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»

Лабораторная работа № 1

Дисциплина Анализ алгоритмов

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Искакова Карина

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Цель данной лабораторной работы — изучить и применить метод динамического программирования на материале агоритмов Левенштейна и Дамерау-Левенштейна, а также получить практические навыки реализации указанных алгоритмов.

Задачами данной лабораторной работы являются:

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. Аналитическая часть

В данном разделе будут рассмотренны алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1. Расстояние Левенштейна

Расстояние Левенштейна — это минимальное количество редакторских операций, которые необходимы для превращения одной строки в другую [1].

Редакторские операции — $\underbrace{\text{вставка (I), замена (R), удаление (D)}}_{\text{штраф}}$

Расстояние Левенштейна применяется в следующих областях:

1) *поисковиках и текстовых редакторах* — автоисправление, автозамена;
2) *биоинформатике* для сравнения генов, хромосом и белков. Например, в белке каждая молекула представляется буквой из ограниченного алфавита, а белки представляются строками, которые можно сравнить.

Для нахождения расстояния Левенштейна используется рекурентная формула для устранения проблемы взаимного выравнивания строк. Пример нахождения расстояния Левенштейна изображен на рис. 1:

Рис. 1. Пример нахождения расстояния Левенштейна

Это не минимальное расстояние Левенштейна, следовательно существует проблема взаимного выравнивания строк. Более удачный вариант и зображен на рис. 2:

Рис. 2. Пример эффективного нахождения расстояния Левенштейна

Математическая формулировка алгоритма

Пусть даны 2 строки $S_1[1..i]$ и $S_2[1..j]$ длиной i и j соответственно. Расстояние в алгоритме Левенштейна можно посчитать с помощью рекурентной формулы:

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} 0, \text{ если } i = 0, j = 0 \\ i, \text{ если } j = 0, i > 0 \\ j, \text{ если } i = 0, j > 0 \\ min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1 & //I \\ D(S_1[1..i-1], S_2[1..j]) + 1 & //D \\ D(S_1[1..i-1], S_2[1..j-1]) + & \\ \begin{bmatrix} 0, \text{ если } S_1[i] == S_2[j], i > 0, j > 0 & //M \\ 1, \text{ иначе} & //R \end{bmatrix} \end{cases} \end{cases}$$

где операции:

1) вставка — I, штраф = 1;
2) замена — R, штраф = 1;
3) удаление — D, штраф = 1;
4) совпадение — M, штраф = 0.

Можно описать поиск расстояния Левенштейна разными алгоритмами:

1) матричное рекурсивное расстояние;
2) рекурсивный расчет по формуле *(проблема - много лишних вычислений)*;
3) рекурсивный алгоритм, заполняющий незаполненные клетки матрицы *(по аналогии с ∞ в алгоритме Дейкстры поиска расстояний в графе)*.

1.2. Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна включает также операцию перестановки двух соседних символов - *транспозицию (T)*.

$D('ab', 'ba') = 1$ — есть T
$D('ab', 'cd')$ — невозможна транспозиция

Модификаця была введена, так как большинство ошибок пользователей при печати текста - банальные опечатки. Например, *"аглоритм"* вместо *"алгоритм"*.

Математическая формулировка алгоритма

Пусть даны 2 строки $S_1[1..i]$ и $S_2[1..j]$ длиной i и j соответственно.

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

Вывод

Были рассмотрены алгоритмы нахождения расстояния Левенштейна и его усовершенствованный алгоритм нахождения расстояния Дамерау-Левенштейна, принципиальная разница которого — наличие транспозиции, а также области применения данных алгоритмов.

2. Конструкторская часть

2.1. Требования к программе

К программе предоставлены следующие требования:

1) на ввод подается 2 строки;
2) uppercase и lowercase буквы считаются разными;
3) программа должна вывести расстояние и матрицу, если она использовалась;
4) две пустые строки — корректный ввод, программа не должна аваийно завершаться.

2.2. Схемы алгоритмов

В данном разделе будут рассмотрены схемы следующих алгоритмов:

1) матричного алгоритма нахождения расстояния Левенштейна (рис. 3);
2) рекурсивного алгоритма нахождения расстояния Левенштейна (рис. 4);
3) рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы (рис. 5);
4) матричного алгоритма нахождения расстояния Дамерау-Левенштейна (рис. 6);
5) рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна (рис. 7).

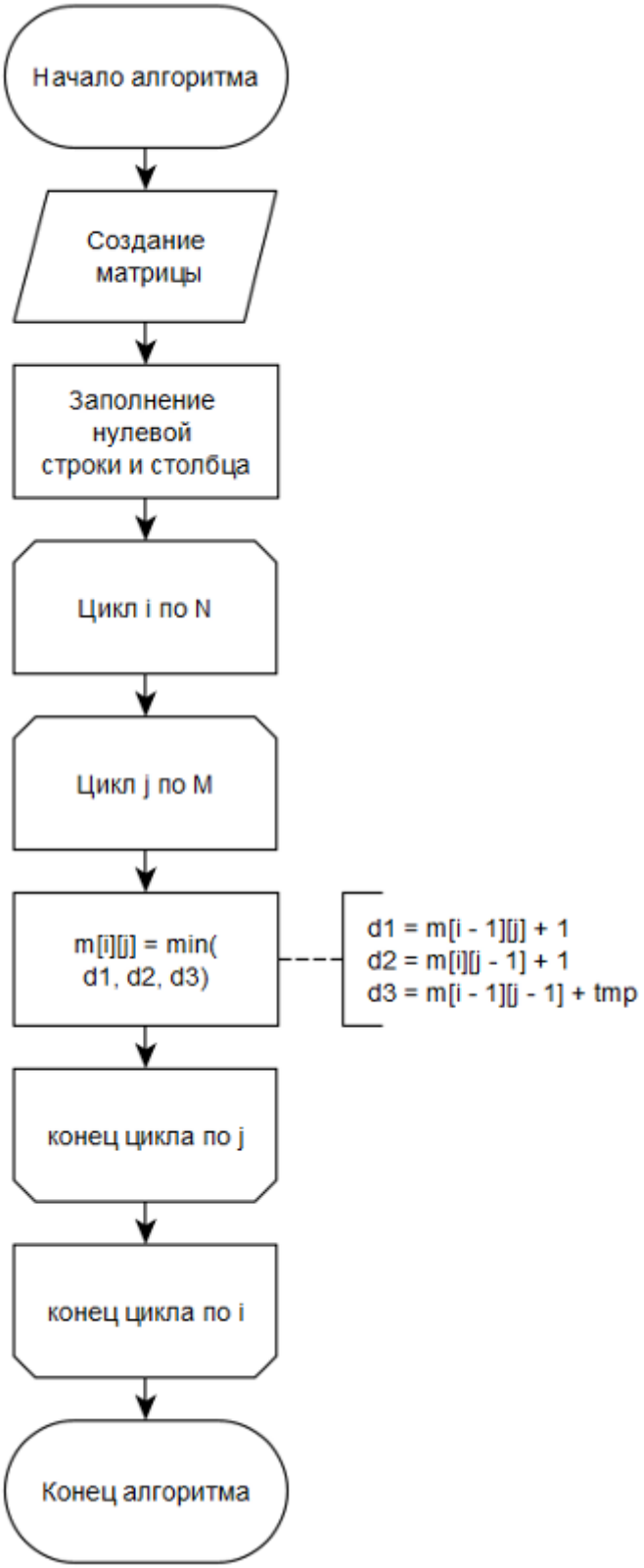

Рис. 3. Схема матричного алгоритма нахождения расстояния Левенштейна

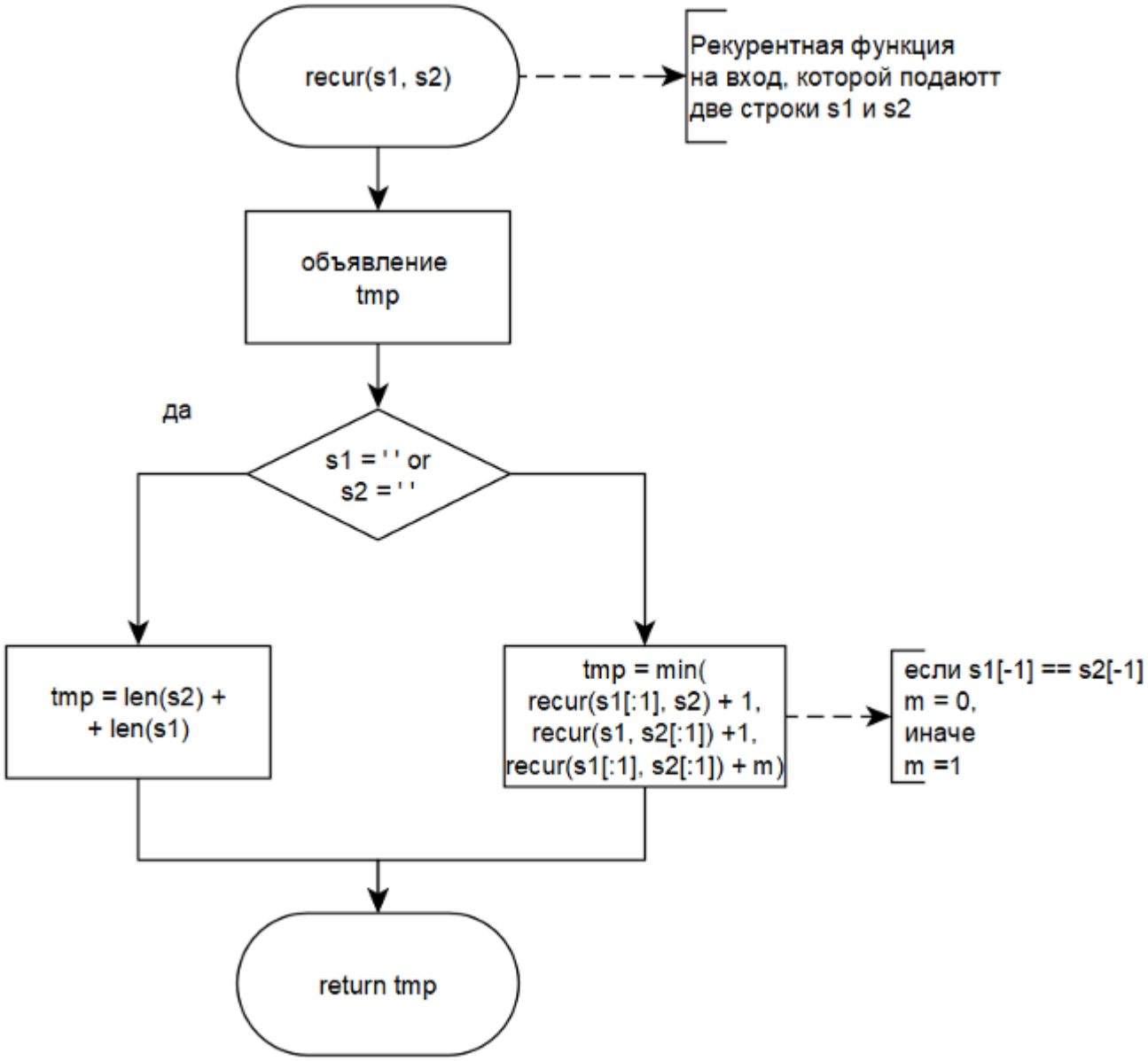

Рис. 4. Схема рекурсивного алгоритма нахождения расстояния Левенштейна

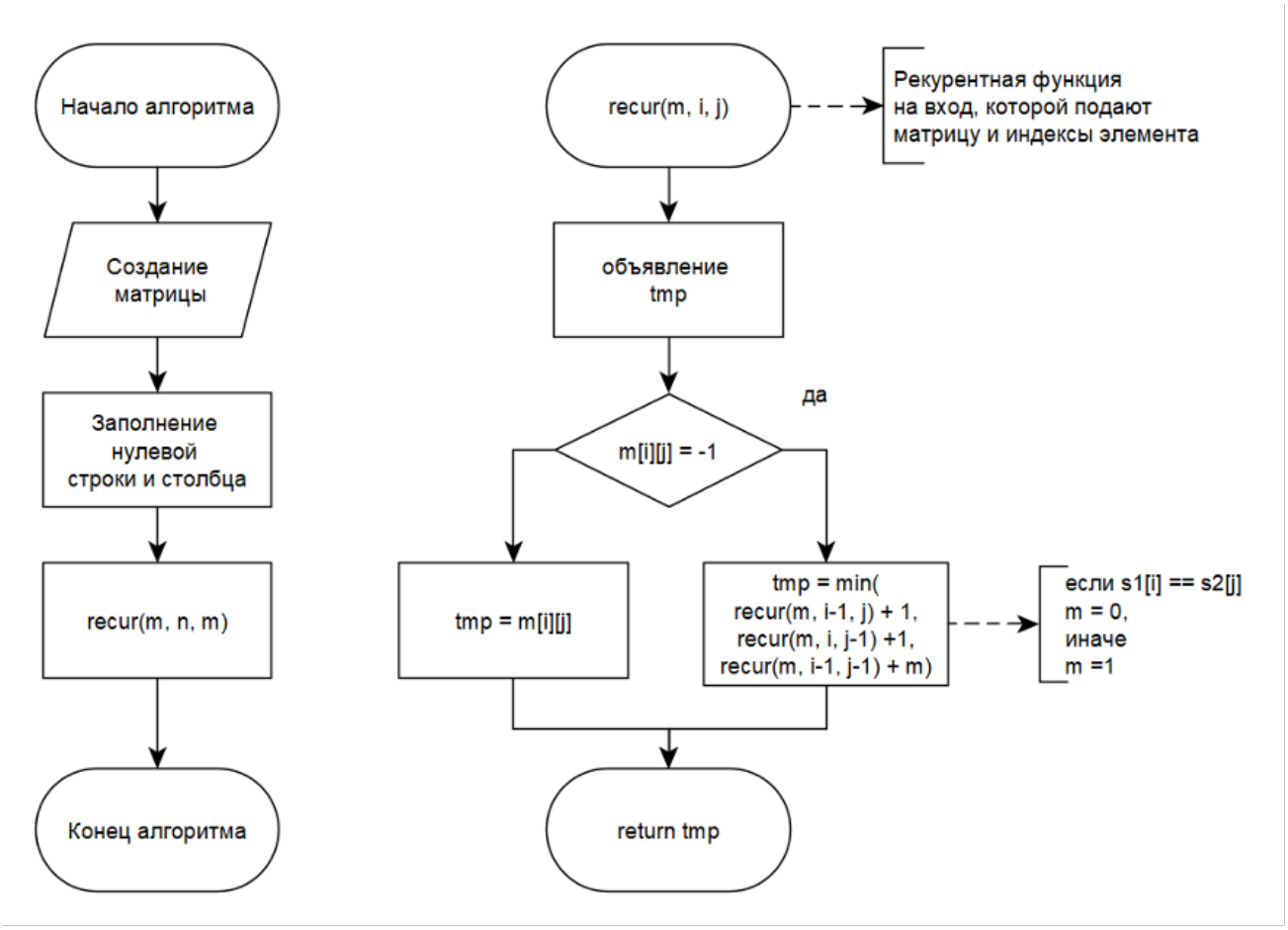

Рис. 5. Схема матрично-рекурсивного алгоритма нахождения расстояния Левенштейна

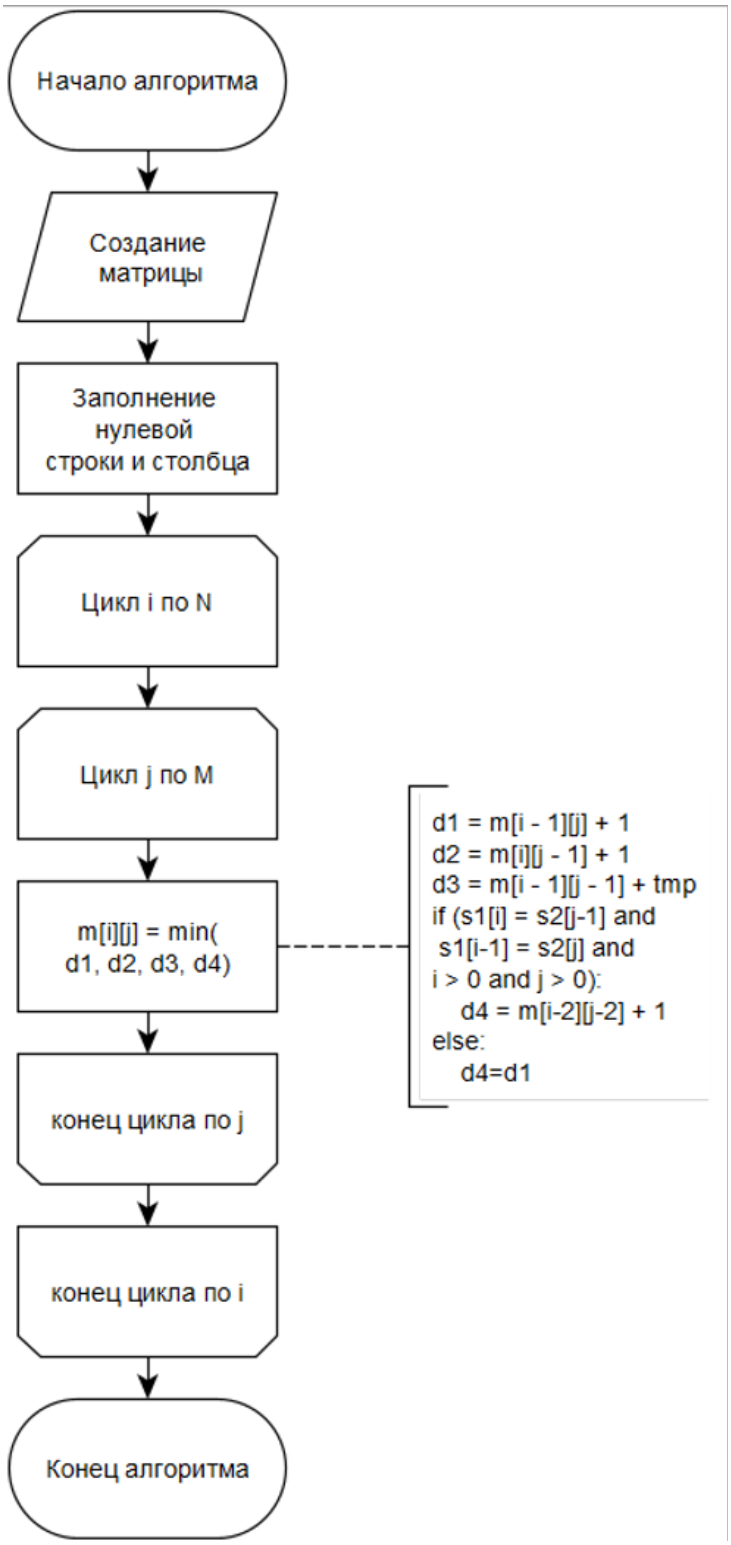

Рис. 6. Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

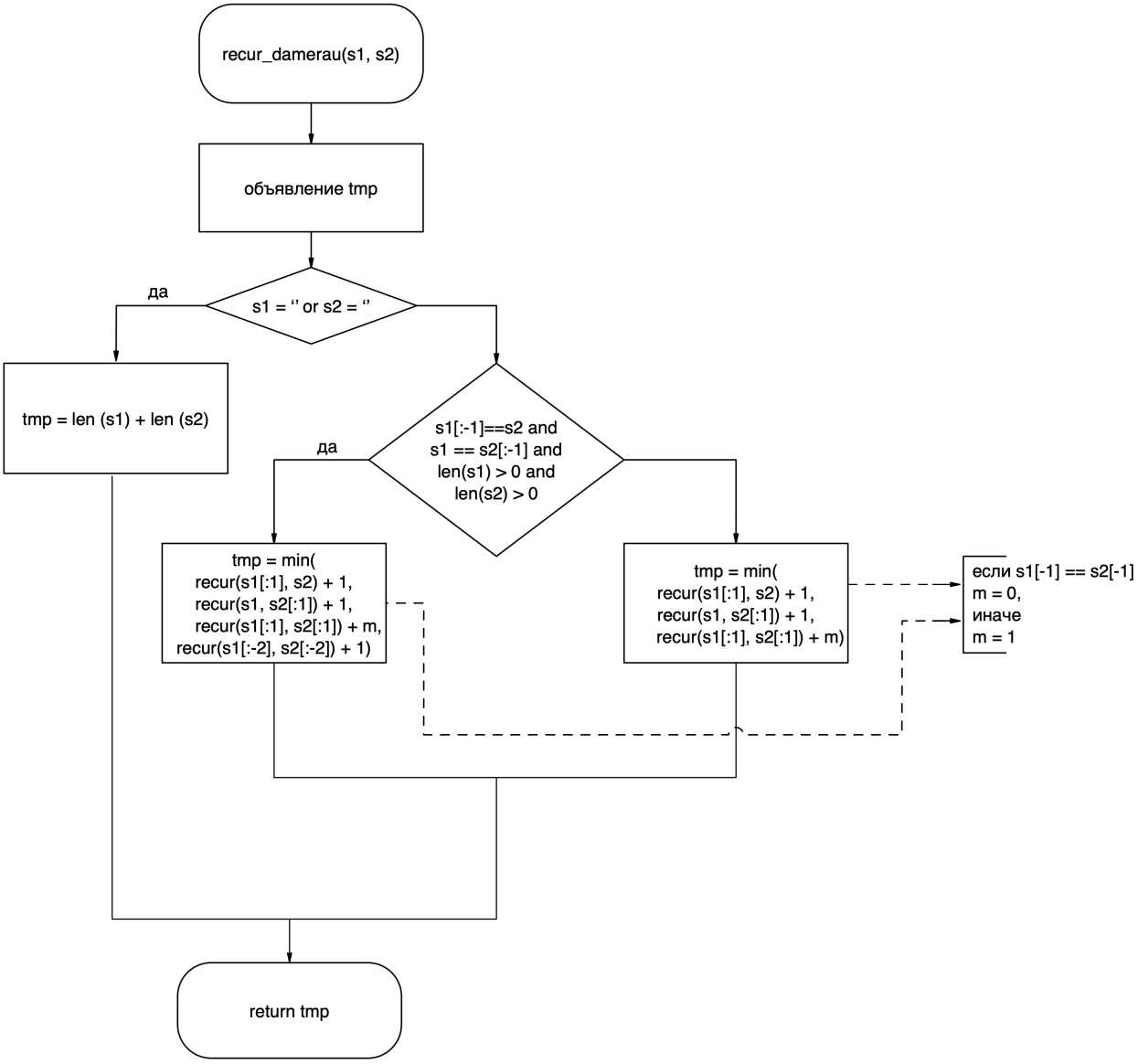

Рис. 7. Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования был выбран python, т.к. данный язык программирования позволяет написать программу за короткое время. Для замера процессорного времени была использована функция process_time(), стандартной библиотеки python – time [2].

3.2. Реализация алгоритмов

В листингах 1-5 представлена реализация алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

В листинге 6 представлена функция для замера времени выполнения заданной функции на заданном количестве итераций на строках указанной длины.

Листинг 1. Реализация алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

```
def calc_dist_matrix(s1, s2):
   matr = np.eye(len(s1) + 1, len(s2) + 1)

   for i in range(len(s1) + 1):
      matr[i][0] = i
   for j in range(len(s2) + 1):
      matr[0][j] = j

   for i in range(len(s1)):
      for j in range(len(s2)):
         d1 = matr[i + 1][j] + 1
         d2 = matr[i][j + 1] + 1
         if s1[i] == s2[j]:
            d3 = matr[i][j]
         else:
            d3 = matr[i][j] + 1
         matr[i + 1][j + 1] = min(d1, d2, d3)
```

Листинг 2. Реализация рекурсивного алгоритма нахождения расстояния Левенштейна.

```
def calc_dist_recur(s1, s2, printable=False):
   if s1 == '' or s2 == '':
      return abs(len(s1) - len(s2))

   tmp = 0 if (s1[-1] == s2[-1]) else 1
   return min(calc_dist_recur(s1[:-1], s2) + 1,
              calc_dist_recur(s1, s2[:-1]) + 1,
              calc_dist_recur(s1[:-1], s2[:-1]) + tmp)
```

Листинг 3. Реализация рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

```
def calc_dist_recur_matrix(s1, s2, printable=False):
   def calc_value(matr, i, j):
      if matr[i][j] != -1:
         return matr[i][j]
      else:
         tmp = 0 if (s1[i - 1] == s2[j - 1]) else 1
         matr[i][j] = min(calc_value(matr, i - 1, j) + 1,
                          calc_value(matr, i, j - 1) + 1,
                          calc_value(matr, i - 1, j - 1) + tmp)
         return matr[i][j]

   matr = np.full((len(s1) + 1, len(s2) + 1), -1)
   for i in range(len(s1) + 1):
      matr[i][0] = i
   for j in range(len(s2) + 1):
      matr[0][j] = j
   value = calc_value(matr, len(s1), len(s2))
```

Листинг 4. Реализация алгоритма нахождения расстояния Дамерау-Левенштейна с заполнением матрицы.

```
def calc_dist_damerau(s1, s2):
   matr = np.eye(len(s1) + 1, len(s2) + 1)

   for i in range(len(s1) + 1):
      matr[i][0] = i
   for j in range(len(s2) + 1):
      matr[0][j] = j

   for i in range(len(s1)):
      for j in range(len(s2)):
         d1 = matr[i + 1][j] + 1
         d2 = matr[i][j + 1] + 1
         if s1[i] == s2[j]:
            d3 = matr[i][j]
         else:
            d3 = matr[i][j] + 1
         if s1[i] == s2[j - 1] and s1[i - 1] == s2[j] and i > 0 and j > 0:
            d4 = matr[i - 1][j - 1] + 1
         else:
            d4 = d1
         matr[i + 1][j + 1] = min(d1, d2, d3, d4)
```

Листинг 5. Реализация рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

```
def calc_dist_damerau_recur(s1, s2, printable=False):

   if s1 == '' or s2 == '':
      return abs(len(s1) - len(s2))

   tmp = 0 if (s1[-1] == s2[-1]) else 1
   if s1[:-1] == s2 and s1 == s2[:-1] and len(s1) > 0 and len(s2) > 0:
      return min(calc_dist_damerau_recur(s1[:-1], s2) + 1,
                 calc_dist_damerau_recur(s1, s2[:-1]) + 1,
                 calc_dist_damerau_recur(s1[:-1], s2[:-1]) + tmp,
                 calc_dist_damerau_recur(s1[:-2], s2[:-2]) + 1)
   else:
      return min(calc_dist_damerau_recur(s1[:-1], s2) + 1,
                 calc_dist_damerau_recur(s1, s2[:-1]) + 1,
                 calc_dist_damerau_recur(s1[:-1], s2[:-1]) + tmp)
```

Листинг 6. Функция подсчета среднего времени выполнения программы для строк длиной length.

```
def time_analyze(function, iterations, length=5):
   t1 = process_time()
   for _ in range(iterations):
      s1 = random_string(length)
      s2 = random_string(length)
      function(s1, s2, False)

   t2 = process_time()
   return (t2 - t1) / iterations
```

3.3. Тестирование функций

Для модульного тестирования реализованных алгоритмах (см. листинги 1-5) была использована стандартная библиотека языка python – unittest. Модульные тесты приведены в листингах 7-8. Все функции протестированы на пустые входящие строки, а также на различные другие входные данные.

Для алгоритма поиска расстояния Дамерау-Левенштейна существует дополнительные отдельные тесты на транспозицию.

Листинг 7. Проверка на пустоту строк

```
def test_empty(self):
   self.assertEqual(self.function('', ''), 0)
   self.assertEqual(self.function('a', ''), 1)
   self.assertEqual(self.function('', 'a'), 1)
```

Листинг 8. Проверка на выполнение операций

```
def test_different(self):
    # Match
    self.assertEqual(self.function('a', 'a'), 0)
    self.assertEqual(self.function('c', 'c'), 0)
    # Delete
    self.assertEqual(self.function('ab', 'a'), 1)
    self.assertEqual(self.function('op', 'o'), 1)
    # Insert
    self.assertEqual(self.function('a', 'ab'), 1)
    self.assertEqual(self.function('o', 'op'), 1)
    # Replace
    self.assertEqual(self.function('ab', 'a'), 1)
    self.assertEqual(self.function('op', 'od'), 1)
```

Все тесты пройдены успешно.

4. Экспериментальная часть

При запуске программы первое, что видит пользователь, – это интуитивно понятный интерфейс.

4.1. Интерфейс

На рис. 8 представлено главное меню программы. В зависимости от выбранного пункта меню запускается соответствующий алгоритм

```
Меню:
 1) Матричное нахождение расстояния Левенштейна
 2) Рекурсивное нахождение расстояния Левенштейна
 3) Рекурсивное нахождение расстояния Левенштейна с заполнением матрицы
 4) Матричное нахождение расстояния Дамерау-Левенштейна
 5) Рекурсивное нахождение расстояния Дамерау-Левенштейна
 5) Анализ времени
 7) Анализ времени 1 из методов
```

Рис. 8. Главное меню программы

На рис. 9-12 приведены примеры работы программы при вводе строк *«увлечение»* и *«развлечения»* при выборе пунктов меню 1-5.

```
  Введите первое слово: увлечение
  Введите второе слово: развлечения
Матрица:
    р а з в л е ч е н и я
  0 1 2 3 4 5 6 7 8 9 10 11
у 1 1 2 3 4 5 6 7 8 9 10 11
в 2 2 2 3 3 4 5 6 7 8 9 10
л 3 3 3 3 4 3 4 5 6 7 8 9
е 4 4 4 4 4 4 3 4 5 6 7 8
ч 5 5 5 5 5 5 4 3 4 5 6 7
е 6 6 6 6 6 6 5 4 3 4 5 6
н 7 7 7 7 7 7 6 5 4 3 4 5
и 8 8 8 8 8 8 7 6 5 4 3 4
е 9 9 9 9 9 9 8 7 6 5 4 4
Операция:
I I R M M M M M M M R
Возврат:  4.0
```

Рис. 9. Пример работы программы при выборе пункта 1

```
  Введите первое слово: увлечение
  Введите второе слово: развлечения
увлечение развлечения

Возврат:  4
```

Рис. 10. Пример работы программы при выборе пункта 2

```
  Введите первое слово: увлечение
  Введите второе слово: развлечения
Матрица

    р а з в л е ч е н и я
  0 1 2 3 4 5 6 7 8 9 10 11
у 1 1 2 3 4 5 6 7 8 9 10 11
в 2 2 2 3 3 4 5 6 7 8 9 10
л 3 3 3 3 4 3 4 5 6 7 8 9
е 4 4 4 4 4 4 3 4 5 6 7 8
ч 5 5 5 5 5 5 4 3 4 5 6 7
е 6 6 6 6 6 6 5 4 3 4 5 6
н 7 7 7 7 7 7 6 5 4 3 4 5
и 8 8 8 8 8 8 7 6 5 4 3 4
е 9 9 9 9 9 9 8 7 6 5 4 4
Операция:
I I R M M M M M M M R
Возврат:  4
```

Рис. 11. Пример работы программы при выборе пункта 3

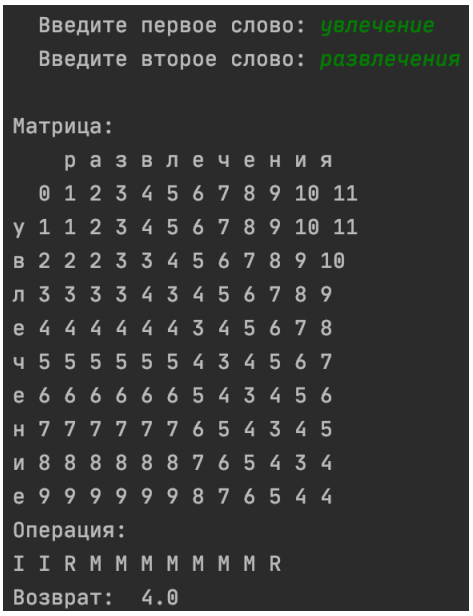

Рис. 12. Пример работы программы при выборе пункта 4

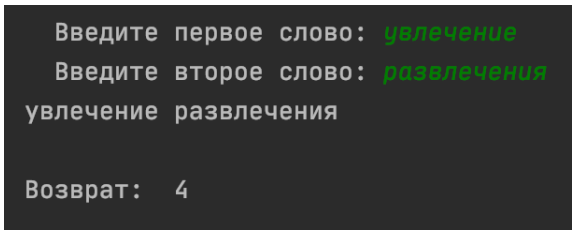

Рис. 13. Пример работы программы при выборе пункта 5

4.2. Сравнение алгоритмов по времени работы реализаций

Для сравнения в программе необходимо провести замеры процессорного времени для выполнения n – количества вычислений для строк одинаковой длины. Результаты замера времени предоставлены в таблице 1.

Таблица 1. Сравнение алгоритмов по времени

Алгоритм	Длина строк	Среднее время, сек
Левенштейн матричный	5	0.00008750
Левенштейн матричный	10	0.00029531
Левенштейн матричный	15	0.00063480
Левенштейн рекурсивный	5	0.00123438
Левенштейн рекурсивный	7	0.03080775
Левенштейн рекурсивный	10	5.73437500
Левенштейн рекурсивный с заполнением матрицы	5	0.00012969
Левенштейн рекурсивный с заполнением матрицы	10	0.00046875
Левенштейн рекурсивный с заполнением матрицы	15	0.00098420
Дамерау-Левенштейн матричный	5	0.00008906
Дамерау-Левенштейн матричный	10	0.00032188
Дамерау-Левенштейн матричный	15	0.00067355
Дамерау-Левенштейн рекурсивный	5	0.00159060
Дамерау-Левенштейн рекурсивный	7	0.03289530
Дамерау-Левенштейн рекурсивный	10	5.47348705

Замеры времени усреднялись для каждого набора одинаковых экспериментов. Для этого все вычисления производились на случайных строках при 1000-10000 итераций, кроме рекурсивных, для которых количество итераций было в пределах 20-50, это связано с временем выполнения данных алгоритмов.

Вывод: у матричных алгоритмов время пропорционально квадрату длины строк (при увеличении строки в два раза, время увеличивается в четыре раза). Рекурсивный алгоритм показывает наихудшее время, это связано с повторным вычислением одних и тех же значений (каждый вызов порождает три новых вызова, у которых могут дублироваться входные значения). Матричные алгоритмы по поиску расстояния Левенштейна и Дамерау-Левенштейна показывают самое лучшее время. Первый алгоритм немного быстрее, однако необходимо учитывать то, что второй решает другую задачу.

4.3. Сравнение алгоритмов по затраченной памяти

Пусть длина строки S1 - n, длина строки S2 - m, тогда затраты памяти на приведенные выше алгоритмы будут следующими.

Затраты памяти **матричного алгоритма нахождения расстояния Левенштейна**:

1) строки S1, S2 - (m + n) * sizeof(char);

2) матрица - ((m + 1) * (n + 1)) * sizeof(int);

3) длины строк - 2 * sizeof(int);

4) вспомогательная переменная - sizeof(int).

Затраты памяти **рекурсивного алгоритма нахождения расстояния Левенштейна** (для каждого вызова):

1) строки S1, S2 - (m + n) * sizeof(char);

2) длины строк - 2 * sizeof(int);

3) вспомогательные переменные - 2 * sizeof(int);

4) адрес возврата.

Затраты памяти рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы:

1) матрица - ((m + 1) * (n + 1)) * sizeof(int);

2) строки S1, S2 - (m + n) * sizeof(char).

Для каждого рекурсивного вызова:

1) передача данных - 2 * sizeof(int*) + sizeof(int**);

2) вспомогательная переменная - sizeof(int).

3) адрес возврата

Чтобы получить итоговую оценку затрачиваемой памяти на рекурсивные вызовы необходимо затрачиваемую память для одного рекурсивного вызова умножить на максимальную глубину рекурсии, которая равна сложению длин входных строк.

Затраты памяти **матричного алгоритма нахождения расстояния Дамерау-Левенштейна**:

1) строки S1, S2 - (m + n) * sizeof(char);

2) матрица - ((m + 1) * (n + 1)) * sizeof(int);

3) длины строк - 2 * sizeof(int);

4) вспомогательная переменная - sizeof(int).

Затраты памяти **рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна**:

1) строки S1, S2 - (m + n) * sizeof(char);

2) длины строк - 2 * sizeof(int);

3) вспомогательные переменные - 2 * sizeof(int);

4) адрес возврата.

Для каждого рекурсивного вызова:

1) передача данных - 2 * sizeof(int*) + sizeof(int**);

2) вспомогательная переменная - sizeof(int).

3) адрес возврата

Вывод: при большой длине строк, матричные методы занимают огромное количество памяти, в отличие от рекурсивного алгоритма. Сравнивая алгоритмы по затрачиваемой памяти, можно прийти к тому, что память для рекурсивного алгоритма зависит от n (длины строк) как $4 * n2 + 164 * n$, в то время, как память, необходимая для матричного алгоритма, – $(8 * n^2 + 2 * n + 180)$, это говорит нам о том, что до n равного 31 включительно по памяти выигрывает матричный алгоритм, а начиная с длины строки 32 и более, рекурсивный алгоритм тратит меньше памяти, чем матричный.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна. Реализованы 4 алгоритма поиска этих расстояний, приведен программный код реализации алгоритмов нахождения расстояний.

Было выполнено сравнение разных алгоритмов нахождения расстояния Левенштейна по затраченным ресурсам. Было установлено, что рекурсивный алгоритм занимает гораздо меньше памяти при работе со строками большой длины, чем матричные алгоритмы. Однако матричные алгоритмы отмечаются своим быстродействием.

Цель работы достигнута. Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна применены на практике, получены практические навыки реализации этих алгоритмов.

Список литературы

[1] Вычисление расстояния Левенштеина // [Электронныи ресурс]. Режим доступа: https://foxford.ru/wiki/informatika/vychislenie-rasstoyaniyalevenshteyna, свободный (дата обраще- ния: 06.09.21).

[2] Официальный сайт Python, документация // [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html, свободный (дата обращения: 06.09.21).