s2[0]:
    return RecursiveLevenshtein(s1[1:], s2[1:])
    return 1 + min(
  RecursiveLevenshtein(s1[1:], s2),
  RecursiveLevenshtein(s1[1:], s2[1:]),
  RecursiveLevenshtein(s1, s2[1:])
  )
```

Листинг 3.2 — Итерационный алгоритм нахождения расстояния Левенштейна с кешэм

```
  def CacheLevenshtein(s1, s2):
    cacheRows = len(s1) + 1
    cacheCols = len(s2) + 1
    cache = [[0] * cacheCols for _ in range(cacheRows)]
```

```
    for i in range(1, cacheCols):
      cache[0][i] = i
    for i in range(1, cacheRows):
      cache[i][0] = i

    for i in range(1, cacheRows):
      for j in range(1, cacheCols):
        cache[i][j] = min(cache[i - 1][j] + 1, cache[i - 1][j - 1]  + (0
           if s1[i - 1] == s2[j - 1] else 1), cache[i][j - 1] + 1)

    return cache[cacheRows - 1][cacheCols - 1]
  )
```

Листинг 3.3 — Итерационный алгоритм нахождения расстояния Дамерау-Левенштейна с кешэм

```
  def CacheDamerauLevenshtein(s1, s2):
    cacheRows = len(s1) + 1
    cacheCols = len(s2) + 1
    cache = [[0] * cacheCols for _ in range(cacheRows)]
    for i in range(1, cacheCols):
      cache[0][i] = i
    for i in range(1, cacheRows):
      cache[i][0] = i
    for i in range(1, cacheRows):
      for j in range(1, cacheCols):
        if i >= 2 and j >= 2 and s1[i - 1] == s2[j - 2] and s1[i - 2] ==
           s2[j - 1]:
          cache[i][j] = min(
            cache[i - 1][j] + 1,
            cache[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else 1),
            cache[i - 2][j - 2] + 1,
            cache[i][j - 1] + 1,
            )
        else:
          cache[i][j] = min(
            cache[i - 1][j] + 1,
            cache[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else 1),
            cache[i][j - 1] + 1,
            )

    return cache[cacheRows - 1][cacheCols - 1]
```

3.3 Функциональные тесты

В табличке 3.1 приведены тесты для алгоритмов поиска расстояние Левенштейна и Дамерау–Левенштейна.

Таблица 3.1 — Функциональные тесты

Входные данные		**Расстояния и алгоритм**				
		Левенштейн			Дамерау–Левенштейн	
Строка 1	Строка 2	Рекурсивный	С кешем	Ожидаемое	С кешем	Ожидаемое
дмитрий	андрей	5	5	5	5	5
река	мука	2	2	2	2	2
1234	2134	3	3	3	2	2

Все тесты пройдены успешно

Вывод

В ходе технологической части работы были разработаны алгоритмы поиска расстояния Левенштейна и Дамерау–Левенштейна, а также проведено их тестирование.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства,на котором проводились замеры:

— Процессор: AMD Ryzen 7 5800H (16) @ 4.46 ГГц;

— Оперативная память: 16 ГБ;

— Операционная система: Arch Linux x86_64.

При проведении замеров ноутбук был включён в сеть и были запущены только системные приложения и jupyter notebook.

4.2 Временные характеристики

В таблице (4.1) приведены временные характеристики, полученные в результате замеров времени алгоритмов (3.1) и (3.2). Для каждой длины замер проводился 100 раз, при этом для каждой итерации создавались 2 случайные строки заданной длины и использовались как аргументы к алгоритмам. Итоговое время взято как среднее арифметическое всех полученных замеров.

Таблица 4.1 — Временные характеристики

Длина строки, символы	Рекурсивный алгоритм, мкс	Итерационный алгоритм с кешем, мкс
2	151	98
3	452	99
4	1 540	120
5	7 077	138
6	29 669	155
7	138 545	177
8	658 211	203
9	2 952 534	223
10	20 510 843	245

Полученные замеры также можно увидеть на графике (4.1):

Как видно из таблицы (4.1) и графика (4.1), рекурсивный алгоритм существенно уступает итерационному по временным характеристикам.

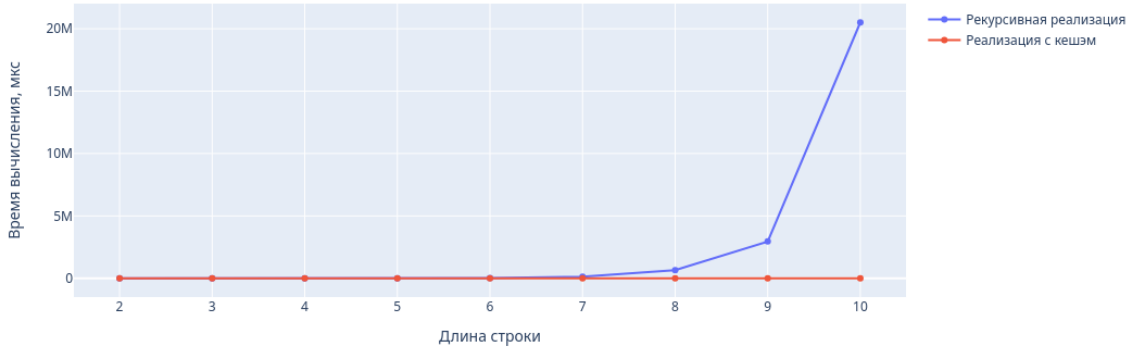

Рисунок 4.1 — График зависимости времени работы алгоритмов от длины строк

4.3 Ёмкостные характеристики

В таблице (4.2) приведены временные характеристики, полученные в результате замеров памяти алгоритмов (3.1) и (3.2) с помощью модуля tracemalloc. Так как для рекурсивной реализации размер выделяемой памяти существенно зависит от глубины рекурсии, то есть от входных строк, то для каждой длины строки для рекурсивного алгоритма алгоритм был проведён 10 раз на случайных строках. Для итерационного алгоритма выделяемая память не зависит от входных строк(при одинаковой длине), так как размер матрицы кеша зависит только от размеров строк. За характеристику памяти взята пиковая память, то есть максимальный объём памяти выделенный функции во время её выполнения.

Таблица 4.2 — Временные характеристики

Длина строки, символы	Рекурсивный алгоритм, байты	Итерационный алгоритм с кешем, байты
2	48	232
3	247	288
4	288	392
5	312	480
6	404	584
7	498	704
8	674	904
9	1 429	1 056
10	2 201	1 224

Также зависимость из таблицы отражена на графике (4.2).

Как видно из графика (4.2) и таблицы (4.2), при небольших размерах строки рекурсивная реализация выигрывает итерационную, одна начиная с длин строк 9, вариант с кешем становится эффективнее, так как реализация рекурсии требует больший объем памяти под входные строки и вызовы.

Рисунок 4.2 — График зависимости требуемой памяти для работы алгоритмов от длины входных строк

4.4 Вывод

В данном разделе было проведено исследование временных и ёмкостных характеристик алгоритмов рекурсивного поиска расстояния Левенштейна и итерационного поиска расстояние Левенштейна с кешем.

По результатам исследования оказалось, что рекурсивная реализация существенно проигрывает итерационной по времени. Что касается ёмкостных характеристик, то при длине входных строк ≤ 8 рекурсивная реализация эффективная, однако при длине ≥ 9 эффективнее становится итерационный вариант из–за того, что рекурсивный алгоритм делает много вызовов функции.

ЗАКЛЮЧЕНИЕ

Цель - изучить методы динамического программирование на примере алгоритмов расстояния Левенштейна и Дамерау-Левенштейна - была выполнена. При этом в ходе работы были выполнены следующие задачи:

— Рассмотрены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;

— Разработаны рекурсивный алгоритм расстояния Левенштейна и итерационные алгоритмы расстояний Левенштейна и Дамерау-Левенштейна с применением методов динамического программирования;

— Реализованы разработанные алгоритмы на языке python3;

— Проведено исследования различий в ёмкостной и временной характеристиках между рекурсивной и итерационной реализаций алгоритмов расстояния Левенштейна.

В результате исследования было выявлено, что сложность алгоритма поиска расстояния Левенштейна быстро растёт при увеличении длины строк. При этом итерационная реализация алгоритм с кешированием на несколько порядков быстрее рекурсивного варианта, за счёт того, что не пересчитывает заново значения для подстрок, как в рекурсивной реализации. С точки зрения ёмкостных характеристик, то при маленьких значениях длины строк(≤ 8) выигрывает рекурсия, однако при длинах (≥ 8) итерационный вариант оказывается эффективнее.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. И. В. Левенштейн Двоичные коды с исправлением выпадений, вставок и замещений символов [Текст] / И. В. Левенштейн // Доклады АН СССР. — 1965. — № 4. — С. 845-848.

2. Python / [Электронный ресурс] // Python : [сайт]. — URL: https://www.python.org/ (дата обращения: 20.09.2024).

3. Jupyter Notebook: The Classic Notebook Interface / [Электронный ресурс] // Jupyter : [сайт]. — URL: https://jupyter.org/ (дата обращения: 20.09.2024).

4. time — Time access and conversions / [Электронный ресурс] // Python 3.12.6 documentation : [сайт]. — URL: https://docs.python.org/3/library/time.html#time.process_time_ns (дата обращения: 21.09.2024).

5. Plotly Open Source Graphing Library for Python / [Электронный ресурс] // Plotly : [сайт]. — URL: https://plotly.com/python/ (дата обращения: 20.09.2024).

6. tracemalloc — Trace memory allocations / [Электронный ресурс] // Python 3.12.6 documentation : [сайт]. — URL: https://docs.python.org/3/library/tracemalloc.html#tracemalloc.get_traced_memory (дата обращения: 20.09.2024).