



**Министерство образования и науки 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)**

ОТЧЕТ
По лабораторной работе №1
По курсу «Анализ алгоритмов»
Тема: «Расстояние Левенштейна»

Студент: Зейналов З. Г.
Группа: ИУ7-51
Преподаватель: Волкова Л.Л.

Москва, 2019г.

Оглавление

Введение	3
Задачи работы:.....	4
1. Аналитическая часть.....	5
1.1 Описание алгоритмов	5
2. Конструкторская часть.....	6
2.1 Разработка алгоритмов.....	7
2.2 Сравнительный анализ рекурсивной и нерекурсивной реализаций	13
3 Технологическая часть.....	14
3.1 Требования к программному обеспечению.....	14
3.2 Средства реализации.....	14
3.3 Листинг кода	15
3.4 Описание тестирования	17
4 Экспериментальная часть	18
4.1 Примеры работы.....	18
4.2 Результаты тестирования.....	20
4.3 Постановка эксперимента по замеру времени [и памяти]	21
4.4 Сравнительный анализ на материале экспериментальных данных	22
Заключение	23

Введение

Расстояние Левинштейна - 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 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. Аналитическая часть

В данном разделе будут представлены описания алгоритмов, формулы и оценки сложностей алгоритмов

1.1 Описание алгоритмов

Расстояние Левинштейна или редакционное расстояние между строками s_1 и s_2 как \min количество редакторских операций, необходимых для преобразования одной строки в другую.

Редакторские операции:

$$1 \text{ символ (стоимость} = 1) \left\{ \begin{array}{l} I - \text{insert (вставка)} \\ D - \text{delete (удаление)} \\ R - \text{replace (замена)} \end{array} \right.$$

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

Формула для нахождения расстояния с использованием матрицы:

$$D(i, j) = \begin{cases} 0, \text{ если } i == 0, & j == 0 \\ i, \text{ если } i > 0, & j == 0 \\ j, \text{ если } j > 0 & i == 0 \\ \min \left\{ \begin{array}{l} D(s_1[1 \dots i], s_2[1 \dots j - 1]) + 1 \\ D(s_1[1 \dots i - 1], s_2[1 \dots j]) + 1 \\ D(s_1[1 \dots i - 1], s_2[1 \dots j - 1]) + \begin{cases} 0, s_1[i] == s_2[j] \\ 1, \text{ иначе} \end{cases} \end{array} \right. \end{cases}$$

Расстояние Дamerau-Левинштейна вводится дополнительная операция перестановки или транспозиция, 2 буквы, стоимость = 1. Если индексы позволяют, и если соседние буквы $s_1[i]$ совпадает с $s_2[j - 1]$ и $s_1[i - 1] = s_2[j]$, то в минимум включается перестановка (транспозиция).

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 алгоритмы так же допускают рекурсивную реализацию.

2. Конструкторская часть

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

2.1 Разработка алгоритмов

Ниже будут приведены схемы реализованных алгоритмов:

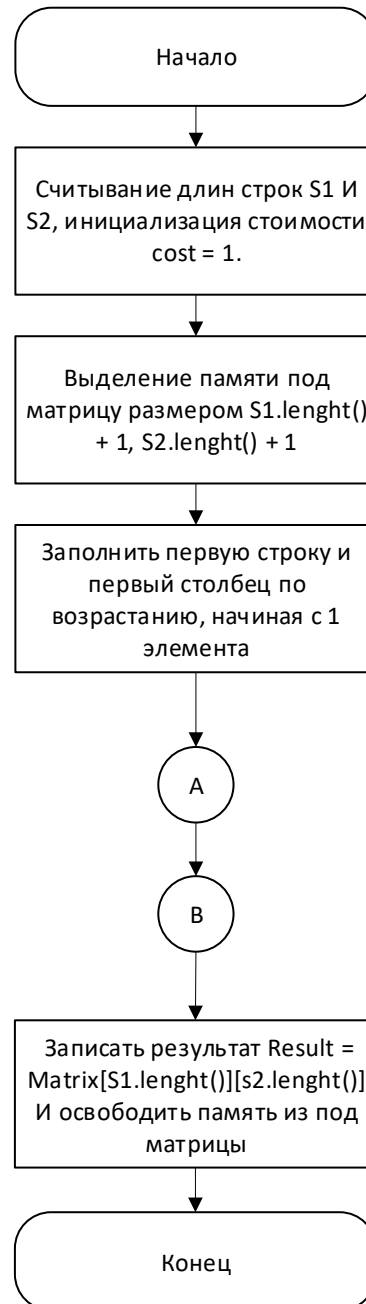


Рисунок 1. Алгоритм Левенштейна часть 1

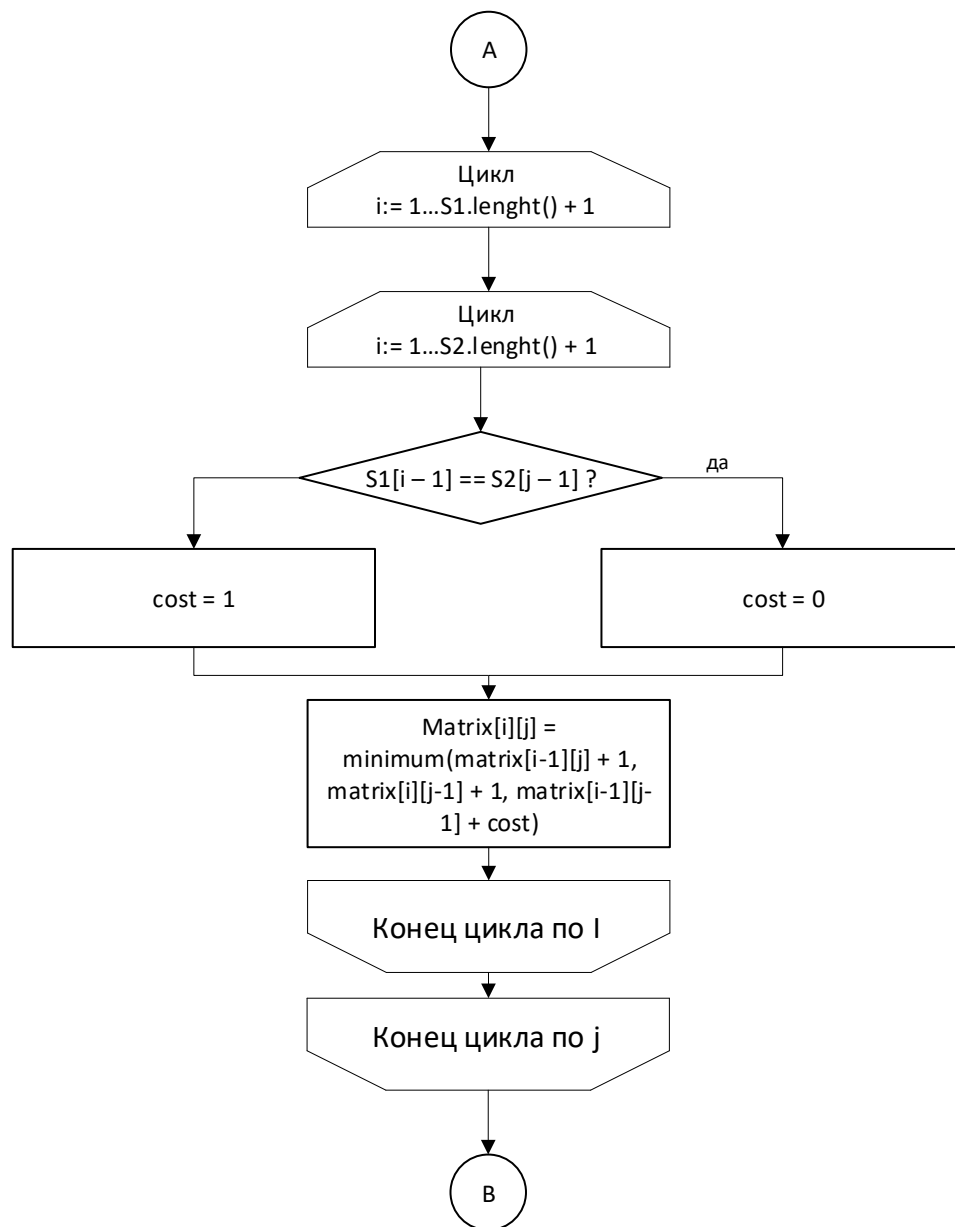


Рисунок 2. Алгоритм Левенштейна часть 2

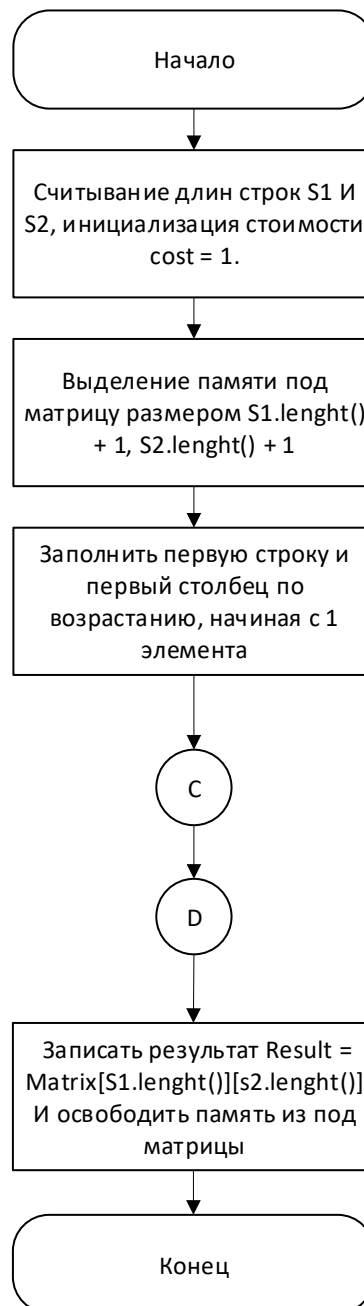


Рисунок 3. Алгоритм Дамерау-Левенштейна часть1

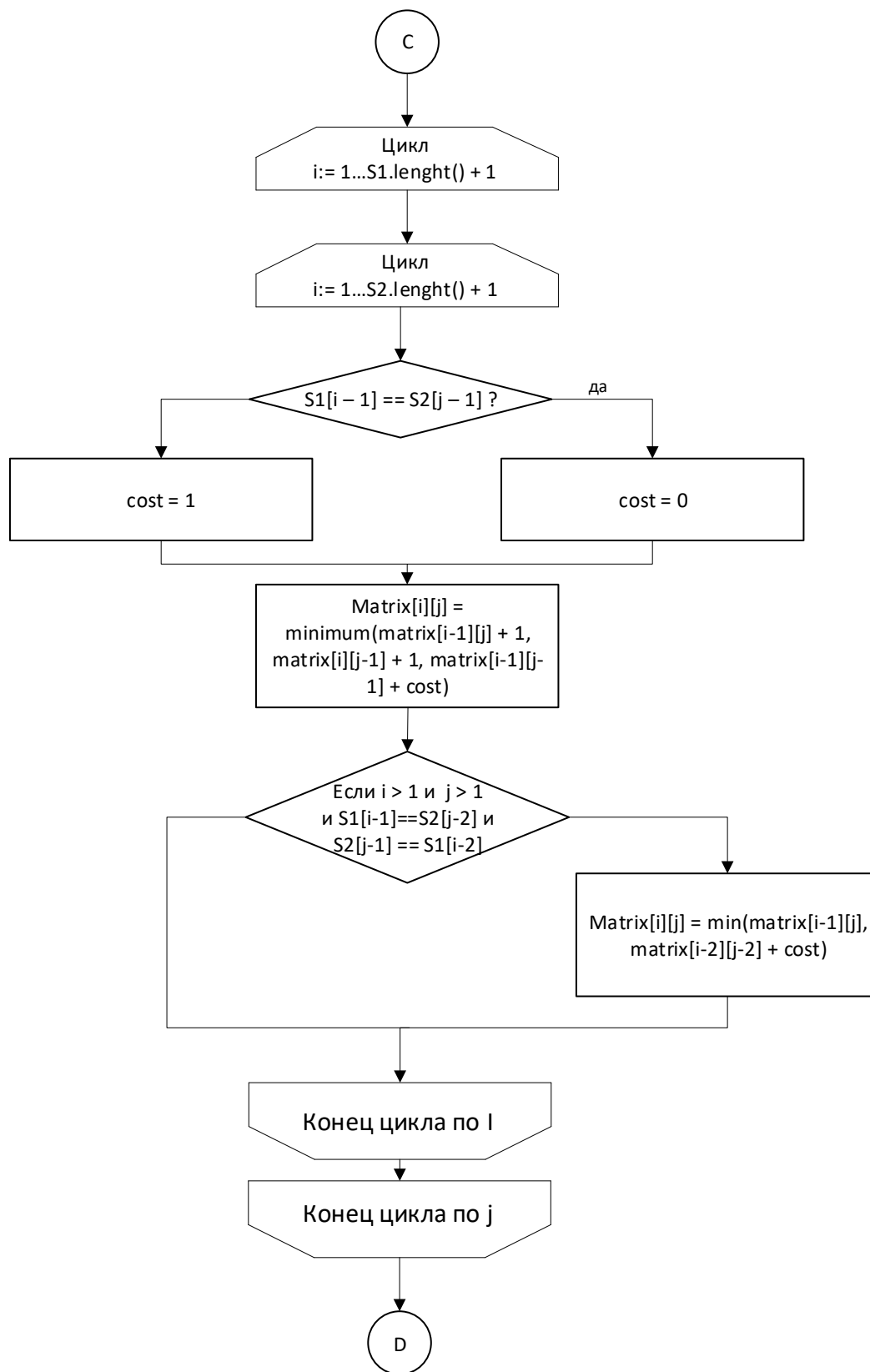


Рисунок 4. Алгоритм Дамерау-Левенштейна часть 2

L – Длина, $S\{n\}$ – взятие подстроки от 0, до n – го символа строки S

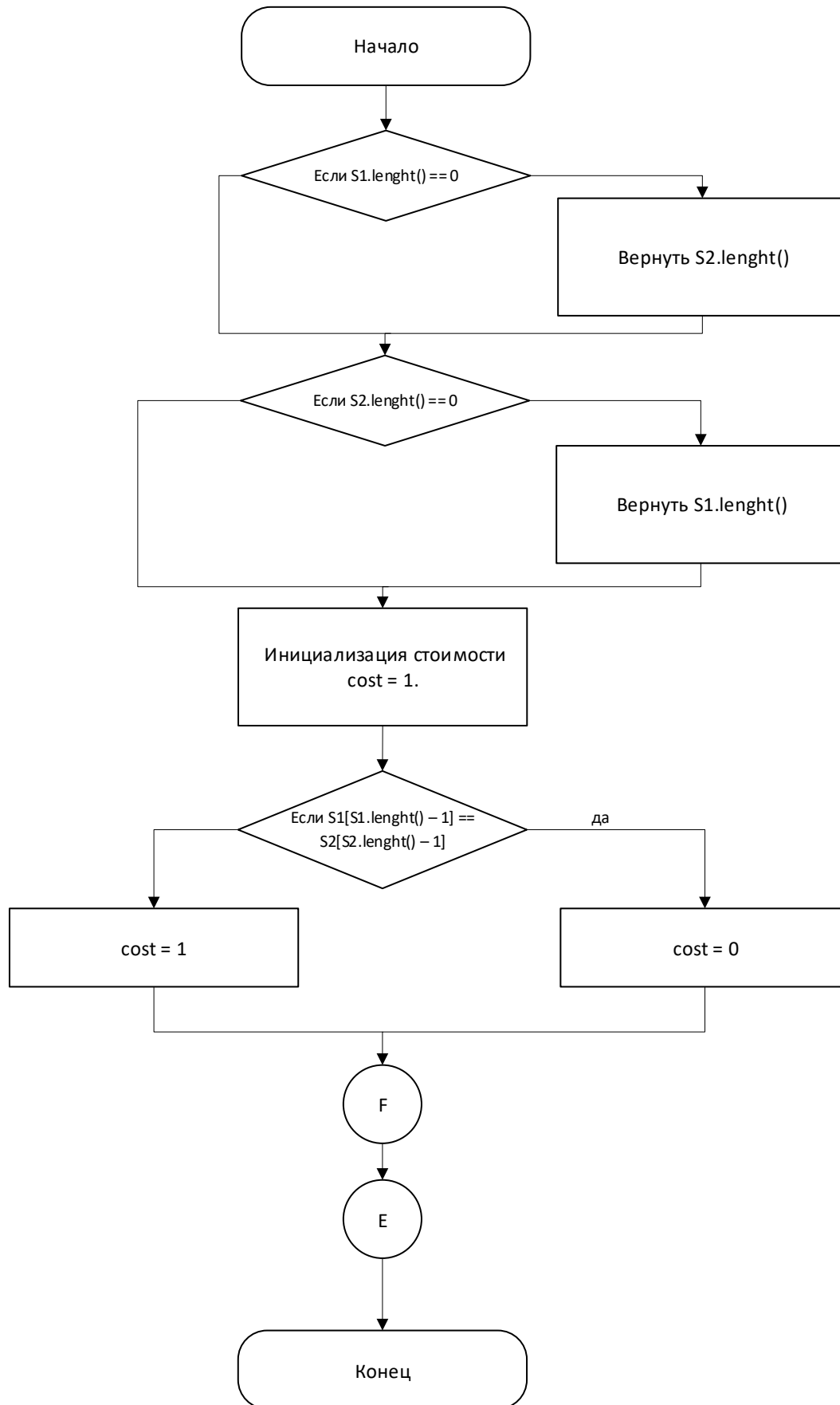


Рисунок 5. Алгоритм Дамерау-Левенштейна рекурсивно часть 1

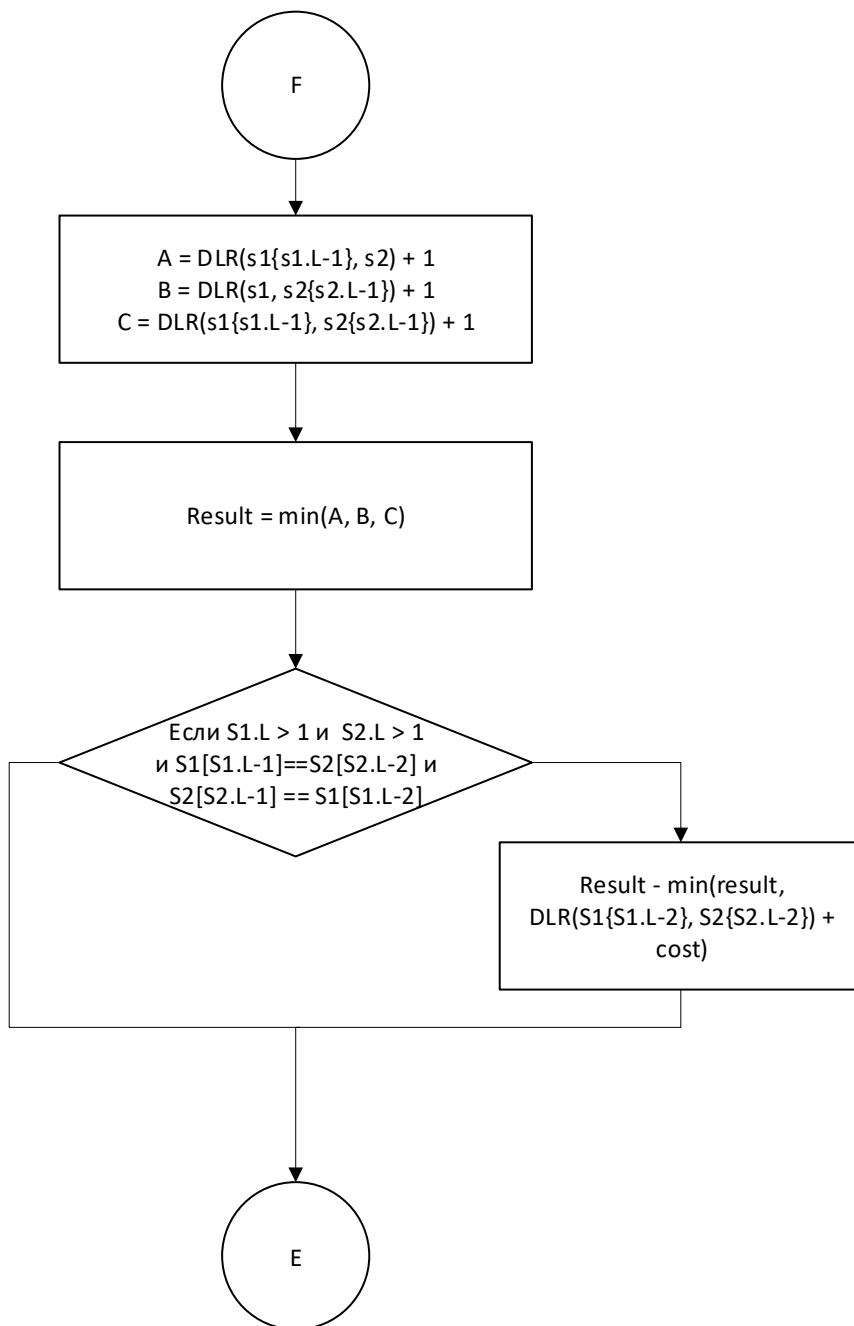


Рисунок 6. Алгоритм Дамерау-Левенштейна рекурсивно часть 2

2.2 Сравнительный анализ рекурсивной и нерекурсивной реализаций

Рекурсивный алгоритм, по сравнению с матричной реализацией работает медленнее, так как при исчерпывание всех возможных комбинаций, возникает ситуация, когда рекурсивные вызовы функций абсолютно идентичны и происходит нерациональная трата ресурсов.

Например:

В Рекурсивной реализации алгоритма Левенштейна при заданных словах «скат», «кот» будет выполнено 2 одинаковых рекурсивных вызова («ска», «ко»). Неполный фрагмент работы:

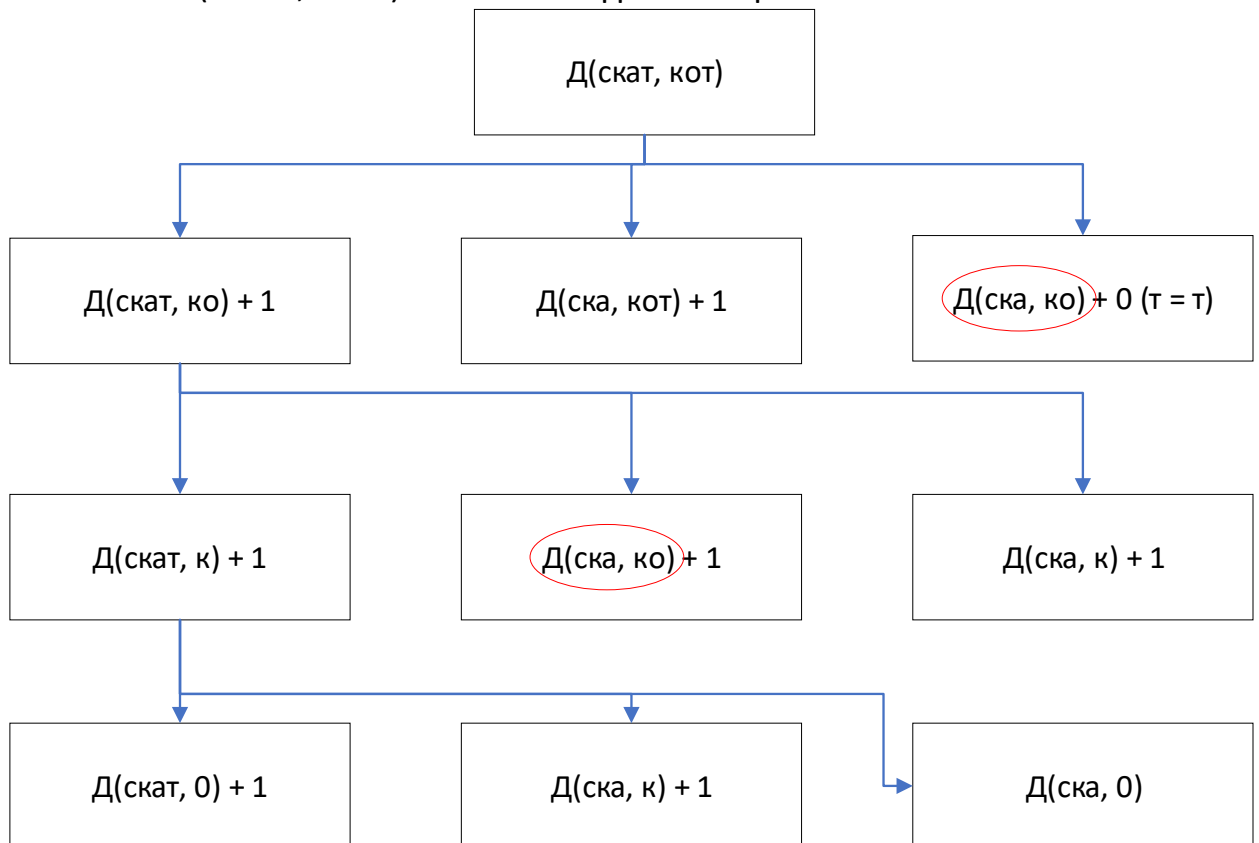


Рисунок 7. Минус рекурсивного алгоритма

Затраты в памяти реализаций алгоритмов так же отличаются:

Пусть $n = s1.length()$; $m = s2.length()$;

Матричная реализация Левенштейна:

- $(n + 1) * (m + 1) * \text{sizeof}(\text{int}) + 6 * \text{int} + 1$ вызов функции $\min()$

Матричная реализация Дамерау-Левенштейна:

- $(n + 1) * (m + 1) * \text{sizeof}(\text{int}) + 6 * \text{int} + 2$ вызова функции $\min()$

Рекурсивная реализация Дамерау-Левенштейна:

- $2 * \text{sizeof}(\text{int}) + 2$ вызова $\min()$ + 4 вызова самой себя

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 C++ в связи с его широким функционалом и быстротой работы, а так же благодаря привычному для меня синтаксису и семантики языка. Среда разработки - Qt. Время работы процессора замеряется с помощью функции:

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

Листинг 1. Функция замера времени

3.3 Листинг кода

```
void fillBaseMatrix(int s1, int s2, int **matrix)
{
    for (int i = 1; i < s1 + 1; i++)
    {
        matrix[i][0] = i;
    }
    for (int j = 1; j < s2 + 1; j++)
    {
        matrix[0][j] = j;
    }
}

int LevensteinMatrix(string s1, string s2)
{
    int row_count = s1.length();
    int column_count = s2.length();
    int **matrix = allocateMatrix(row_count, column_count);
    fillBaseMatrix(s1.length(), s2.length(), matrix);
    int cost = 0;
    for (int i = 1; i < row_count + 1; i++)
    {
        for (int j = 1; j < column_count + 1; j++)
        {
            if (s1[i - 1] == s2[j - 1])
            {
                cost = 0;
            }
            else
            {
                cost = 1;
            }
            matrix[i][j] = minimum(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1, matrix[i - 1][j - 1] + cost);
        }
    }
    //cout << "=== Levenstein Matrix ===" << endl;
    printMatrix(row_count, column_count, matrix);
    int result = matrix[s1.length()][s2.length()];
    clearMatrix(matrix, row_count);
    return result;
}
```

Листинг 2. Реализация Левенштейна с помощью матрицы

```

int DamerauLevensteinRecursion(string s1, string s2)
{
    if (s1.length() == 0)
        return s2.length();
    if (s2.length() == 0)
        return s1.length();

    int cost = 1;
    if (s1[s1.length() - 1] == s2[s2.length() - 1])
        cost = 0;
    else
        cost = 1;
    int result = minimum(DamerauLevensteinRecursion(s1.substr(0,
s1.length() - 1), s2) + 1, DamerauLevensteinRecursion(s1, s2.substr(0,
s2.length() - 1)) + 1,
                        DamerauLevensteinRecursion(s1.substr(0,
s1.length() - 1), s2.substr(0, s2.length() - 1)) + cost);
    if (((s1.length() > 1 && s2.length() > 1) && s1[s1.length() - 1] ==
s2[s2.length() - 2]
        && s2[s2.length() - 1] == s1[s1.length() - 2]))
    {
        result = __min(result, DamerauLevensteinRecursion(s1.substr(0,
s1.length() - 2), s2.substr(0, s2.length() - 2)) + cost);
    }
    //unsigned tick2 = tick();
    //cout << tick1 << " " << tick2 << endl;
    return result;
}

```

Листинг 3. Реализация Дамерау-Левенштейна рекурсивно


```

int DamerauLevinsteinMatrix(string s1, string s2)
{
    int row_count = s1.length();
    int column_count = s2.length();
    int **matrix = allocateMatrix(row_count, column_count);
    fillBaseMatrix(s1.length(), s2.length(), matrix);
    int cost = 1;
    for (int i = 1; i < row_count + 1; i++)
    {
        for (int j = 1; j < column_count + 1; j++)
        {
            if (s1[i - 1] == s2[j - 1])
            {
                cost = 0;
            }
            else
            {
                cost = 1;
            }
            matrix[i][j] = minimum(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1, matrix[i - 1][j - 1] + cost);
            if ((i > 1 && j > 1) && s1[i - 1] == s2[j - 2] && s2[j - 1] == s1[i - 2])
            {
                matrix[i][j] = __min(matrix[i][j], matrix[i - 2][j - 2] + cost);
            }
        }
    }
    int result = matrix[s1.length()][s2.length()];
    clearMatrix(matrix, row_count);
    return result;
}

```

Листинг 4. Реализация алгоритма Дамерау-Левенштейна

3.4 Описание тестирования

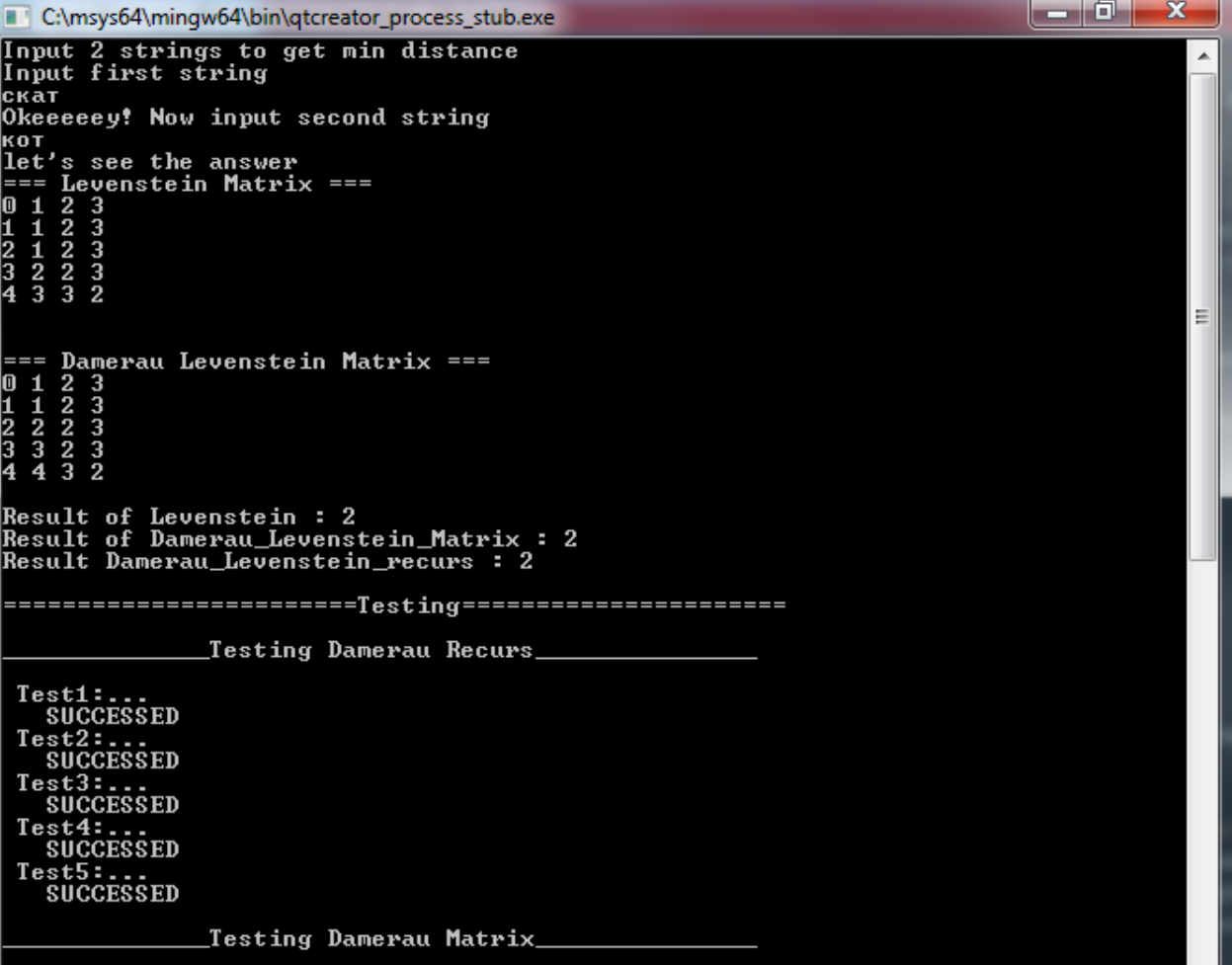
Тестирование будет проведено на следующих примерах, включая пустые строки и строки, содержащие пробелы:

- 1) Развлечения, Увлечение
- 2) Скат, Кот
- 3) " ", " "
- 4) " ", " "
- 5) "Darek", " "

4 Экспериментальная часть

В данном разделе будут приведены примеры работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе экспериментальных данных.

4.1 Примеры работы



```
C:\msys64\mingw64\bin\qtcreator_process_stub.exe
Input 2 strings to get min distance
Input first string
скат
Okeeeeeey! Now input second string
кот
let's see the answer
=== Levenstein Matrix ===
0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2

=== Damerau Levenstein Matrix ===
0 1 2 3
1 1 2 3
2 2 2 3
3 3 2 3
4 4 3 2

Result of Levenstein : 2
Result of Damerau_Levenstein_Matrix : 2
Result Damerau_Levenstein_recurs : 2

=====Testing=====

Testing Damerau Recurs
Test1:...
SUCCEEDED
Test2:...
SUCCEEDED
Test3:...
SUCCEEDED
Test4:...
SUCCEEDED
Test5:...
SUCCEEDED

Testing Damerau Matrix
```

Скриншот 1. Пример работы на словах «скат» и «кот»

```
C:\msys64\mingw64\bin\qtcreator_process_stub.exe
Input 2 strings to get min distance
Input first string
снєкрр
Okeeeey! Now input second string
снєкрр
let's see the answer
=== Levenstein Matrix ===
0 1 2 3 4 5 6
1 0 1 2 3 4 5
2 1 0 1 2 3 4
3 2 1 0 1 2 3
4 3 2 1 1 1 2
5 4 3 2 1 2 2
6 5 4 3 2 2 2

=== Damerau Levenstein Matrix ===
0 1 2 3 4 5 6
1 0 1 2 3 4 5
2 1 0 1 2 3 4
3 2 1 1 1 2 3
4 3 2 1 1 2 3
5 4 3 2 2 1 2
6 5 4 3 3 2 1

Result of Levenstein : 2
Result of Damerau_Levenstein_Matrix : 1
Result Damerau_Levenstein_recurs : 1

=====Testing=====
Testing Damerau Recurs_____

Test1:...
SUCCEEDED
Test2:...
SUCCEEDED
Test3:...
SUCCEEDED
Test4:...
SUCCEEDED
Test5:...
SUCCEEDED
```

Скриншот 2. Пример различия работы двух алгоритмов (Пользователь напечатал одну букву раньше другой)

4.2 Результаты тестирования

```
int mainTest(void)
{
    string strings[10] = {"Razvlecheniya", "Uvlechenie", "Skat",
    "kot", " ", " ", " ", "Darek", "", "", ""};
    cout << "\n_____Testing Damerau Recurs_____ \n"
<< endl;
    DamerauLevensteinRTesting(strings);
    cout << "\n_____Testing Damerau Matrix_____ \n"
<< endl;
    DamerauLevensteinMTesting(strings);
    cout << "\n_____Testing Levenstein_____ \n" <<
endl;
    LevensteinMTesting(strings);
    return 0;
}
```

Листинг 5. Функция тестирования

```
=====Testing=====
_____Testing Damerau Recurs_____
Test1:...
SUCCEEDED
Test2:...
SUCCEEDED
Test3:...
SUCCEEDED
Test4:...
SUCCEEDED
Test5:...
SUCCEEDED
_____Testing Damerau Matrix_____
Test1:...
SUCCEEDED
Test2:...
SUCCEEDED
Test3:...
SUCCEEDED
Test4:...
SUCCEEDED
Test5:...
SUCCEEDED
_____Testing Levenstein_____
Test1:...
SUCCEEDED
Test2:...
SUCCEEDED
Test3:...
SUCCEEDED
Test4:...
SUCCEEDED
Test5:...
SUCCEEDED
```

Скриншот 3. Результат тестирования

№	S1	S2	Ожидаемый результат	Полученный результат
1	Пустая строка	Пустая строка	0, 0, 0	0, 0, 0
2	Пустая строка	Darek	5, 5, 5	5, 5, 5
3	Darek	Пустая строка	5, 5, 5	5, 5, 5
4	Пробел	Пробел	0, 0, 0	0, 0, 0
5	Молот	Молот	0, 0, 0	0, 0, 0
6	Спектр	Спеткр	2, 1, 1	2, 1, 1
7	Спект	Спектр	1, 1, 1	1, 1, 1
8	Спектр	Сбектр	1, 1, 1	1, 1, 1
9	Спектр	Спект	1, 1, 1	1, 1, 1

Таблица 1. Результаты тестирования

В таблице 1 в столбцах результатов указаны 3 значения через запятую, соответствующие реализациям алгоритмов Левенштейна, Дамерау-Левенштейна и рекурсивного алгоритма Дамерау-Левенштейна. Во время тестирования программа корректно сработала на ввод пустых строк и строк содержащие одинаковые символы, а так же на операции удаления («Спектр», «Спект»), добавлении («Спект», «Спектр»), замены («Спектр», «Сбектр») и операции перестановки («Спектр», «Спеткр»).

4.3 Постановка эксперимента по замеру времени [и памяти]

Для произведения замеров времени выполнения реализаций алгоритмов будет использована следующая формула $t = \frac{Tn}{N}$, где t – время выполнения, N – количество замеров. 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.

4.4 Сравнительный анализ на материале экспериментальных данных

Ниже приведены графики зависимости временных затрат (в тиках процессора) от размеров входных данных.

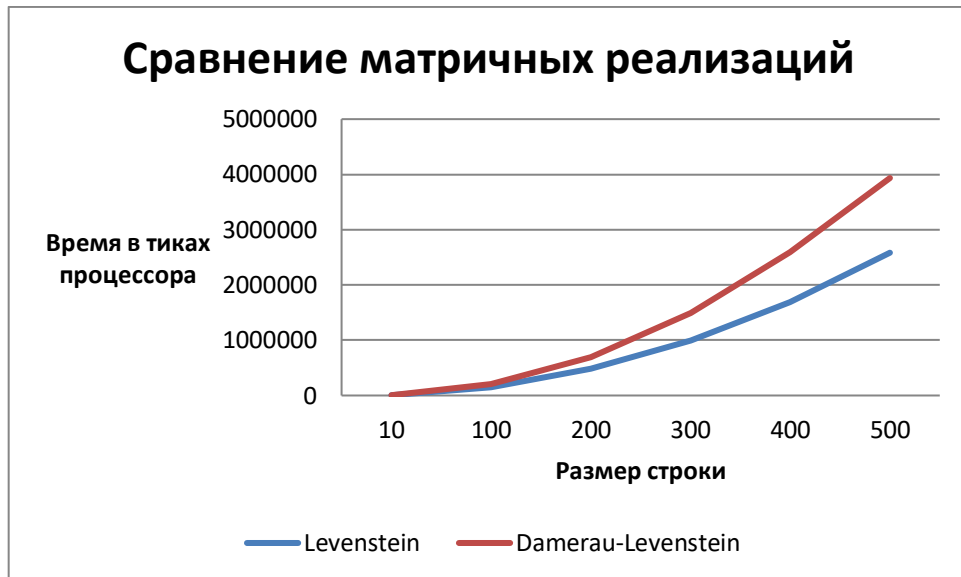


График 1. Сравнение матричных реализаций

По графику видно, что алгоритм Дамерау-Левенштейна работает медленнее алгоритма Левенштейна, так как в нем есть дополнительные проверки.



График 2. Сравнение матричной и рекурсивной реализации алгоритма Дамерау-Левенштейна

В результате проведенного эксперимента был получен следующий вывод:

Рекурсивный алгоритм Дамерау-Левенштейна работает гораздо дольше итеративной реализации, начиная с длины строк = 6, время его работы увеличивается в геометрической прогрессии. Итеративный алгоритм значительно превосходит его по эффективности.

Заключение

В ходе работы были изучены и реализованы в матричной форме алгоритмы Левенштейна и Дамерау-Левенштейна и в рекурсивной алгоритм Дамерау-Левенштейна. Выполнено сравнение рекурсивного и итеративного алгоритмов Левенштейна. Изучены зависимости времени выполнения алгоритмов от длин строк. Также были реализованы 3 описанных алгоритма нахождения расстояний Левенштейна и Дамерау-Левенштейна.