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

Название: Расстояние Левенштейна и Дамерау – Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-54б _____ Ларин В.Н.
(группа) (подпись, дата) (фамилия, и.о.)

Преподаватель _____ Волкова Л.Л.
(подпись, дата) (фамилия, и.о.)

Москва, 2021

СОДЕРЖАНИЕ

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.

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

Цель данной лабораторной работы:

1) Изучение метода динамического программирования на материале алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

2) Оценка реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Для достижения данных целей были выделены следущие задачи:

1) Изучение алгоритмов Левенштейна и Дамерау-Левенштейна;

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

3) Получение практических навыков реализации указанных алгоритмов: матричные и рекурсивные 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;

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

1 Аналитический раздел

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

Цены операций могут зависеть от вида операции (вставка (insert), удаление (delete), замена (replace) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

— $w(a,b)$ — цена замены символа a на символ b.

— $w(\lambda,b)$ — цена вставки символа b.

— $w(a,\lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

— $w(a,a) = 0$.

— $w(a,b) = 1,\ a \neq b$.

— $w(\lambda,b) = 1$.

— $w(a,\lambda) = 1$.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i,j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \text{i > 0, j > 0} \\ \quad D(i-1,j-1)+m(a[i],b[j]) & (1.2) \\ \} & \end{cases}, \tag{1.1}$$

а функция 1.2 определена как:

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

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена из следующих соображений:

1) Для перевода из пустой строки в пустую требуется ноль операций;

2) Для перевода из пустой строки в строку a требуется $|a|$ операций;

3) Для перевода из строки a в пустую требуется $|a|$ операций;

4) Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

   а) Сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

   б) Сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b;

   в) Сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;

   г) 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.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i,j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i,j)$.

Также можно заметить, что для заполнения каждой новой строки матрицы, необходима только одна предыдущая строка, что дает дополнительную оптимизацию алгоритма по памяти.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшированием

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с помощью кэширования. Суть данного метода заключается в заполнении матрицы при выполнении рекурсии. 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 возвращается полученное ранее значение.

1.4 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1, & \\ \quad d_{a,b}(i-1,j)+1, & \\ \quad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \quad (1.3)$$

Формула выводится по тем же соображениям, что и формула (1.1). Как и в случае с рекурсивным методом, прямое применение этой формулы неэффективно по времени исполнения, то аналогично методу из 1.3 производится добавление матрицы для хранения промежуточных значений рекурсивной формулы.

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего

возможность перестановки соседних символов. Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторский раздел

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояние Левештейна и Дамерау - Левенштейна. На рисунках 2.1, 2.2, 2.3 представлены рассматриваемые алгоритмы.

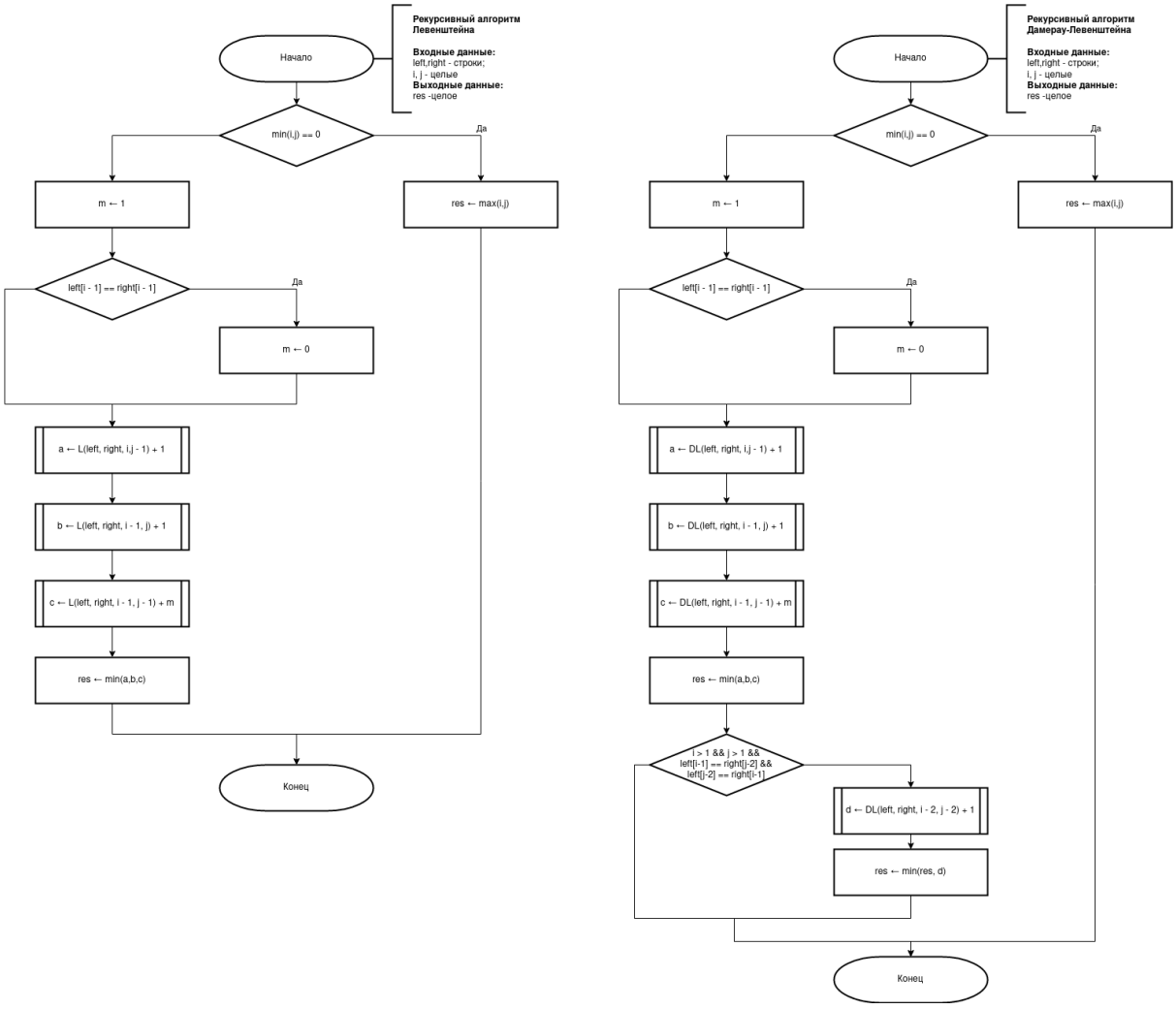

Рисунок 2.1 — Схема рекурсивных алгоритмов Левенштейна и Дамерау–Левенштейна

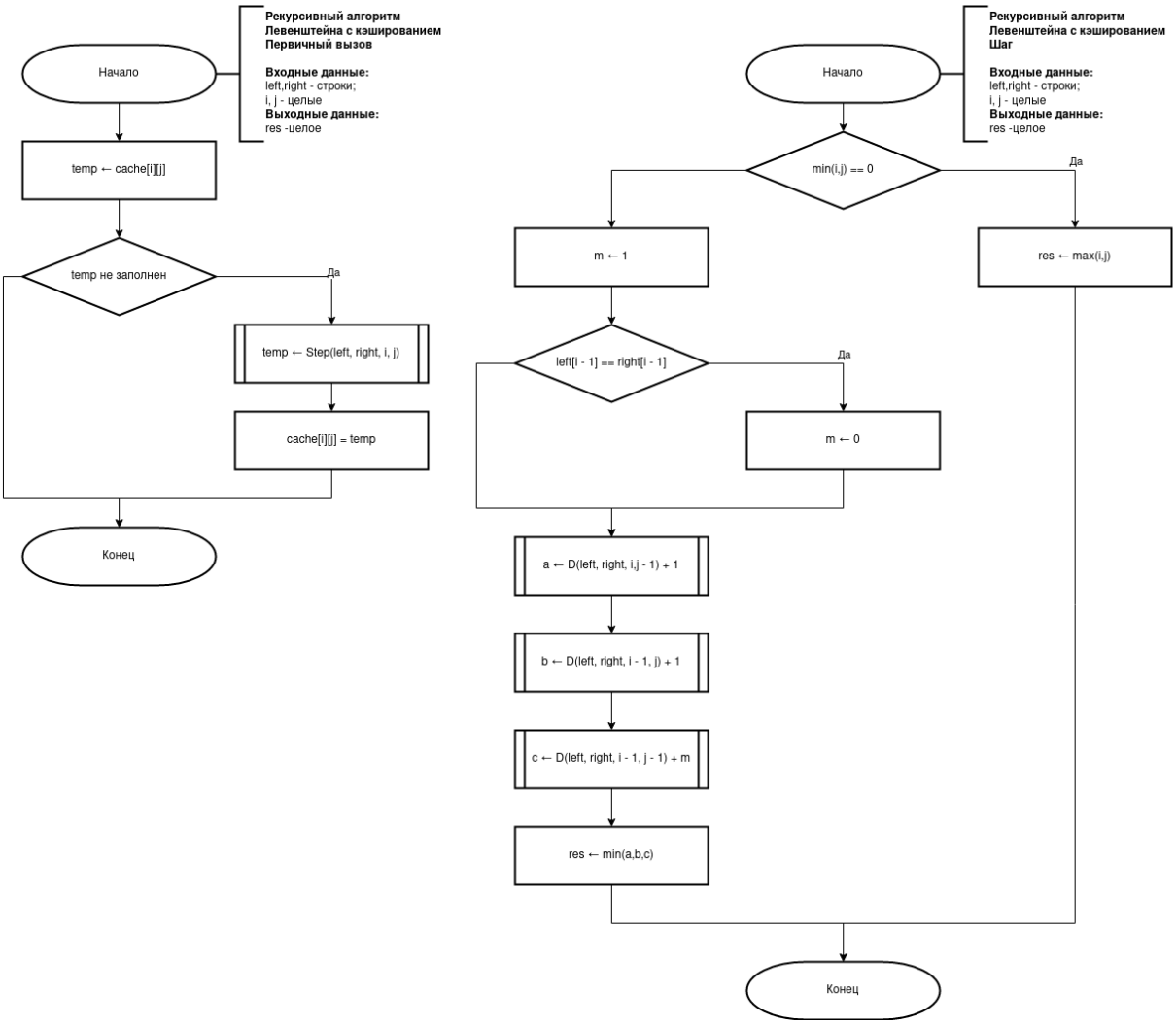

Рисунок 2.2 — Схема рекурсивного алгоритма Левенштейна с кэшированием

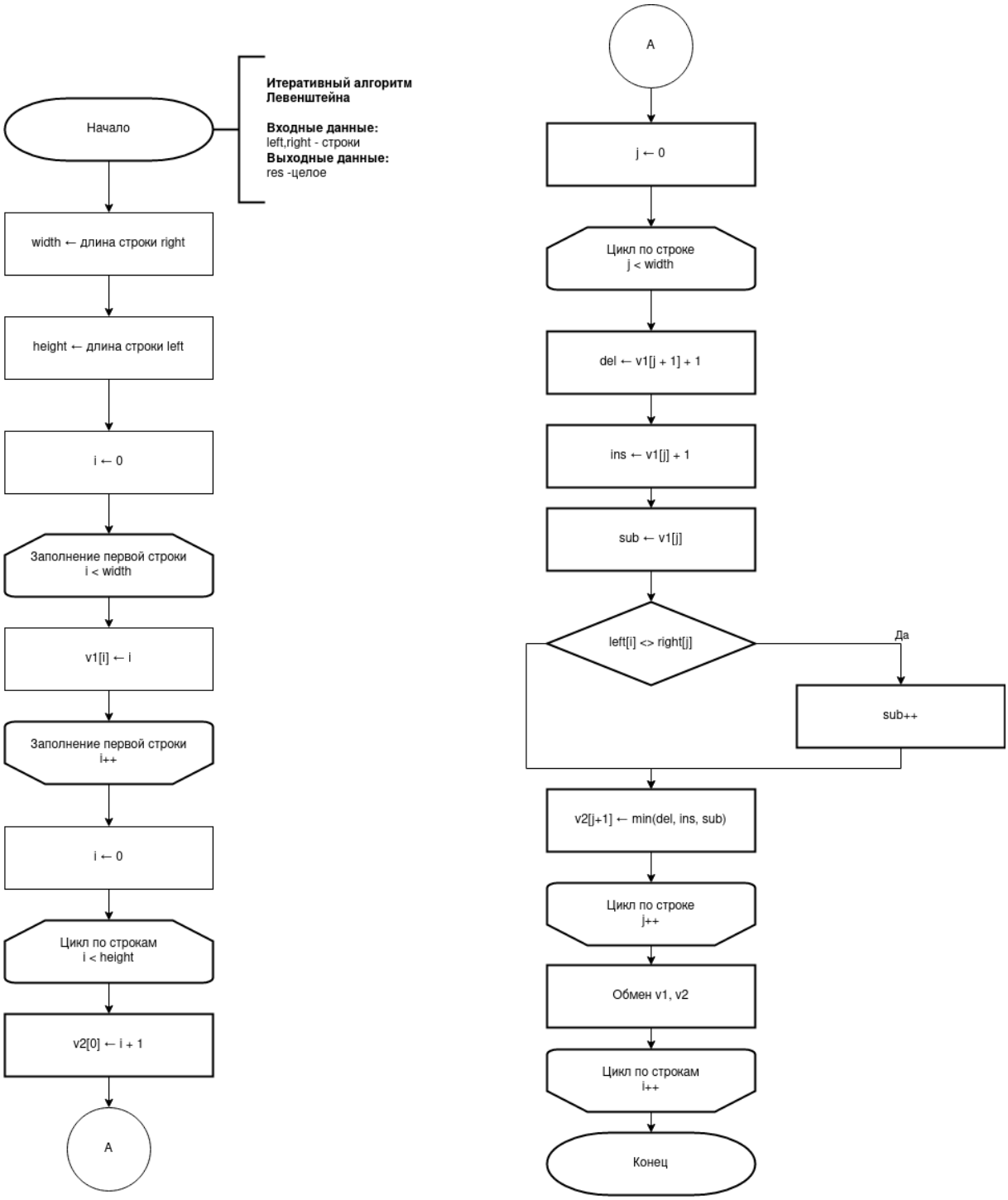

Рисунок 2.3 — Схема итеративного алгоритма Левенштейна

2.2 Вывод

На основе теоретических данных, полученные в аналитическом разделе были построены схемы иследуемых алгоритмов.

3 Технологический раздел

3.1 Требование к ПО

Программа должна отвечать следующем требованиям:

1) На вход подаются две строки в любой раскладке;
2) ПО должно выводить полученное расстояние.

3.2 Средства реализации

Основным средством разработки является язык программирования. Был выбран язык программирования C++. Данный выбор обоснован высокой скоростью работы языка, поддержкой строгой типизации [2]. Для сборки проекта был выбран инструмент CMake [3]. В качестве среды разработки был выбран инструмент JetBrains Clion [4].

Для подгрузки тестовых данных в формате JSON [5] в программу была использован модуль PropertyTree библиотеки Boost [6].

3.3 Реализация алгоритмов

В листинге 3.1 представлена реализация рекурсивного алгоритма Левенштейна. В листинге 3.2 представлена реализация рекурсивного алгоритма Левенштейна с кэшированием. В листинге 3.3 представлена реализация итеративного алгоритма Левенштейна. В листинге 3.4 представлена реализация рекурсивного алгоритма Дамерау–Левенштейна.

Листинг 3.1 — Рекурсивный алгоритм Левенштейна

```
1  int rec_l(const std::string &left, const std::string &right, const int i,
       const int j) {
2      if (std::min(i, j) == 0) {
3          return std::max(i, j);
4      }
5
6      int m = left[i - 1] == right[j - 1] ? 0 : 1;
7
8      return std::min({
9              rec_l(left, right, i, j - 1) + 1,
10             rec_l(left, right, i - 1, j) + 1,
11             rec_l(left, right, i - 1, j - 1) + m
12     });
13 }
```

Листинг 3.2 — Рекурсивный алгоритм Левенштейна с кэшированием

```
1  class Rec_Mem_l {
```

```
2       const int NoData = -1;
3
4       std::vector<std::vector<int>> _buffer;
5       std::string _left;
6       std::string _right;
7
8       int _step(int i, int j);
9
10      int __step(int i, int j);
11
12  public:
13      int operator()(const std::string &left, const std::string &right);
14  };
15
16  int Rec_Mem_l::_step(int i, int j) {
17
18      int temp = _buffer[i][j];
19      if (temp != NoData) {
20          return temp;
21      }
22
23      temp = __step(i, j);
24      _buffer[i][j] = temp;
25      return temp;
26
27  }
28
29  int Rec_Mem_l::__step(int i, int j) {
30
31      if (std::min(i, j) == 0) {
32          return std::max(i, j);
33      }
34
35      int m = _left[i - 1] == _right[j - 1] ? 0 : 1;
36      std::initializer_list<int> res = {
37              _step(i, j - 1) + 1,
38              _step(i - 1, j) + 1,
39              _step(i - 1, j - 1) + m
40      };
41
42      return std::min(res);
43  }
44
45  int Rec_Mem_l::operator()(const std::string &left, const std::string
        &right) {
46      _left = left;
47      _right = right;
```

```
48
49      int i = left.size(), j = right.size();
50      _buffer.resize(i + 1);
51      for (auto &row: _buffer) {
52          row.resize(j + 1);
53          for (auto &el: row) {
54              el = NoData;
55          }
56      }
57
58      return _step(i, j);
59  }
```

Листинг 3.3 — Итеративный алгоритм Левенштейна

```
1   int iter_l(const std::string &left, const std::string &right) {
2       int width = right.size();
3       int height = left.size();
4       std::vector<int> v1(width + 1);
5       std::vector<int> v2(width + 1);
6
7       for (size_t i = 0; i < width; i++) {
8           v1[i] = i;
9       }
10
11      for (size_t i = 0; i < height; i++) {
12          v2[0] = i + 1;
13          for (size_t j = 0; j < width; j++) {
14              int del = v1[j + 1] + 1;
15              int ins = v2[j] + 1;
16              int sub = v1[j];
17              if (left[i] != right[j]) {
18                  sub++;
19              }
20
21              v2[j + 1] = std::min({del, ins, sub});
22          }
23          std::swap(v1, v2);
24      }
25      return v1[width];
26  }
```

Листинг 3.4 — Рекурсивный алгоритм Дамерау–Левенштейна

```
1   int rec_dl(const std::string &left, const std::string &right, const int i,
        const int j) {
2
3       if (std::min(i, j) == 0) {
```

```
4           return std::max(i, j);
5       }
6
7       int m = left[i - 1] == right[j - 1] ? 0 : 1;
8
9       std::vector<int> res = {
10              rec_dl(left, right, i, j - 1) + 1,
11              rec_dl(left, right, i - 1, j) + 1,
12              rec_dl(left, right, i - 1, j - 1) + m
13      };
14      if (i > 1 && j > 1 && left[i - 1] == right[j - 2] && left[i - 2] ==
            right[j - 1]) {
15          res.push_back(
16                  rec_dl(left, right, i - 2, j - 2) + 1
17          );
18      }
19
20      return *std::min_element(res.begin(), res.end());
21  }
```

3.4 Тестовые данные

В таблице 3.1 представлены тестовые данные для алгоритмов Левенштейна и Дамерау–Левенштейна.

Таблица 3.1 — Тестовые данные для алгоритмов Левенштейна и Дамерау–Левенштейна

№	S_1	S_2	Левенштейна	Дамерау–Левенштейна
1	« »	« »	0	0
2	«same»	«same»	0	0
3	«hello»	«gol»	4	4
4	«qwer»	«rewq»	4	3
5	«music»	«muisc»	2	1
6	«memory»	«memxory»	1	1
7	«memxory»	«memory»	1	1
8	«mexory»	«memory»	1	1

3.5 Вывод

На основе схем из конструкторского раздела были разработаны реализации требуемых алгоритмов.

4 Экспериментальный раздел

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

— Операционная система Ubuntu 21.04;

— Память 16 GiB (4,5 GiB выделено для нужд графического ядра)

— Процессор AMD® Ryzen 5 5500u with radeon graphics × 12

4.2 Время выполнения алгоритмов

Для замеров времени использовалась стандартная функция языка clock [7]. Данная функция возвращает суммарное процессорное время, использованное программой. В случае ошибки, функция возвращает значение -1. На листинге 4.1 показан способ применения данной функции при замерах.

Листинг 4.1 — Замер времени функции

```
1 auto start = clock();
2 for (int j = 0; j < counts; ++j) {
3     rec_l(left, right);
4 }
5 res.rl = double(clock() - start) / counts;
```

Результаты тестирования приведены в таблице 4.1 .Прочерк в таблице означает что тестирование для этого набора данных не выполнялось.

Представлены зависимости времени работы рекурсивных алгоритмов Левенштейна и Дамерау–Левенштейна на рисунках 4.1 и 4.2

На рисунках 4.3 и 4.4 представлена зависимость времени работы реализаций алгоритмов Левенштейна итеративного и рекурсивного с кэшем.

4.3 Использование памяти

Максимальная глубина стека при вызове рекурсивных функций рассчитывается по формуле (4.1).

Использование памяти при итеративной реализации алгоритма Левенштейна может быть найдено с помощью формулы (4.2).

Таблица 4.1 — Таблица зависимости времени работы реализаций алгоритмов от длины входных слов

Длина слов	Рекурсивный Левенштейна	Итеративный Левенштейна	Рекурсивный с кэшем Левенштейна	Рекурсивный Дамерау–Левенштейна
1	0.5	0.3	0.5	0.7
2	0.2	0.4	0.5	1.3
3	0.9	0.6	0.7	6.7
4	4.6	0.8	1.1	34.4
5	23.8	1	1.5	183.9
6	127.6	1.3	2.1	961.9
7	689.6	1.7	3.2	4756.3
8	3488.9	2.8	6.7	34719.2
9	23001.5	2.4	4.9	145662
10	111428	3	5.9	797734
20	–	10.3	22.5	–
30	–	22.5	46.1	–
50	–	59.9	132.8	–
100	–	236.4	530.1	–
200	–	921	2047.2	–

$$M_{recursive} = (n \cdot lvar + ret + ret_{int}) \cdot depth \tag{4.1}$$

Где:

n – количество аллоцированных локальных переменных;
$lvar$ – размер переменной типа int
ret – адрес возврата;
ret_{int} – возвращаемое значение;
$depth$ – максимальная глубина стека вызова, которая равна$|S_1| + |S_2|$.

$$M_{iter} = |S_1| + |S_2| + (|S_2| + 1) \cdot 2 \cdot lvar + n \cdot lvar + ret + ret_{int} \tag{4.2}$$

Где $(|S_2| + 1) \cdot 2 \cdot lvar$ – место в памяти под матрицу расстояний.

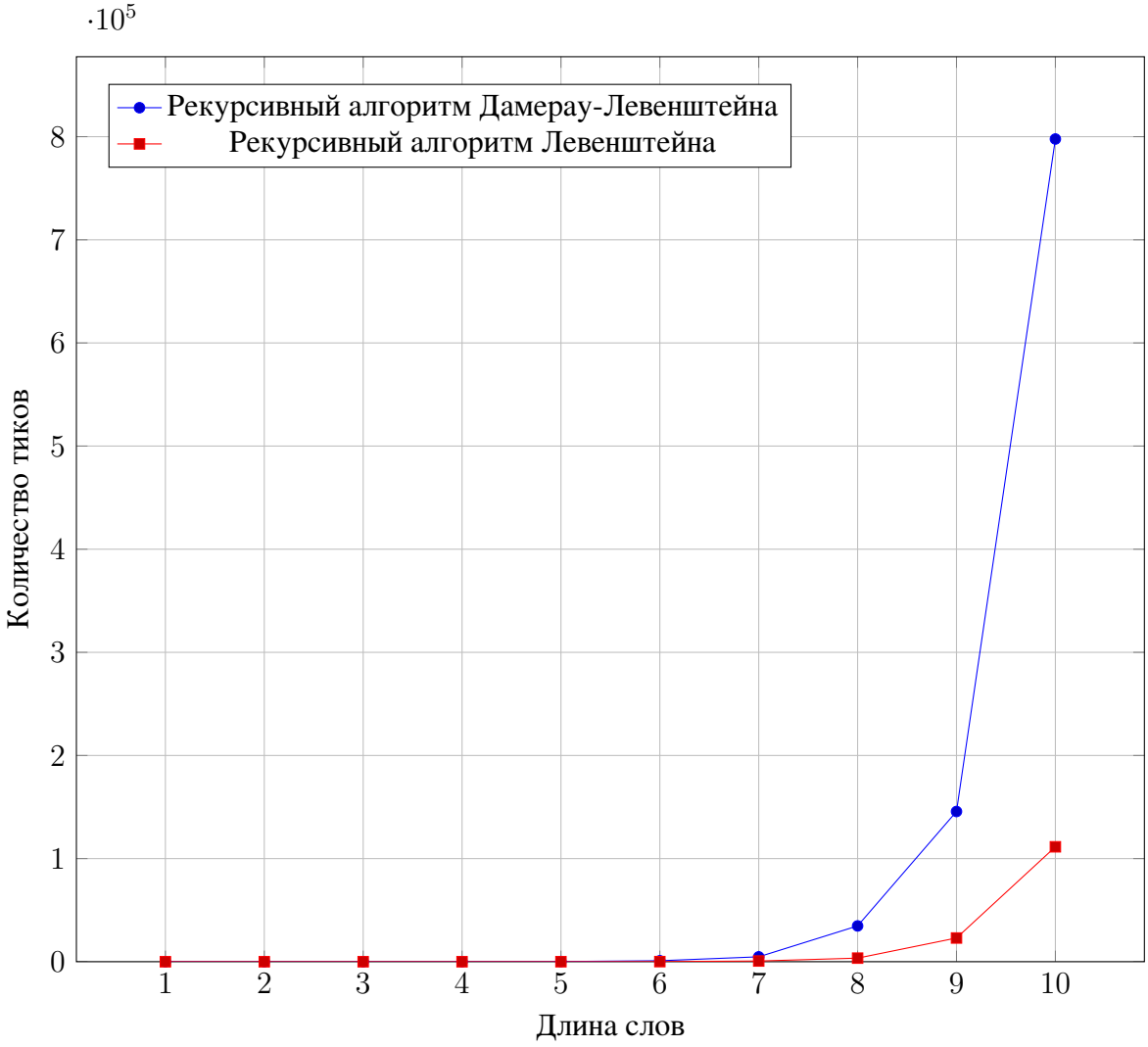

Рисунок 4.1 — Зависимость времени работы реализаций рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна от времени

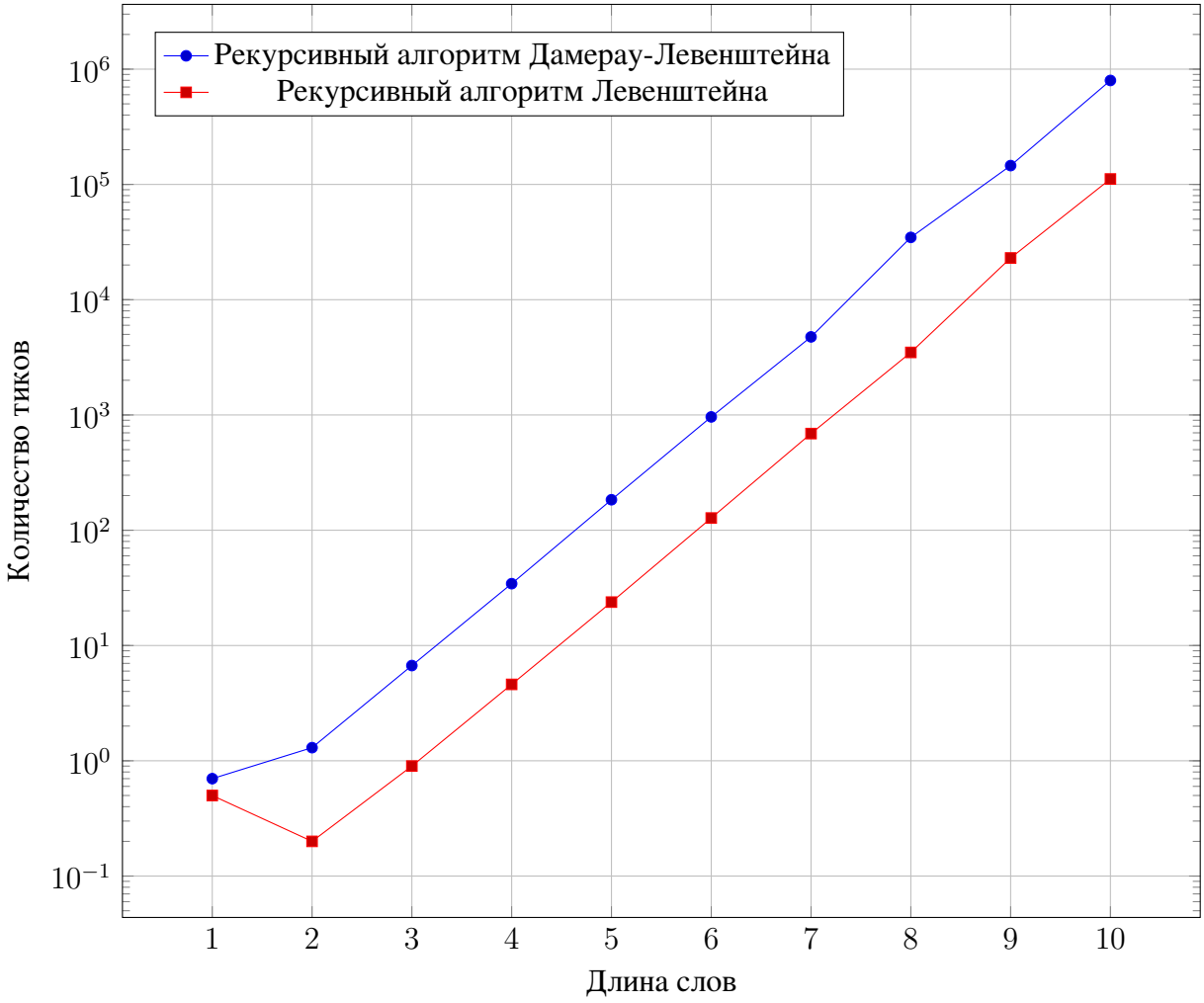

Рисунок 4.2 — Зависимость времени работы реализаций рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна от времени в логарифмической шкале

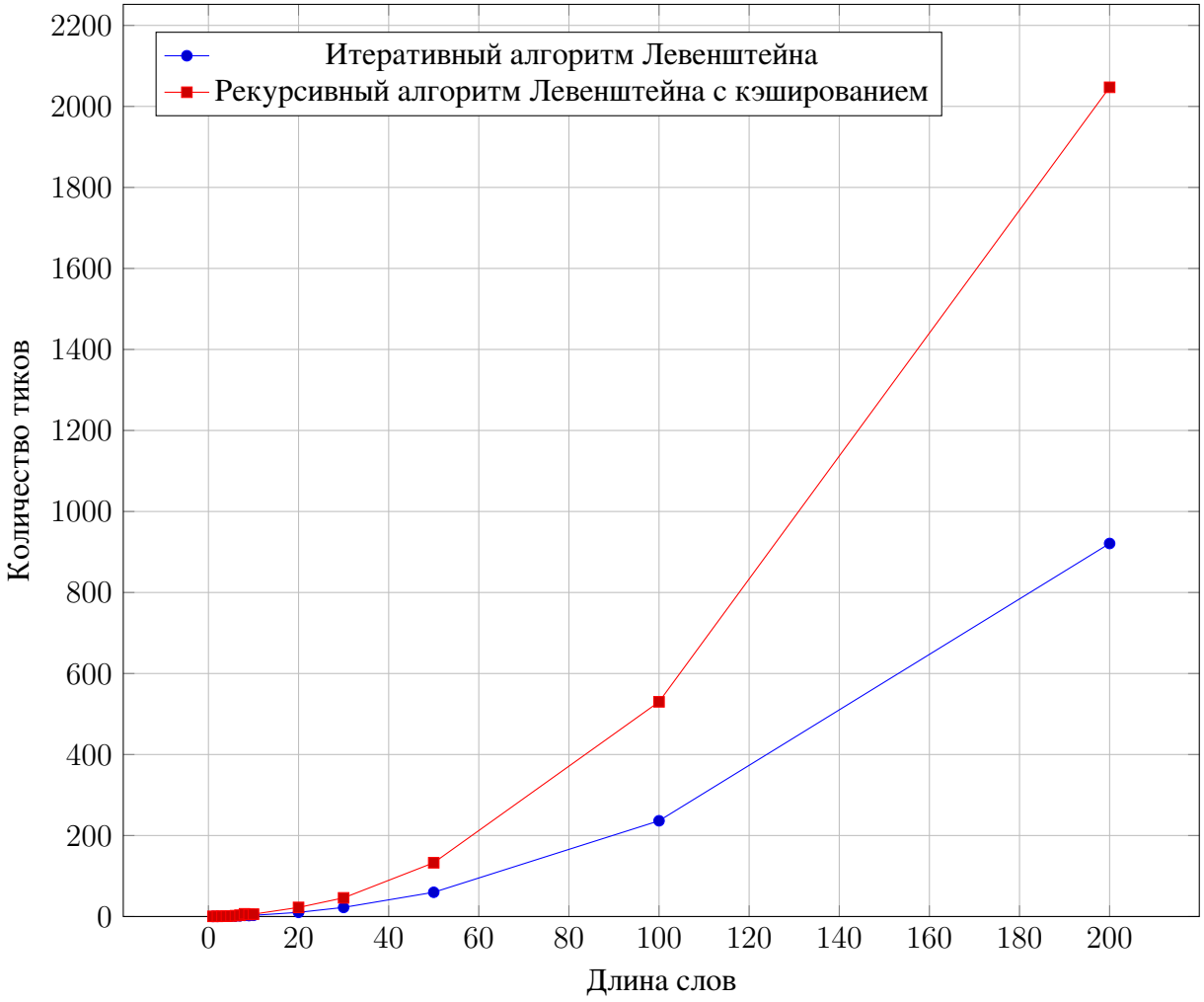

Рисунок 4.3 — Зависимость времени работы реализаций алгоритмов Левенштейна итеративного и рекурсивного с кэшем

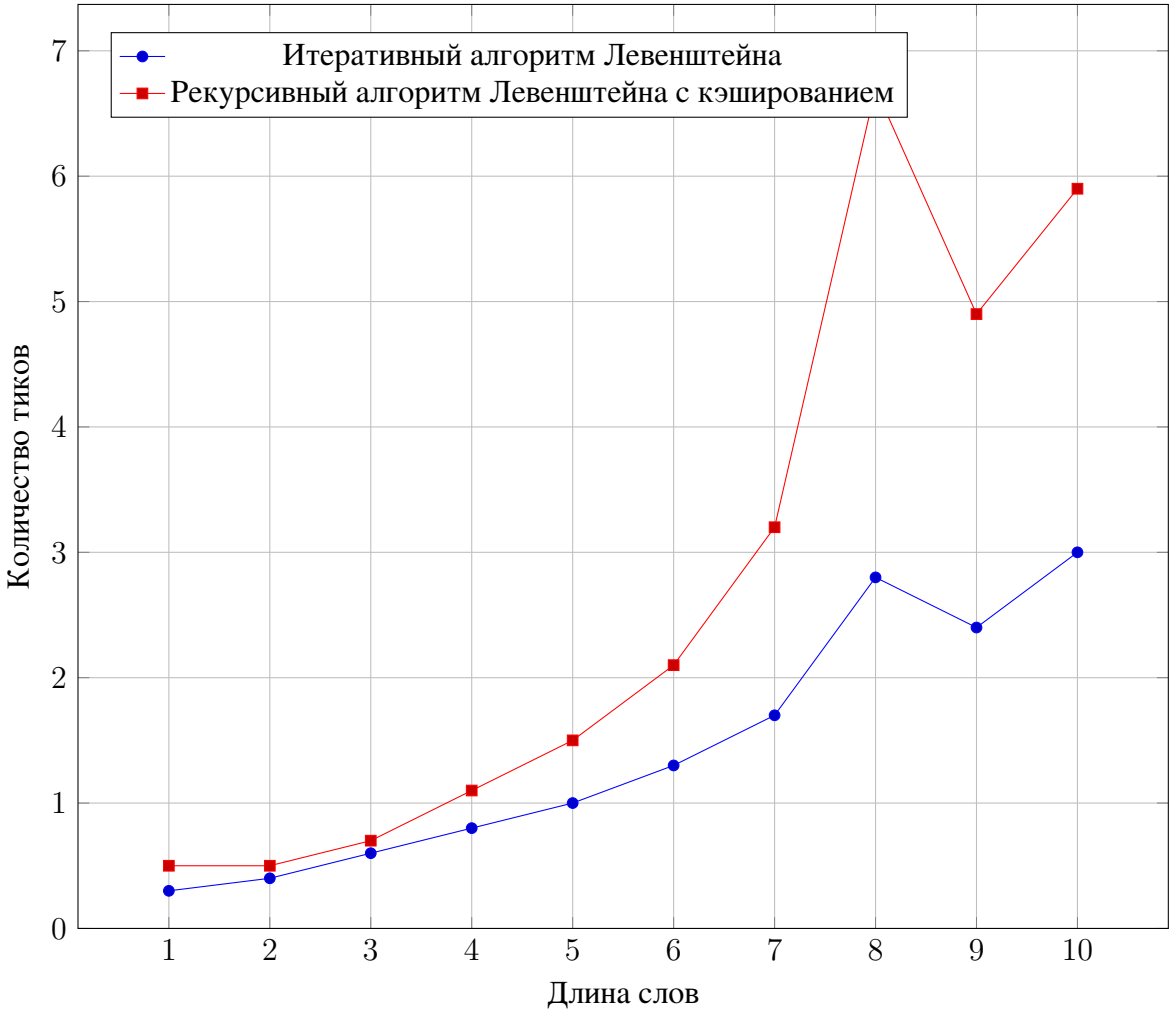

Рисунок 4.4 — Зависимость времени работы реализаций алгоритмов Левенштейна итеративного и рекурсивного с кэшем

4.4 Вывод

Рекурсивный алгоритм Левенштейна работает дольше итеративной реализации – время этого алгоритма увеличивается экспонентально с ростом размера строк. Рекурсивный алгоритм с кэшированием превосходит простой рекурсивный алгоритм по времени. Расстояние Дамерау – Левенштейна по результатом замеров работает дольше в отличии от расстояния Левенштейна. Однако, в системах автоматического исправления текста, где транспозиция встречается чаще, расстояние Дамерау – Левенштейна будет наиболее эффективным алгоритмом. По расходу памяти все реализации проигрывают рекурсивной за счет большого количества выделенной памяти под матрицу расстояний.

ЗАКЛЮЧЕНИЕ

В рамках лабораторной работы были рассмотрены два алгоритма нахождения редакторского расстояния – расстояние Левенштейна и расстояние Дамерау – Левенштейна. Во время аналитического изучения алгоритмов были выявлены смысловые различия между двумя алгоритмами – расстояние Дамерау – Левенштейна более эффективно в системах автоматической замены текста, где наиболее часто встречающаяся редакторская операция – это транспозиция. В других случаях, если алгоритмы работают не с буквами в естественном языке, рациональнее использовать алгоритм расстояние Левенштейна. Самая оптимальная реализация по памяти – рекурсивный алгоритм, самая оптимальная реализация по времени – итеративный алгоритм, использующий матрицу расстояний. В ходе лабораторной работы получены навыки динамического программирования, реализованы изученные алгоритмы.

СПИСОК ЛИТЕРАТУРЫ

1. Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов. — Докл. АН СССР, 1965. — С. 845–848. — ISBN: 9785457707351. — Режим доступа: http://mi.mathnet.ru/dan31411.

2. Страуструп Б. Программирование. Принципы и практика использования C++. — ООО ИД Вильямс, 2011. — ISBN: 9785457707351. — Режим доступа: https://books.google.ru/books?id=kttwBgAAQBAJ.

3. CMake [Электронный ресурс]. — Режим доступа: https://cmake.org/ (дата обращения: 10.10.2021).

4. CLion: A Cross-Platform IDE for C and C++ by JetBrains [Электронный ресурс]. — Режим доступа: https://www.jetbrains.com/clion/ (дата обращения: 10.10.2021).

5. Introducing JSON [Электронный ресурс]. — Режим доступа: https://www.json.org/json-en.html (дата обращения: 10.10.2021).

6. Boost C++ Libraries [Электронный ресурс]. — Режим доступа: https://www.boost.org/ (дата обращения: 10.10.2021).

7. Clock: Интерактивная система просмотра системных руководств [Электронный ресурс]. — Режим доступа: https://www.opennet.ru/man.shtml?topic=clock&category=3&russian=0 (дата обращения: 10.10.2021).