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»

ОТЧЕТ

по Лабораторной работе №1

по курсу «Анализы Алгоритмов»

на тему: «Динамическое программирование»

Студент группы ИУ7-54Б _____ (Подпись, дата) Разин А. В. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

ВВЕДЕНИЕ

При работе со словами, необходимо их сравнивать, причем необходима конкретная метрика, которая покажет, насколько посимвольно одно слово отличается от другого. Одной из таких метрик является Расстояние Левенштейна. Данное расстояние является метрикой, измеряющей по модулю разность между двумя последовательностями символов. [**levenshtein**]

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0 – 1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна активно используется и по сей день:

— исправление ошибок в слове(в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

— сравнение текстовых файлов утилитой `diff`;

— для сравнения геномов, хромосом и белков в биоинформатике.

Данная метрика была модифицирована Фредриком Дамерау, путем введения операции перестановки соседних символов. [**Damerau**]

1 Аналитическая часть

Целью данной лабораторной работы является описание и исследование алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна.

Для поставленной цели необходимо выполнить следующие задачи.

1) Исследовать расстояние Левенштейна.

2) Разработка алгоритмов поиска расстояний Левенштейна, Дамерау—Левенштейна.

3) Создать программное обеспечение, реализующее следующие алгоритмы.

4) Провести исследование, затрачиваемого процессорного времени и памяти при различных реализациях алгоритмов.

5) Провести сравнительный анализ алгоритмов.

1.1 Расстояние Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. [**levenshtein**]

Вводятся операции нескольких типов:

1) I (англ. insert) — операция вставки символа.

2) D (англ. delete) — операция удаления символа.

3) R (англ.replace) — операция замены символа.

Также введем символ λ, обозначающий пустой символ строки, не входящий ни в одну из рассматриваемых строк.

Будем считать стоимость каждой вышеизложенной операции равной 1:

— $D(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $D(\lambda, b) = 1$;

— $D(a, \lambda) = 1$.

Также обозначим совпадение символов как M (англ. match), таким образом $D(a, a) = 0$. Существует проблема взаимного выравнивания строк, в случае когда строки различной длины существует множество способов сопоставить соответствующие символы.

Решим проблему введением рекуррентной формулы, обозначим:

1) $L1$ — длина S_1.

2) $L2$ — длина S_2.

3) $S_1[1...i]$ — подстрока S_1 длиной i, начиная с первого символа.

4) $S_2[1...j]$ — подстрока S_2 длиной j, начиная с первого символа.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M и N соответственно рассчитывается по рекуррентной формуле (1.1).

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} \max(i, j), & \text{если } \min(i, j) = 0, \\ \min(D(S_1[1..i], S_2[1..j-1]) + 1, & \\ \quad D(S_1[1..i-1], S_2[1..j]) + 1, & \text{иначе,} \\ \quad D(S_1[1..i-1], S_2[1..j-1]) + & \\ \quad +m(S_1[i], S_2[j])), & \end{cases} \tag{1.1}$$

где сравнение символов строк S_1 и S_2 рассчитывается как:

$$m(a, b) = \begin{cases} 0, & \text{если a = b,} \\ 1, & \text{иначе.} \end{cases} \tag{1.2}$$

Обозначим результат подсчета расстояния Левенштейна для подстрок $S_1[1..i], S_2[1..j]$ как $Lev(S_1[1..i], S_2[1..j])$.

В расстоянии Дамерау—Левенштейна вводится еще одна операция, обозначим ее как S (англ. swap), данная операция применима, только если $S_1[i] = S_2[j-1]$ и $S_1[i-1] = S_2[j]$. Рекуррентная формула данной метрики выглядит следующим образом:

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} min \begin{cases} Lev(S_1[1..i], S_2[1..j]), \\ D(S_1[1..i-2], S_2[1..j-2]) + 1, \end{cases} & \text{если i > 1, j > 1,} \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ Lev(S_1[1..i], S_2[1..j]) & \text{иначе.} \end{cases} \tag{1.3}$$

Идея в том, что после замены пары символов местами, полученные пары были поэлементно равны друг другу.

Вывод

В данном разделе были рассмотрены цели и задачи работы, а также введены необходимые обозначения для поиска расстояний Левенштейна и Дамерау—Левенштейна и выведены рекуррентные отношения для поиска их значения.

2 Конструкторская часть

Рассмотрим различные реализации алгоритмов поиска расстояния Левенштейна и Дамерау—Левенштейна для строк S_1, S_2, каждая имеет длину N, M соответственно.

2.1 Реализации алгоритмов.

В данной части работы будут рассмотрены различные реализации алгоритмов поиска редакционных расстояний и представлены схемы алгоритмов данных реализаций.

2.1.1 Использование матрицы

Вводится матрица M с $N + 1$ на $M + 1$ элементами. Номер строки матрицы i соответствует длине подстроки S_1, номер столбца j — длине подстроки S_2. Значение $M[i][j]$ соответствует расстоянию Левенштейна между подстроками, $S_1[i]$ и $S_2[j]$ соответственно. После чего используется формула (1.1) или (1.3) для расчета значения в данной клетке, таким образом для получения значения, необходимо рассмотреть диагональную, верхнюю и нижнюю клетки, в случае формулы (1.3) также рассматривается клетка $M[i-2][j-2]$.

2.1.2 Использование рекурсии

При использовании рекуррентной формулы возможно использование рекурсии. На вход подаются 2 строки, а также их длины на данный момент, получив строки с длинами i и j, необходимо рассмотреть те же строки с длинами $(i-1, j-1)$, $(i, j-1)$, $(i-1, j)$ и рассчитать их стоимость с помощью формул (1.1) и (1.3), в (1.3) также рассматриваются строки с длинами $(i-2, j-2)$. После чего необходимо выбрать вариант с наименьшей стоимость из возможных.

2.1.3 Использование рекурсии с мемоизацией

Заметим, что метод, описанный в пункте 2.1.2, можно ускорить, если запоминать уже посчитанные значения, и использовать их в дальнейшем. Таким образом вводится матрциа M, аналогичная матрице из пункта 2.1.1, все ячейки которой заполняются $+\infty$, после чего перед вычислением текущего значения происходит проверка: в случае, если

значение $M[i][j] = +\infty$, вычисляется текущее значение и заносится в матрицу, иначе вычисления значения не происходит и значение сразу берется из ячейки $M[i][j]$.

2.1.4 Схемы алгоритмов

Рисунок 2.1 – Схема алгоритма поиска расстояния Левенштейна с помощью матрицы

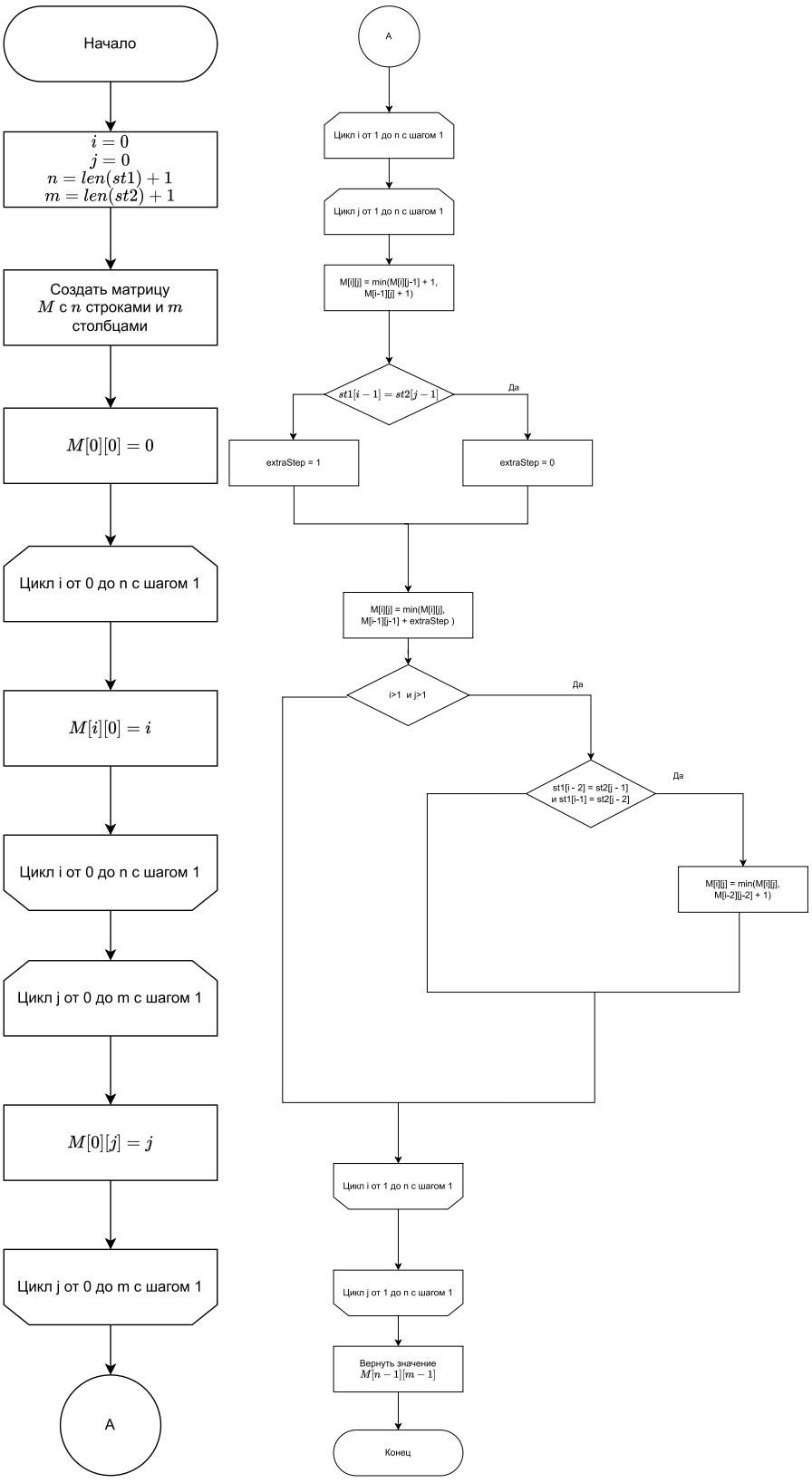

Рисунок 2.2 – Схема алгоритма поиска расстояния Дамерау—Левенштейна с помощью матрицы

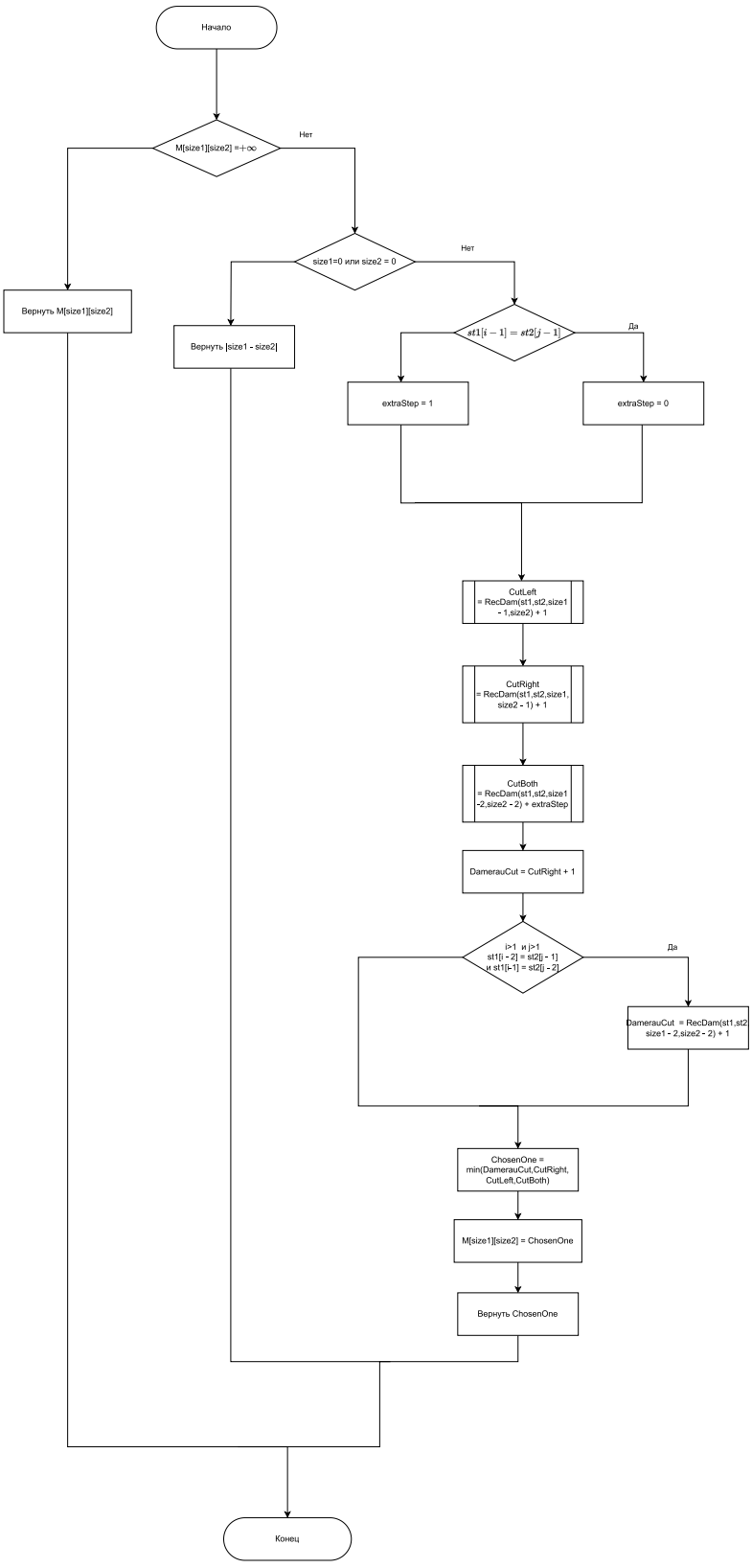

Рисунок 2.3 – Схема алгоритма поиска расстояния Дамерау—Левенштейна с помошью рекурсии с мемоизацией

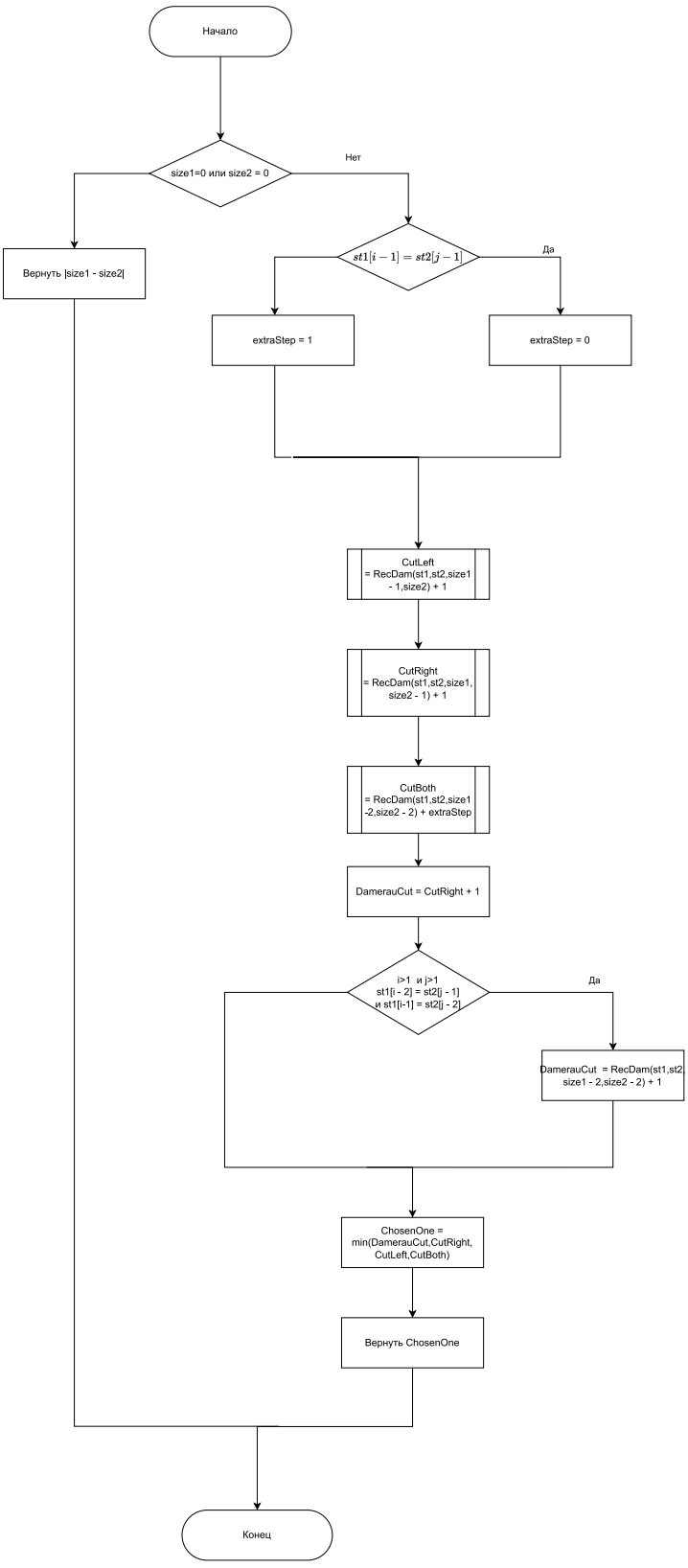

Рисунок 2.4 – Схема алгоритма поиска расстояния Дамерау—Левенштейна с помошью рекурсии

2.1.5 Структуры данных

Для реализации выбранных алгоритмов были использованы следующие структуры данных:

1) Матрица — массив векторов типа int.

2) Строка — массив типа wchar.

3) Длина строки — целое значение типа size_t.

Вывод

На данном этапе были рассмотрены алгоритмы и записаны их схемы, также были описаны используемые структуры данных.

3 Технологическая часть

В данном разделе будут описаны средства реализации, модули программы, а также листинги, модульные и функциональные тесты.

3.1 Средства реализации

Алгоритмы для данной лабораторной работы были реализованы на языке C++, при использовании компилятора gcc версии 10.5.0, так как в стандартной библиотеке приведенного языка присутствует функция `clock_gettime`, которая (при использовании макропеременной `CLOCK_THREAD_CPUTIME_ID`) позволяет рассчитать процессорное время конкретного потока.[**cpp-time**]

3.2 Листинги реализаций алгоритмов

Стоит отметить, что все используемые выше алгоритмы реализованы как метода класса *Matrix*, рекурсивные части алгоритмов были вынесены в отдельные функции. Листинги исходных кодов программ 4.1–4.6 приведены в приложении.

3.3 Тестирование

При написании тестов использовалась библиотека *gtest*, позволяющая писать модульные тесты, которые очень удобны в данном случае. Данные тесты приведены в листинге 4.7. При реализации данных тестов вычисленные значения сравниваются со значениями, заранее известными для соответствующих входных данных, с помощью приведенной библиотеки.

Также данные тесты рассмотрены в таблице 3.1.

Все тесты были успешно пройдены.

Таблица 3.1 – Модульные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау—Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
wwwwwwc	bbbbbbc	6	6	6	6
AB	BA	2	1	1	1
KAABKA	AKAAK	3	3	3	3
ABC	BCA	2	2	2	2
ВФА	АВФ	2	2	2	2
ADF	ABFDSADADF	7	7	7	7

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

1) Процессор Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 2592 МГц, ядер: 6, логических процессоров: 12.
2) Оперативная память: 16 ГБайт.
3) Операционная система: Майкрософт Windows 10 Pro. [**windows**]
4) Использованная подсистема: WSL2. [**WSL2**]

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного программного обеспечения, показаны результаты вычислений расстояний Дамерау–Левенштейна и Левенштейна между словами «fds»,«asd», заметим что в случае поднятого флага вывода матриц происходит вывод матриц, использованных в расчетах.

```
7
Введите первое слово:
fds
Введите второе слово:
asd

Полученная матрица:
0 1 2 3
1 1 2 3
2 2 2 2
3 3 2 3

Полученное расстояние Левенштейна с помощью матрицы:
3

Полученная матрица:
0 1 2 3
1 1 2 3
2 2 2 2
3 3 2 2

Полученное расстояние Дамерау-Левенштейна с помощью матрицы:
2

Полученное расстояние Дамерау-Левенштейна с помощью рекурсии:
2

Полученное расстояние Дамерау-Левенштейна с помощью рекурсии с мемоизацией:
2
```

Рисунок 4.1 – Демонстрация работы программы

4.3 Временные характеристики

Результаты исследования замеров по времени приведены в таблице 4.1. Введем следующие обозначения для чтения таблиц:

1) n — длина анализируемых строк.
2) ДМ — реализация алгоритма поиска расстояния Дамерау—Левенштейна с использованием матрицы.
3) ДР — рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна.
4) ДРМ — рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна с использованием мемоизации.
5) ЛМ — реализация алгоритма поиска расстояния Левенштейна с использованием матрицы.

Заметим, что некоторые поля в данной таблице имеют значение «∼», это обусловлено тем, что дальнейший расчет значений столбца «ДР» окажется слишком долгим, полученных данных достаточно для проведения исследования.

Замеры проводились на одинаковых длин строк от 0 до 60 с шагом 3, для получения достоверных результатов замеры времени для каждой пары строк проводились 100 раз, после чего усреднялись. Все результаты вычислений приведены в миллисекундах.

Приведенные графики 4.2–4.4 получены при помощи анализа данных таблицы 4.1.

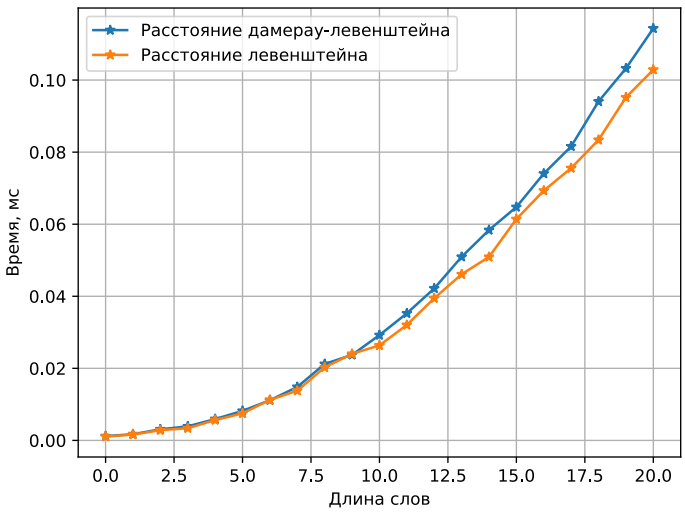

Рисунок 4.2 – Сравнение по времени нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна

Таблица 4.1 – Полученная таблица замеров по времени различных реализаций алгоритмов поиска редакционных расстояний

n	ДМ	ДР	ДРМ	ЛМ
0	0.00118	0.00106	0.00109	0.00106
3	0.00172	0.00219	0.00203	0.00162
6	0.00308	0.12611	0.00388	0.00284
9	0.0039	13.261	0.00534	0.00338
12	0.00592	2489.9	0.00845	0.00565
15	0.0082	∼	0.0115	0.00751
18	0.01115	∼	0.01612	0.01125
21	0.01484	∼	0.02665	0.01379
24	0.02121	∼	0.0322	0.02028
27	0.02374	∼	0.03873	0.02401
30	0.02922	∼	0.04384	0.02638
33	0.03526	∼	0.05304	0.03204
36	0.04217	∼	0.06269	0.03945
39	0.05094	∼	0.07313	0.0461
42	0.05838	∼	0.08418	0.05092
45	0.06476	∼	0.09546	0.06141
48	0.07404	∼	0.11081	0.06931
51	0.08159	∼	0.12869	0.07557
54	0.09406	∼	0.14051	0.0834
57	0.10324	∼	0.15499	0.09519
60	0.11433	∼	0.16834	0.10287

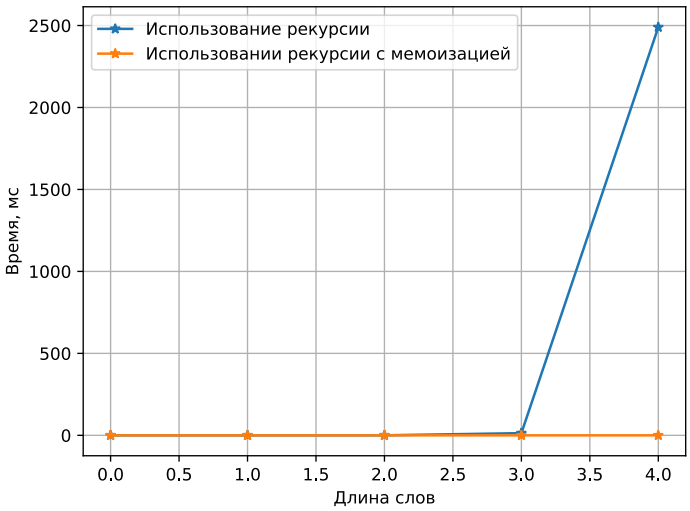

Рисунок 4.3 – Сравнение по времени реализации рекурсивного поиска расстояния Дамерау—Левенштейна с мемоизацией и без

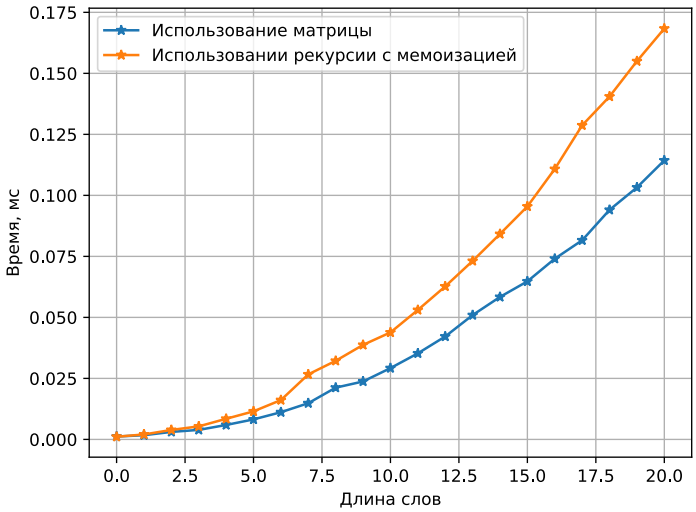

Рисунок 4.4 – Сравнение по времени реализации рекурсивного поиска расстояния Дамерау—Левенштейна с мемоизацией, с поиском расстояния с использоавнием матрицы

4.4 Характеристики по памяти

Введем следующие обозначения:

1) $size_1$ — длина строки S_1.
2) $size_2$ — длина строки S_2,
3) $sizeof$ — функция вычисляющая размер в байтах.
4) $wstring$ — строковый тип.
5) $wstring\&$ — указатель на строковый тип.
6) $matrix\&$ — указатель на матрицу.
7) int — целочисленный тип.
8) $size\_t$ — беззнаковый целочисленный тип.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау—Левенштейна равна сумме входящих строк, так как дерево рекурсии будет расти пока длина хотя бы одной строки не будет равна 0. Таким образом при рекурсивном поиске данного расстояния затраченная память будет рассчитываться аналогично примеру (4.1).

$$(n+m)\cdot(2\cdot sizeof(wstring\&)+sizeof(int)+6\cdot sizeof(size\_t))+2\cdot sizeof(wstring), \quad (4.1)$$

где:

— $2 \cdot sizeof(string)$ — хранение двух строк;

— $2 \cdot sizeof(int)$ — хранение размеров рассматриваемых подстрок;

— $2 \cdot sizeof(string\&)$ — хранение указателей на строки;

— $6 \cdot sizeof(size\_t)$ — вспомогательные переменные;

— $sizeof(int)$ — адрес возврата.

Заметим, что в данном случае считается, что в типе *string* хранятся все необходимые данные для задания строки(т.е. $size + 1$ символ).

Для рекурсивного алгоритма с кешированием поиска расстояния Дамерау—Левенштейна будет теоретически схож с расчетом в формуле (4.1), но также учитывается матрица, соответственно, максимальный расход памяти равен:

$$\begin{aligned}(n+m) \cdot (2 \cdot sizeof(string\&) + sizeof(int) + \\ + sizeof(matrix\&) + 6 \cdot sizeof(size\_t)) + \\ + (2 \cdot sizeof(string) + sizeof(matrix)).\end{aligned} \tag{4.2}$$

Затраты памяти на хранение матрицы описаны в (4.3).

$$\begin{aligned}sizeof(matrix) = sizeof(int) \cdot (n+1) \cdot (m+1) + \\ + sizeof(int) \cdot 2 + sizeof(int\&) \cdot (n+1) + sizeof(int\&\&).\end{aligned} \tag{4.3}$$

В данном случае считается, что:

— $sizeof(int) \cdot (n+1) \cdot (m+1)$ — хранение данных матрицы;

— $sizeof(int) \cdot 2$ — хранение размеров матрицы;

— $sizeof(int\&) \cdot (n+1)$ — хранение строк матрицы;

— $sizeof(int\&\&)$ — хранение указателя на матрицу;

Использование памяти при итеративной реализации алгоритма поиска расстояния Левенштейна равно:

$$\begin{aligned}sizeof(matrix) + sizeof(matrix\&) + 2 \cdot sizeof(wstring\&) + \\ + 2 \cdot sizeof(wstring) + sizeof(int) + 3 \cdot sizeof(int),\end{aligned} \tag{4.4}$$

где:

— $sizeof(matrix) + sizeof(matrix\&)$ — хранение матрицы и ее указателя в кадре стека;

— $2 \cdot sizeof(wstring) + 2 \cdot sizeof(wstring)$ — хранение строк и их указателей в кадре стека;

— $(n+1) \cdot (m+1) \cdot size(int)$ — хранение матрицы;

— $3 \cdot sizeof(int)$ — вспомогательные переменные.

— $sizeof(int)$ — адрес возврата.

Использование памяти при итеративной реализации алгоритма поиска расстояния Дамерау—Левенштейна аналогично приведенному в (4.4).

По приведенным формулам затрат по памяти в программе были написаны соответствующие функции для подсчета расходуемой памяти, результаты расчетов, которых представлены в таблице 4.2, где размеры строк находятся в диапазоне от 10 до 200 с шагом 10.

Таблица 4.2 – Полученная таблица замеров по памяти различных реализаций алгоритмов поиска редакционных расстояний

n	ДМ	ДР	ДРМ	ЛМ
0	200	64	132	200
10	760	2464	3652	760
20	2120	4864	7972	2120
30	4280	7264	13092	4280
40	7240	9664	19012	7240
50	11000	12064	25732	11000
60	15560	14464	33252	15560
70	20920	16864	41572	20920
80	27080	19264	50692	27080
90	34040	21664	60612	34040
100	41800	24064	71332	41800
110	50360	26464	82852	50360
120	59720	28864	95172	59720
130	69880	31264	108292	69880
140	80840	33664	122212	80840
150	92600	36064	136932	92600
160	105160	38464	152452	105160
170	118520	40864	168772	118520
180	132680	43264	185892	132680
190	147640	45664	203812	147640
200	163400	48064	222532	163400

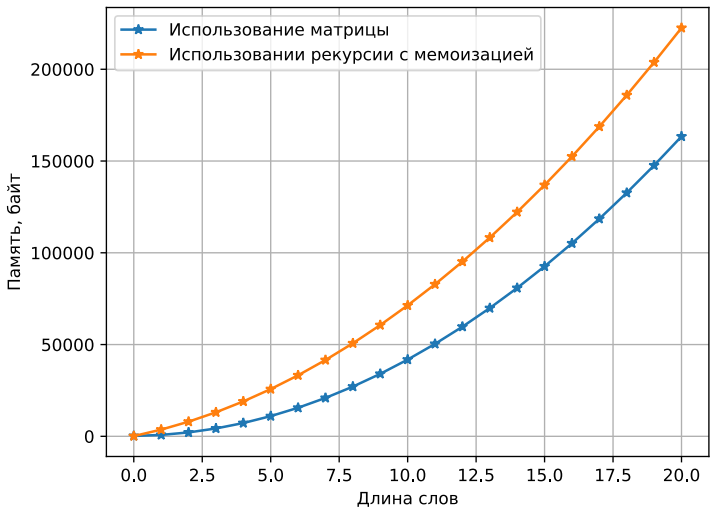

Рисунок 4.5 – Сравнение по памяти алгоритмов поиска расстояния Левенштейна и Дамерау—Левенштейна — итеративной и рекурсивной реализации

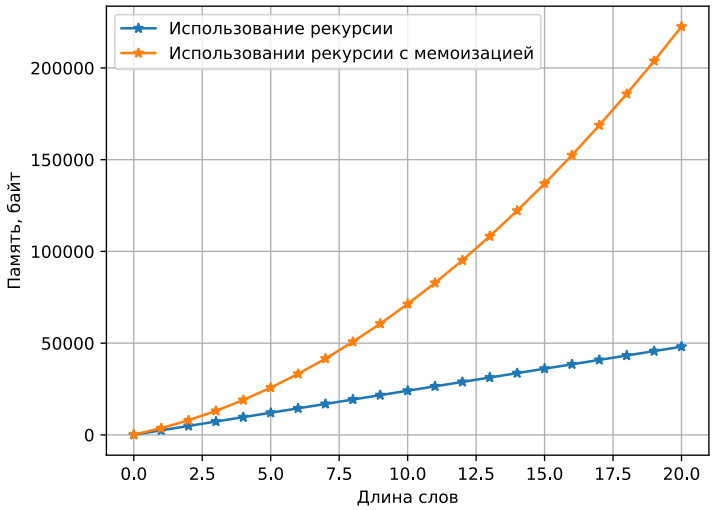

Рисунок 4.6 – Сравнение по памяти алгоритмов поиска расстояния Дамерау—Левенштейна — рекурсивной реализации с мемоизацией и без

4.5 Результаты анализа различных реализаций

4.5.1 Сравнение по времени исполнения

Имеет смысл сравнить поиск расстояний Левенштейна и Дамерау—Левенштейна при использовании матрицы, используя рисунок 4.2. Заметим, что при использовании алгоритма поиска расстояния Дамерау—Левенштейна, при рассмотрении слов длины больше 50 затрачивается в 1.1 раз больше времени, чем при использовании алгоритма Левенштейна. Данная закономерность обусловлена дополнительной операцией обмена символов, которую необходимо учитывать при поиске расстояния Дамерау—Левенштейна.

Также необходимо сравнить реализации рекурсивного поиска расстояния Дамерау—Левенштейна с мемоизацией и без, полученный график приведен на картинке 4.3. Заметим, что рекурсия с мемоизацией, за счет сохранения информации о уже рассчитанных расстояниях подстрок показала себя в 29455 раз лучше, чем рекурсия без мемоизации.

Стоит отметить, что при сравнении реализации рекурсивного поиска расстояния Дамерау—Левенштейна с мемоизацией и поиска с использованием матрицы (графики замеров представлены на рисунке 4.4), реализация с использованием матрицы показала себя более эффективной по времени, затратив в 1,47 раз меньше миллисекунд. Наименее затратным по времени является итеративная реализация алгоритма нахождения расстояния Левенштейна.

4.5.2 Сравнение затраченной памяти

Проанализировав таблицу 4.2, можно сделать вывод, что по расходу памяти итеративные алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в итеративном растет как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк. Также стоит отметить, что при малых длинах слов рекурсивные алгоритмы уступают итеративным, это обусловлено большим количеством вспомогательных переменных в реализации рекурсивного алгоритма. Наиболее эффективным по памяти является рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна.

Рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна будет более затратной по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау—Левенштейна, но менее затратным по памяти по отношению к итеративному алгоритму Дамерау—Левенштейна.

ЗАКЛЮЧЕНИЕ

В результате исследования было определено, что время алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна растет в геометрической прогрессии при увеличении длин строк. Лучшие показатели по времени дает матричная реализация алгоритма нахождения расстояния Дамерау-Левенштейна и его рекурсивная реализация с мемоизацией. При этом итеративная реализация с использованием матрицы занимают довольно много памяти при большой длине строк.

Цели данной лабораторной работы были достигнуты, а именно описание и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной целей были выполнены следующие задачи.

1) Описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна.
2) Создано программное обеспечение, реализующее следующие алгоритмы.
   - нерекурсивный метод поиска расстояния Левенштейна;
   - нерекурсивный метод поиска расстояния Дамерау-Левенштейна;
   - рекурсивный метод поиска расстояния Дамерау-Левенштейна;
   - рекурсивный с кешированием метод поиска расстояния Дамерау-Левенштейна.
3) Выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов.
4) Проведены анализ затрат работы программы по времени и по памяти, выяснены влияющие на них характеристики.

./lib.bib

ПРИЛОЖЕНИЕ А

Приведены листинги реализаций расчета редакционных расстояний.

Листинг 4.1 – Метод нахождения расстояния Левенштейна с использованием матрицы

```
1     size_t Matrix::findMatrixDistLev(const std::wstring&
         st1, const std::wstring& st2)
2     {
3         int n = st1.size() + 1;
4         int m = st2.size() + 1;
5         if (n > _n || m > _m)
6             assert(0);
7         _table[0][0] = 0;
8         for (int i = 1; i < n; i++)
9             _table[i][0] = i;
10        for (int j = 1; j < m; j++)
11            _table[0][j] = j;
12        for (int i = 1; i < n; i++)
13        {
14            for (int j = 1; j < m; j++)
15            {
16
17                _table[i][j] = std::min(_table[i][j - 1] +
                     1, _table[i - 1][j] + 1);
18                int extraStep = 0;
19                if (st1[i - 1] != st2[j - 1])
20                    extraStep++;
21                _table[i][j] = std::min(_table[i][j],
                     _table[i - 1][j - 1] + extraStep);
22            }
23        }
24        return _table[n - 1][m - 1];
25    }
```

Листинг 4.2 – Метод нахождения расстояния Дамерау—Левенштейна с использованием матрицы

```
1     size_t Matrix::findMatrixDistDamerau(const std::wstring&
         st1, const std::wstring& st2)
2     {
3         int n = st1.size() + 1;
4         int m = st2.size() + 1;
5         if (n > _n || m > _m)
```

```
6             assert(0);
7         _table[0][0] = 0;
8         for (int i = 1; i < n; i++)
9             _table[i][0] = i;
10        for (int j = 1; j < m; j++)
11            _table[0][j] = j;
12        for (int i = 1; i < n; i++)
13        {
14            for (int j = 1; j < m; j++)
15            {
16
17                _table[i][j] = std::min(_table[i][j - 1] +
                      1, _table[i - 1][j] + 1);
18                int extraStep = 0;
19                if (st1[i - 1] != st2[j - 1])
20                    extraStep++;
21
22                _table[i][j] = std::min(_table[i][j],
                      _table[i - 1][j - 1] + extraStep);
23                if (i > 1 && j > 1)
24                {
25                    if (st1[i - 1] == st2[j - 2] && st1[i -
                          2] == st2[j - 1])
26                        _table[i][j] =
                              std::min(_table[i][j], _table[i -
                              2][j - 2] + 1);
27                }
28            }
29        }
30        return _table[n - 1][m - 1];
31    }
```

Листинг 4.3 – Функция нахождения расстояния Дамерау—Левенштейна с использованием рекурсии

```
1     size_t RecurseDistDamerau(const std::wstring& st1, const
          std::wstring& st2, int size1, int size2)
2     {
3         if (size1 == 0)
4             return size2;
5         if (size2 == 0)
6             return size1;
7
```

```
8         size_t extraStep = 0;
9         if (st1[size1 - 1] != st2[size2 - 1])
10            extraStep++;
11
12        size_t cutLeft = RecurseDistDamerau(st1, st2, size1
              - 1, size2) + 1;
13        size_t cutRight = RecurseDistDamerau(st1, st2,
              size1, size2 - 1) + 1;
14        size_t cutBoth = RecurseDistDamerau(st1, st2, size1
              - 1, size2 - 1) + extraStep;
15        size_t DamerauCut = cutBoth + 1;
16        if (size1 > 1 && size2 > 1 && st1[size1 - 1] ==
              st2[size2 - 2]
17            && st1[size1 - 2] == st2[size2 - 1])
18            DamerauCut = RecurseDistDamerau(st1, st2, size1
                  - 2, size2 - 2) + 1;
19
20        cutBoth = std::min(DamerauCut, cutBoth); //need
              static here
21        size_t chosenOne = std::min(std::min(cutRight,
              cutLeft), cutBoth);
22        return chosenOne;
23    }
```

Листинг 4.4 – Функция нахождения расстояния Дамерау—Левенштейна с использованием рекурсии с мемоизацией

```
1     size_t RecurseDistMemDamerau(const std::wstring& st1,
2     const std::wstring& st2,
3     int lastIndex1,
4     int lastIndex2,
5     Matrix& mat)
6     {
7         //std::cout<< "i " << "j " << lastIndex1 << " " <<
              lastIndex2 << std::endl;
8         if (mat._table[lastIndex1][lastIndex2] != INF)
9             return mat._table[lastIndex1][lastIndex2];
10
11        if (lastIndex1 == 0)
12        {
13            mat._table[lastIndex1][lastIndex2] = lastIndex2;
14            return lastIndex2;
15        }
```

```
16          if (lastIndex2 == 0)
17          {
18              mat._table[lastIndex1][lastIndex2] = lastIndex1;
19              return lastIndex1;
20          }
21
22
23
24          int extraStep = 0;
25          if (st1[lastIndex1 - 1] != st2[lastIndex2 - 1])
26              extraStep++;
27
28          int cutLeft = RecurseDistMemDamerau(st1, st2,
               lastIndex1 - 1, lastIndex2,mat) + 1;
29          int cutRight = RecurseDistMemDamerau(st1, st2,
               lastIndex1, lastIndex2 - 1,mat) + 1;
30          int cutBoth = RecurseDistMemDamerau(st1, st2,
               lastIndex1 - 1, lastIndex2 - 1,mat) + extraStep;
31          int DamerauCut = cutBoth + 1;
32          if (lastIndex1 > 1 && lastIndex2 > 1 &&
               st1[lastIndex1 - 2] == st2[lastIndex2 - 1]
33              && st1[lastIndex1 - 1] == st2[lastIndex2 - 2])
34              DamerauCut = RecurseDistMemDamerau(st1, st2,
                    lastIndex1 - 2, lastIndex2 - 2,mat) + 1;
35          cutBoth = std::min(DamerauCut, cutBoth);
36          int chosenOne = std::min(std::min(cutRight,
               cutLeft), cutBoth);
37          mat._table[lastIndex1][lastIndex2] = chosenOne;
38          return chosenOne;
39      }
```

Листинг 4.5 – Метод нахождения расстояния Дамерау—Левенштейна с использованием функции 4.4

```
1       const int INF = 1e8;
2       size_t Matrix::findRecurseDistMemDamerau(const
           std::wstring& st1, const std::wstring& st2)
3       {
4           for (int i = 0; i < _n; i++)
5               for (int j = 0; j < _m; j++)
6                   _table[i][j] = INF;
7
8           return RecurseDistMemDamerau(st1, st2, st1.size(),
```

```
            st2.size(), *this);
9    }
```

Листинг 4.6 – Метод нахождения расстояния Дамерау—Левенштейна с использованием функции 4.3

```
1 size_t Matrix::findRecurseDistDamerau(const std::wstring&
     st1, const std::wstring& st2)
2 {
3     return RecurseDistDamerau(st1, st2, st1.size(),
         st2.size());
4 }
```

ПРИЛОЖЕНИЕ Б

Приведена реализация тестов для разработанного ПО.

Листинг 4.7 – Модульные тесты

```
1     //
2     // Created by Андрей on 10/09/2023.
3     //
4     #include <gtest/gtest.h>
5     #include "../../src/table.hpp"
6
7     //Тестирование при словах одинаковой длины
8     TEST(LenTest, EQLENTEST) {
9         std::wstring st1 = L"wwwwwwc";
10        std::wstring st2 = L"bbbbbbc";
11        Matrix mat(st1.size(),st2.size());
12
13        int DamerauLen = 6;
14        int LevLen  = 6;
15
16        int lenMatDamerau =  mat.findMatrixDistDamerau(st1,
              st2);
17        int lenRecurseMemDamerau =
              mat.findRecurseDistMemDamerau(st1, st2);
18        int lenRecurseDamerau =
              mat.findRecurseDistDamerau(st1, st2);
19
20        int lenMatLev = mat.findMatrixDistLev(st1,st2);
21
22        ASSERT_EQ(DamerauLen,lenMatDamerau);
23        ASSERT_EQ(DamerauLen,lenRecurseDamerau);
24        ASSERT_EQ(DamerauLen,lenRecurseMemDamerau);
25        ASSERT_EQ(LevLen,lenMatLev);
26    }
27
28    //Тестирование с различными результатами при использовани
          и различных расстояний
29    TEST(LenTest, DIFLENTEST) {
30        std::wstring st1 = L"AB";
31        std::wstring st2 = L"BA";
32        Matrix mat(st1.size(),st2.size());
33
34        int DamerauLen = 1;
```

```
35          int LevLen  = 2;
36
37          int lenMatDamerau =  mat.findMatrixDistDamerau(st1,
                st2);
38          int lenRecurseMemDamerau =
                mat.findRecurseDistMemDamerau(st1, st2);
39          int lenRecurseDamerau =
                mat.findRecurseDistDamerau(st1, st2);
40
41          int lenMatLev = mat.findMatrixDistLev(st1,st2);
42
43          ASSERT_EQ(DamerauLen,lenMatDamerau);
44          ASSERT_EQ(DamerauLen,lenRecurseDamerau);
45          ASSERT_EQ(DamerauLen,lenRecurseMemDamerau);
46          ASSERT_EQ(LevLen,lenMatLev);
47      }
48
49
50      //Различающийся результат расстояний при словах различной
            длины
51      TEST(LenTest, LongWords) {
52          std::wstring st1 = L"KAABKA";
53          std::wstring st2 = L"AKAAK";
54          Matrix mat(st1.size(),st2.size());
55
56          int DamerauLen = 3;
57          int LevLen  = 3;
58
59          int lenMatDamerau =  mat.findMatrixDistDamerau(st1,
                st2);
60          int lenRecurseMemDamerau =
                mat.findRecurseDistMemDamerau(st1, st2);
61          int lenRecurseDamerau =
                mat.findRecurseDistDamerau(st1, st2);
62
63          int lenMatLev = mat.findMatrixDistLev(st1,st2);
64
65          ASSERT_EQ(DamerauLen,lenMatDamerau);
66          ASSERT_EQ(DamerauLen,lenRecurseDamerau);
67          ASSERT_EQ(DamerauLen,lenRecurseMemDamerau);
68          ASSERT_EQ(LevLen,lenMatLev);
```

```
69      }
70
71      //Тестирование подобное тесту на русском языке
72      TEST(LenTest, English) {
73          std::wstring st1 = L"BCA";
74          std::wstring st2 = L"ABC";
75          Matrix mat(st1.size(),st2.size());
76
77          int DamerauLen = 2;
78          int LevLen  = 2;
79
80          int lenMatDamerau =  mat.findMatrixDistDamerau(st1,
                st2);
81          int lenRecurseMemDamerau =
                mat.findRecurseDistMemDamerau(st1, st2);
82          int lenRecurseDamerau =
                mat.findRecurseDistDamerau(st1, st2);
83
84          int lenMatLev = mat.findMatrixDistLev(st1,st2);
85
86          ASSERT_EQ(DamerauLen,lenMatDamerau);
87          ASSERT_EQ(DamerauLen,lenRecurseDamerau);
88          ASSERT_EQ(DamerauLen,lenRecurseMemDamerau);
89          ASSERT_EQ(LevLen,lenMatLev);
90      }
91
92      //Использование русских букв
93      TEST(LenTest, Russian) {
94          std::wstring st1 = L"ВФА";
95          std::wstring st2 = L"АВФ";
96          Matrix mat(st1.size(),st2.size());
97
98          int DamerauLen = 2;
99          int LevLen  = 2;
100
101         int lenMatDamerau =  mat.findMatrixDistDamerau(st1,
                st2);
102         int lenRecurseMemDamerau =
                mat.findRecurseDistMemDamerau(st1, st2);
103         int lenRecurseDamerau =
                mat.findRecurseDistDamerau(st1, st2);
```

```
104
105         int lenMatLev = mat.findMatrixDistLev(st1,st2);
106
107         ASSERT_EQ(DamerauLen,lenMatDamerau);
108         ASSERT_EQ(DamerauLen,lenRecurseDamerau);
109         ASSERT_EQ(DamerauLen,lenRecurseMemDamerau);
110         ASSERT_EQ(LevLen,lenMatLev);
111     }
112
113     //Большая разница в длине слов
114     TEST(LenTest, BIGSIZEDIFF) {
115         std::wstring st1 = L"ADF";
116         std::wstring st2 = L"ABFDSADADF";
117         Matrix mat(st1.size(),st2.size());
118
119         int DamerauLen = 7;
120         int LevLen  = 7;
121
122         int lenMatDamerau =  mat.findMatrixDistDamerau(st1,
              st2);
123         int lenRecurseMemDamerau =
              mat.findRecurseDistMemDamerau(st1, st2);
124         int lenRecurseDamerau =
              mat.findRecurseDistDamerau(st1, st2);
125
126         int lenMatLev = mat.findMatrixDistLev(st1,st2);
127
128         ASSERT_EQ(DamerauLen,lenMatDamerau);
129         ASSERT_EQ(DamerauLen,lenRecurseDamerau);
130         ASSERT_EQ(DamerauLen,lenRecurseMemDamerau);
131         ASSERT_EQ(LevLen,lenMatLev);
132     }
```