Федеральное государственное бюджетное образовательное учреждение
высшего образования «Московский государственный технический университет
имени Н. Э. Баумана (национальный исследовательский университет»)

Курс: «Анализ алгоритмов»

Лабораторная работа №1

Тема работы: «Расстояние Левенштейна и Дамерау-Левенштейна»

Студент: Волков Е. А.
Преподаватели: Волкова Л. Л.
Строганов Ю. В.
Группа: ИУ7-55Б

2019

Содержание

Введение

При создании программного продукта может возникнуть необходимость сравнить текстовый запрос пользователя с некоторой базой слов и если совпадение не будет найдено, найти ближайшие к написанному слова.

Цель лабораторной работы: изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

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 временно́й эффективности рекурсивной и нерекурсивной 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 Аналитический раздел

В данном разделе анализируются наиболее популярные алгоритмы для нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Описание алгоритмов

Определим редакторские операции над строкой s:

1. I - вставка (insert) - цена 1;
2. D - удаление (delete) - цена 1;
3. R - замена (replace) - цена 1;
4. M - совпадение (match) - цена 0.

Расстояние Левенштейна — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Этот алгоритм активно применяется в следующих областях:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированого текста или речи);
2) для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы;
3) в биоинформатике для сравнения генов, хромосом и белков [4].

1.1.1 Расстояние Левенштейна

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

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

где $m(S_1[i], S_2[i])$ равна нулю, если $S_1[i] = S_2[j]$ и единице в противном случае. [2]

1.1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау - Левенштейна является модификацией расстояния Левенштейна. Оно получается, если к списку разрешённых операций добавить транспозицию (два соседних символа меняются местами). Цена транспозиции, также как и для вставки, удаления и замены, равняется 1 [3].

Пусть, как в формуле 1, S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна $d(S_1, S_2)$ можно подсчитать по следующей рекуррентной формуле $d(S_1, S_2) = D(M, N)$, где:

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

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2 Конструкторский раздел

В данном разделе содержатся схемы алгоритмов.

2.1 Разработка алгоритмов

На рис. 1 приведена схема матричного алгоритма Левенштейна.

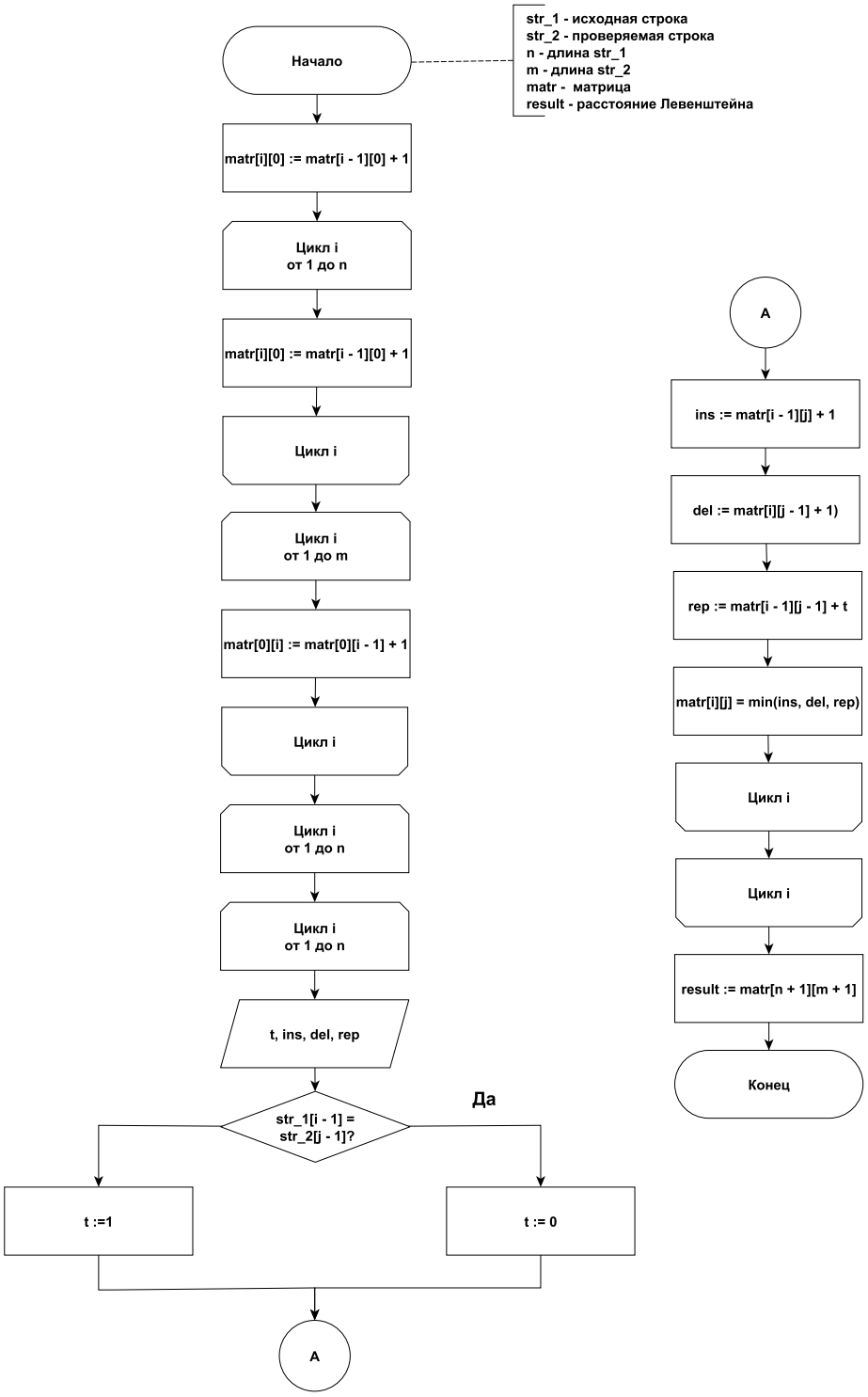

Рис. 1: Матричный алгоритм нахождения расстояния Левенштейна

На рис. 2 приведена схема матричного алгоритма Левенштейна.

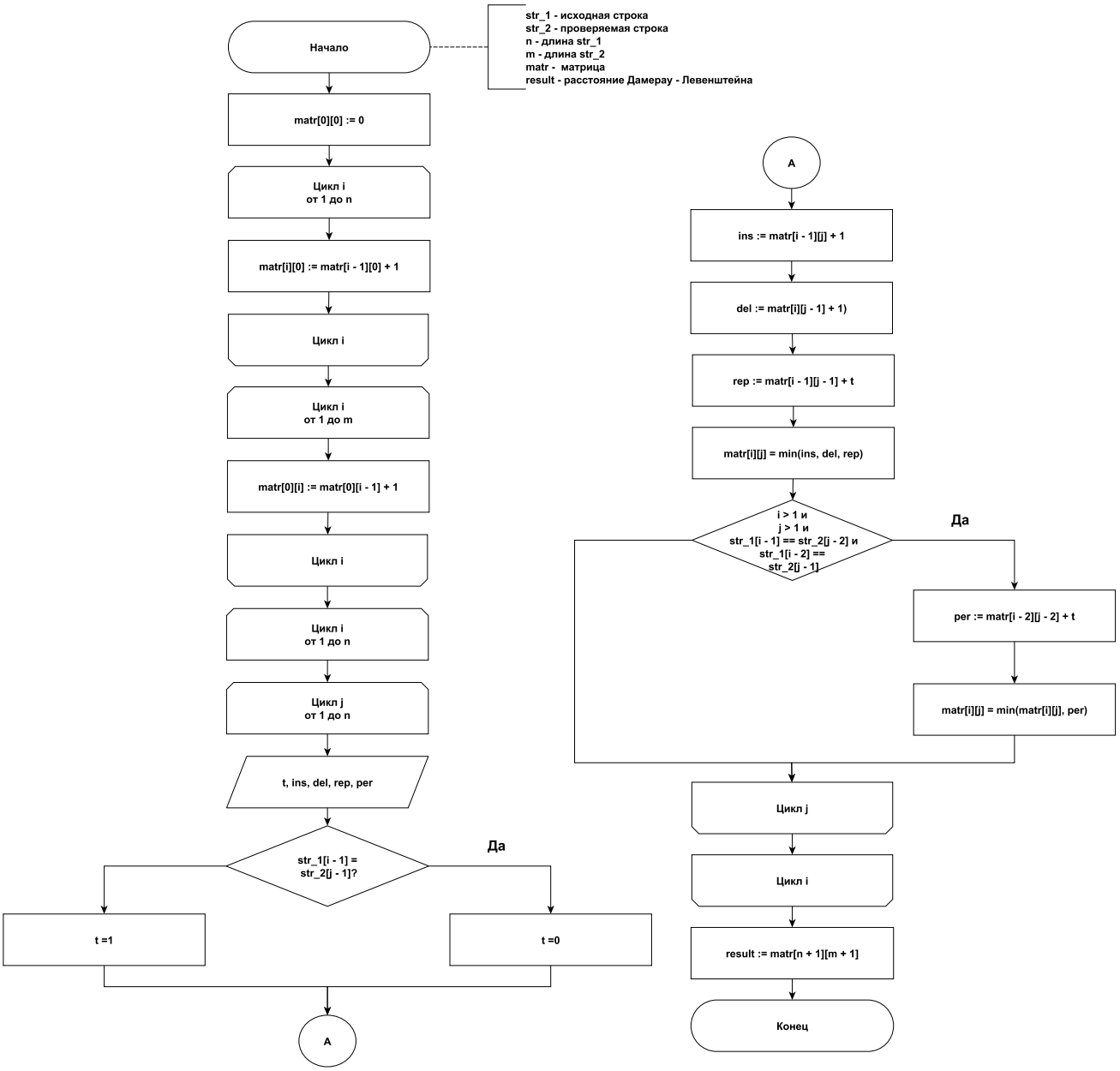

Рис. 2: Матричный алгоритм нахождения расстояния Дамерау- Левенштейна

На рис. 3 приведена схема рекурсивного алгоритма Дамерау-Левенштейна.

Вывод

В данном разделе были представлены схемы матричного алгоритма Левенштейна и матричного и рекурсивного алгоритмов Дамерау-Левенштейна.

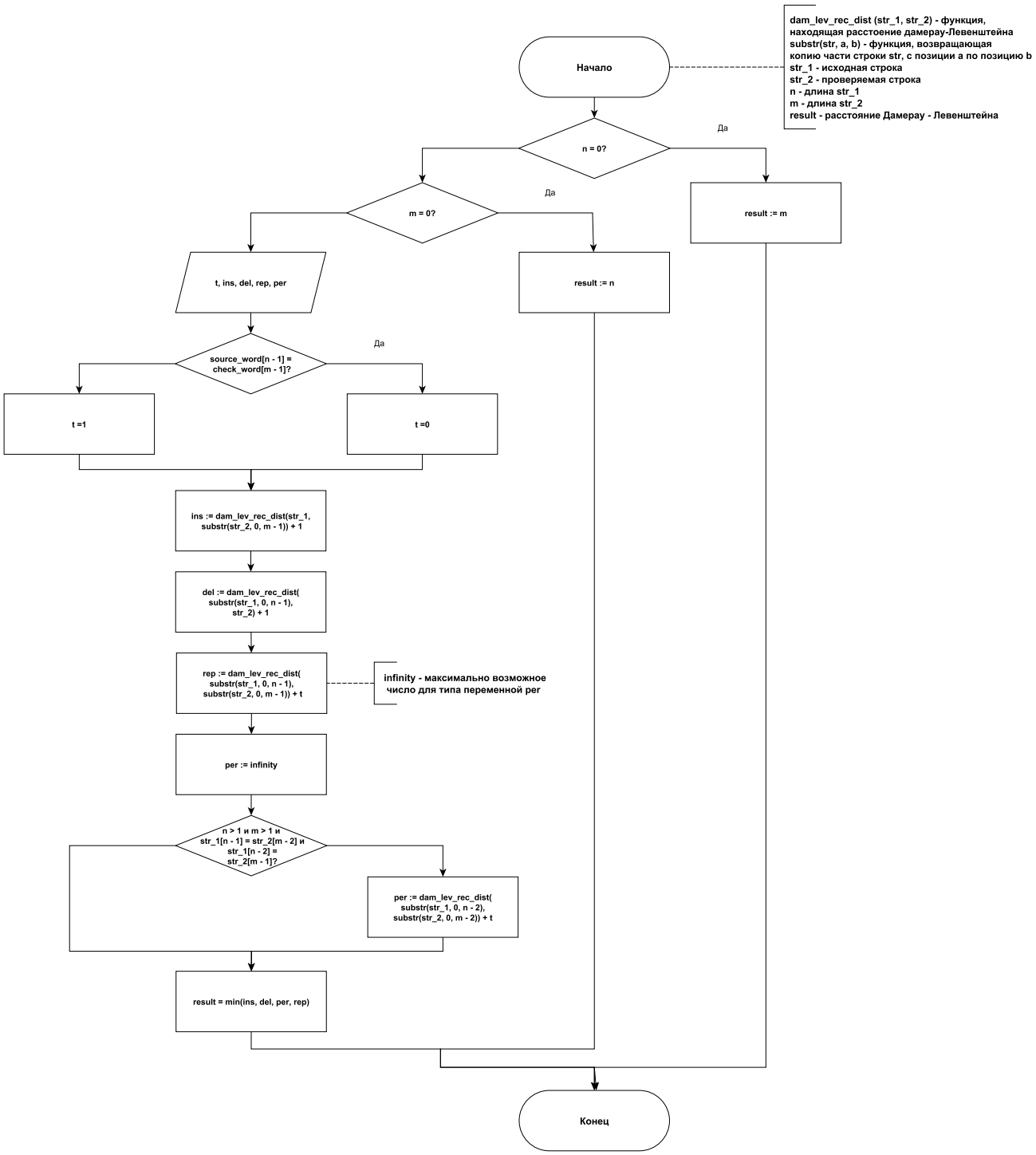

Рис. 3: Рекурсивный алгоритм нахождения расстояния Дамерау- Левенштейна

3 Технологический раздел

В данном разделе будут описаны требования к программному обеспечению и средства реализации, приведён листинг программ и проведен сравнительный анализ потребления алгоритмами памяти.

3.1 Требования к программному обеспечению

Входные данные: две строки (возможно пустые) с любыми символами.

Выходные данные: численное значение расстояний Левенштейна и Дамерау - Левенштейна

На рис. 4 приведена функциональная схема определения расстояния Левенштейна (Дамерау - Левенштейна).

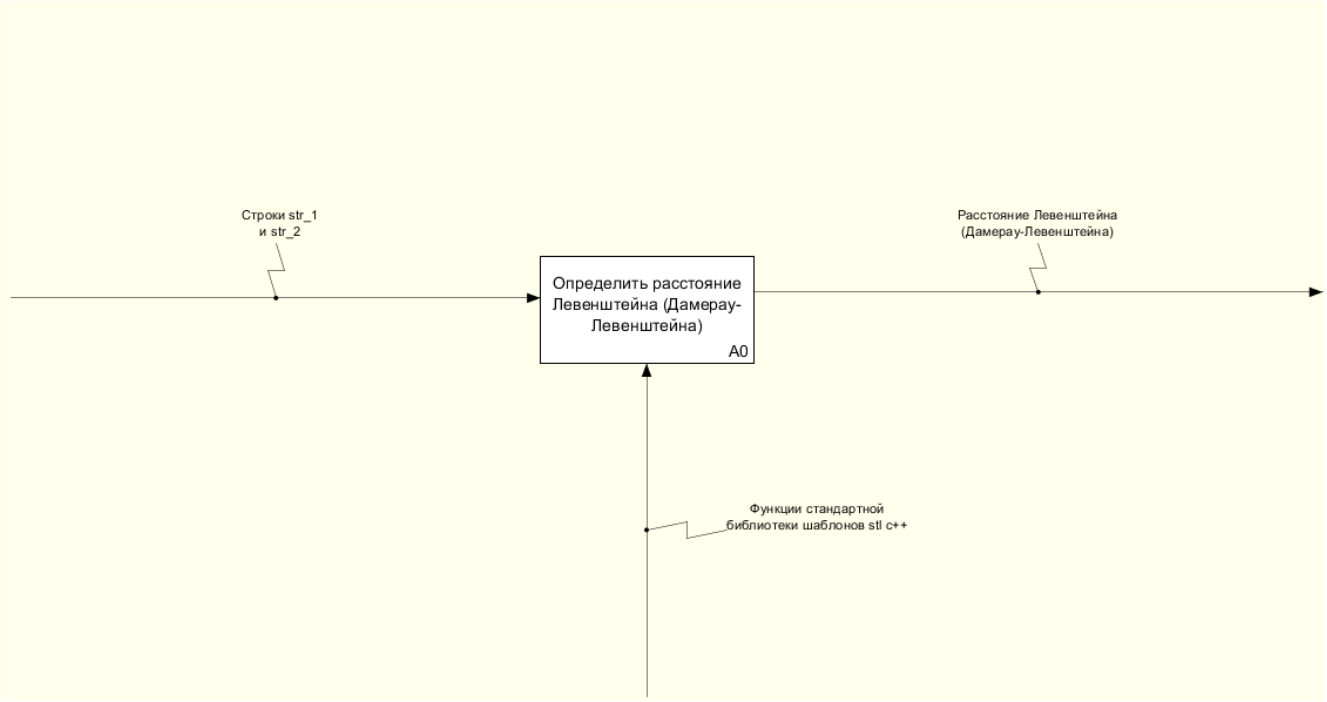

Рис. 4: Функциональная схема определения расстояния Левенштейна (Дамерау - Левенштейна)

3.2 Средства реализации

Программа написана на языке C++, т. к. этот язык предоставляет программисту широкие возможности реализации самых разнообразных алгоритмов, обладает высокой эффективностью и значительным набором стандартных классов и процедур. В качестве среды разработки использовался фреймворк QT 5.13.1.

Для обработки строк и матриц были использованы стандартные контейнерные классы std::string и std::vector.

Для замера времени выполнения программы использовалась библиотека ctime.

3.3 Листинг программы

В листингах 1, 2 и 3 представлены функции, описывающие матричную реализацию алгоритма Левенштейна и матричную и рекурсивную реализацию алгоритма Дамерау - Левенштейна

Листинг 1: Матричный алгоритм Левенштейна

```
1
2  int lev_matr_dist(string source_word, string check_word) {
3      size_t n = source_word.size() + 1;
4      size_t m = check_word.size() + 1;
5      Matrix matr(n, Vector(m, 0));
6
7      for(size_t i = 1; i < n; i++) {
8          matr[i][0] = matr[i - 1][0] + 1;
9      }
10
11     for(size_t i = 1; i < m; i++) {
12         matr[0][i] = matr[0][i - 1] + 1;
13     }
14
15     for (size_t i = 1; i < n; i++) {
16         for(size_t j = 1; j < m; j++) {
17             source_word[i - 1] == check_word[j - 1] ? t = 0 : t = 1;
18             matr[i][j] = min(min(matr[i - 1][j] + 1, matr[i][j - 1] + 1),
19                     matr[i - 1][j - 1] + t);
20         }
21     }
22
23     //print_matr(matr, source_word, check_word);
24     return matr[n - 1][m - 1];
25 }
```

Листинг 2: Матричный алгоритм Дамерау - Левенштейна

```
1
2  int dam_lev_matr_dist(string source_word, string check_word) {
3      size_t n = source_word.size() + 1;
4      size_t m = check_word.size() + 1;
5      Matrix matr(n, Vector(m, 0));
6
7      for(size_t i = 1; i < n; i++) {
8          matr[i][0] = matr[i - 1][0] + 1;
9      }
10
11     for(size_t i = 1; i < m; i++) {
12         matr[0][i] = matr[0][i - 1] + 1;
13     }
14
15     for (size_t i = 1; i < n; i++) {
16         for(size_t j = 1; j < m; j++) {
17             t = 0;
18             source_word[i - 1] == check_word[j - 1] ? t = 0 : t = 1;
```

```
            matr[i][j] = min(min(matr[i - 1][j] + 1, matr[i][j - 1] + 1),
                    matr[i - 1][j - 1] + t);

            if (i > 1 && j > 1 && source_word[i - 1] == check_word[j - 2] &&
                    source_word[i - 2] == check_word[j - 1]) {
                matr[i][j] = min(matr[i][j], matr[i - 2][j - 2] + t);
            }
        }
    }

    print_matr(matr, source_word, check_word);
    return matr[n - 1][m - 1];
}
```

Листинг 3: Матричный алгоритм Дамерау - Левенштейна

```

int dam_lev_rec_dist(string source_word, string check_word) {

    size_t n = source_word.size();
    size_t m = check_word.size();

    if (!n) {
        return m;
    }
    if (!m) {
        return n;
    }

    int t = 0;
    source_word[n - 1] == check_word[m - 1] ? t = 0 : t = 1;

    int ins = dam_lev_rec_dist(source_word, check_word.substr(0, m - 1)) +
        1;
    int del = dam_lev_rec_dist(source_word.substr(0, n - 1), check_word) +
        1;
    int rep = dam_lev_rec_dist(source_word.substr(0, n - 1),
                               check_word.substr(0, m - 1)) + t;
    int per = std::numeric_limits<short>::max();

    if (n > 1 && m > 1 && source_word[n - 1] == check_word[m - 2] &&
            source_word[n - 2] == check_word[m - 1]) {
        per = dam_lev_rec_dist(source_word.substr(0, n - 2),
                                          check_word.substr(0, m - 2)) + t;
    }

    return min(min(min(ins, del), rep), per);
}
```

3.4 Тестовые данные

Программа должна корректно находить расстояния Левенштейна и Дамерау-Левенштейна при следующих входных данных, представленных в таблице 1.

Таблица 1: Тестовые данные

№	Исходное слово	Проверяемое слово	Расстояние Левенштейна	Расстояние Дамерау - Левенштейна
1	∅	∅	0	0
2	∅	a	1	1
3	a	∅	1	1
4	a	a	1	1
5	skat	kot	2	2
6	red	erd	2	1
7	student	sutednt	3	2
8	mouse	mouse	0	0
9	elephant	relevant	3	3
10	abc	def	3	3
11	coffee	soda	1	1
12	annas	max	4	4
13	levenshtein	meilenstein	4	4
14	notebook	ontbeooko	5	3
15	market	street	3	3
16	ball	ballon	2	2
17	bbball	ball	2	2
18	doctor	odtcor	3	2
19	factory	fcatroy	4	2
20	park	krap	4	3

Расстояние Дамерау-Левенштейна, посчитанное рекурсией и расстояние Дамерау-Левенштейна, посчитанное с помощью матрицы должны совпадать.

3.5 Сравнительный анализ рекурсивной и нерекурсивной реализаций

Расчёт потребляемой памяти для матричного алгоритма Дамерау-Левенштейна для двух строк размерами n и m приведен в таблице 2:

Таблица 2: Расчёт используемой памяти для матричного алгоритма Дамерау-Левенштейна

Объект	Занимаемая память (формула)	Занимаемая память (байт)
Матрица	sizeof(std::vector)	24
Строки матрицы	(n + 1)*sizeof(std::vector)	(n + 1)*24
Ячейки матрицы	(n + 1)*(m + 1)*sizeof(int)	(n + 1)*(m + 1)*4
Вспомогательные переменные	3*sizeof(int)	3*4

Таким образом, общая формула будет выглядеть так:

$$Memory = 24 + (n + 1) * 24 + (n + 1) * (m + 1) * 4 + 3 * 4. \quad (2)$$

В алгоритме Левенштейна используется на одну вспомогательную переменную меньше, поэтому формула будет отличаться незначительно:

$$Memory = 24 + (n + 1) * 24 + (n + 1) * (m + 1) * 4 + 2 * 4.$$

Следовательно, можно считать потребление памяти матричным алгоритмом Левенштейна приблизительно равным потреблению памяти матричным алгоритмом Дамерау-Левенштейна.

В рекурсивном алгоритме Дамерау-Левенштейна максимальное количество занятой памяти зависит от глубины рекурсии. Если n >= m, то рекурсия будет иметь глубину n. Расчёт памяти для i-го рекурсивного вызова представлен в таблице 3.

Таблица 3: Расчёт используемой памяти для i-го рекурсивного вызова алгоритма Дамерау-Левенштейна

Объект	Занимаемая память (формула)	Занимаемая память (байт)
Две строки	sizeof(std::string)*2	32*2
Элементы строк	(m + n - i)*sizeof(char)	m + n - i
Область в стеке для нового рекурсивного вызова	адрес возврата и два указателя на строку	8 + 8*2
Вспомогательные переменные	7*sizeof(int)	7*4

Общая формула будет выглядеть так:

$$Memory = \sum_{i=0}^{n+m}(32 * 2 + m + n - i + 8 * 3 + 7 * 4). \quad (3)$$

Или более просто:

$$Memory = (32 * 2 + m + n + 8 * 3 + 7 * 4) * (n + m) - \sum_{i=0}^{n+m} i.$$

Оценить количество потребляемой алгоритмами памяти на строках фиксированной длины можно по таблицам 4 и 5 (возмем маленькую строку длиной 5 символов, среднюю - длиной 50 символов и большую - длиной 1000 символов).

Таким образом рекурсивный алгоритм выигрывает по памяти в сравнении с итеративными алгоритмами при обработке строк большой длины.

Вывод

В данном разделе были рассмотрены требования к программному обеспечению, в качестве средств реализации выбраны язык C++ и фреймворк QT

Таблица 4: Расчёт используемой памяти для матричного алгоритма Дамерау-Левенштейна для строк длиной 5, 50 и 1000 символов

Объект	Формула	n = 5, m = 5	n = 50, m = 50	n = 1000, m = 1000
Матрица	24	24	24	24
Строки матрицы	(n + 1)*24	144	1224	24024
Ячейки матрицы	(n + 1)*(m + 1)*4	144	10404	4008004
Вспомогательные переменные	3*4	12	12	12
Итого (байт)	24 + (n + 1)*24 + (n + 1)*(m + 1)*4 + 3*4	324	11664	4032064

Таблица 5: Расчёт используемой памяти для рекурсивного алгоритма Дамерау-Левенштейна для строк длиной 5, 50 и 1000 символов

Объект	Формула	n = 5, m = 5	n = 50, m = 50	n = 1000, m = 1000
Две строки	32*2*(n + m)	640	6400	128000
Элементы строк	$(m + n) * (m + n) - \sum_{i=0}^{n+m} i$	55	4950	1999000
Область в стеке для нового рекурсивного вызова	8*3*(n + m)	240	2400	48000
Вспомогательные переменные	7*4*(n + m)	280	2800	56000
Итого (байт)	$(32*2+m+n+8*3+7*4*(m+n)-\sum_{i=0}^{n} i$	1215	16550	2231000

версии 5.13.1, приведён листинг программы и тестовые данные. Было проанализировано потребление каждым из алгоритмов памяти, были выведены формулы для вычисления памяти на строках определенной длины. Аналитическое сравнение потребления памяти алгоритмами на строках длины 5, 50 и 1000 символов показало, что матричный алгоритм Дамерау-Левенштейна эффективен на строках длиной 5 и 50 символов, а рекурсивный - на строках длиной 1000 символов.

4 Исследовательский раздел

4.1 Примеры работы

На рис. 5 приведен пример работы программы с пустой строкой.

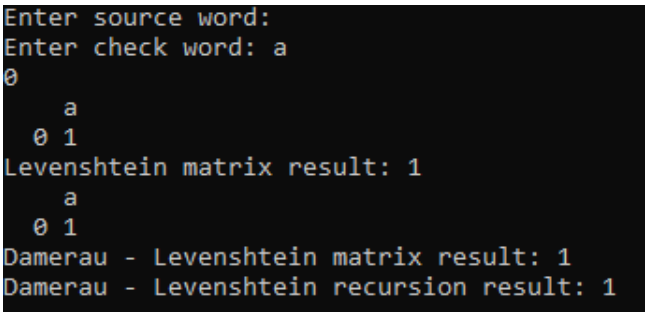

Рис. 5: Пример работы программы для пустого слова и слова "a"

На рис. 6 приведен пример со словами, не имеющими переставленных символов (алгоритмы Левенштейна и Дамерау - Левенштейна дают одинаковые результаты).

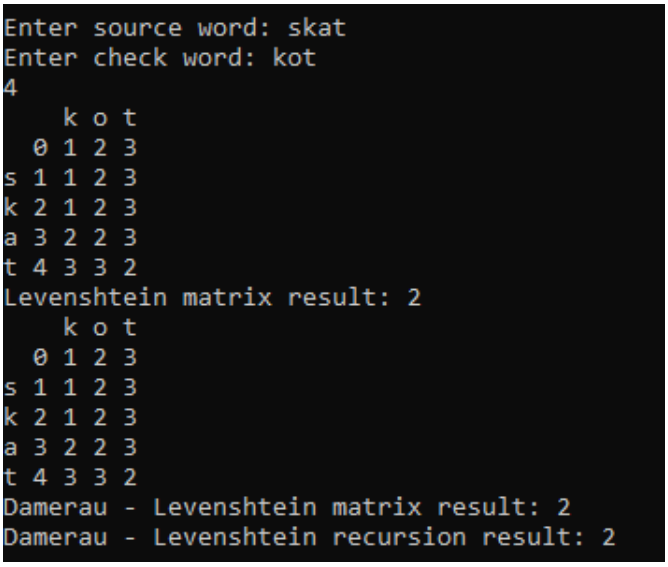

Рис. 6: Пример работы программы для слов "skat"и "kot"

На рис. 7 приведен пример со словами, имеющими перестановки(алгоритмы Левенштейна и Дамерау - Левенштейна дают разные результаты).

4.2 Результаты тестирования

Все тесты из раздела 3.4 успешно пройдены.

4.3 Постановка эксперимента

Необходимо выполнить следующие замеры времени:

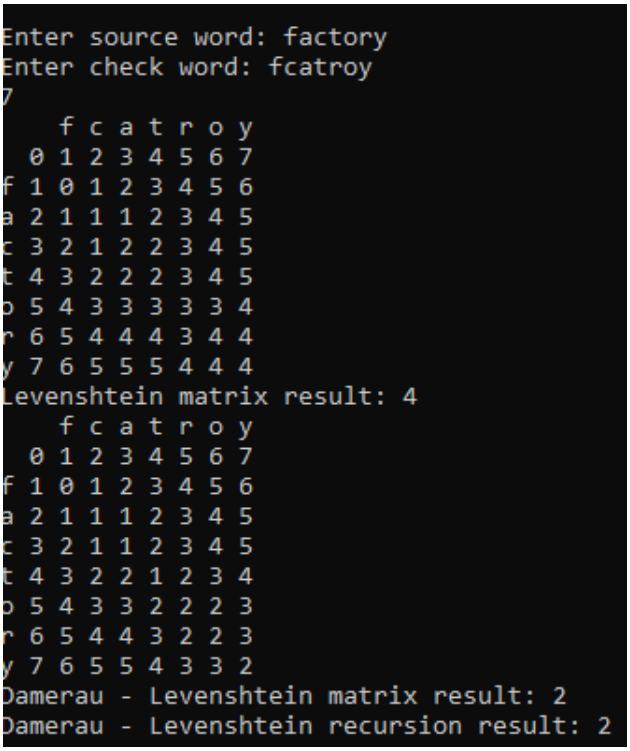

Рис. 7: Пример работы программы для слов "factory"и "fcatroy"

1. сравнить время вычисления расстояния Левенштейна матрично и вычисление расстояния Дамерау-Левенштейна матрично на строках длины от 1 до 1001 с шагом 100;

2. сравнить время вычисления расстояния Дамерау-Левенштейна рекурсивно и вычисление расстояния Дамерау-Левенштейна матрично на строках длины от 1 до 10 с шагом 1.

4.4 Сравнительный анализ на материале экспериментальных данных

На рис. 8 приводятся графики времени выполнения матричных Левенштейна и Дамерау-Левенштейна.

Как видно, алгоритм Левенштейна немного выигрывает у алгоритма Дамерау-Левенштейна (на длине в 1000 символов эффективнее на 20%), так как дополнительная проверка на перестановку символов увеличивает время работы алгоритма Дамерау-Левенштейна. Также видно, что обе функции имеют один характер роста.

На рис. 9 приводятся графики времени выполнения матричного и рекурсивного Дамерау-Левенштейна.

Как видно из этих зависимостей, уже на строках длины 8, рекурсивный алгоритм начинает проигрывать.Время выполнения рекурсивного алгоритма растет экспоненциально, пропорционально количеству рекурсивных вызовов.

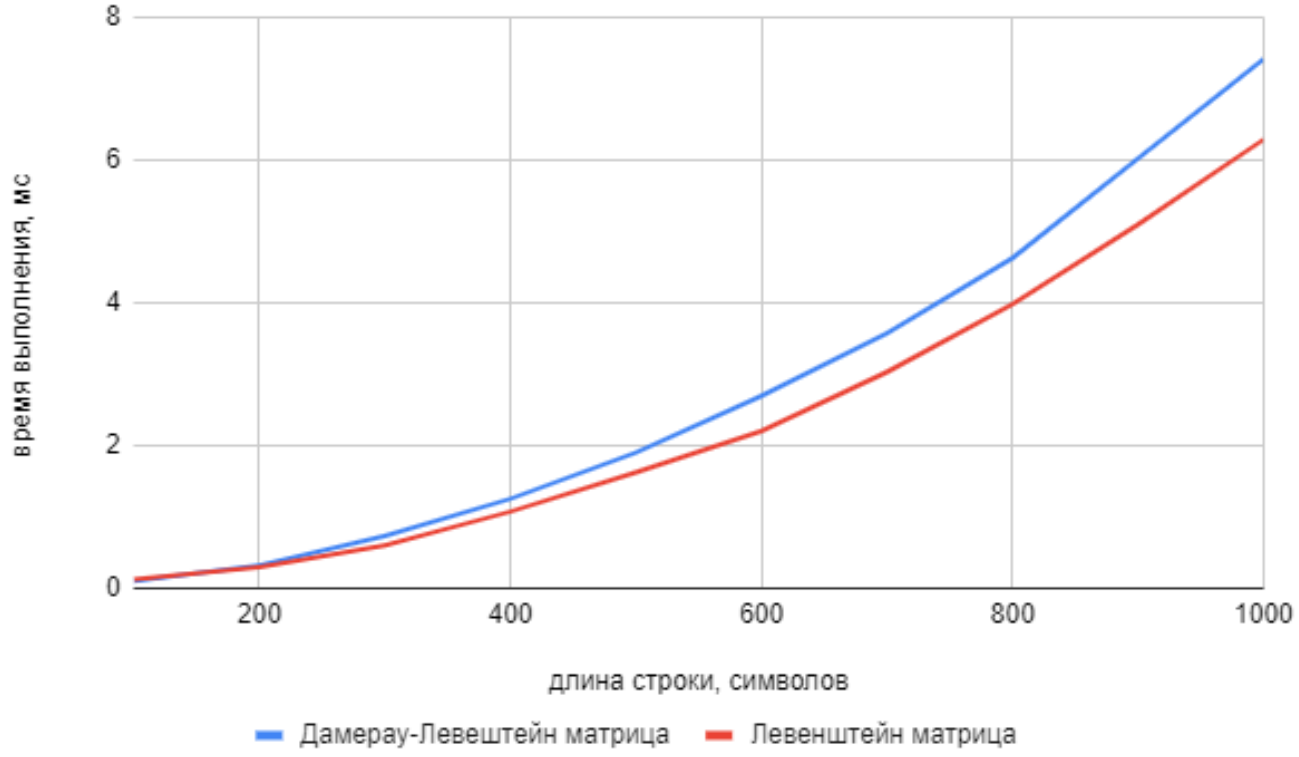

Рис. 8: Сравнение времени выполнения для матричного Левенштейна и матричного Дамерау-Левенштейна

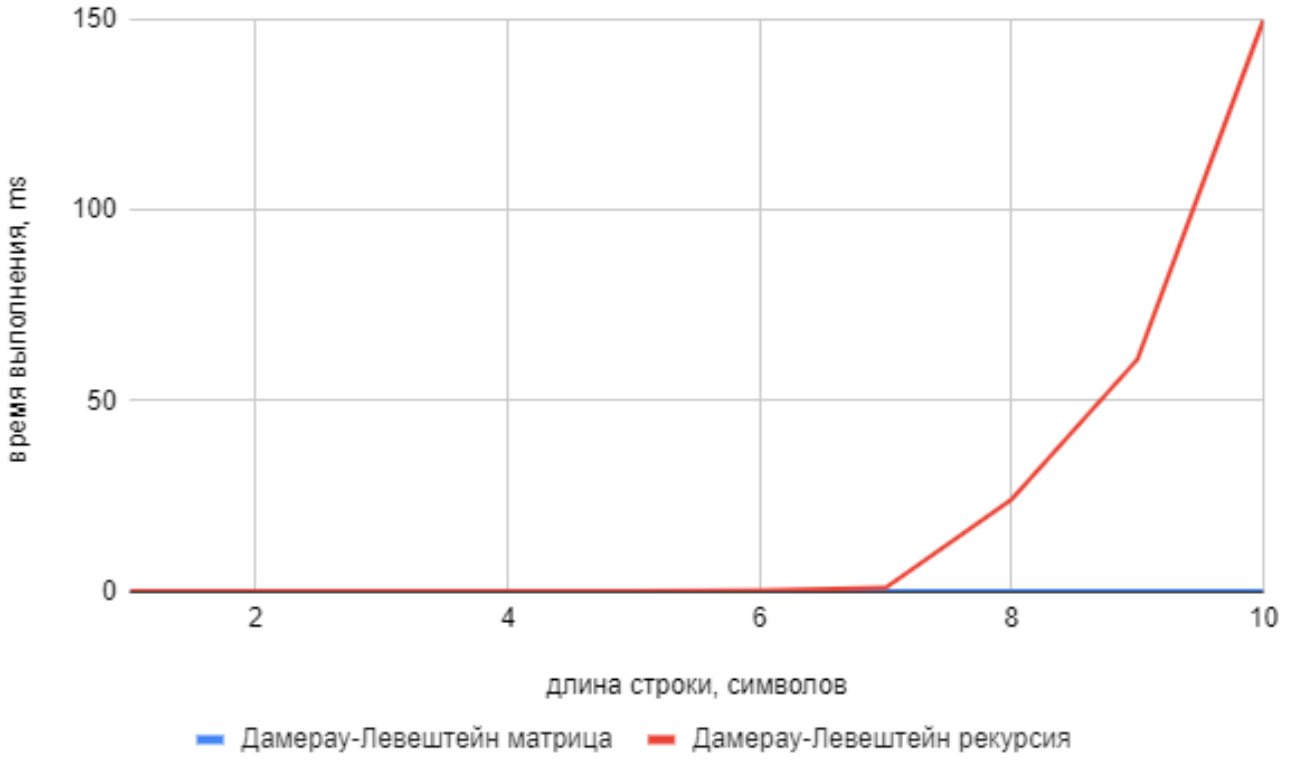

Рис. 9: Сравнение времени выполнения для матричного Дамерау-Левенштейна и рекурсивного Дамерау-Левенштейна

Проблемой рекурсивного алгоритма является и то, что параллельные вызовы могут искать расстояния между одними и теми же строками, выполняя одина-

ковые вычисления много раз подряд. В связи с этим, использование рекурсивного алгоритма на строках большой длины нецелесообразно.

На рисунке 10 представлен график потребления алгоритмами памяти, построенные на основе формул 2 и 3 на строках длины от 1 до 100 с шагом 1.

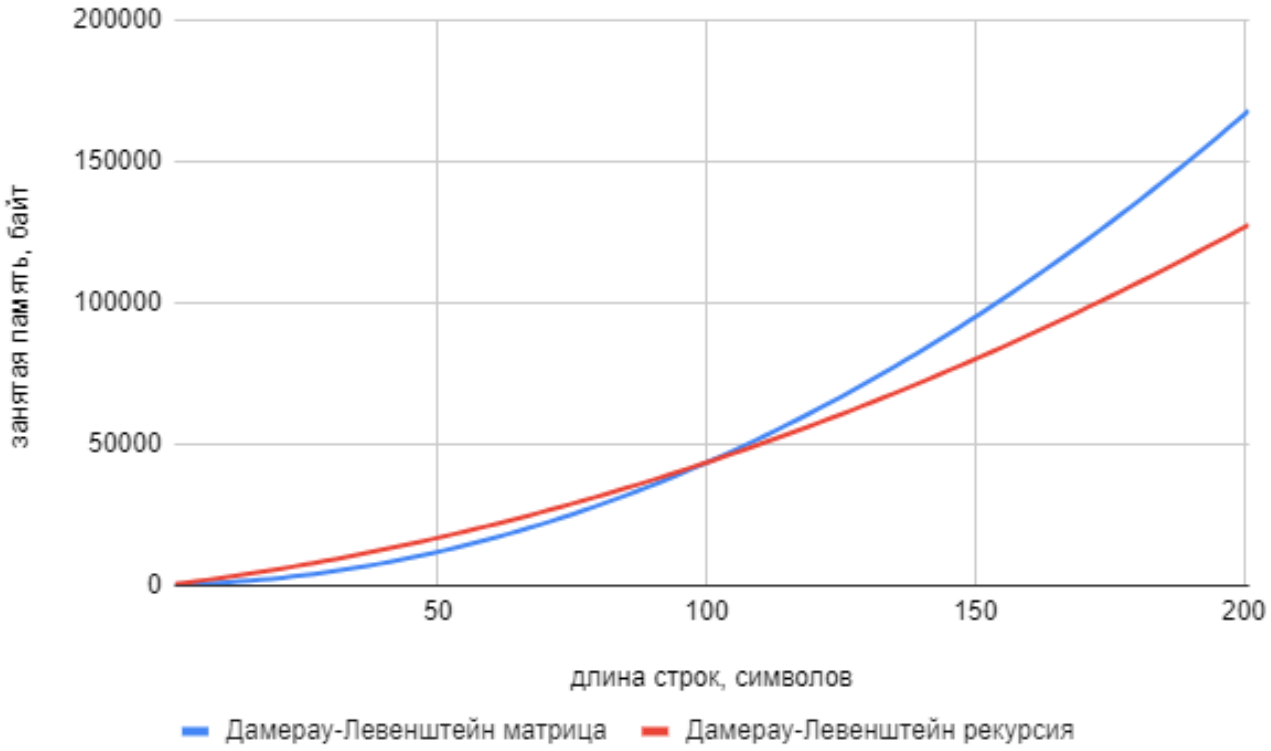

Рис. 10: Сравнение используемой памяти для матричного Дамерау-Левенштейна и рекурсивного Дамерау-Левенштейна

Как видно, на строках длиной до приблизительно 100 символов, эффективнее оказывается матричный алгоритм, при более длинных строках эффективнее рекурсивный (на длине в 200 символов эффективнее на 30%).

Вывод

В результате тестирования программой были успешно пройдены все заявленные тесты. Эксперименты замера времени показали, что самым эффективным по времени выполнения является матричный алгоритм Левенштейна, а самым убыточным рекурсивный алгоритм Дамерау-Левенштейна, время выполнения которого растет экспоненциально. Построение графика потребления алгоритмами памяти по формулам из технологического раздела показало, что для строк длиной до 100 символов матричный алгоритм расходует меньше памяти, а на строках большей длины эффективнее себя показывает рекурсивный алгоритм Дамерау-Левенштейна.

Заключение

В ходе данной работы было выполнено следующее:

1) изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2) применены методы динамического программирования для матричной реализации указанных алгоритмов;
3) получены практические навыки реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4) проведен сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5) экспериментально подтверждено различие во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6) описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе.

Экспериментальный анализ эффективности алгоритмов по времени показал, что самым эффективным является матричный алгоритм Левенштейна, а самым убыточным рекурсивный алгоритм Дамерау-Левенштейна, использование которого на строках длинней 8-10 символов нецелесообразно. Матричный алгоритм Дамерау-Левенштейна незначительно (на 1000 символов приблизительно на 20%) проигрывает алгоритму Левенштейна, но это оправдано, т. к. алгоритм Дамерау-Левенштейна решает более широкую задачу.

Сравнение потребления алгоритмами памяти было проведено аналитически. До строк длиной 100 символов матричный алгоритм расходует меньше памяти, на строках большей длины эффективнее себя показывает рекурсивный алгоритм Дамерау-Левенштейна (так, матричные алгоритмы хуже на 30% на строках длиной 200). Однако, выигрыш в использовании памяти не оправдывает практическое применение рекурсивного алгоритма на больших объемах данных.

Список литературы

[1] Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход.- М.:Техносфера, 2009.

[2] Методика идентификации пассажира по установленным данным В. М . Черненький , Ю. Е . Гапанюк [Электронный ресурс]. – Режим доступа: http://www.engjournal.ru/articles/89/89.pdf, свободный – (07.10.2019)

[3] Библиография в LaTeX [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/114997/, свободный – (26.09.2019) https://habr.com/ru/post/114997/

[4] Нечеткий поиск, расстояние левенштейна алгоритм [Электронный ресурс]. – Режим доступа: https://steptosleep.ru/antananarivo-106/, свободный – (01.10.2019)