МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнила: Овчинникова Анастасия, ИУ7-55Б

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

Москва, 2019

Оглавление

Введение **2**

Аналитическая часть **4**

Конструкторская часть **8**
Требования к входным данным 8
Требования к программе 8
Схемы алгоритмов 9
Выбор языка программирования 9
Сведения о модулях программы 9
Тесты 17

Исследовательская часть **20**
Постановка эксперимента 20
Сравнительный анализ на материале экспериментальных данных 20
Вывод 23

Заключение **24**

Список литературы **25**

Введение

Расстояние Левенштейна (редакторское расстояние) - это минимальное количество операций (вставки одного символа, удаления одного символа и замены одного символа на другой), которое необходимо для преобразования строки s1 в строку s2. Расстояние Левенштейна широко используется в теории информации, биоинформатике и компьютерной лингвистике:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- для сравнения текстовых файлов утилитой diff и ей подобными;
- в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Дамерау-Левенштейна - это расстояние Левенштейна, расширенное перестановкой соседних символов.

Целью данной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Задачи данной лабораторной:

- изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- применение метода динамического программирования для матричной реализации указанных алгоритмов;
- получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной и в рекурсивной версиях;
- сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- 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;

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

Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Вводятся следующие разрешенные редакторские операции над строкой:

- I (англ. insert) - вставка;
- D (англ. delete) - удаление;
- R (англ. replace) - замена;
- M (англ. match) - совпадение.

При нахождении расстояния Дамерау-Левенштейна добавляется еще одна операция:

- T (англ. transpose) - транспозиция (перестановка соседних символов).

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда редакционное расстояние можно подсчитать по следующей рекуррентной формуле:

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

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

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

Оба алгоритма можно реализовать с помощью таблицы (матрично).

Пусть имеются строки s_1 = 'скат' и s_2 = 'кот'. Необходимо вычислить редакторское расстояние между строками s1 и s2.

Для вычисления расстояния составляется следующая таблица (символ ∅ обозначает пустую строку):

	∅	к	о	т
∅				
с				
к				
а				
т				

При заполнении таблицы для каждой из операций (вставка, удаление, замена, совпадение, транспозиция) нужно учитывать штрафы:

- Вставка - штраф 1;
- Удаление - штраф;
- Замена - штраф 1;
- Транспозиция - штраф 1;
- Совпадение - штраф 0.

В каждую клетку заносится значение $D(s_1[i], s_2[j])$, где $i >= 0, j >= 0$ номера текущей строки и текущего столбца соответственно (нумерация начинается с 0).

На следующем шаге заполняются тривиальные случаи: нулевой столбец и нулевая строка (строки не являются частью таблицы).

	∅	к	о	т
∅	0	1	2	3
с	1			
к	2			
а	3			
т	4			

Для $i > 0, j > 0$ выбирается минимальный ход из трех возможных:

x	y
z	$min(y+1, z+1, x+fine)$

$$\text{где } fine = \begin{cases} 1 & \text{, если } s_1[i] = s_2[j] \\ 0 & \text{, иначе.} \end{cases}$$

Так, для $i = 1, j = 1$:

x =0	y = 1
z = 1	$min(y+1=2, z+1=2, x+fine=x+0=1)$

Заносим в таблицу полученное значение:

	∅	к	о	т
∅	0	1	2	3
с	1	1		
к	2			
а	3			
т	4			

Проделав данную операцию для всех клеток, получим:

	∅	к	о	т
∅	0	1	2	3
с	1	1	2	3
к	2	1	2	3
а	3	2	2	3
т	4	3	3	**2**

Искомое редакторское расстояние выделено жирным шрифтом.

При нахождении расстояния Дамерау-Левенштейна матричным способом при вычислении нетривиальных случаев необходимо учитывать еще один возможный ход:

g

x	y
z	$min(y+1, z+1, x+fine, g+1)$

$$g \text{ учитывается, только если } \begin{cases} len(s_1) >= 2 \\ len(s_2) >= 2 \\ s_1[i] = s_2[j-1] \\ s_2[j] = s_1[i-1] \end{cases}$$

где $len(s)$ - длина строки s.

Конструкторская часть

Требования к входным данным

1. На вход программе подается две строки.
2. Строки могут быть пустыми.
3. Строчные и прописные буквы считаются разными.

Требования к программе

Программа представляет собой консольное приложение с меню для выбора пользователем необходимого действия. Меню содержит следующие пункты:

- "1 - расстояние Левенштейна матрично";
- "2 - расстояние Левенштейна рекурсивно";
- "3 - расстояние Дамерау-Левенштейна матрично";
- "4 - расстояние Дамерау-Левенштейна рекурсивно";
- "5 - все сразу (1-4)";
- "6 - временной анализ";
- "7 - тесты".

Для выбора соответствующих пунктов меню пользователь вводит соответствующую цифру (1-7). При вводе любого другого символа программа должна корректно завершаться.

При выборе пунктов 1-5 программа попросит пользователя ввести две строки. Ввод пустых строк считается корректым, программа при этом не должна аварийно завершаться.

Схемы алгоритмов

Схема матричного алгоритма для вычисления расстояния Левенштейна представлена на рисунке 1. Схема матричного алгоритма для вычисления расстояния Дамерау-Левенштейна представлена на рисунке 2. Схема рекурсивного алгоритма для вычисления расстояния Левенштейна представлена на рисунке 3. Схема рекурсивного алгоритма для вычисления расстояния Дамерау-Левенштейна представлена на рисунке 4.

Выбор языка программирования

В качестве языка программирования для реализации программы был выбран язык C++ и фреймворк Qt, потому что:

- язык C++ имеет высокую вычислительную производительность;
- язык C++ поддерживает различные стили программирования;
- стандартные строки C++ (std::string) не поддерживают юникод, поэтому вместо std::string были использованы строки Qt (QString);
- в Qt существует удобный инструмент для тестирования - QtTest - который позволяет собирать тесты в группы, собирать результаты выполнения тестов, а также уменьшить дублирование кода при схожих объектах тестирования.

Для замеров времени использовалась функция clock() модуля ctime. Эта функция возвращает количество временных тактов, прошедших с начала запуска программы. С помощью макроса CLOCKS_PER_SEC можно узнать количество пройденных тактов за 1 секунду.

Сведения о модулях программы

Программа состоит из следующих файлов:

- devenshteindistance.h, devenshteindistance.cpp - заголовочный файл и файл, в котором расположена реализация алгоритмов;
- main.cpp - главный файл программы, в котором расположена реализация меню;
- testdistance.h, testdistance.cpp - файл и заголовочный файл, в котором расположена реализация тестов.

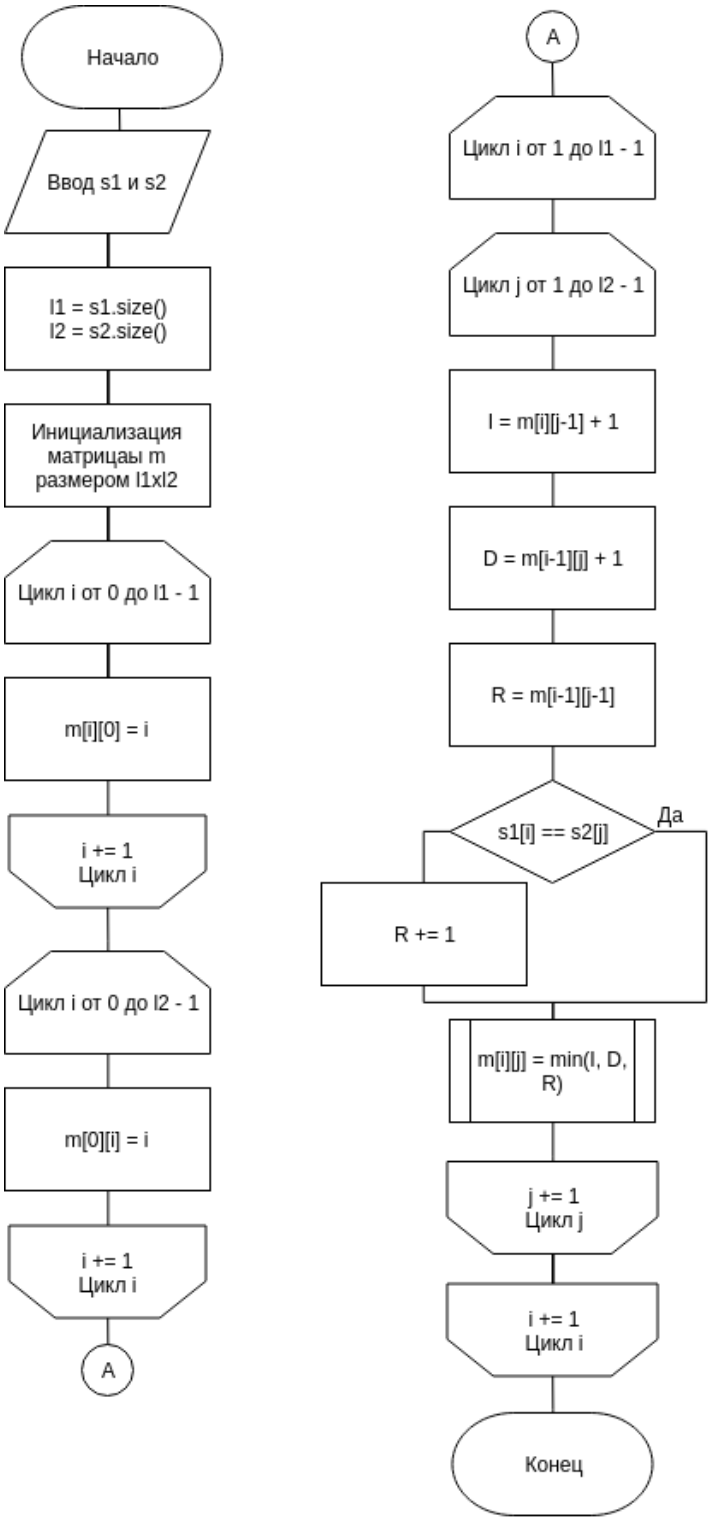


Рис. 1: Матричный алгоритм для вычисления расстояния Левенштейна

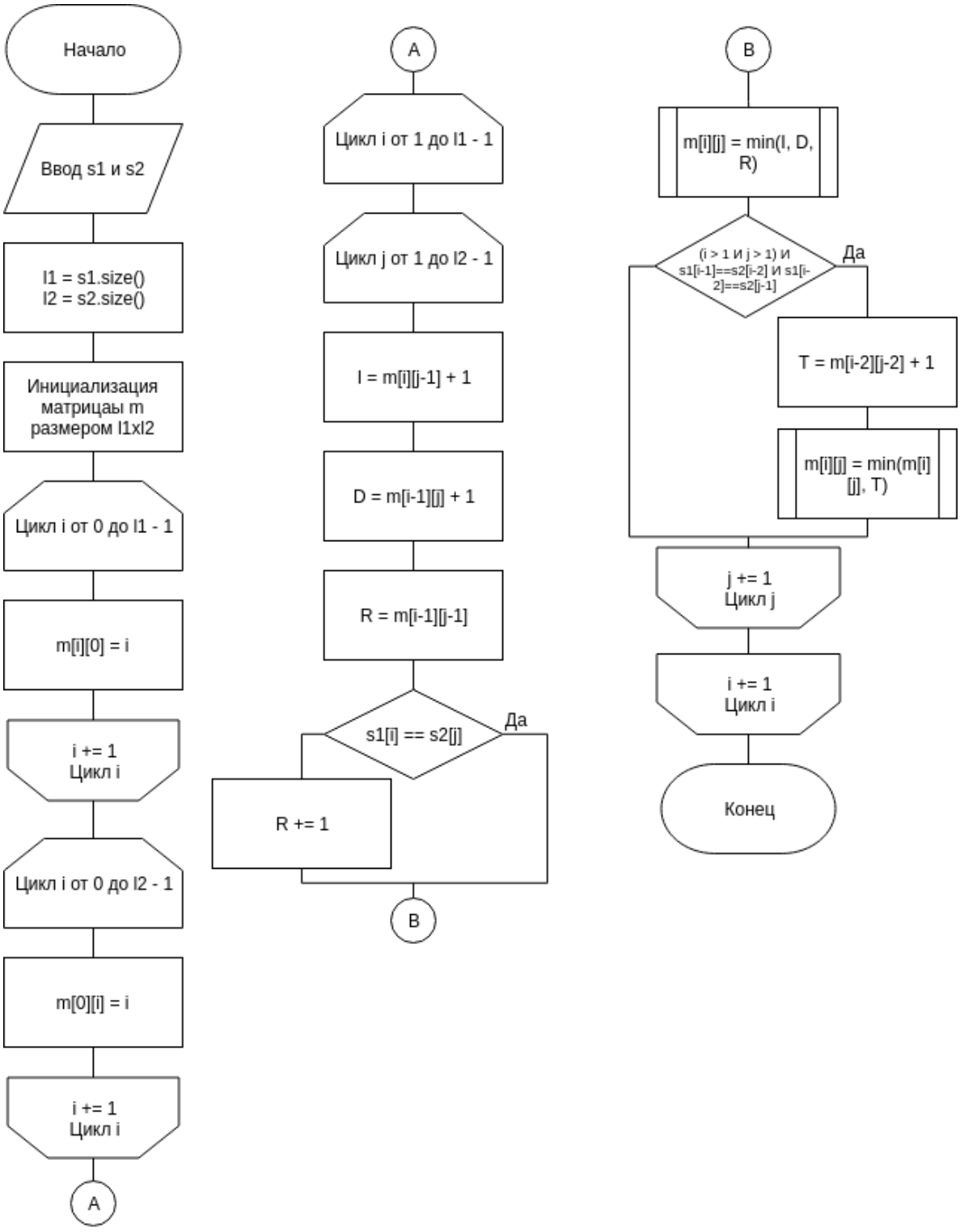


Рис. 2: Матричный алгоритм для вычисления расстояния Дамерау-Левенштейна

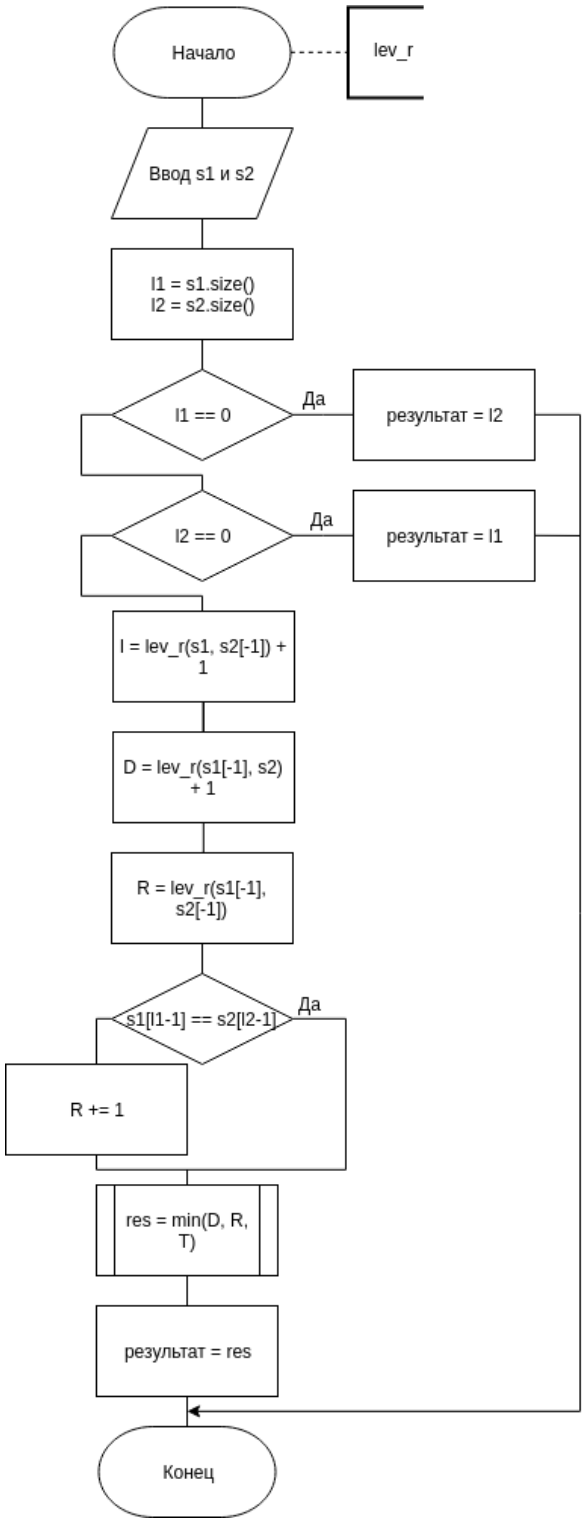


Рис. 3: Рекурсивный алгоритм для вычисления расстояния Левенштейна

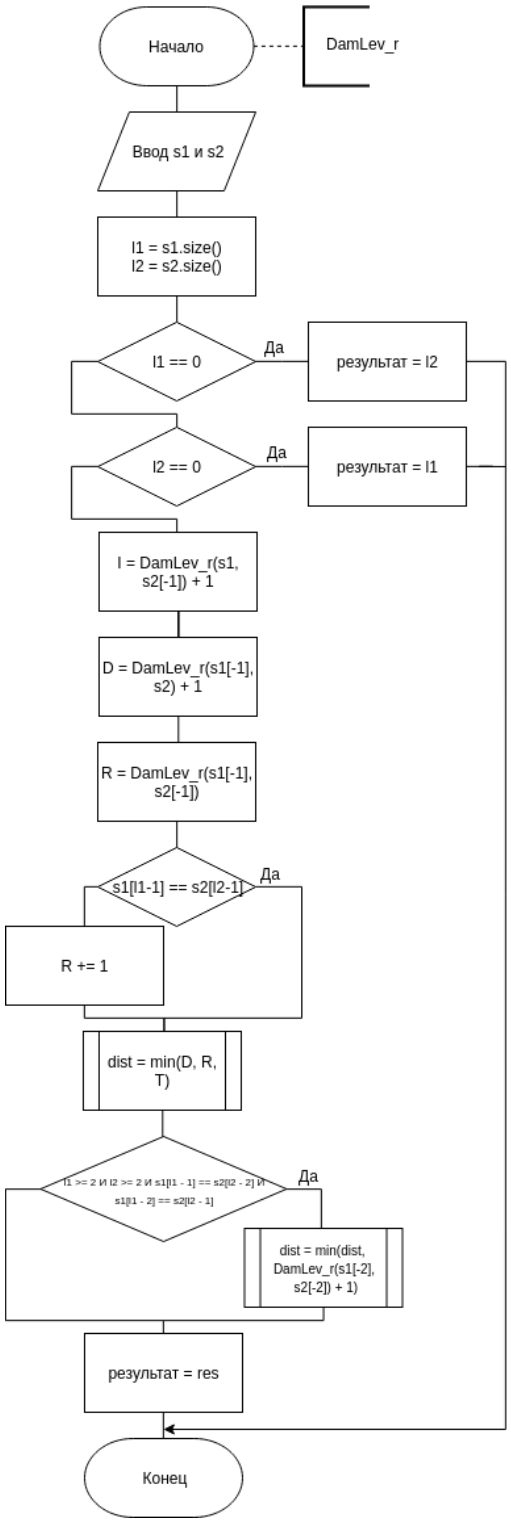


Рис. 4: Рекурсивный алгоритм для вычисления расстояния Дамерау-Левенштейна

Листинг 1: Функция для нахождения расстояния Левенштейна матрично

```
int LevenshteinDistance::levenshteinDistance_m(QString s1, QString s2)
{
    int l1 = s1.size() + 1;
    int l2 = s2.size() + 1;

    int * matrix = (int*) calloc(l1 * l2, sizeof (int));

    for (int i = 0; i < l1; ++i)
    {
        for (int j = 0; j < l2; ++j)
        {
            matrix[i*l2 + j] = 0;
        }
    }

    for (int i = 1; i < l2; ++i)
        matrix[i] = i;
    for (int i = 1; i < l1; ++i)
        matrix[i*l2] = i;

    for (int i = 1; i < l1; ++i)
    {
        for (int j = 1; j < l2; ++j)
        {
            int r1 = matrix[(i - 1)*l2 + j] + 1;
            int r2 = matrix[i*l2 + (j - 1)] + 1;

            int fine = 1;
            if (s1.at(i - 1) == s2.at(j - 1))
            {
                fine = 0;
            }
            int r3 = matrix[(i - 1)*l2 + (j - 1)] + fine;
            matrix[i*l2 + j] = myMin(r1, r2, r3);
        }
    }
    if (printMatrixFlag)
    {
        cout << "Matrix: " << endl;
        printMatrix(matrix, l1, l2);
    }
    int d = matrix[(l1 - 1)*l2 + (l2 - 1)];
    free(matrix);
    return d;
}
```

Листинг 2: Функция для нахождения расстояния Левенштейна рекурсивно

```
int LevenshteinDistance::levenshteinDistance_r(QString s1, QString s2)
{
```

```
3       if (s1.size() == 0 && s2.size() == 0)
4           return 0;
5       if (s1.size() == 0)
6           return s2.size();
7       if (s2.size() == 0)
8           return s1.size();
9
10      QString newS1 = s1.left(s1.size() - 1);
11      QString newS2 = s2.left(s2.size() - 1);
12      int r1 = levenshteinDistance_r(newS1, s2) + 1;
13      int r2 = levenshteinDistance_r(s1, newS2) + 1;
14      int r3 = levenshteinDistance_r(newS1, newS2);
15      if (s1.at(s1.size() - 1) != s2.at(s2.size() - 1))
16          r3 += 1;
17      int distance = myMin(r1, r2, r3);
18      return distance;
19  }
```

Листинг 3: Функция для нахождения расстояния Дамерау-Левенштейна матрично

```
1   int LevenshteinDistance::damerauLevenshteinDistance_m(QString s1, QString
        s2)
2   {
3       int l1 = s1.size() + 1;
4       int l2 = s2.size() + 1;
5
6       int * matrix = (int*) calloc(l1 * l2, sizeof (int));
7
8       for (int i = 0; i < l1; ++i)
9       {
10          for (int j = 0; j < l2; ++j)
11          {
12              matrix[i*l2 + j] = 0;
13          }
14      }
15
16      for (int i = 1; i < l2; ++i)
17          matrix[i] = i;
18      for (int i = 1; i < l1; ++i)
19          matrix[i*l2] = i;
20
21      for (int i = 1; i < l1; ++i)
22      {
23          for (int j = 1; j < l2; ++j)
24          {
25              int r1 = matrix[(i - 1)*l2 + j] + 1;
26              int r2 = matrix[i*l2 + (j - 1)] + 1;
27
28              int fine = 1;
29              if (s1.at(i - 1) == s2.at(j - 1))
30              {
```

```
31                  fine = 0;
32              }
33              int r3 = matrix[(i - 1)*l2 + (j - 1)] + fine;
34              int distance = myMin(r1, r2, r3);
35
36              if (i > 1 && j > 1)
37              {
38                  if (s1.at(i-1) == s2.at(j-2) && s2.at(j-1) == s1.at(i-2))
39                      distance = min(distance, matrix[(i-2)*l2 + (j-2)] +
                            1);
40              }
41              matrix[i*l2 + j] = distance;
42          }
43      }
44      if (printMatrixFlag)
45      {
46          cout << "Matrix: " << endl;
47          printMatrix(matrix, l1, l2);
48      }
49      int d = matrix[(l1 - 1)*l2 + (l2 - 1)];
50      free(matrix);
51      return d;
52  }
```

Листинг 4: Функция для нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1   int LevenshteinDistance::damerauLevenshteinDistance_r(QString s1, QString
        s2)
2   {
3       if (s1.size() == 0 && s2.size() == 0)
4           return 0;
5       if (s1.size() == 0)
6           return s2.size();
7       if (s2.size() == 0)
8           return s1.size();
9
10      int l1 = s1.size();
11      int l2 = s2.size();
12
13      QString newS1 = s1.left(l1 - 1);
14      QString newS2 = s2.left(l2 - 1);
15      int r1 = levenshteinDistance_r(newS1, s2) + 1;
16      int r2 = levenshteinDistance_r(s1, newS2) + 1;
17      int r3 = levenshteinDistance_r(newS1, newS2);
18      if (s1.at(l1 - 1) != s2.at(l2 - 1))
19          r3 += 1;
20      int distance = myMin(r1, r2, r3);
21
22      if (l1 >= 2 && l2 >= 2 && s1.at(l1 - 1) == s2.at(l2 - 2) &&
23              s1.at(l1 - 2) == s2.at(l2 - 1))
```

```
24      {
25          newS1 = s1.left(l1 - 2);
26          newS2 = s2.left(l2 - 2);
27          distance = min(distance, damerauLevenshteinDistance_r(newS1,
                newS2) + 1);
28      }
29
30      return distance;
31  }
```

Тесты

Тестирование проводилось с помощью модуля QtTest. Для этого был написал класс **TestDistance**, который тестирует написанные алгоритмы. Тестирование алгоритмов проводилось в три шага.

Сначала каждый из алгоритмов тестировался оттельно на общем для всех четырех алгоритмов наборе тестов. Набор тестов приведен в листинге 5. Затем проводились тесты, специфичные для каждого алгоритма (для каждой пары алгоритмов). Набор тестов приведен в листинге 6. После этого сравнивались результаты работы двух функций: функция нахождения расстояния Левенштейна матрично с функцией расстояния Левенштейна рекурсивно, функция нахождения расстояния Дамерау-Левенштейна матрично с функцией нахождения расстояния Дамерау-Левенштейна рекурсивно. Набор тестов приведен в листинге 7. При этом использовасля генератор случайных строк. Он приведен в листинге 8.

Листинг 5: Общие тесты

```
1  void TestDistance::testEmptyString()
2  {
3      for (int i = 0; i < methodsToTest; ++i)
4      {
5          QVERIFY((testObj.*methodsPtr[i])("", "") == 0);
6          QVERIFY((testObj.*methodsPtr[i])("1", "") == 1);
7          QVERIFY((testObj.*methodsPtr[i])("12", "") == 2);
8          QVERIFY((testObj.*methodsPtr[i])("", "123456789") == 9);
9      }
10 }
11
12 void TestDistance::testSameStrings()
13 {
14     for (int i = 0; i < methodsToTest; ++i)
15     {
16         QVERIFY((testObj.*methodsPtr[i])("abc", "abc") == 0);
17         QVERIFY((testObj.*methodsPtr[i])("ABC", "ABC") == 0);
18         QVERIFY((testObj.*methodsPtr[i])("", "") == 0);
19         QVERIFY((testObj.*methodsPtr[i])("a", "a") == 0);
20     }
```

```
}

void TestDistance::testDifferentStrings()
{
    for (int i = 0; i < methodsToTest; ++i)
    {
        QVERIFY((testObj.*methodsPtr[i])("a", "") == 1);
        QVERIFY((testObj.*methodsPtr[i])("", "1") == 1);
        QVERIFY((testObj.*methodsPtr[i])("b", "c") == 1);
        QVERIFY((testObj.*methodsPtr[i])("bc", "b") == 1);
        QVERIFY((testObj.*methodsPtr[i])("bc", "c") == 1);
        QVERIFY((testObj.*methodsPtr[i])("ab", "cd") == 2);
        QVERIFY((testObj.*methodsPtr[i])("ab", "AB") == 2);
        QVERIFY((testObj.*methodsPtr[i])("Cd", "cd") == 1);

        QVERIFY((testObj.*methodsPtr[i])("skat", "kot") == 2);
        QVERIFY((testObj.*methodsPtr[i])("skat", "kat") == 1);
        QVERIFY((testObj.*methodsPtr[i])("skatskat", "skat") == 4);
        QVERIFY((testObj.*methodsPtr[i])("SKAT", "skat") == 4);
        QVERIFY((testObj.*methodsPtr[i])("SkAt", "skat") == 2);
    }
}
```

Листинг 6: специфичные тесты

```
void TestDistance::test_levenshteinDistance_r()
{
    QVERIFY(testObj.levenshteinDistance_r("ac", "ca") == 2);
    QVERIFY(testObj.levenshteinDistance_r("abc", "cba") == 2);
}

void TestDistance::test_damerauLevenshteinDistance_m()
{
    QVERIFY(testObj.damerauLevenshteinDistance_m("ac", "ca") == 1);
    QVERIFY(testObj.damerauLevenshteinDistance_m("abc", "cba") == 2);
}
```

Листинг 7: сравнение результатов работы двух алгоритмов

```
void TestDistance::testPairsEmpty()
{
    for (int i = 0; i < testsPerPair; ++i)
    {
        QString s1 = "";
        QString s2 = randomString(i);
        QVERIFY(testObj.levenshteinDistance_m(s1, s2) == testObj.
            levenshteinDistance_r(s1, s2));
        QVERIFY(testObj.levenshteinDistance_m(s2, s1) == testObj.
            levenshteinDistance_r(s2, s1));
    }
}
```

```
11
12 void TestDistance::testPairEqualLen()
13 {
14     for (int i = 0; i < testsPerPair; ++i)
15     {
16         QString s1 = randomString(i);
17         QString s2 = randomString(i);
18         QVERIFY(testObj.levenshteinDistance_m(s1, s2) == testObj.
               levenshteinDistance_r(s1, s2));
19         QVERIFY(testObj.levenshteinDistance_m(s2, s1) == testObj.
               levenshteinDistance_r(s2, s1));
20     }
21 }
22
23 void TestDistance::testPairDifferentLen()
24 {
25     for (int i = 0; i < testsPerPair; ++i)
26     {
27         int l1 = 1 + rand() % 10;
28         int l2 = 1 + rand() % 10;
29         QString s1 = randomString(l1);
30         QString s2 = randomString(l2);
31         QVERIFY(testObj.levenshteinDistance_m(s1, s2) == testObj.
               levenshteinDistance_r(s1, s2));
32         QVERIFY(testObj.levenshteinDistance_m(s2, s1) == testObj.
               levenshteinDistance_r(s2, s1));
33     }
34 }
```

Листинг 8: Генератор случайных строк

```
1 QString TestDistance::randomString(int size)
2 {
3     string symbols = "
         abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ1234567890";
4     int symbolsSize = symbols.size();
5     QString res = "";
6     for (int i = 0; i < size; ++i)
7     {
8         res += symbols.at(rand() % (symbolsSize - 1));
9     }
10    return res;
11 }
```

Все написанные тесты были пройдены.

Исследовательская часть

Постановка эксперимента

В рамках данной работы были проведены следующие эксперименты.

1. Сравнение по времени матричных алгоритмов для вычисления расстояний Левенштейна и Дамерау-Левенштейна. Сравнение проводилось для строк длиной от 1 до 1000 с шагом 50. Каждый эксперимент проводился 100 раз.

2. Сравнение по времени рекурсивного и итеративного алгоритмов для вычисления расстояния Дамерау-Левенштейна. Сравнение проводилось для строк длиной от 1 до 10 с шагом 1. Каждый эксперимент проводился 10 раз.

Измерения проводились на компьютере HP Pavilion Notebook на базе Intel Core i5-7200U, 2.50 Гц с 6 Гб оперативной памяти под управлением операционной системы Linux Mint.

Сравнительный анализ на материале экспериментальных данных

Результаты замеров времени для сравнения времени работы матричных алгоритмов для вычисления расстояний Левенштейна и Дамерау-Левенштейна приведены в таблице 1. Зависимость времени работы матричных алгоритмов для вычисления рассояния Левенштейна и Дамерау-Левенштейна от длины входных строк представлена и на рисунке 5.

Условные обозначения в таблицах и на графиках:

- Lev(M) - время работы матричного алгоритма для вычисления расстояния Левенштейна в секундах;

- DamLev(M) - время работы матричного алгоритма для вычисления расстояния Дамерау-Левенштейна в секундах;

- Lev(R) - время работы рекурсивного алгоритма для вычисления расстояния Левенштейна в секундах;

Таблица 1: Сравнение времени работы матричных алгоритмов для вычисления расстояний Левенштейна и Дамерау-Левенштейна

Длина строки	Lev(M), с	DamLev(M), с
0	2.58e-06	5.4e-07
50	0.00032896	0.00017596
100	0.00067026	0.00071314
150	0.00100624	0.00159782
200	0.00176384	0.0028283
250	0.00273216	0.00448456
300	0.00392608	0.00650698
350	0.00531646	0.00876206
400	0.00692414	0.0114267
450	0.0087609	0.0145368
500	0.0108672	0.0179466
550	0.0131632	0.0217669
600	0.0156044	0.0257943
650	0.0182917	0.0303514
700	0.0212066	0.0356014
750	0.0244535	0.0403816
800	0.0281559	0.0460019
850	0.0314529	0.0519328
900	0.0354612	0.0581385
950	0.0392116	0.0649824
1000	0.0433937	0.0723375

- DamLev(R) - время работы рекурсивного алгоритма для вычисления расстояния Дамерау-Левенштейна в секундах.

Результаты сравнения по времени рекурсивного и итеративного алгоритмов для вычисления расстояния Дамерау-Левенштейна приведены в таблице 2 и на рисунке 6.

Замеры времени для более длинных строк не проводились, так как время работы рекурсивного алгоритма увеличивается экспоненциально, пропорционально количеству рекурсивных вызовов (максимальная глубина рекурсии равна длине самого длинного слова).

При увеличении длинны входных строк становится видна очень сильная разница по времени выполнения между рекурсивным и матричным алгоритмами. Уже при длине строки в 10 символов матричная реализация оказывается на порядок быстрее.

Рис. 5: Сравнение работы матричных алгоритмов для вычисления расстояния Левенштейна и Дамерау-Левенштейна

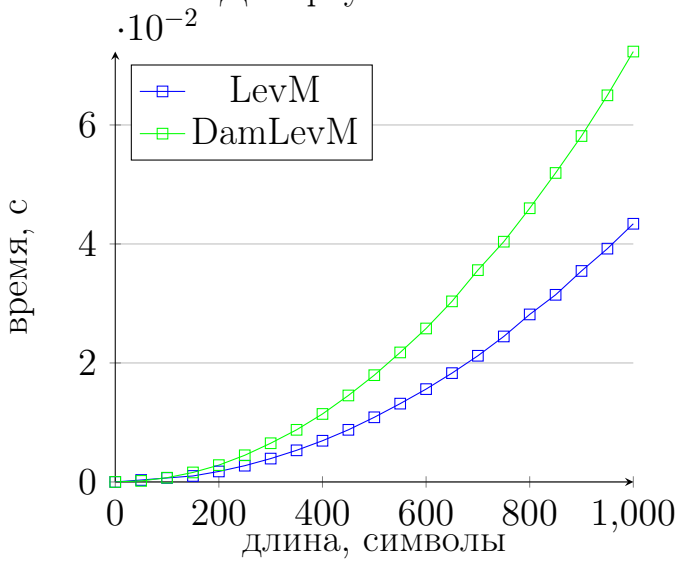
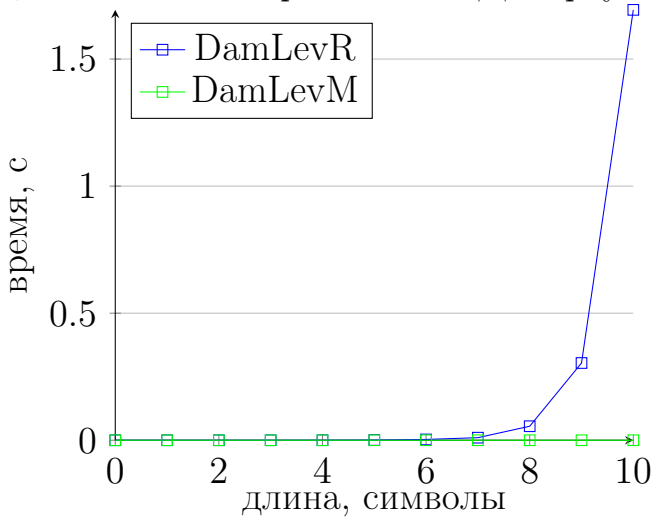


Таблица 2: Сравнение времени выполнения рекурсивного и итеративного алгоритмов для вычисления расстояния Дамерау-Левенштейна

Длина строки	DamLev(M), с	DamLev(R), с
0	2.2e-06	1.9e-06
1	1.9e-06	3.6e-06
2	2.7e-06	1.04e-05
3	3.6e-06	3.97e-05
4	4.7e-06	0.0001972
5	6.4e-06	0.0008738
6	8.2e-06	0.0030926
7	1.11e-05	0.0101053
8	1.38e-05	0.0550824
9	1.69e-05	0.303918
10	2.1e-05	1.69359

Рис. 6: Сравнение времени выполнения рекурсивного и итеративного алгоритмов для вычисления расстояния Дамерау-Левенштейна



Вывод

Из приведенных графиков можно увидеть, что матричный алгоритм вычисления расстояния Левенштейна и матричный алгоритм вычисления расстояния Дамерау-Левенштейна сравнимы между собой и имеют одинаковый характер (время их выполненя в зависимости от длины входных строк растет одинаково). Однако алгоритм Дамерау-Левенштейна имеет большее время выполнения, так как имеет более сложную логику.

Кроме того, проведенный эксперимент показал, что рекурсивные алгоритмы становятся неприемлемыми для использования при длинах строк больше 10 символов.

Заключение

В ходе данной работы был изучен метод динамического программирования на примере алгоритмов Левенштейна и Дамерау-Левенштейна. Были получены практические навыки реализации данных алгоритмов в рекурсивных и матричных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате проведенных исследований были получены выводы, что матричная реализация алгоритмов не только значительно выигрывает по времени при росте длины строк, но и имеет допустимое время отклика при длине строк более 10 симоволов, в то время как рекурсивная реализация имеет недопустимо большое время отклика.

Список литературы

1. Дж. Маконелл. Анализ алгоритмов. Активный обучающий подход. - М.:Техносфера, 2009.