Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №1

по курсу «Анализ алгоритмов»

Расстояние Левенштейна и Дамерау-Левенштейна

Выполнил: Сорокин А.П., гр. ИУ7-52Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2019 г.

Оглавление

Введение

Задача определения такого минимума актуальна, так как она решает множество проблем в теории информации и компьютерной лингвистике, например:

- исправление ошибок в словах при вводе (при в поисковых ситсемах, базах данных, программах автоматического определения текста);
- сравнении текстовых файлов (к примеру, утилита diff);
- сравнение белков, генов и хромосом в биоинформатике.

1. Аналитическая часть

1.1 Задачи

Цель лабораторной работы: исследовать расстояния Левенштейна и Дамерау-Левенштейна. Для достижения этой цели были поставлены следующие задачи:

- изучить алгоритмы вычисления расстояний между строками;
- применить методы динамического программирования для матричной реализации алгоритмов;
- сравнить матричную и рекурсивную реализацию алгоритмов;
- оценить эффективность каждой из реализаций по времени и памяти.

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);
- совпадение (M - match).

У каждой операции есть так называемая "цена или "штраф"за её выполнение. Цена каждой операции равна 1, кроме операции совпадения, цена которой равна 0, т. к. при равенстве символов не требуется никаких действий. Соответственно, задача нахождения расстояния Левенштейна заключается в нахождении такой последовательности операции, приводящик одну строку к другой, суммарная цена которых минимальна.

Таким образом, если заданы две строки S_1 и S_2 с длинами m и n соответственно над некоторым алфавитом, то расстояние Левенштейна $D(S_1, S_2)$ между данными строками можно вычислить по следующей рекуррентной формуле [5]:

$$D(S_1[1..m], S_2[1..n]) = \begin{cases} m \; if \; n = 0 \\ n \; if \; m = 0 \\ min \begin{cases} D(S_1[1..m-1], S_2[1..n] + 1) \\ D(S_1[1..m], S_2[1..n-1] + 1) \\ D(S_1[1..m-1], S_2[1..n-1] + (S_1[m] \neq S_2[n])) \end{cases} \end{cases} \tag{1.1}$$

Соотношения в рекурретной формуле отвечают за соотвествующие разрешённые операции:

1. вставка;
2. удаление;
3. замена или совпадение в зависимости от результата $(S_1[m] \neq S_2[n])$.

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. В своих исследованиях Ф. Дамерау показал, что наиболее частой ошибкой при вводе текста является перестановка двух соседних букв слов [2]. "Цена"данной операции также равняется 1. При вычислении расстояния Левенштейна в такой ситуации потребовалось бы дважды заменить символ. Суммарная цена этих двух операций равнялась бы 2, а транспозиция добавляет в суммарную цену лишь 1. Исходя из этого, можно утверждать, что расстояние Дамерау-Левенштейна даёт лучший результат в сравнении с расстоянием Левенштейна.

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

$$D(S_1[1..m-2], S_2[1..n-2]) + 1 \tag{1.2}$$

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

$$\begin{cases} m > 2, n > 2 \\ S_1[m] = S_2[n-1] \\ S_1[m-1] = S_2[n] \end{cases} \tag{1.3}$$

Таким образом получаем следующую рекурретную формулу:

$$D(S_1[1..m], S_2[1..n]) = \begin{cases} m \; if \; n = 0 \\ n \; if \; m = 0 \\ \left[\begin{array}{ll} min \begin{cases} D(S_1[1..m-1], S_2[1..n] + 1) \\ D(S_1[1..m], S_2[1..n-1] + 1) \\ D(S_1[1..m-1], S_2[1..n-1] + (S_1[m] \neq S_2[n])) \\ D(S_1[1..m-2], S_2[1..n-2]) + 1 \end{cases} & \text{if } (1.3) \\ min \begin{cases} D(S_1[1..m-1], S_2[1..n] + 1) \\ D(S_1[1..m], S_2[1..n-1] + 1) \\ D(S_1[1..m-1], S_2[1..n-1] + (S_1[m] \neq S_2[n])) \end{cases} & \text{otherwise} \end{array} \right. \end{cases} \tag{1.4}$$

2. Конструкторская часть

2.1 Схемы алгоритмов

Рис. 2.1: Алгоритм Левенштейна

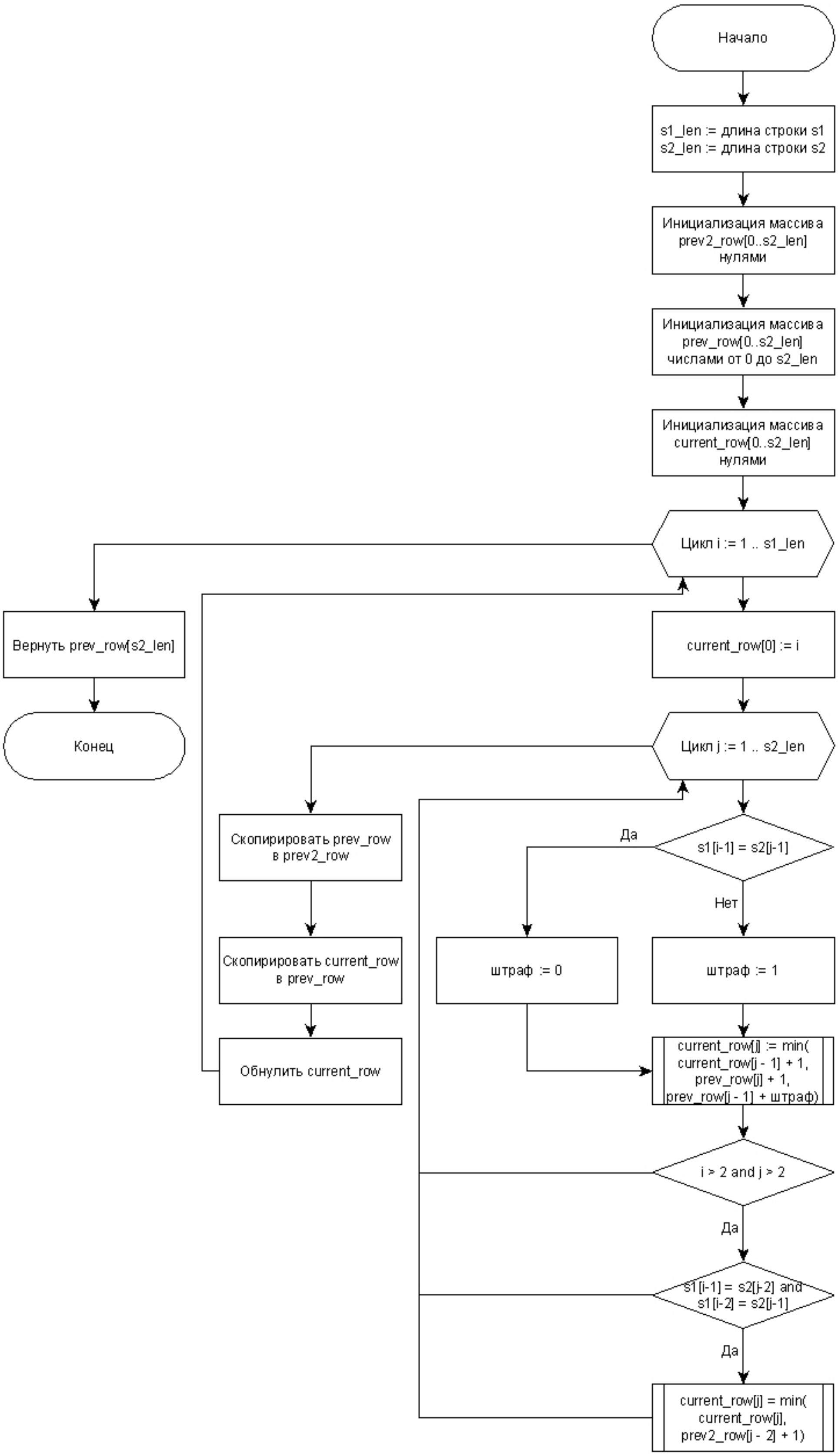


Рис. 2.2: Матричный алгоритм Дамерау-Левенштейна

2.2 Оценка используемой памяти алгоритмами

В каждом алгоритме используются символьные массивы для хранения строк. Пусть для хранения символа используется один байт. Тогда для всех алгоритмов для хранения строк длин m, n требуется $N + M$

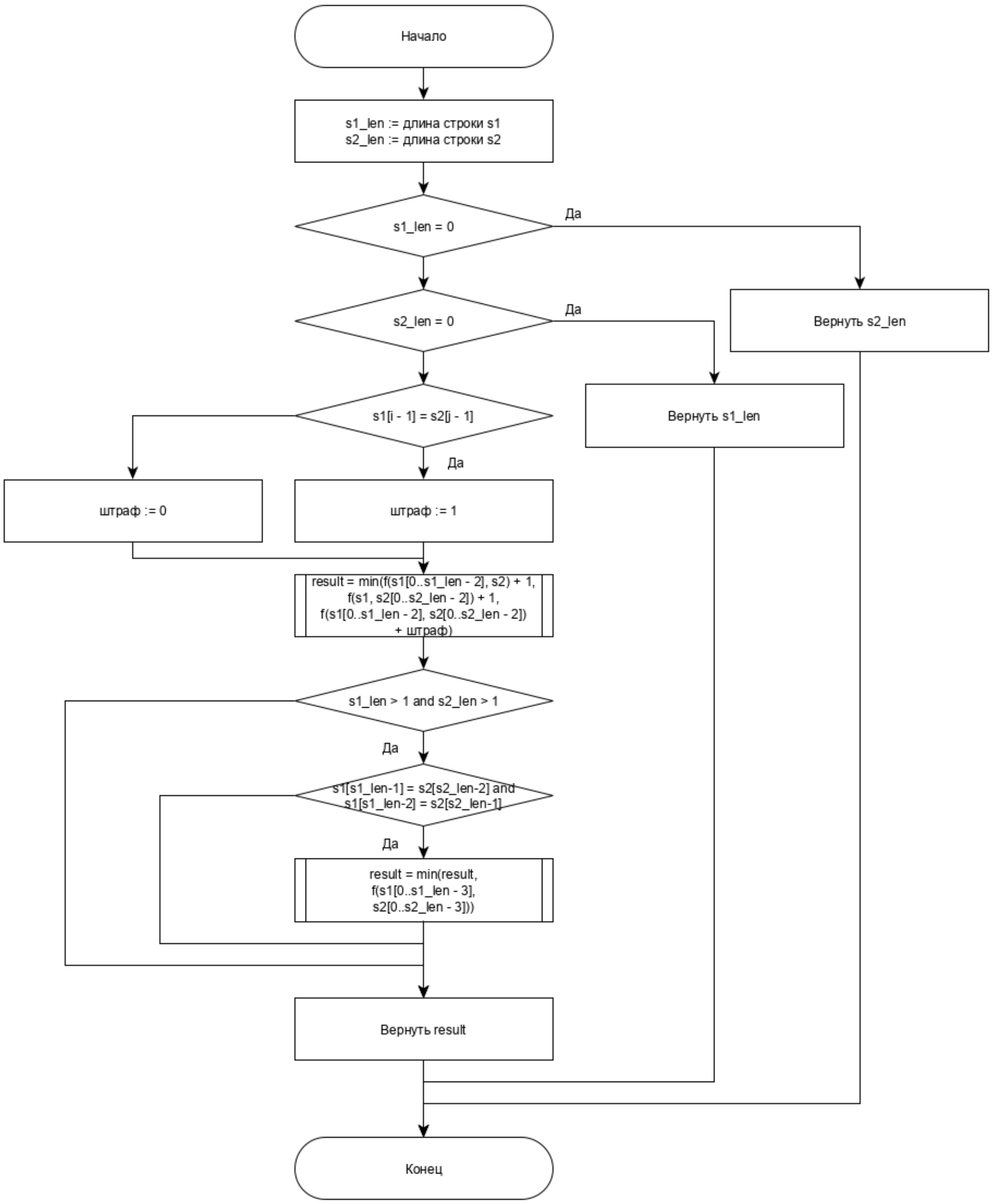


Рис. 2.3: Рекурсивный алгоритм Дамерау-Левенштейна

байтов памяти.
В матричном алгоритме Левенштейна дополнительно требуется два массива под хранение значений текущей и предыдущей строк обрабатываемой матрицы. Строки матрицы имеют длину $M + 1$, для хранения целого числа требуется 4 байта, следовательно требуется $8(M + 1)$ байтов памяти.
В матричном алгоритме Дамерау-Левенштейна требуется также хранить идущую за предыдущей строку при выполнении операции транспозиции, следовательно суммарно требуется $12(M + 1)$ байтов памяти.
В итоге для матричных алгоритмов Левенштейна и Дамерау-Левенштейна требуется $N + 9(M + 1)$ и $N + 13(M + 1)$ байтов памяти соответственно.

В рекурсивном алгоритме каждый раз при вызове функции создаются новые параметры. При сокращении первой строки будет выделено

3. Технологическая часть

3.1 Требования к программному обеспечению

На вход подаются две строки, символы которой входят в таблицу Юникода (UTF-16). На выход программа выдаёт три числовых значения, которые являются результатами вычисления расстояний тремя методам: матричными алгоритмоми Левенштейна и Дамерау-Левенштейна и рекурсивным алгоритмом Дамерау-Левенштейна. В качестве результата для матричных алгоритмов также выводится матрица вычислений.

3.2 Средства реализации

Для реализации программы был использован язык Python [3], так как данный язык позволяет проще работать с символьными строками: для того чтобы отбросить последние символы строки (что требуется при реализации рекурсивного алгоритма) имеется возможность использовать встроенные методы - срезы. Для написания функции замера времени был использован язык C [4], так как данный язык имеет в составе стандартной библиотеки встроенную функцию замера процессорного времени в тиках.

3.3 Листинг кода

Листинг 3.1: Расстояние Левенштейна (матрично)

```
def str_distance(s1, s2, to_print=False):
    s1_len = len(s1)
    s2_len = len(s2)

    # initialization of first two rows
    prev_row = [i for i in range(s2_len + 1)] # first row - [0, 1, ..., n]
    current_row = [0] * (s2_len + 1)

    if to_print:
        print(prev_row)

    for i in range(1, s1_len + 1): # row loop
        # current row fill
        current_row[0] = i
        for j in range(1, s2_len + 1): # column loop
            match_fault = int(s1[i - 1] != s2[j - 1]) # symbol match
            current_row[j] = min(current_row[j - 1] + 1, # horizontal
                                 prev_row[j] + 1, # vertical
                                 prev_row[j - 1] + match_fault) # diagonal

        if to_print:
            print(current_row)

        # row switching
        prev_row = current_row
        current_row = [0] * (s2_len + 1)

    return prev_row[-1] # value in bottom right corner of table
```

Листинг 3.2: Расстояние Дамерау-Левенштейна (матрично)

```
def str_distance(s1, s2, to_print=False):
```

```
    s1_len = len(s1)
    s2_len = len(s2)

    # initialization of first two rows
    prev2_row = [0] * (s2_len + 1)
    prev_row = [i for i in range(s2_len + 1)]
    current_row = [0] * (s2_len + 1)

    if to_print:
        print(prev_row)

    for i in range(1, s1_len + 1): # row loop
        # current row fill
        current_row[0] = i
        for j in range(1, s2_len + 1): # column loop
            match_fault = int(s1[i - 1] != s2[j - 1]) # if symbol matches
            current_row[j] = min(current_row[j - 1] + 1, # horizontal
                                 prev_row[j] + 1, # vertical
                                 prev_row[j - 1] + match_fault) # diagonal

            # transposition check
            if i > 2 and j > 2:
                if s1[i - 1] == s2[j - 2] and s1[i - 2] == s2[j - 1]:
                    current_row[j] = min(current_row[j],
                                         prev2_row[j - 2] + 1)

        if to_print:
            print(current_row)

        # row switching
        prev2_row = prev_row
        prev_row = current_row
        current_row = [0] * (s2_len + 1)

    return prev_row[-1] # value in bottom right corner of table
```

Листинг 3.3: Расстояние Дамерау-Левенштейна (рекурсивно)

```
def str_distance(s1, s2):
    s1_len = len(s1)
    s2_len = len(s2)

    if s1_len == 0:
        return s2_len
    if s2_len == 0:
        return s1_len

    match_fault = int(s1[-1] != s2[-1])

    result = min(str_distance(s1[:-1], s2) + 1,
                 str_distance(s1, s2[:-1]) + 1,
                 str_distance(s1[:-1], s2[:-1]) + match_fault)

    if s1_len > 1 and s2_len > 1:
        if s1[-1] == s2[-2] and s1[-2] == s2[-1]:
            result = min(result, str_distance(s1[:-2], s2[:-2]) + 1)

    return result
```

Листинг 3.4: Функция замера времени

```
unsigned long long tick(void)
{
  unsigned long long d;
  __asm__ __volatile__("rdtsc" : "=A"(d));
  return d;
```

```
}
```

3.4 Тесты

Для проверки корректности работы были подготовлены следующие функциональные тесты:

Строка 1	**Строка 2**	**Ожидание**	**Результат**
<пустая>	<пустая>	0 0 0	0 0 0
<пустая>	а	1 1 1	1 1 1
а	<пустая>	1 1 1	1 1 1
а	а	0 0 0	0 0 0
а	б	1 1 1	1 1 1
азы	базы	1 1 1	1 1 1
компютер	компьютер	1 1 1	1 1 1
данны	данные	1 1 1	1 1 1
email.ru	mail.ru	1 1 1	1 1 1
programmmer	programmer	1 1 1	1 1 1
mail.rus	mail.ru	1 1 1	1 1 1
ашибка	ошибка	1 1 1	1 1 1
алгоритм	алгорифм	1 1 1	1 1 1
копия	копии	1 1 1	1 1 1
укрсовой	курсовой	2 1 1	2 1 1
аглоритм	алгоритм	2 1 1	2 1 1
унивре	универ	2 1 1	2 1 1
курс	курсовой	4 4 4	4 4 4
курсовой	курс	4 4 4	4 4 4
курсовой	курсовик	2 2 2	2 2 2
код	закодировать	9 9 9	9 9 9
закодировать	код	9 9 9	9 9 9
ccoders	recoding	5 5 5	5 5 5
header	subheader	3 3 3	3 3 3
subheader	header	3 3 3	3 3 3
subheader	overheader	4 4 4	4 4 4

Таблица 3.1: Функциональные тесты

4. Экспериментальная часть

4.1 Примеры работы

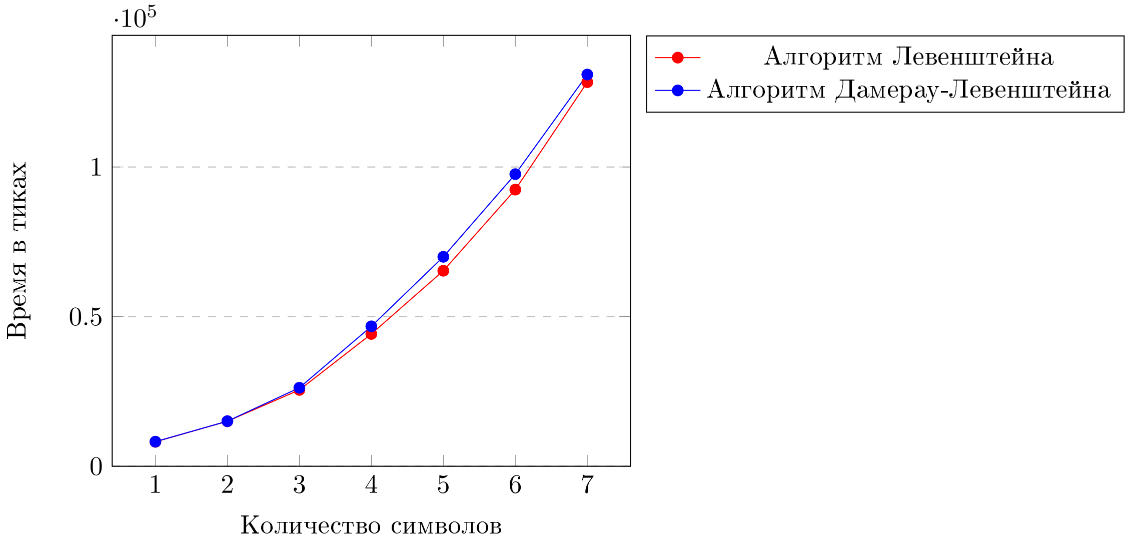
```
Введите первую строку: облако
Введите вторую строку: олбакл

Левенштейн:
[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 1, 1, 2, 3, 4]
[3, 2, 1, 2, 2, 3, 3]
[4, 3, 2, 2, 2, 3, 4]
[5, 4, 3, 3, 3, 2, 3]
[6, 5, 4, 4, 4, 3, 3]
3

Дамерау-Левенштейн (матричный):
[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 1, 1, 2, 3, 4]
[3, 2, 1, 1, 2, 3, 3]
[4, 3, 2, 2, 1, 2, 3]
[5, 4, 3, 3, 2, 1, 2]
[6, 5, 4, 4, 3, 2, 2]
2

Дамерау-Левенштейн (рекурсивный): 2
```

Рис. 4.1: Пример работы программы

4.2 Сравнение работы алгоритмов Левенштейна и Дамерау-Левенштей

Для сравнения времени работы алгоритмов Левенштейна и Дамерау-Левенштейна были использованы строки длиной от 1 до 7 с шагом 1. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов.

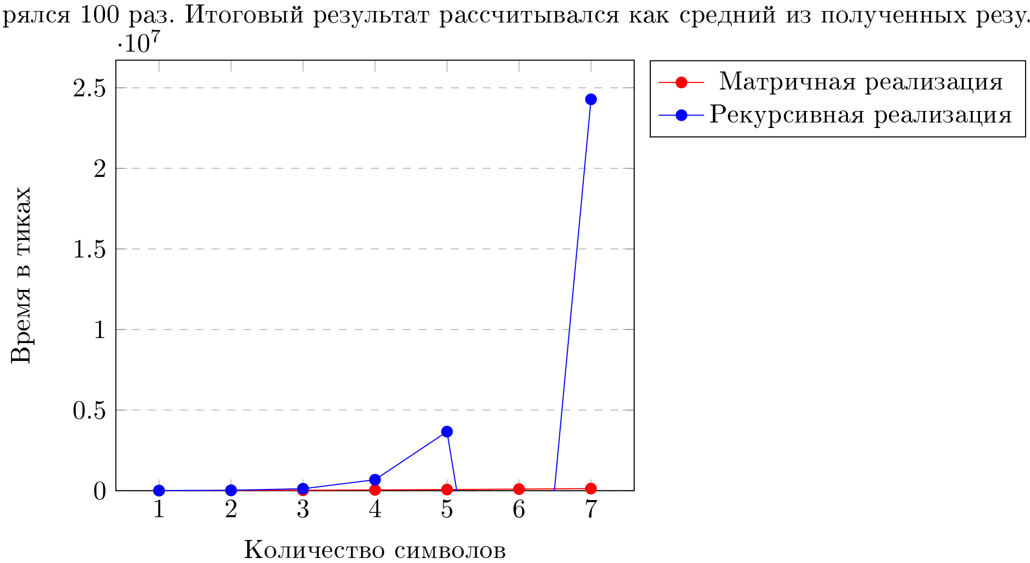
Длина слова	Левенштейн	Дамерау-Левенштейн (матричный)	Дамерау-Левенштейн (рекурсивный)
1	8158	8212	6116
2	15034	15057	25171
3	25505	26227	120905
4	44206	46749	679218
5	65335	69994	3664830
6	92444	97609	-23501767
7	128347	130896	24282663



Алгоритм Левенштейна выигрывает по времени в среднем не более, чем на 10 процентов. При этом алгоритм Дамерау-Левенштейна даёт наилучший результат. Исходя из этого, можно сделать вывод о том, что из матричных методов эффективнее использовать алгоритм Дамерау-Левенштейна.

4.3 Сравнение работы реализаций алгоритма Дамерау-Левенштейна

Для сравнения времени работы матричной и рекурсивной реализаций алгоритма Дамерау-Левенштейна были использованы строки длиной от 1 до 7 с шагом 1. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов.



Время выполнения рекурсивного алгоритма резко возрастает с увеличением длины слов: так при длине слова 5 рекурсивный алгоритм выполняется в 50 раз дольше, чем матричный. Рекурсивный алгоритм выигрывает по времени только при длине слов, равной 1 (на 37 процентов). Можно сделать вывод о том, что матричный алгоритма значительно эффективнее рекурсивного.

Заключение

Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна между строками были изучены и реализованы: были реализованы три варианта алгоритма для получения навыка динамического программирования.
Были исследованы затраты данных вариантов реализации по времени и памяти. Экспериментально было подтверждено, что рекурсивный вариант реализации алгоритма значительно проигрывает матричным вариантам при росте длины входных строк по обоим показателям.

Литература

[1] Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. В. И. Левенштейн.

[2] A technique for computer detection and correction of spelling errors. Damerau Fred J.

[3] https://www.python.org/doc/ [Электронный ресурс]

[4] https://creference.com/ [Электронный ресурс]

[5] Indexing methods for approximate dictionary searching. Journal of Experimental Algorithmics, 2011. L. M. Boytsov