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	«Фундаментальные Науки»
КАФЕДРА	ФН-12 «Математическое моделирование»

ОТЧЕТ

ПО Лабораторной работе №2

по дисциплине «Типы и структуры данных»

Тема: «Редакционные расстояния»

Выполнил студент гр. ФН12-31Б: _____ Лямин И.С.
дата, подпись Ф.И.О.

Проверил преподаватель: _____ Волкова Л. Л.
дата, подпись Ф.И.О.

Москва, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной лабораторной работе будут реализованы алгоритмы поиска минимального расстояния Левенштейна и Дамерау-Левенштейна на языке програмирования C++. Цель работы – выполнить оценку ресурсной эффективности алгоритмов Левенштейна, Дамерау-Левенштейна, рекурсивных алгоритмов и рекурсивного алгоритма с кэшем и их реализации.

Для достижения цели необходимо выполнить следующие задачи.

1. Описать математическую основу расстояния Левенштейна и Дамерау—Левенштейна,
2. Описать модель вычисления,
3. Реализовать программу для расчёта расстояний Дамерау-Левенштейна и Левенштейна,
4. Выполнить оценку трудоёмкости реализации алгоритмов,
5. Реализовать разработанные алгоритмы в программном обеспечении с 2-мя режимами работы – одиночного расчёта и массированного замера процессорного времени,
6. Выполнить замеры процессорного времени выполнения реализации каждого алгоритма в зависимости от длины строк,
7. Выполнить сравнительный анализ рассчитанных трудоёмкостей и результатов замера процессорного времени,

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна — минимальное количество редакционных операций вставки, удаления, замены одного символа, необходимое для преобразования одной строки к другой.

Формула поиска расстояния Левенштейна [1]:

$$D_{s1,s2}(i,j) = \begin{cases} max \begin{Bmatrix} i \\ j \end{Bmatrix}, & i = 0 \text{ or } j = 0, \\ min \begin{Bmatrix} D_{s1,s2}(i-1,j)+1, \\ D_{s1,s2}(i,j-1)+1, \\ D_{s1,s2}(i-1,j-1)+n_{i,j} \end{Bmatrix}, & i > 0, j > 0, \end{cases} \quad (1)$$

где $s1, s2$ – сравниваемые строки;
$D_{s1,s2}(i,j)$ – расстояние Левенштейна для подстрок строк $s1, s2$, где подстрока строки $s1$ — часть строки, начинающаяся с элемента строки с индексом 0 и заканчивающаяся элементом строки с индексом $i-1$, подстрока строки $s2$ — часть строки, начинающаяся с элемента строки с индексом 0 и заканчивающаяся элементом строки с индексом $j-1$.
Функция $n_{i,j}$:

$$n_{i,j} = \begin{cases} 0, & s1[i] = s2[j], \\ 1, & s1[i] \neq s2[j], \end{cases} \quad (2)$$

где $s1[i]$ — элемент строки $s1$ с индексом i, $s2[j]$ — элемент строки $s2$ с индексом j.

1.2 Расстояние Дамерау–Левенштейна

Расстояние Дамерау–Левенштейна — минимальное количество редакционных операций вставки, удаления, замены одного символа, перестановки двух соседних символов, необходимое для преобразования одной строки к другой.

Формула поиска расстояния Дамерау — Левенштейна:

$$d_{s1,s2}(i,j) = \begin{cases} max \begin{Bmatrix} i \\ j \end{Bmatrix}, & i = 0 \text{ or } j = 0, \\ min \begin{Bmatrix} d_{s1,s2}(i-1,j)+1, \\ d_{s1,s2}(i,j-1)+1, \\ d_{s1,s2}(i-1,j-1)+n_{i,j}, \\ d_{s1,s2}(i-2,j-2)+1, \end{Bmatrix}, & i > 1, j > 1, m_{i,j} = 1, \\ min \begin{Bmatrix} d_{s1,s2}(i-1,j)+1, \\ d_{s1,s2}(i,j-1)+1, \\ d_{s1,s2}(i-1,j-1)+n_{i,j} \end{Bmatrix}, & \text{else}, \end{cases} \tag{3}$$

где $s1, s2$ – сравниваемые строки;
$d_{s1,s2}(i,j)$ — расстояние Дамерау — Левенштейна для подстрок строк $s1, s2$, где подстрока строки $s1$ — часть строки, начинающаяся с элемента строки с индексом 1 и заканчивающаяся элементом строки с индексом i, подстрока строки $s2$ — часть строки, начинающаяся с элемента строки с индексом 1 и заканчивающаяся элементом строки с индексом j;

Функция $m_{i,j}$ имеет вид:

$$m_{i,j} = \begin{cases} 0, & s1[i-1]! = s2[j] \text{ or } s1[i]! = s2[j-1], \\ 1, & s1[i-1] == s2[j] \text{ and } s1[i] == s2[j-1], \end{cases} \tag{4}$$

где $s1[i]$ — элемент строки $s1$ с индексом i, $s2[j]$ — элемент строки $s2$ с индексом j.

2 Конструкторская часть

2.1 Описание алгоритмов

Алгоритм поиска редакционного расстояния, основанный на формуле нахождения растояния Левенштейна, представлен на блок-схеме алгоритма на рисунке 1. Алгоритм поиска редакционного расстояния, основанный на формуле нахождения расстояния Дамерау–Левенштейна, представлен на рис блок-схемы алгоритма 2. Рекурсивный алгоритм поиска редакционного расстояния, основанный на формуле нахождения расстояния Левенштейна, представлен на рисунке блок-схемы алгоритма 3.

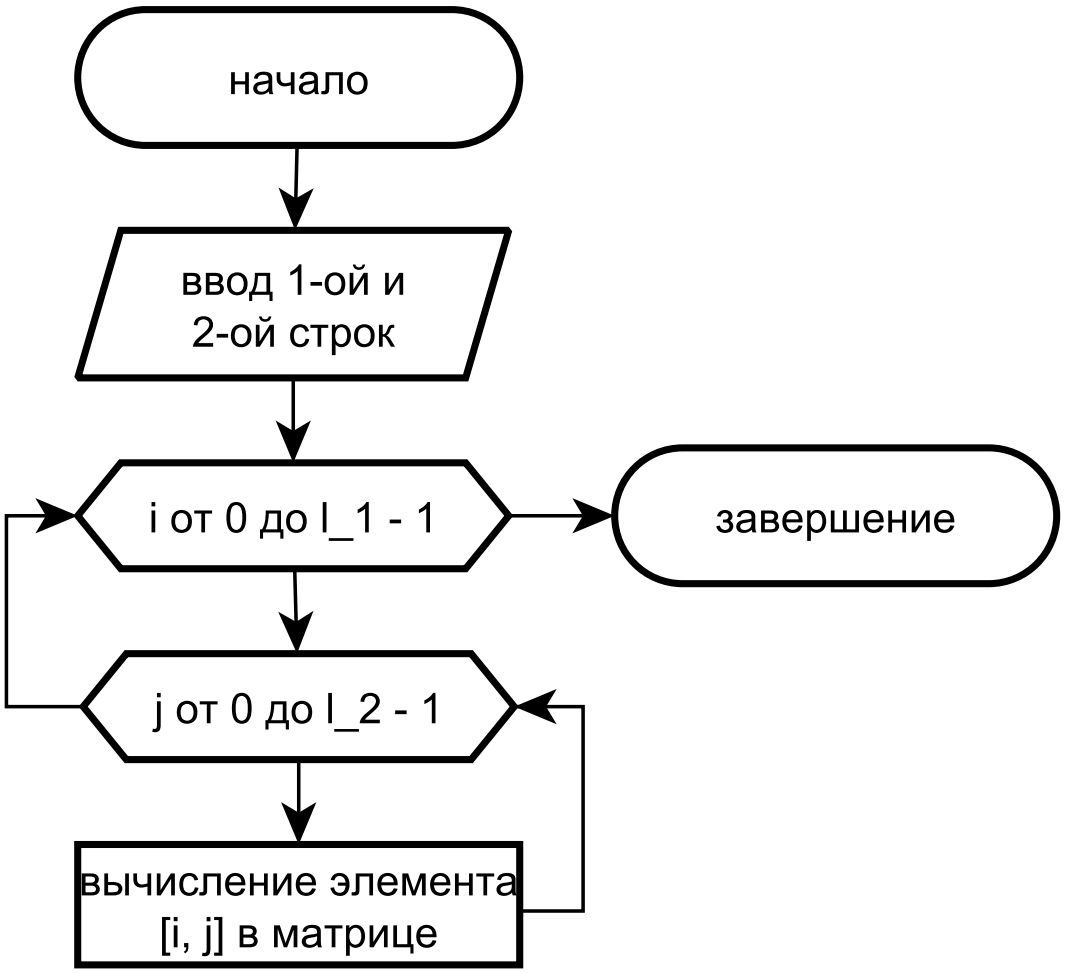

Рисунок 1 — Схема алгоритма функции поиска расстояния Левенштейна

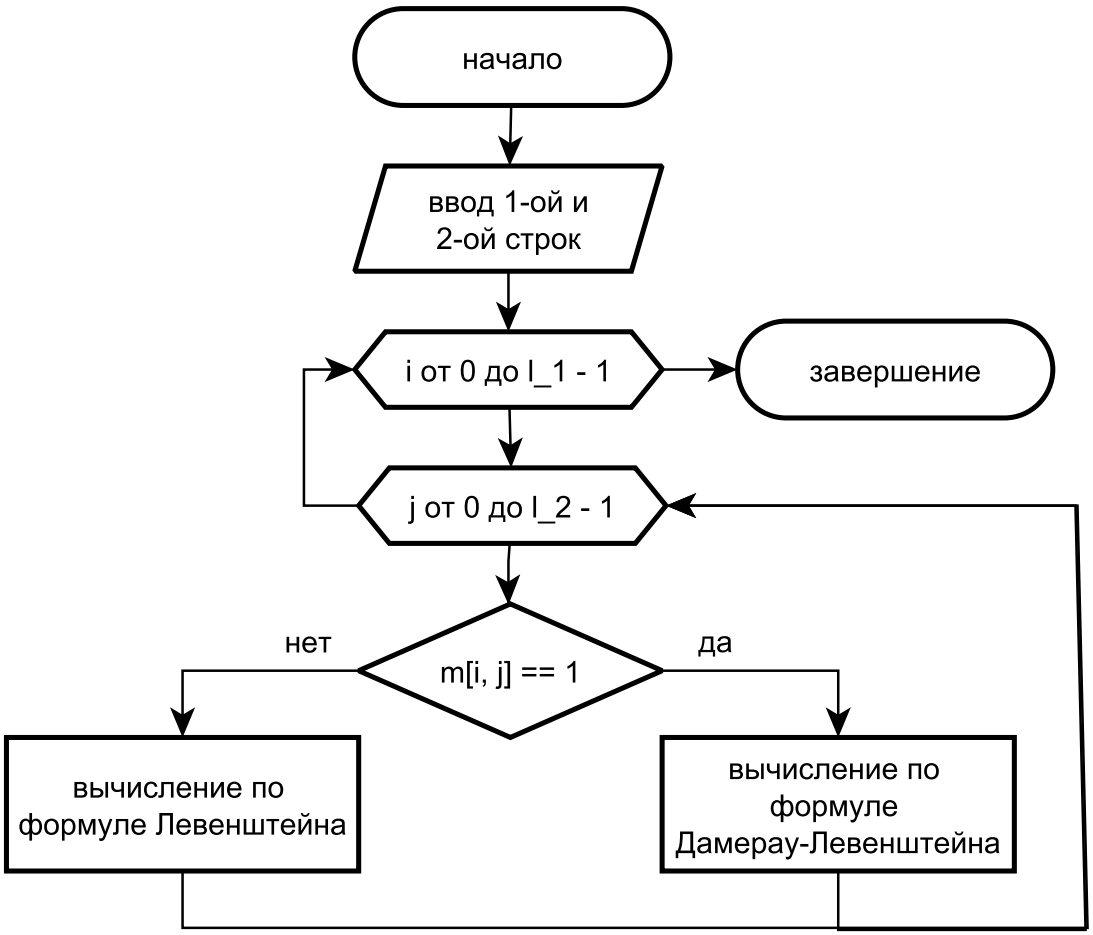

Рисунок 2 — Схема алгоритма функции поиска расстояния Дамерау–Левенштейна

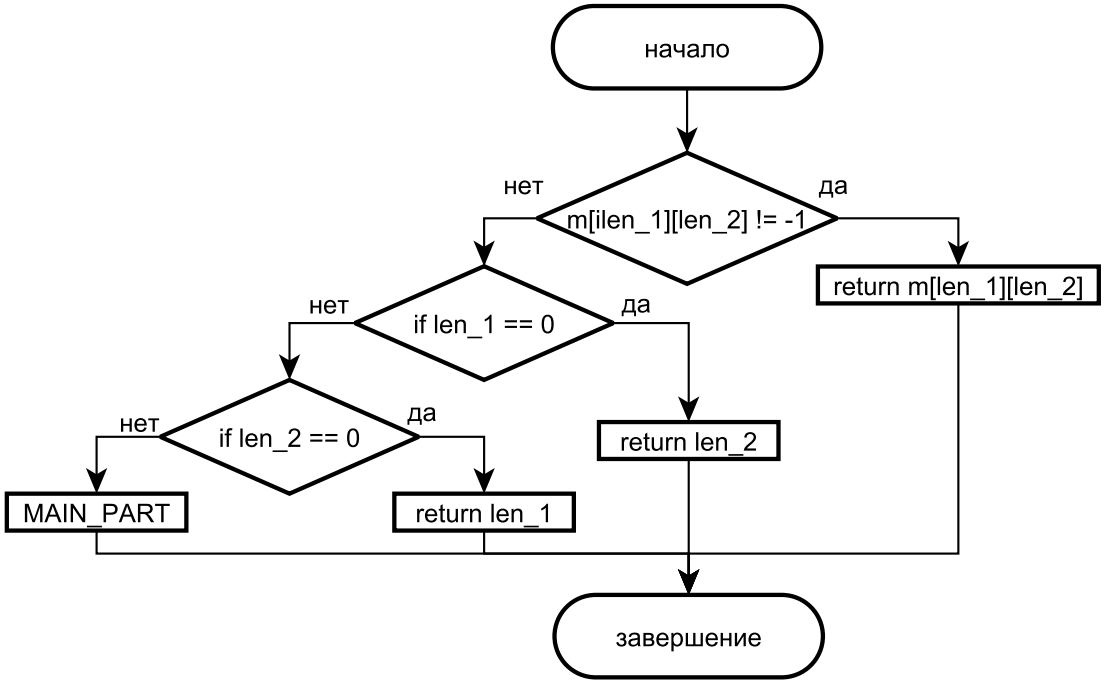

Рисунок 3 — Схема рекурсивного алгоритма поиска расстояния Левенштейна

Подпрограмма MAIN_PART выполняет следующую работу: return min(alg (str_1, str_2, len_2, len_1 - 1) + 1, alg(m, len_2 - 1, len_1) + 1, alg(m, len_2 - 1, len_1 - 1) + n), где alg — рекусивный алгоритм.

Рекурсивный алгоритм с кэшем поиска редакционного расстояния, основанный на формуле нахождения расстояния Левенштейна представлен на схеме алгоритма на рисунок 4. Рекурсивный алгоритм для расстояния Дамерау-Левенштейна аналогичен и отличен только тем, что при $m_{i,j} == 1$, часть MAIN_PART примет вид: return min(alg(str_1, str_2, len_2, len_1 - 1) + 1, alg(str_1, sre_2, len_2 - 1, len_1) + 1, alg(str_1, str_2, len_2 - 1, len_1 - 1) + n,
alg(str_1, str_2, len_2 - 2, len_1 - 2) + 1);

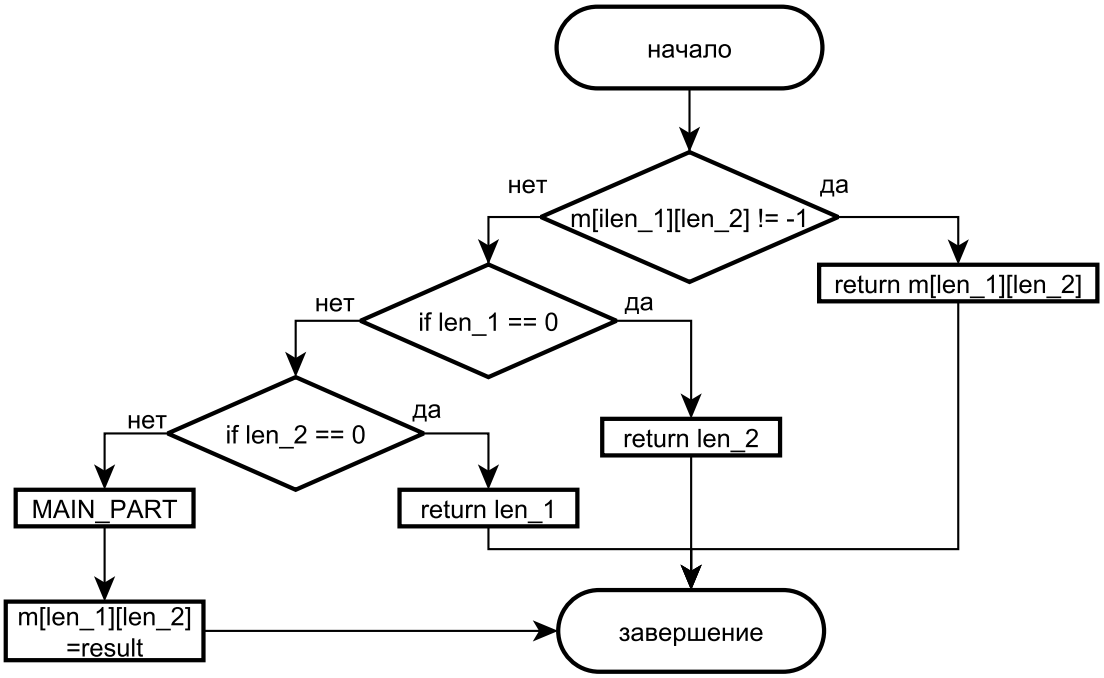

Рисунок 4 — Схема алгоритма функции рекурсивного алгоритма поиска растояния Левенштейна с кэшем

2.2 Анализ сложностей

В таблице 1 представлены асимптотические сложности разобранных алгоритмов по времени и по памяти. Рекурсивный алгоритм затрачивает на порядок больше времени для вычисления (таблица№1) относительно обычному алгоритму и алгоритму с кэшем. А обычный алгоритм и алгоритм с кэшем затрачивают одинаковое количество ресурсов, что по памяти, что по времени.

Таблица 1 — Асимптотические сложности разобранных алгоритмов

Алгоритм	Сложность по времени	Сложность по памяти
Поиск расстояния Левенштейна	$O(n \cdot m)$	$O(n \cdot m)$
Поиск расстояния Дамерау–Левенштейна	$O(n \cdot m)$	$O(n \cdot m)$
Рекурсивный алгоритм Поиска расстояния Дамерау–Левенштейна	$O(4^{max(n,m)})$	$O(n \cdot m + 20 \cdot (n + m))$
Рекурсивный алгоритм с кэшем	$O(n \cdot m)$	$O(n \cdot m)$

3 Технологическая часть

3.1 Выбор средств реализации

Для программной реализации использовалась среда разработки Visual Studio, язык программирования, на котором была выполнена реализации — C++. Исследование проводилось на ноутбуке (64-разрядная операционная система, процессор x64, частота процессора 3.10 Ггц, оперативная память 16 ГБ)

Для замера времени использовались функции библиотеки Chrono [2].

3.2 Реализация алгоритмов

В листинге 1 можно увидеть программную реализацию описанных алгоритмов.

Листинг 1 — програмная реализация вспомогательной структуры

```
struct matrix {
        public:
        matrix(string str_1, string str_2) {
                this->rows = (str_2.size() + 1);
                this->columns = (str_1.size() + 1);
                this->str_1 = "_" + str_1;
                this->str_2 = "_" + str_2;
                this->c = new int* [rows];
                prepair_matrix();
        }

        ~matrix() {
                delete[] c;
        }

        void print() {
                cout << setiosflags(ios::left);

                for (int i = 0; i < rows; i++) {
                        cout << setw(2) << "|";
                        for (int j = 0; j < columns; j++) {
                                cout << setw(3) << c[i][j] << " ";
                        }
                        cout << "|" << endl;
                }
        }

        int* operator[] (int row) {
                return (this->c[row]);
        }

```

```
32          string str(int ind) {
33                  if (ind == 1) {
34                          return str_1;
35                  }
36                  return str_2;
37          }
38
39          int ro() {
40                  return rows;
41          }
42
43          int co() {
44                  return columns;
45          }
46
47          private:
48          string str_1;
49          string str_2;
50          int rows;
51          int columns;
52          int** c;
53
54          void prepair_matrix() {
55                  for (int i = 0; i < rows; i++) {
56                          c[i] = new int[columns];
57                          for (int j = 0; j < (columns); j++) {
58                                  if (i == 0) {
59                                          c[i][j] = j;
60                                  }
61                                  else if (j == 0) {
62                                          c[i][j] = i;
63                                  }
64                                  else {
65                                          c[i][j] = -1;
66                                  }
67                          }
68                  }
69          }
70  };
```

Листинг 2 — Программная реализация разработанных алгоритмов

```
void aloritm_Lev(matrix* m) {
        int n;
        for (int i = 0; i < (*m).ro(); i++) {
                for (int j = 0; j < (*m).co(); j++) {
                        //formula for finding the Levenshtein distance
                        if (i == 0) {
                                (*m)[i][j] = j;
                        }
                        else if (j == 0) {
                                (*m)[i][j] = i;
                        }
                        else {
                                if ((*m).str(2)[i] == (*m).str(1)[j]) n = 0;
                                else n = 1;
                                (*m)[i][j] = min(min((*m)[i][j - 1] + 1, (*m)[i - 1][j] + 1), ((*m)[i - 1][j - 1] + n));
                        }
                }
        }
}

void algoritm_Dam_Lev(matrix* m) {
        int n;
        for (int i = 0; i < (*m).ro(); i++) {
                for (int j = 0; j < (*m).co(); j++) {
                        if (i == 0) {
                                (*m)[i][j] = j;
                        }
                        else if (j == 0) {
                                (*m)[i][j] = i;
                        }
                        else {
                                if ((*m).str(2)[i] == (*m).str(1)[j]) n = 0;
                                else n = 1;
                                if ((j > 1) && (i > 1) && ((*m).str(2)[i] == (*m).str(1)[j - 1]) && ((*m).str(2)[i - 1] == (*m).str(1)[j])) {
                                        (*m)[i][j] = min(min((*m)[i][j - 1] + 1, (*m)[i - 1][j] + 1), min((*m)[i - 1][j - 1] + n, (*m)[i - 2][j - 2] + 1));
                                }
                                else {
                                        (*m)[i][j] = min(min((*m)[i][j - 1] + 1, (*m)[i - 1][j] + 1), (*m)[i - 1][j - 1] + n);
                                }
                        }
                }
        }
```

```
43 }
44
45 int alg_Lev_rec(matrix* m, int len_2, int len_1) {
46         int n;
47
48         //formula for finding the Levenshtein distance by recirsion
49         if (len_1 == 0) {
50                 return len_2;
51         }
52         else if (len_2 == 0) {
53                 return len_1;
54         }
55         else {
56                 if ((*m).str(2)[len_2] == (*m).str(1)[len_1]) n = 0;
57                 else n = 1;
58                 return min(min(alg_Lev_rec(m, len_2, len_1 - 1) + 1,
   ↪ alg_Lev_rec(m, len_2 - 1, len_1) + 1), alg_Lev_rec(m, len_2 -
   ↪ 1, len_1 - 1) + n);
59         }
60 }
61
62 int alg_Dam_Lev_rec(matrix* m, int len_2, int len_1) {
63         int n;
64
65         //formula for finding the Levenshtein distance by recirsion
66         if (len_1 == 0) {
67                 return len_2;
68         }
69         else if (len_2 == 0) {
70                 return len_1;
71         }
72         else {
73                 if ((*m).str(2)[len_2] == (*m).str(1)[len_1]) n = 0;
74                 else n = 1;
75                 if ((len_1 > 1) && (len_2 > 1) && ((*m).str(2)[len_2]
   ↪  == (*m).str(1)[len_1 - 1]) && ((*m).str(2)[len_2 - 1] == (*m).
   ↪ str(1)[len_1])) {
76                         return min(min(alg_Dam_Lev_rec(m, len_2,
   ↪ len_1 - 1) + 1, alg_Dam_Lev_rec(m, len_2 - 1, len_1) + 1), min(
   ↪ alg_Dam_Lev_rec(m, len_2 - 1, len_1 - 1) + n, alg_Dam_Lev_rec(m
   ↪ , len_2 - 2, len_1 - 2) + 1));
77                 }
78                 return min(min(alg_Dam_Lev_rec(m, len_2, len_1 - 1) +
   ↪  1, alg_Dam_Lev_rec(m, len_2 - 1, len_1) + 1), alg_Dam_Lev_rec(
   ↪ m, len_2 - 1, len_1 - 1) + n);
79         }
80 }
81
82 int alg_Lev_rec_cash(matrix* m, int len_2, int len_1) {
83         if ((*m)[len_2][len_1] != -1) {
84                 return (*m)[len_2][len_1];
85         }
```

```
86
87          int n;
88          if (len_1 == 0) {
89                  return len_2;
90          }
91          else if (len_2 == 0) {
92                  return len_1;
93          }
94          else {
95                  if ((*m).str(2)[len_2] == (*m).str(1)[len_1]) {
96                          n = 0;
97                  }
98                  else {
99                          n = 1;
100                 }
101
102                 (*m)[len_2][len_1] = min(min(alg_Lev_rec_cash(m,
    ↪ len_2, len_1 - 1) + 1, alg_Lev_rec_cash(m, len_2 - 1, len_1) +
    ↪ 1), alg_Lev_rec_cash(m, len_2 - 1, len_1 - 1) + n);
103                 return (*m)[len_2][len_1];
104         }
105 }
```

3.3 Тестирование программы

В таблице 2 представлены описания тестов по методологии чёрного ящика, все тесты пройдены успешно.

Таблица 2 — Описание тестов по методологии чёрного ящика

	Описание теста	**Входные данные**	**Ожидаемый результат**	**Полученный результат**
1	проверка на обработку не валидных данных	3	оповещение о некорректности данных и запрос новых	оповещение о некорректности данных и запрос новых
2	проверка на коректность работы алгоритмов	1 123456 132546	Левенштейн: 3 Дамерау-Левенштейн: 2	Левенштейн: 3 Дамерау-Левенштейн: 2
3	проверка на единичные строки	1 2 2	Левенштейн: 0 Дамерау-Левенштейн: 0	Левенштейн: 0 Дамерау-Левенштейн: 0
4	проверка на коректность работы с матрицей 1 на n	1 1 123456	Левенштейн: 5 Дамерау-Левенштейн: 5	Левенштейн: 5 Дамерау-Левенштейн: 5

4 Исследовательская часть

4.1 Замеры процессорного времени выполнения реализации алгоритмов

На рисунке 5 можно увидеть графики, иллюстрирующие зависимость процессорного времени выполнения реализации алгоритма расчета редакционного расстояния от длины строки для рекурсивного алгоритма и для обычного с кэшем. Пусть длины строк совпадают.

Рисунок 5 — Визуализация зависимости процессорного времени при выполнении реализации рекурсивного и обычного алгоритма поиска расстояния Левенштейна от длины строк.

Из этого можно сделать вывод, что рекурсивный алгоритм намного более затратный с точки зрения числа выполненных операций (затраченного на расчёт времени). Так, для строк длиной 7 затрачено в 140 раз больше времени.

Визуализацию проведенного замеров затрачиваемого процессорного времени обычным алгоритмом с кэшем и рекурсивным с кэшем, можно увидеть на рисунке 6.

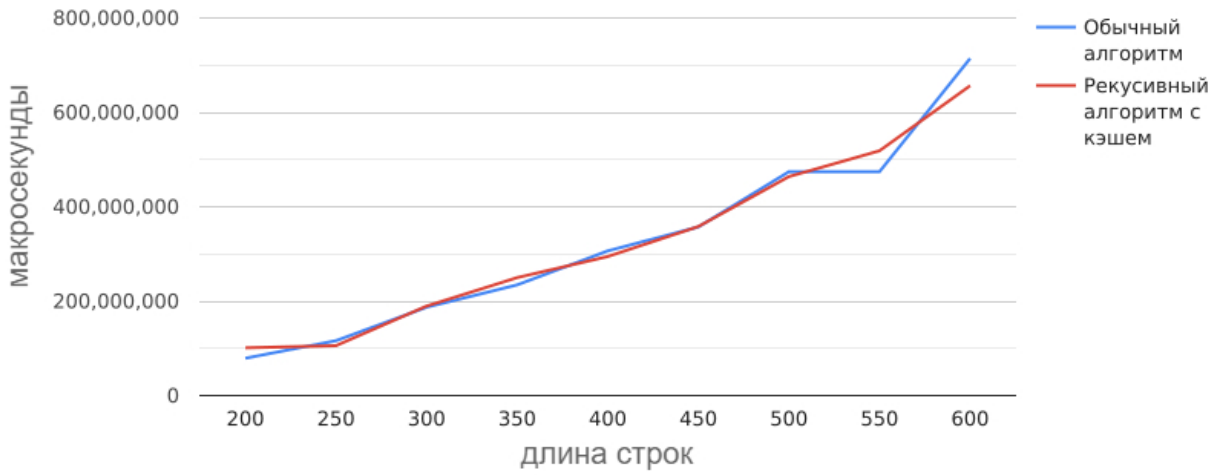

Рисунок 6 — Визуализация зависимости процессорного времени выполнения реализации рекурсивного с кэшем и обычного алгоритмов поиска расстояния Левенштейна от длины строк.

Проанализировав графики, можно отметить, что процессорное время, затрачиваемое на выполнение реализации алгоритмов поиска расстояния Дамерау – Левенштейна, как рекурсивного с кэшем, так и нерекурсивного, намного меньше, чем то время, которое затрачивается на выполнение реализации рекурсивного алгоритма поиска расстояния Левенштейна даже на строках маленькой длины.

Рекурсивный алгоритм с кэшем и обычный алгоритм с кэшем показывают примерно равные результаты.

ЗАКЛЮЧЕНИЕ

В результате лабораторной работы были выполнены все поставленные задачи.

1. Описана математическая основа расстояния Левенштейна и Дамерау—Левенштейна,
2. Описана модель вычисления,
3. Реализована программу для расчёта расстояний Дамерау-Левенштейна и Левенштейна,
4. Выполнена оценка трудоёмкости реализации алгоритмов,
5. Реализованы разработанные алгоритмы в программном обеспечении с двумя режимами работы – одиночного расчёта и массированного замера процессорного времени,
6. Выполнены замеры процессорного времени выполнения реализации каждого алгоритма в зависимости от длины строк,
7. Выполнен сравнительный анализ рассчитанных трудоёмкостей и результатов замера процессорного времени,

Цель работы достигнута: выполнена оценка ресурсной эффективности алгоритмов нахождения расстояния Левенштейна и Дамерау—Левенштейна.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ульянов М. В. Ресурсно эффективные компьютерные алгоритмы / Учебное пособие 2007.

2. Microsoft. GetProcessTimes function. [Электронный ресурс] - URL:https://learn.microsoft
ru/cpp/standard-library/processthreadsapi/chrono

   (дата обращения: 20.09.2024).

3. AlgoLib. [Электронный ресурс] - URL: https://www.algolib.narod/Math/Matrix

   (дата обращения: 20.09.2024).