*Федеральное государственное бюджетное образовательное учреждениевысшего профессионального образования*

|  |  |
| --- | --- |
| Gerb-BMSTU_01 | ***«Московский государственный технический университет  имени Н.Э. Баумана (национальный исследовательский институт)»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

**Отчёт**

**по лабораторной работе 1**

**Дисциплина: Анализ Алгоритмов**

**Тема лабораторной работы работы: Расстояния Левенштейна и Дамерау-Левенштейна**

Студентки гр. ИУ7-51б **\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сушина А.Д.**

(Подпись, дата) (И.О. Фамилия)

Преподаватель  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Волкова Л.Л.**

(Подпись, дата) (И.О. Фамилия)

Москва, 2019г

Оглавление

[Введение 3](#_Toc22478944)

[1. Аналитическая часть 4](#_Toc22478945)

[1.1. Описание алгоритмов 4](#_Toc22478946)

[2. Конструкторская часть 5](#_Toc22478947)

[2.1.Разработка алгоритмов 5](#_Toc22478948)

[2.2.Сравнительный анализ рекурсивной и нерекурсивной реализаций 9](#_Toc22478949)

[3.Технологическая часть 10](#_Toc22478950)

[3.1.Требования к программному обеспечению 10](#_Toc22478951)

[3.2.Средства реализации 10](#_Toc22478952)

[3.3.Листинг кода 10](#_Toc22478953)

[3.4.Описание тестирования 12](#_Toc22478954)

[4.Экспериментальная часть 13](#_Toc22478955)

[4.1.Примеры работы 13](#_Toc22478956)

[4.2.Результаты тестирования 14](#_Toc22478957)

[4.3.Постановка эксперимента по замеру времени 15](#_Toc22478958)

[Заключение 16](#_Toc22478959)

# Введение

**Расстояние Левенштейна** (редакционное расстояние, дистанция редактирования) - 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.

**Расстояние Дамерау-Левенштейна**— это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

**Цель работы:** изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

**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 временнóй 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

## 1.1. Описание расстояний

**Расстояния Левенштейна**

Расстояние Левенштейна или редакторское расстояние считается как минимальное количество редакторских операций вставки, удаления и замены, необходимых для преобразования строки s1 в строку s2.

Редакторские операции:

* INSERT — вставкам (штраф 1)
* DELETE — удаление (штраф 1)
* REPLACE — замена (штраф 1)
* MATCH — сопадение (штраф 0)

Считается, что элементы строк нумеруются с первого, как принято в математике, а не с нулевого, как принято во многих языках программирования.

Пусть s1 и s2— две строки (длиной M и N соответственно) над некоторым афвалитом, тогда редакционное расстояние (расстояние Левенштейна) d(s1, s2) можно подсчитать по следующей рекуррентной формуле:

где операция min возвращает наименьший из аргументов.

**Расстояние Дамерау-Левенштейна**

Вводится дополнительна операция: перестановка или транспозиция двух букв со штрафом 1.

Если индексы позволяют и если две соседние буквы , то в минимум включается перестановка.

Пусть s1 и s2— две строки (длиной M и N соответственно) над некоторым афвалитом, тогда редакционное расстояние (расстояние Дамерау-Левенштейна) d(s1, s2) можно подсчитать по следующей рекуррентной формуле:

Расстояние Левенштейна и его обобщения активно применяется:

* 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. Здесь роль «символов» играют строки, а роль «строк»— файлы.
* в биоинформатике для сравнения генов, хромосом и белков.

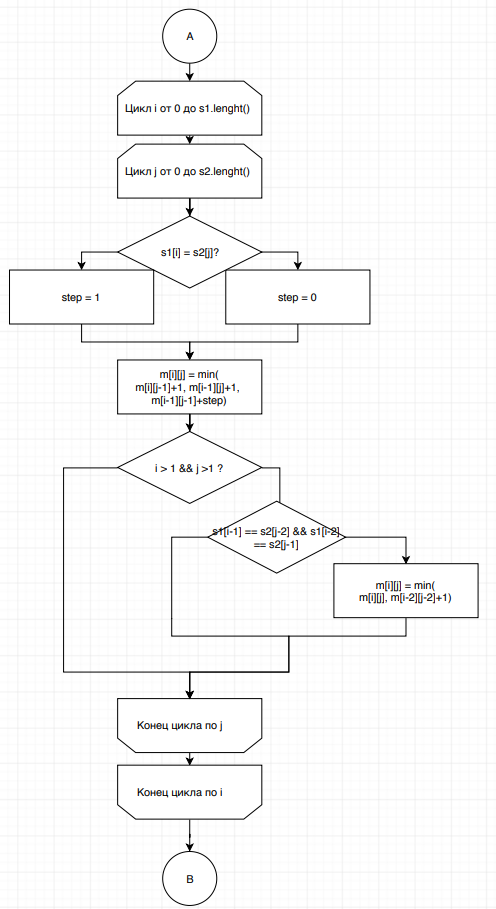
# 2. Конструкторская часть

## 2.1.Разработка алгоритмов

В разделе представлены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна на основании матрчного расчета, а также рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна (рис. 1-3).

|  |  |
| --- | --- |
|  |  |

Рис 1. Схема алгоритма Левенштейна



|  |  |
| --- | --- |
|  |  |

Рис. 2 Схема алгоритма Дамерау-Левенштейна(часть 1)

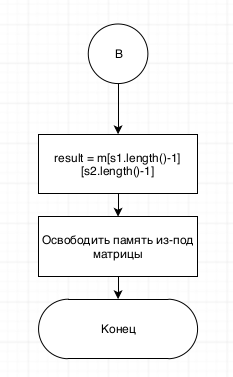
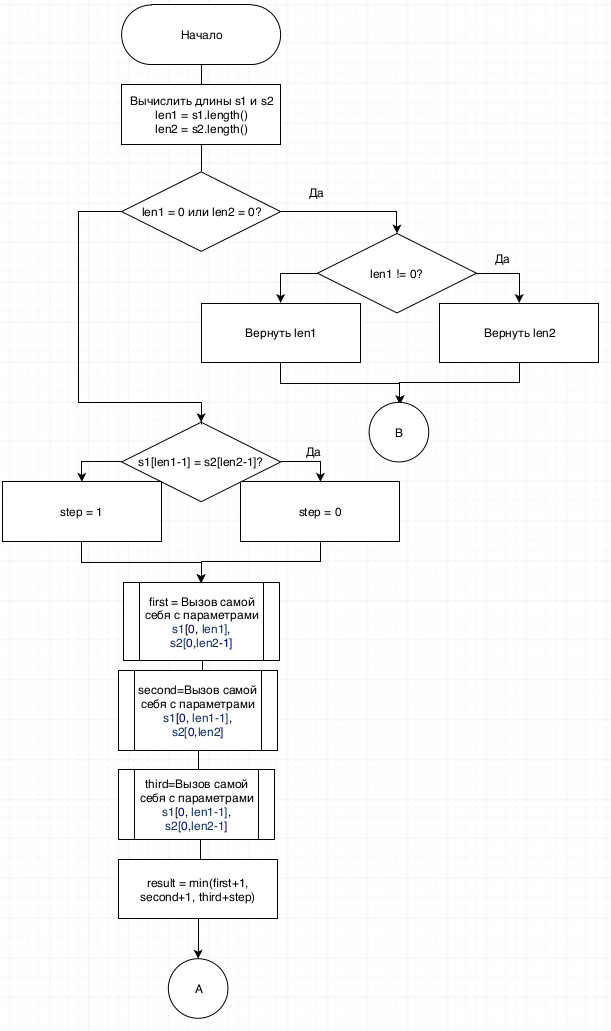


Рис 3. Схема алгоритма Дамерау-Левенштейна(часть 2)

Рис 3. Рекурсивный алгоритм Дамерау-Левенштейна(часть 1)

## 

Рис. 4 Алгоритм Дамерау-Левенштейна (рекурсивно)

## 2.2.Сравнительный анализ рекурсивной и нерекурсивной реализаций

* 1. Рекурсивная реализация по сравнению с матричной работает медленнее. При постоянном рекурсивном вызове исчерпываются возможные комбинации и некоторые вызовы функции абсолютно идентичны. Это не рациональная трата ресурсов.
  2. К тому же, на рекурсивный вызов функции тратится большое количество памяти в стеке, что, при большой глубине рекурсии, может привести к ошибкам.
  3. Затраты по памяти алгоритмов различаются.
* Матричная реализация алгоритма Левенштейна:

*(n+1)\*(m+1)\*sizeeof(int) (матрица) + 6\*sizeof(int)(локальные переменные) + 2 вызова std::min*

* Матричная реализация алгоритма Дамерау-Левенштейна:

*(n+1)\*(m+1)\*sizeeof(int) (матрица) + 6\*sizeof(int)(локальные переменные) + 3 вызова std::min*

* Рекурсивная реалзация алгоритма Дамерау-Левенштейна:

На 1 вызов приходится

*8\*sizeof(int)(локальные переменные) + 3 вызова std::min + 4 вызова самой себя*

Пиковое значение выделенной памяти оценивается как количество памяти на один вызов, умноженное на высоту дерева — сумму длин двух строк.

При больших длинах строк по памяти эффективнее рекурсивная реализация.

# 3.Технологическая часть

## 3.1.Требования к программному обеспечению

* 1. На вход в программу поступает 2 строки некоторой длины. На выходе необходимо получить три числа, являющиеся результатами работы трех вышеупомянутых алгоритмов. Для матричных реализаций требуется вывести матрицу решений. Также требуется замерить время работы каждого из алгоритмов.

## 3.2.Средства реализации

* 1. Для реализации был выбран язык C++. Этот язык позволяет решить задачу с минимальными затратами по памяти. Этот язык работает быстрее аналогов, он удобен, а так же знаком мне. Среда разработки — Qt creator.   
     Для реализации я выбрала паттерн Интерфейс, так как он позволяет реализовать единый интерфейс для всех алгоритмов.

## 3.3.Листинг кода

* 1. Матричная реализация алгоритма нахождения расстояния Левенштейна:

|  |
| --- |
| class Livinshtein {  public:  Livinshtein(){}  int **count**(std::string s1, std::string s2, int print\_flag = 0) {  allocate\_matrix(matrix, s1.length()+1,s2.length()+1);  matrix[0][0] = 0;  for (size\_t i = 0; i < s1.length()+1; i++) {  matrix[i][0] = i;  }  for (size\_t j = 0; j < s2.length()+1; j++) {  matrix[0][j] = j;  }  for (size\_t i = 1; i < s1.length()+1; i++) {  for (size\_t j = 1; j < s2.length()+1; j++) {  int step = s1[i-1] == s2[j-1] ? 0 : 1;  matrix[i][j] = std::min(std::min(matrix[i-1][j]+1,  matrix[i][j-1]+1),  matrix[i-1][j-1]+step);  }  }  if (print\_flag)  {  print\_matrix(s1, s2);  }  int answ = matrix[ s1.length()][s2.length()];  free\_matrix(matrix, s1.length()+1);  return answ;  }  private:  int \*\*matrix;  }; |

Матричная реализация алгоритма нахождения расстояния Дамерау-Левенштейна:

|  |
| --- |
| class DamerauLiv {  public:  DamerauLiv() {}  int **count**(std::string s1, std::string s2, int print\_flag = 0) {  allocate\_matrix(matrix, s1.length()+1,s2.length()+1);  matrix[0][0] = 0;  for (size\_t i = 0; i < s1.length()+1; i++) {  matrix[i][0] = i;  }  for (size\_t j = 0; j < s2.length()+1; j++) {  matrix[0][j] = j;  }  for (size\_t i = 1; i < s1.length()+1; i++) {  for (size\_t j = 1; j < s2.length()+1; j++) {  int step = s1[i-1] == s2[j-1] ? 0 : 1;  matrix[i][j] = std::min(std::min(matrix[i-1][j]+1,  matrix[i][j-1]+1),  matrix[i-1][j-1]+step);  if (i > 1 && j > 1){  if (s1[i-1] == s2[j-2] && s1[i-2] == s2[j-1]) {  matrix[i][j] = std::min(matrix[i][j],  matrix[i-2][j-2]+1);  }  }  }  }  if (print\_flag)  {  print\_matrix(s1,s2);  }  int answ = matrix[s1.length()][s2.length()];  free\_matrix(matrix, s1.length()+1);  return answ;  }  private:  int \*\*matrix;  }; |

Рекурсивная реализация алгоритма нахождения расстояния Дамерау-Левенштейна:

|  |
| --- |
| class DamerauLivRec {  public:  DamerauLivRec() {}  int **count**(std::string s1, std::string s2) {  size\_t len1 = s1.length();  size\_t len2 = s2.length();  if (len1 == 0 || len2 == 0){  return len1? len1: len2;  }  size\_t step = 1;  if (s1[len1-1] == s2[len2-1]) {  step = 0;  }  int first = this->count(s1.substr(0, len1), s2.substr(0,len2-1)) + 1;  int second = this->count(s1.substr(0, len1-1), s2.substr(0,len2)) + 1;  int third = this->count(s1.substr(0, len1-1), s2.substr(0,len2-1)) + step;  int returnal = std::min(std::min(first, second), third);  if (len1 > 1 && len2 > 1){  if (s1[len1-1] == s2[len2-2] && s1[len1-2] == s2[len2-1]) {  int fourth = this->count(s1.substr(0, len1-2), s2.substr(0, len2-2)) + 1;  return std::min(returnal, fourth);  }  }  return returnal;  }  }; |

## 3.4.Описание тестирования

Тестирование будет реализовано отдельной функцией в программе, которую можно запустить по желанию пользователя.

Для каждого алгоритма реализована своя функция тестирования, так как алгоритмы Левенштейна и Дамерау-ЛеТвенштейна дают разные результаты в некоторых случаях, так как это разные алгоритмы и они ищут разные расстояния.

Тестируем по следующим данным:

1. Проверка работы: "skat" , "kot"

2. Проверка работы с пустыми строками: "skat", ""; "", "kot"

3. Проверка работы с идентичными строками: "kot", "kot"; "",""

4. Проверка работы с перестановкой: "skat", "ksat"; "sksksk", "ksksks"

5. Проверка работы с полностью несовпадающими строками: "aaaaaa", "kot"

6. Проверка работы с пробелами: «kt kt kt», «tk tk tk»

# 4.Экспериментальная часть

## 4.1.Примеры работы

Приведем примеры работы программы.

Пример 1 (см. рис. 6):

Запустим программу на строках skat и kot. Ожидаемый результат: 2,2,2(вставка+замена)  
Так же выводятся матрицы, построенные в процессе вычислений

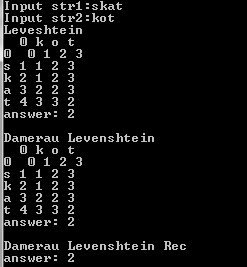


Рис. 6 - Пример 1 работы программы

Пример 2(см. рис 7):

Запустим программу на строках skat и ksаt. Ожидаемый результат: 2,1,1  
В этот раз ответы отличаются, так как перестановка дает меньший результат.



Рис. 7 — Пример 2 работы программы

## 4.2.Результаты тестирования

Таблица 1.

Тесты и ожидаемый результат тестирования алгоритмов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № Теста | Первое слово | Второе слово | Ожидаемый результат | | |
| Левенштейна | Дамерау-Левенштейа | Дамерау-Левенштейа рекурсивно |
| 1 | skat | kot | 2 | 2 | 2 |
| 2 | skat |  | 4 | 4 | 4 |
| 3 |  | kot | 3 | 3 | 3 |
| 4 | kot | kot | 0 | 0 | 0 |
| 5 |  |  | 0 | 0 | 0 |
| 6 | skot | ksot | 2 | 1 | 1 |
| 7 | sksksk | ksksks | 2 | 2 | 2 |
| 8 | aaaaaa | kot | 6 | 6 | 6 |
| 9 | kt kt kt | tk tk tk | 6 | 3 | 3 |

## 

Тесты и ожидаемые результаты тестирования представлены в таблице.

* 1. Рис 8 — Результаты выполнения тестов
  2. Все тесты прошли успешно.

## 4.3.Постановка эксперимента по замеру времени

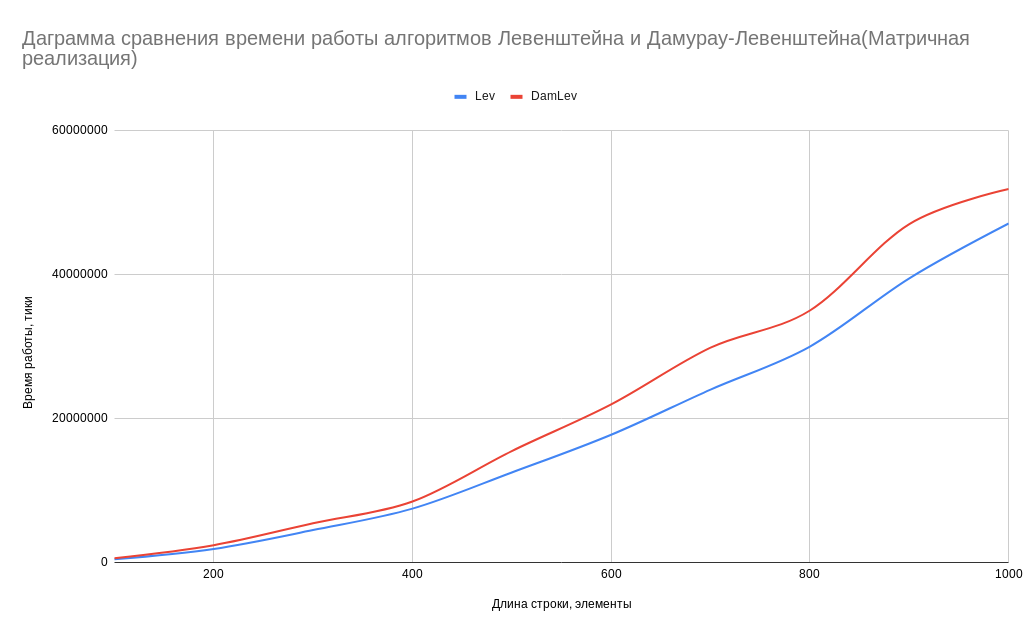
Для произведения замеров времени выполнения реализаций алгоритмов будет использована формула , где N – количество замеров, t – время выполнения реализации алгоритма, Tn — время выполнения N замеров. Неоднократное измерение времени необходимо для построения более гладкого графика и получения усредненного значения времени.

Количество замеров будет взято равным 100.

Тестирование будет проведено на рандомных строках одинаковой длины: для сравнения матричных реализации длины строк принимают значения 0-1000 с шагом 100, для сравнения матричной и рекурсивной реализации - 0-10 с шагом 1.

Результаты замеров представлены на рис. 9-10.

|  |
| --- |
| Рис. 9 График зависимости времени работы алгоритмов от времени |

Рис. 10 График зависимости ремени работы матричных алгоритмов от времени на рандомных строках одинаковое

В результате проведенного эксперимента было выяснено следующее:

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.

Алгоритм Дамерау-Левештейна работает меленнее алгоритма Левенштейна, но незначительно. Это объясняется наличием дополнительного сравнения в алгоритме Дамерау-Левенштейна

# Заключение

В ходе работы были изучены алгоритмы Левенштейна и Дамерау-Левенштейна. Выполнено тестирование и 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(в матричной и рекурсивной формах).