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 предмету: «Анализ алгоритмов»

Расстояние Левенштейна

Студент: Гасанзаде М.А.,
Группа: ИУ7-56Б

Москва, 2019 г.

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Измеряется для двух строк, широко используется в теории информации и компьютерной лингвистике.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей, впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

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.

Еще пару лет назад Фредерик Дамерау мог бы гарантировать, что большинство ошибок при наборе текста — как раз и есть транспозиции. Поэтому именно данная метрика дает наилучшие результаты на практике.

Задачи работы:

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

2) применение метода динамического программирования для матричной реализации указанных алгоритмов;

3) получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;

4) сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5) экспериментальное подтверждение различий во временной эффективности рекурсивной и не рекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

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

1. Аналитическая часть

В данном разделе будут представлены описание расстояния Левенштейна, Дамерау-Левенштейна, формулы и оценки сложностей данных алгоритмов.

1.1 Описание алгоритмов

Расстояние Левенштейна или редакционное расстояние между строками s1 и s2 как min количество редакторских операций, необходимых для преобразования одной строки в другую.

Редакторские операции:

$$1 \text{ символ (стоимость} = 1) \begin{cases} I - insert \text{ (вставка)} \\ D - delete \text{ (удаление)} \\ R - replace \text{ (замена)} \end{cases}$$

$$\exists M - match \text{ (совпадение) стоимость} = 0$$

Формула для нахождения расстояния с использованием матрицы:

$$D(i,j) = \begin{cases} 0, \text{если } i == 0, \quad j == 0 \\ i, \text{если } i > 0, \quad j == 0 \\ j, \text{если } j > 0 \quad i == 0 \\ min \begin{cases} D(s1[1 \dots i], s2[1 \dots j-1]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j-1]) + \begin{bmatrix} 0, s1[i] == s2[j] \\ 1, \text{иначе} \end{bmatrix} \end{cases} \end{cases}$$

Расстояние Дамерау-Левенштейна вводится дополнительная операция перестановки или транспозиция, 2 буквы, стоимость = 1. Если индексы позволяют, и если соседние буквы $s1[i]$ совпадает с $s2[j-1]$ и $s1[i-1] = s2[j]$, то в минимум включается перестановка (транспозиция).

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise, Infinity} \end{cases}$$

Применение алгоритмов:

- поиск по словарю (частная задача лингвистики);
- биоинформатика;
- поисковые системы, для предложения более подходящего запроса, в случае, если пользователь допустил ошибку или ввел одну букву раньше другой.

Данные формулы так же допускают рекурсивную реализацию [4].

1.2 Вывод:

В данном разделе были рассмотрены формулы нахождения расстояния Левенштейна[2] и Дамерау-Левенштейна[3], который является модификацией первого, учитывающего возможность перестановки соседних символов, а также способы их применения.

2. Конструкторская часть

В данном разделе будут размещены схемы алгоритмов и сравнительный анализ рекурсивной и не рекурсивной реализаций.

2.1 Разработка алгоритмов

Ниже на *рис. 1-3* будут приведены схемы реализованных алгоритмов:

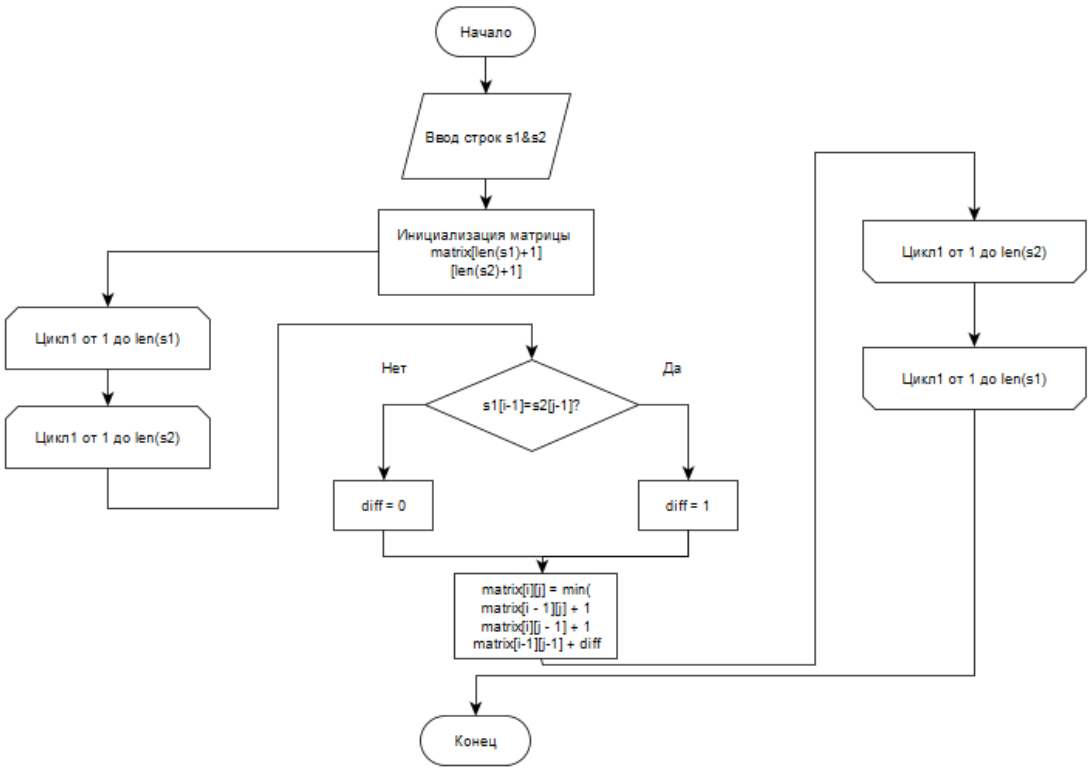

Рисунок 1. Алгоритм стандартного Левенштейна

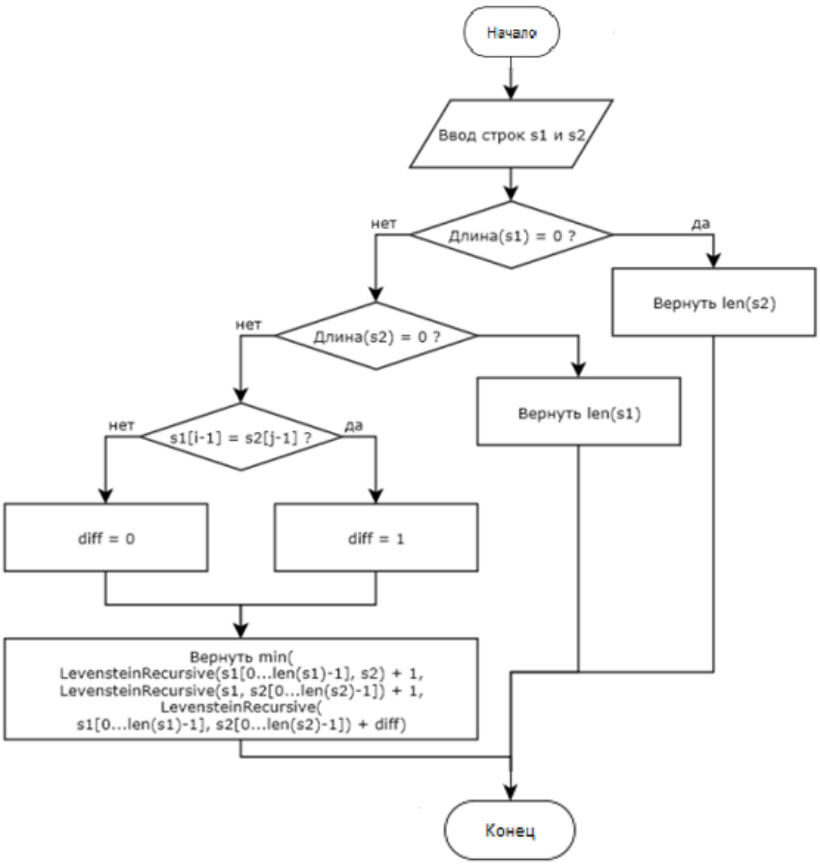

Рисунок 2. Алгоритм Левенштейна рекурсией

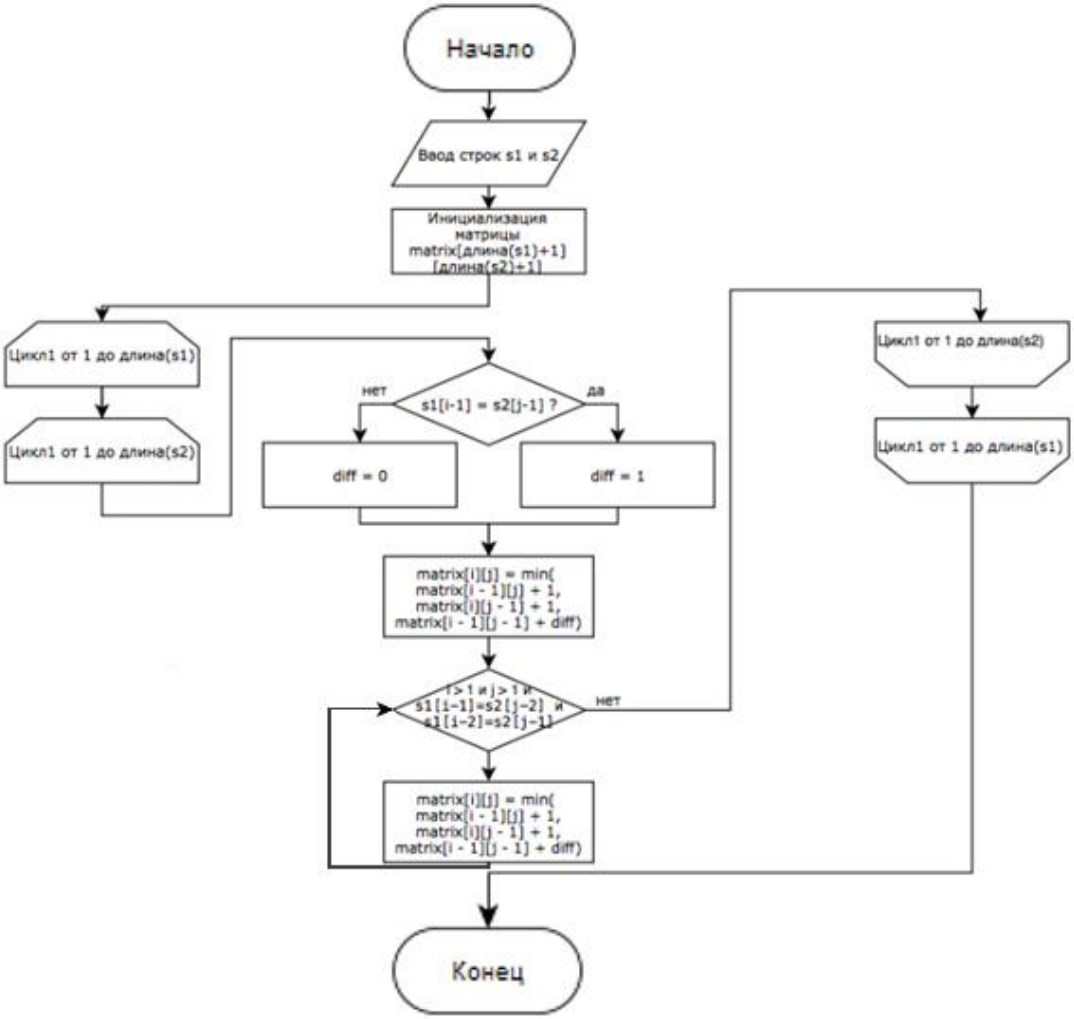

Рисунок 3. Алгоритм Дамерау-Левенштейна

2.2 Вывод

В данном разделе были рассмотрены схемы алгоритмов нахождения расстояния Левенштейна, Дамерау-Левенштейна, а также рекурсивный алгоритм Левенштейна.

3 Технологическая часть

В данном разделе будут приведены Требования к программному обеспечению, средства реализации, листинг кода и примеры тестирования.

3.1 Требования к программному обеспечению

На вход подаются 2 строки, на выходе необходимо получить матрицу и 3 результата, выдаваемых матричными реализациями обоих алгоритмов и рекурсивной реализации алгоритма Дамерау-Левенштейна.

Требуется замерить время работы каждой реализации.

3.2 Средства реализации

В качестве языка программирования был выбран Python в связи с его широким функционалом удобством работы, а из-за привычного для меня синтаксиса. Среда разработки - стандартная IDLE Python. Время работы процессора замеряется с помощью библиотеки time[1] и функции:

Листинг 1. Функция замера времени

```
from time import perf_counter
def getWorkTime(str1, str2, countOperations):
    t1_start = perf_counter()
    for i in range(countOperations):
        levensteinRec(str1, str2)

    t1_stop = perf_counter()
    print("Elapsed time:", t1_stop-t1_start)
    t2_start = perf_counter()
    for i in range(countOperations):
        levens(str1, str2)
    t2_stop = perf_counter()
    print("Elapsed time:", t2_stop-t2_start)

    t3_start = perf_counter()
    for i in range(countOperations):
        lev_modified(str1, str2)
    t3_stop = perf_counter()
    print("Elapsed time:", t3_stop-t3_start)
```

3.3 Листинг кода

Листинг кода был представлен на *листингах. 2, 3, 4.*

Рисунок 2. Реализация стандартного алгоритма Левенштейна

```
def levens(word1, word2):
    size1, size2 = len(word1), len(word2)
    if size1 > size2:
        word1, word2 = word2, word1
        size1, size2 = size2, size1

    current_row = range(size1 + 1)
    for i in range(1, size2 + 1):
        previous_row, current_row = current_row, [i] + [0] * size1
        for j in range(1, size1 + 1):
            add, delete, change = previous_row[j] + 1, current_row[j -
1] + 1, previous_row[j - 1]
            if word1[j - 1] != word2[i - 1]:
                change += 1
            current_row[j] = min(add, delete, change)
    return current_row[size1]
```

Листинг 3. Реализация алгоритма Левенштейна рекурсивно

```
def levensteinRec(str1, str2):
    size1 = len(str1)
    if (size1 == 0):
        return len(str2)

    size2 = len(str2)
    if (size2 == 0):
        return size1

    symbIdent = 0
    if (str1[0] != str2[0]):
        symbIdent = 1

    return min((levensteinRec(str1[1:], str2) + 1),
               (levensteinRec(str1, str2[1:]) + 1),
               (levensteinRec(str1[1:], str2[1:]) + symbIdent))
```

```
def lev_modified(str1, str2):
    size1, size2 = len(str1), len(str2)
    if (size1 < 2 or size2 < 2):
        return levens(str1, str2)

    if size1 > size2:
        str1, str2 = str2, str1
        size1, size2 = size2, size1

    prev_row = range(size1 + 1)
    curr_row = [1] + [0] * size1
    for i in range(1, size1 + 1):
        add, delete, change = prev_row[i] + 1, curr_row[i - 1] + 1,
prev_row[i - 1]
        if str1[i - 1] != str2[0]:
            change += 1
        curr_row[i] = min(add, delete, change)

    for i in range(2, size2 + 1):
        prev_prev_row = prev_row; prev_row = curr_row; curr_row = [i]
+ [0] * size1
        add, delete, change = prev_row[1] + 1, curr_row[0] + 1,
prev_row[0]
        if (str1[0] != str2[i - 1]):
            change += 1
        curr_row[1] = min(add, delete, change)

        for j in range(2, size1 + 1):
            add, delete, change = prev_row[j] + 1, curr_row[j - 1] +
1, prev_row[j - 1]
            if (str1[j - 1] != str2[i - 1]):
                change += 1
            if (str2[i - 1] == str1[j - 2] and str2[i - 2] == str1[j -
1]):
                transp = prev_prev_row[j - 2] + 1
                curr_row[j] = min(add, delete, change, transp)
            else:
                curr_row[j] = min(add, delete, change)

    return curr_row[size1]
```

Листинг 4. Реализация алгоритма Дамерау-Левенштейна

3.4 Описание тестирования

Примеры работы программы с введенными данными (Слово 1 и Слово 2). В качестве вывода приведены результаты работы 3 реализованных алгоритмов. Таблицы с результатами были представлены на *рис.* 1,2,3.

Таблица 1 — Данные тестов

Слово 1	Слово 2	Вывод	Ожидаемый вывод	Результат
doors	**odors**	2 2 1	2 2 1	+
doors	**doors**	0 0 0	0 0 0	+
doors	**door**	1 1 1	1 1 1	+
doors	**norway**	5 5 5	5 5 5	+

Рисунок 2 — Данные тестов

Слово 1	Слово 2	Вывод	Ожидаемый вывод	Результат
abracadabra	**arbadacarba**	6 6 4	6 6 4	+
abracadabra	**abracadabra**	0 0 0	0 0 0	+
abracadabra	**arbitrary**	7 7 7	7 7 7	+
abracadabra	**barack**	8 8 7	8 8 7	+

Рисунок 3 — Данные тестов

Слово 1	Слово 2	Вывод	Ожидаемый вывод	Результат
decade	**facade**	2 2 2	2 2 2	+
decade	**decade**	0 0 0	0 0 0	+
decade	**decaed**	2 2 1	2 2 1	+
decade	**deacon**	4 4 3	4 4 3	+

Все тесты пройдены успешно.

3.4 Вывод

В данном разделе мы рассмотрели листинг кода, а также убедились в безошибочной работе программы.

4. Экспериментальная часть.

В данном разделе будут рассмотрены примеры работ программы.

Память была замерена с помощью библиотеки memory_profiler[6] в 32х битном Python IDLE.

4.1 Постановка эксперимента по замеру времени

Для произведения замеров времени выполнения реализаций алгоритмов будет использована следующая формула $t = \frac{Tn}{N}$, где t – время выполнения, N – количество замеров, Tn – общее время потраченное на замеры. Неоднократное измерение времени необходимо для построения более гладкого графика.

Количество замеров будет взято равным 50.

Тестирование будет проведено на одинаковых входных данных. Для сравнения матричных реализации длины строк 0-500 с шагом 100. Для сравнения матричной и рекурсивной реализации длины строк 0-10 с шагом 1.

4.2 Сравнительный анализ на материале экспериментальных данных

Алгоритмы были протестированы по скорости работы.

Замеры времени отражены на *рис. 4, 5*.

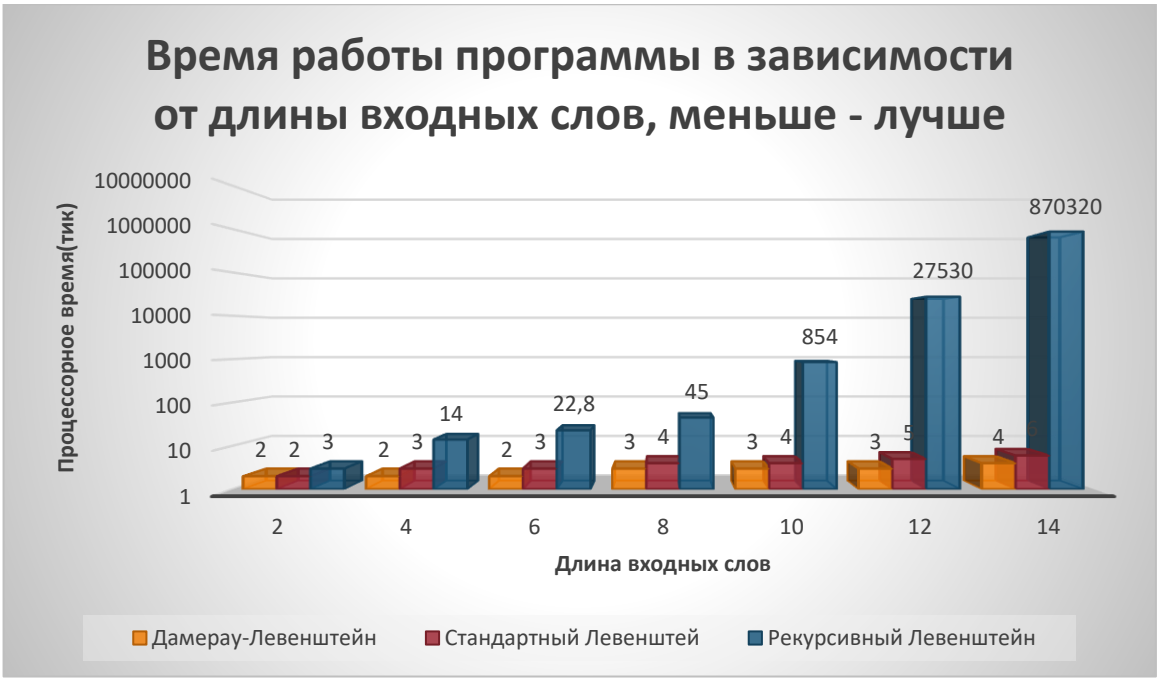

Рисунок 4 – Замеры времени.

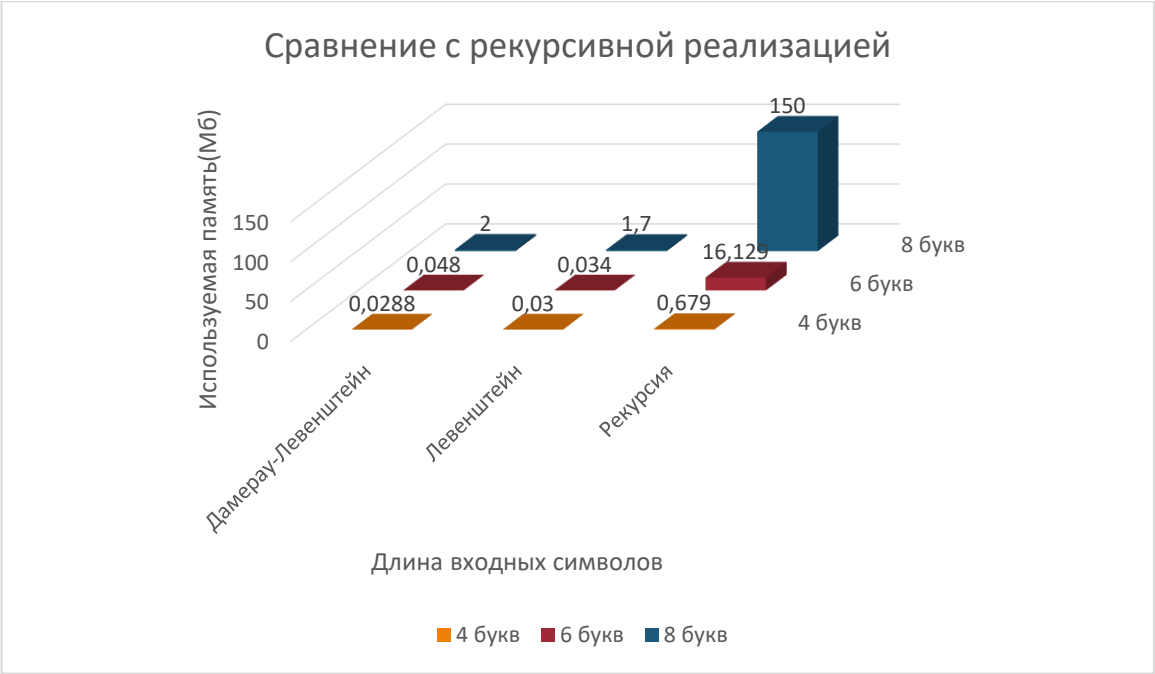

Рисунок 5 – Замеры памяти.

4.3 Сравнительный анализ алгоритмов

На примере алгоритма Левенштейна было показано, что данная рекурсивная реализация алгоритма поиска редакционного расстояния очень сильно

проигрывает по времени работы и памяти, демонстрируя экспоненциальный рост процессорного времени.

На практике, большинство ошибок при печати представляют собой транспозицию соседних символов, и алгоритм Дамерау-Левенштейна, имеющий соответствующую проверку, предпочтительнее. Однако, у алгоритма Дамерау-Левенштейна также может быть рекурсивная реализация.

Принимая во внимание первый абзац, можно сделать вывод о том, что подобная реализация рекурсивного алгоритма Дамерау-Левенштейна будет демонстрировать такой же экспоненциальный рост времени и памяти работы и нецелесообразен для обработки даже не очень больших объёмов текста.

Таким образом, предпочтительней всего для исправления ошибок при пользовательском вводе использовать итеративный алгоритм ДамерауЛевенштейна.

4.4 Вывод

В данном разделе был представлен эксперимент по замеру времени и памяти выполнения каждого алгоритма. По итогам замеров алгоритм нахождения расстояния Левенштейна оказался самым быстрым, а самым медленным – рекурсивнный алгоритм Левешгтейна

Заключение

В данной работе были реализованы и протестированы самые популярные алгоритмы поиска редакционного расстояния - Левенштейна и ДамерауЛевенштейна. Проведён сравнительный анализ рекурсивной и нерекурсивной (итеративной) реализаций.

Итеративная реализация продемонстрировала наилучший результат по времени работы и лишена гипотетической проблемы переполнения стека вызовов.

Рекурсивная же реализация выигрывает лишь по простоте воплощения на языке программирования и удобочитаемости кода.

Список литературы

1. time — Time access and conversions // Python URL: https://docs.python.org/3/library/time.html (дата обращения: 1.11.2019).
2. Расстояние Левенштейна – URL: http://bilgisayarkavramlari.sadievrenseker.com/2010/12/30/levenshtein-mesafe-algoritmasi-levenshtein-distance/ (дата обращения 1.11.2019)
3. Расстояние Дамерау-Левенштейна URL: https://tr.qwertyu.wiki/wizardmhurl/ (дата обращения 1.11.2019)
4. Рекурсивный метод Вагнера-Фишера URL: shorturl.wizardmh (дата обращения 1.11.2019)
5. Метод Вагнера-Фишера URL: https://medium.com/@sddkal/wagner-fischer-algoritmas%C4%B1-e9558b598ba0/ (дата обращения 1.11.2019)
6. Замер памяти в Python3: URL: https://pypi.org/project/memory-profiler/ (дата обращения 5.12.2019)