МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ИМ. Н. Э. БАУМАНА

УДК ______
№ госрегистрации ______
Инв. №______

УТВЕРЖДАЮ

____
головной исполнитель НИР

__________________________ ___________

«______» _____________________ 2019 г.

ОТЧЁТ ПО ДИСЦИПЛИНЕ "АНАЛИЗ АЛГОРИТМОВ"
О ЛАБОРАТОРНОЙ РАБОТЕ №1

по теме:
"Расстояние Левенштейна и Дамерау-Левенштейна"

(промежуточный)

Студент ИУ7-53Б ___ Пудов Дмитрий Юрьевич

Москва 2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

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

Постановка задачи:

— изучить метод метод динамического программирования на материала алгоритмов Левенштейна и Дамерау-Левенштейна;

— применить его;

— получить практические навыки реализации указанных алгоритмов.

1 Аналитическая часть

В данной части будут описаны суть задач нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также математические способы их решения.

Поиск расстояния Левенштейна заключается в определении минимального количества редакционных операций (вставка, удаление, замена одного символа), необходимых для трансформации одной строки в другую.

В задаче о расстоянии Дамерау-Левенштейна во множество редакционных операций добавляется транспозиция - перестановка двух соседних символов.

1.1 Описание алгоритмов

Пусть a и b - строки над некоторым алфавитом длины M и N соответственно. Тогда расстояние Левенштейна определяется формулой $d(a, b) = D(M, N)$, где

$$D(i,j) = \begin{cases} 0 & i = 0, j = 0 \\ i & j = 0, i > 0 \\ j & i = 0, j > 0 \\ min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + 1_{a[i] \neq b[j]} \end{cases} \end{cases}$$

Расстояние Дамерау-Левенштейна для данных строк определяется как $d(a,b) = D(M,N)$, где

$$D(i,j) = \begin{cases} max(i,j) & min(i,j) = 0 \\ min \begin{cases} D(i,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+1_{a[i]\neq b[j]} \\ D(i-2,j-2)+1 \end{cases} & i > 1 \text{ и } j > 1 \\ \text{ и } a[i] = b[j-1] \text{ и } a[i-1] = b[j] & \\ min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+1_{a[i]\neq b[j]} \end{cases} & \text{иначе.} \end{cases}$$

2 Конструкторская часть

В данной части будут приведены схемы алгоритмов Левенштейна в рекурсивной и матричной реализации и Дамерау-Левенштейна.

2.1 Разработка алгоритмов

Далее указаны разработанные схемы алгоритмов Левенштейна и Дамерау-Левенштейна. Будем считать, что известны следующие функции: определения длины строки, поиска максимума и минимума среди нескольких чисел. Для матричных реализаций требуется наличие функции, динамически выделяющей память.

На рис. 2.1 представлена схема алгоритма определения расстояния Левенштейна в рекурсивной реализации.

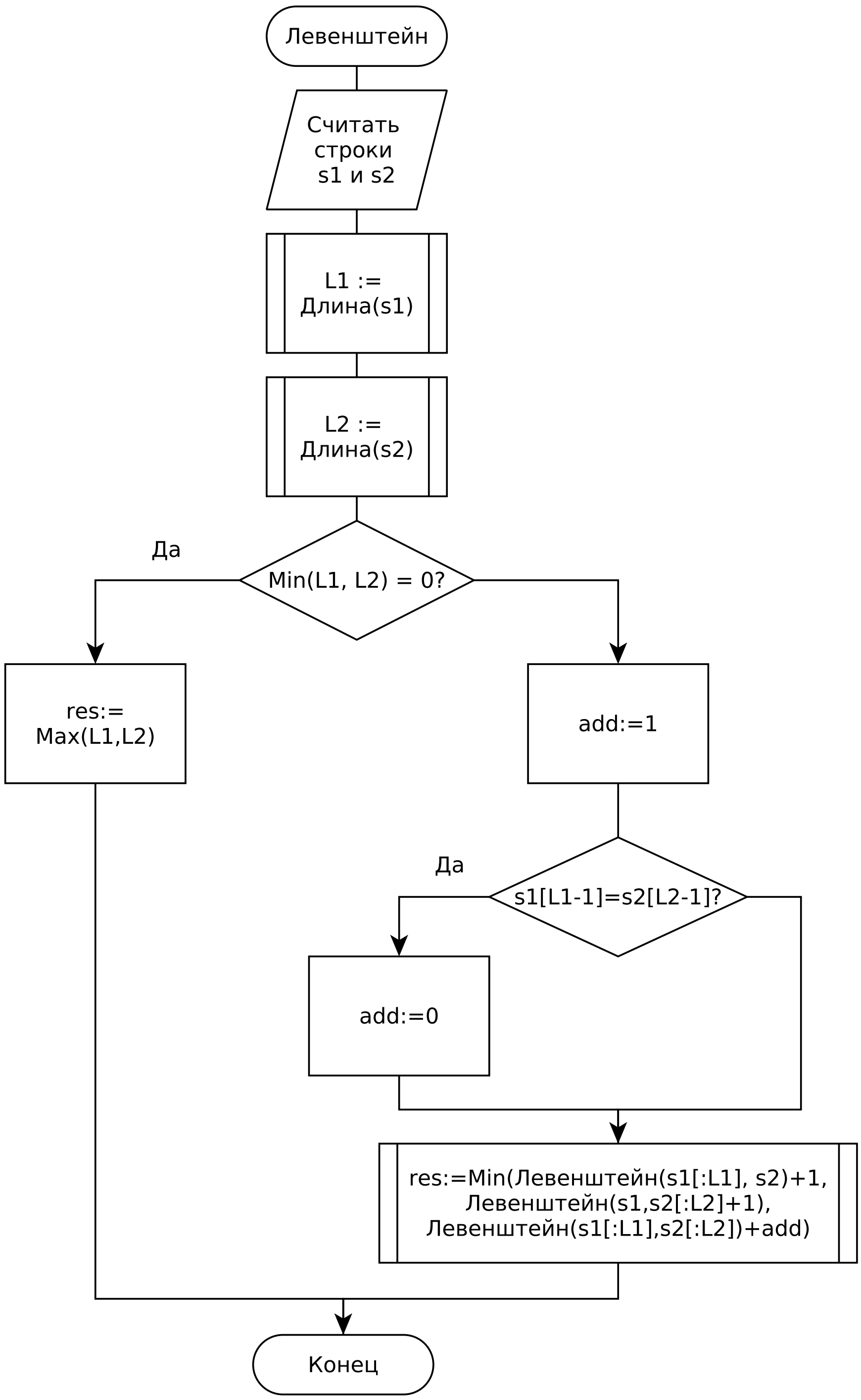


Рисунок 2.1 — Схема алгоритма определения расстояния Левенштейна в рекурсивной реализации.

На рис. 2.2 и на рис. 2.3 представлена 1 часть схемы алгоритма определения расстояния Левенштейна в матричной реализации.

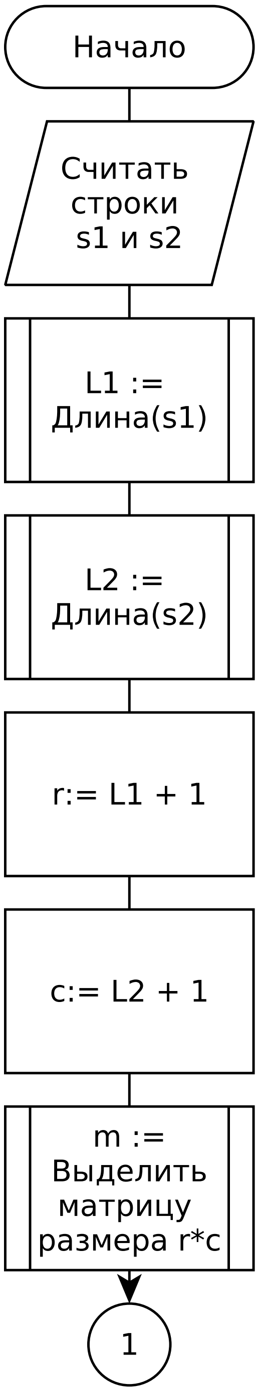


Рисунок 2.2 — Схема алгоритма определения расстояния Левенштейна в матричной реализации. Часть 1.

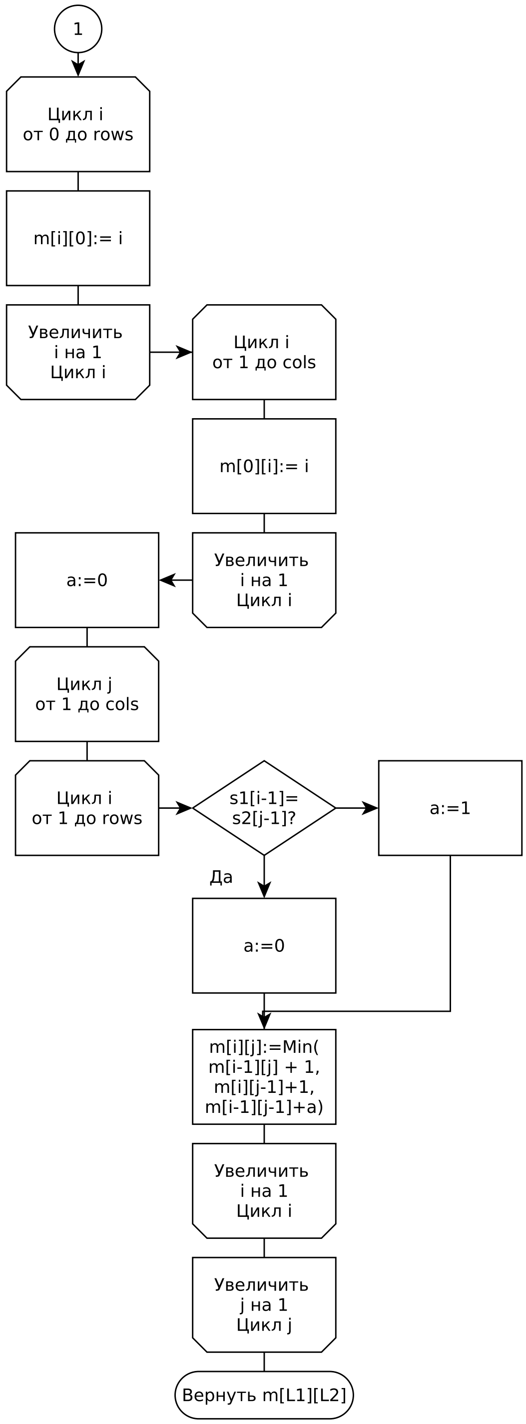


Рисунок 2.3 — Схема алгоритма определения расстояния Левенштейна в матричной реализации. Часть 2.

На рис. 2.4 и рис. 2.5 представлена схема алгоритма определения расстояния Дамерау-Левенштейна в матричной реализации.

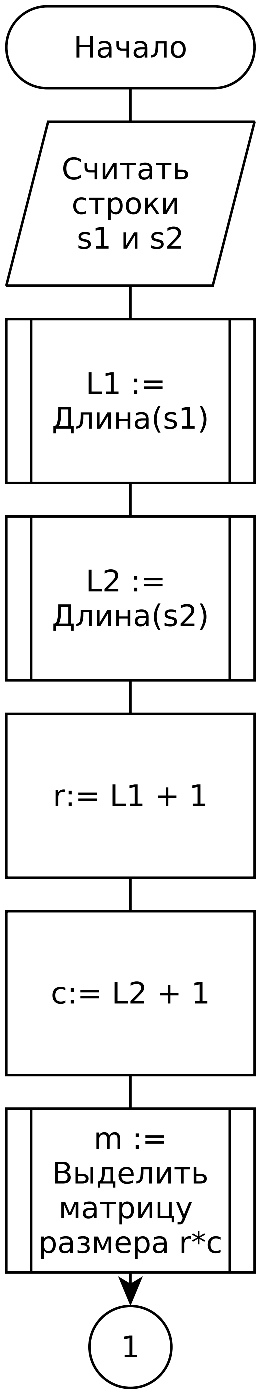


Рисунок 2.4 — Схема алгоритма определения расстояния Дамерау-Левенштейна. Часть 1.

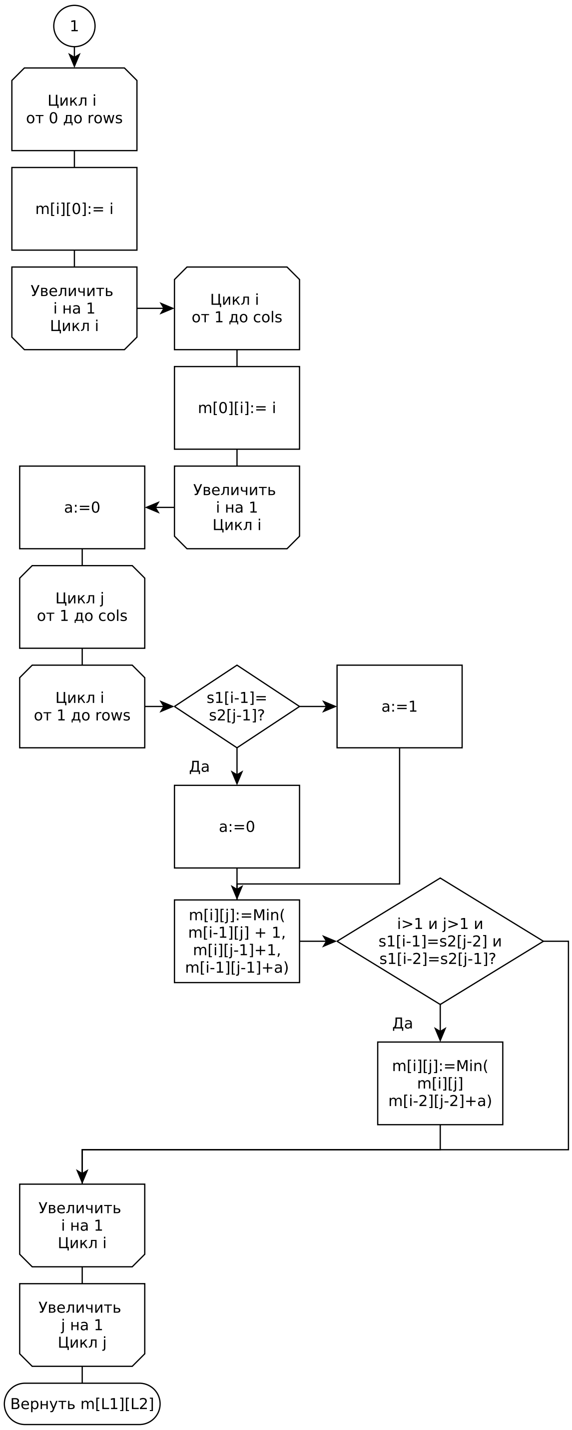


Рисунок 2.5 — Схема алгоритма определения расстояния Дамерау-Левенштейна. Часть 2.

2.2 Сравнительный анализ рекурсивной и нерекурсивной реализаций

Рекурсивная реализация по сравнению с матричной будет иметь большую сложность. В матричной количество итераций заранее известно и будет равно $m * n$, где m - число строк, а n -число столбцов.

При этом в рекурсивной версии сложность может достигать $3^{max(m,n)}$, так как на каждый вызов функции может требоваться ещё 3. При этом в данном дереве рекурсии на некоторых ветвях будут производиться повторные вычисления, что неэффективно.

3 Технологическая часть

В данной части будет описан подход для программной реализации ранее указанных алгоритмов.

3.1 Требования к программному обеспечению

Разработанное ПО должно предоставлять возможность замеров процессорного времени выполнения каждого алгоритма. Требуется вводить 2 строки и выводить матрицу (кроме рекурсивной реализации) и значения расстояний, полученных различными реализациями.

3.2 Средства реализации

Для реализации ПО был выбран язык Go, поскольку в его основном пакете присутствуют удобные средства замера процессорного времени и памяти модуля testing.

3.3 Листинг кода

В данном разделе приведены листинги реализаций ранее указанных алгоритмов на языке Go.

Листинг 3.1 — Матричная версия алгоритма Дамерау-Левенштейна

```
func LevenshteinDamerau(first string, second string) (int, [][] int) {
    lenFirst := len(first)
    lenSecond := len(second)
    rows := lenFirst + 1
    cols := lenSecond + 1
    matrix := allocateMatrix(rows, cols)

    for i := 0; i < rows; i++ {
        matrix[i][0] = i
    }
    for i := 1; i < cols; i++ {
        matrix[0][i] = i
    }

    add := 0
```

```
    for j := 1; j < cols; j++ {
        for i := 1; i < rows; i++ {
            if first[i-1] == second[j-1] {
                add = 0
            } else {
                add = 1
            }
            matrix[i][j] = MinThree(matrix[i-1][j]+1,
                matrix[i][j-1]+1,
                matrix[i-1][j-1]+add)

            if i > 1 && j > 1 && first[i-1] == second[j-2] && first[i-2] ==
               second[j-1] {
                matrix[i][j] = Min(matrix[i][j], matrix[i-2][j-2]+add)
            }
        }
    }

    return matrix[lenFirst][lenSecond], matrix
}
```

Листинг 3.2 — Матричная реализация алгоритма Левенштейна

```
func LevenshteinIterative(first string, second string) (int, [][]int) {
    lenFirst := len(first)
    lenSecond := len(second)
    rows := lenFirst + 1
    cols := lenSecond + 1
    matrix := allocateMatrix(rows, cols)

    for i := 0; i < rows; i++ {
        matrix[i][0] = i
    }
    for i := 1; i < cols; i++ {
        matrix[0][i] = i
    }

    add := 0
    for j := 1; j < cols; j++ {
        for i := 1; i < rows; i++ {
            if first[i-1] == second[j-1] {
                add = 0
            } else {
                add = 1
            }
            matrix[i][j] = MinThree(matrix[i-1][j]+1,
                matrix[i][j-1]+1,
```

```
                    matrix[i-1][j-1]+add)
            }
        }

    return matrix[lenFirst][lenSecond], matrix
}
```

Листинг 3.3 — Рекурсивная версия алгоритма Левенштейна

```
func LevenshteinRecursive(first string, second string) int {
    lenFirst := len(first)
    lenSecond := len(second)
    if Min(lenFirst, lenSecond) == 0 {
        return Max(lenFirst, lenSecond)
    } else {
        add := 1
        if first[lenFirst-1] == second[lenSecond-1] {
            add = 0
        }

        return MinThree(
            LevenshteinRecursive(first[:lenFirst-1], second)+1,
            LevenshteinRecursive(first, second[:lenSecond-1])+1,
            LevenshteinRecursive(first[:lenFirst-1],
                second[:lenSecond-1])+add)
    }
}
```

Листинг 3.4 — Оптимизированная версия рекурсивного алгоритма Левенштейна

```
func levenshteinRecursiveModule(first, second string, result, minval int)
    int {
    lenFirst := len(first)
    lenSecond := len(second)
    if result >= minval {
        return minval
    } else if lenFirst == 0 {
        return result +lenSecond
    } else if lenSecond == 0 {
        return result + lenFirst
    } else {
        add := 1
        if first[lenFirst-1] == second[lenSecond-1] {
            add = 0
        }
        r1 := levenshteinRecursiveModule(first[:lenFirst-1],
            second[:lenSecond-1], result+add, minval)
```

```
        r2 := levenshteinRecursiveModule(first, second[:lenSecond-1], result+1, Min(r1, minval))
        r3 := levenshteinRecursiveModule(first[:lenFirst-1], second, result+1, MinThree(r1, r2, minval))
        return MinThree(r1, r2, r3)
    }
}

func LevenshteinRecursiveOptimized(first string, second string) int {
    minval := Max(len(first), len(second))
    result := 0
    return levenshteinRecursiveModule(first, second, result, minval)
}
```

3.4 Описание тестирования

Тестирование будет проведено для каждой из реализаций со следующими входными данными:

— пустые строки;

— идентичные строки;

— строки одинаковой длины, но с разными символами;

— строки различной длины, но отличных лишь в некотором числе вставок или удалений символа;

— строки различной длины и требующих замены символа.

Для расстояния Дамерау-Левенштейна дополнительный тест будет заключаться в перестановке двух соседних символов.

4 Экспериментальная часть

В данной части будет проведена апробация и тестирование разработанной программы.

4.1 Примеры работы

Программа имеет консольный интерфейс. Далее будут приведены скриншоты.

На рис. 4.1 показаны возможные сценарии запуска программы.

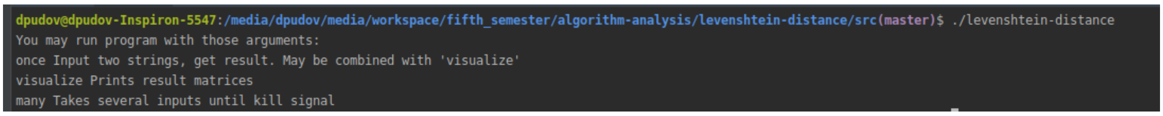


Рисунок 4.1 — Использование программы.

На рис. 4.2 показан пример использования программы с однократным вводом входных данных.

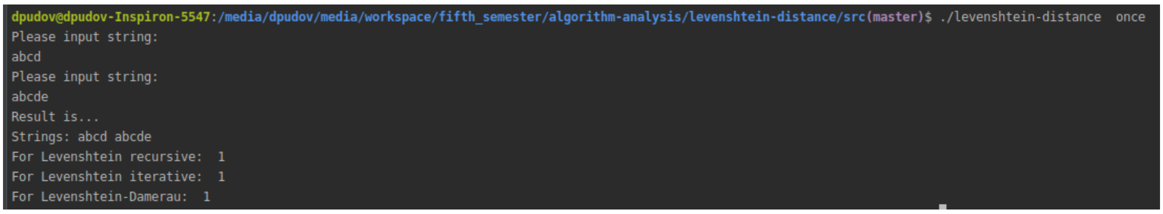


Рисунок 4.2 — Однократное использование.

На рис. 4.3 показан пример использования программы с выводом матриц.

Рисунок 4.3 — Использование с выводом матриц.

На рис. 4.4 показан пример использования программы со множеством входных данных.

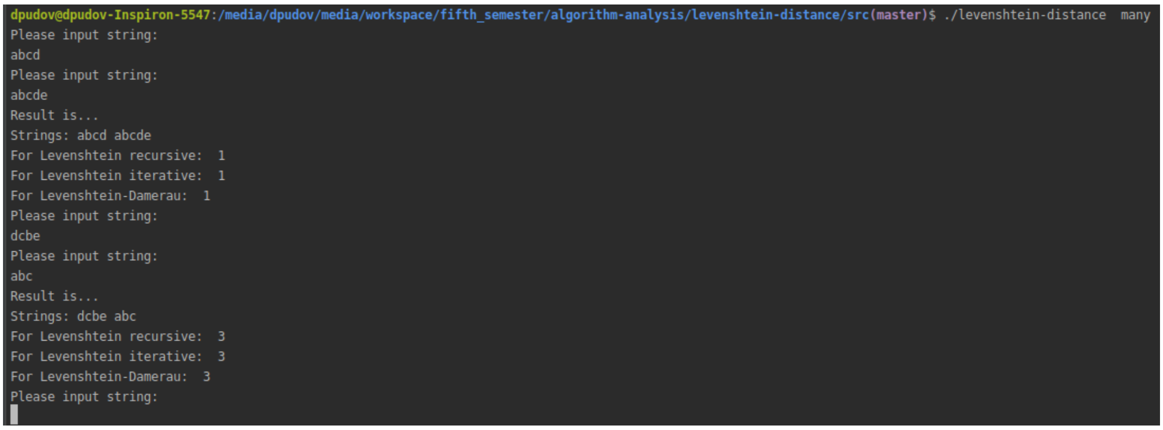


Рисунок 4.4 — Многократное использование.

4.2 Результаты тестирования

В данном разделе будут расположены результаты тестирования ранее описанных реализаций.

В табл. 4.1 отражены результаты тестирования рекурсивной реализации алгоритма поиска расстояния Левенштейна.

Таблица 4.1 — Результаты тестирования рекурсивной реализации.

S1	S2	Ожидаемый результат	Фактический результат
Пустая строка	Пустая строка	0	0
"abc"	"abc"	0	0
"abc"	"def"	3	3
"abcdef"	"abc"	3	3
"abc"	"abcdef"	3	3
"xyzabc"	"abcdef"	6	6
"abkdef"	"abcdefg"	2	2
"abc"	"acb"	2	2

В табл. 4.2 отражены результаты тестирования матричной реализации алгоритма поиска расстояния Левенштейна.

В табл. 4.3 отражены результаты тестирования матричной реализации алгоритма поиска расстояния Дамерау-Левенштейна.

Таблица 4.2 — Результаты тестирования матричной реализации.

S1	S2	Ожидаемый результат	Фактический результат
Пустая строка	Пустая строка	0	0
"abc"	"abc"	0	0
"abc"	"def"	3	3
"abcdef"	"abc"	3	3
"abc"	"abcdef"	3	3
"xyzabc"	"abcdef"	6	6
"abkdef"	"abcdefg"	2	2
"abc"	"acb"	2	2

Таблица 4.3 — Результаты тестирования матричной реализации Дамерау-Левенштейна.

S1	S2	Ожидаемый результат	Фактический результат
Пустая строка	Пустая строка	0	0
"abc"	"abc"	0	0
"abc"	"def"	3	3
"abcdef"	"abc"	3	3
"abc"	"abcdef"	3	3
"xyzabc"	"abcdef"	6	6
"abkdef"	"abcdefg"	2	2
"abc"	"acb"	1	1

4.3 Постановка эксперимента по замеру времени и памяти

Для матричных алгоритмов эксперимент ставился со следующими условиями:

— число повторов эксперимента - 100

— диапазоны сравниваемых строк: от 100 до 1000 с шагом 100

— все строки получены случайным образом.

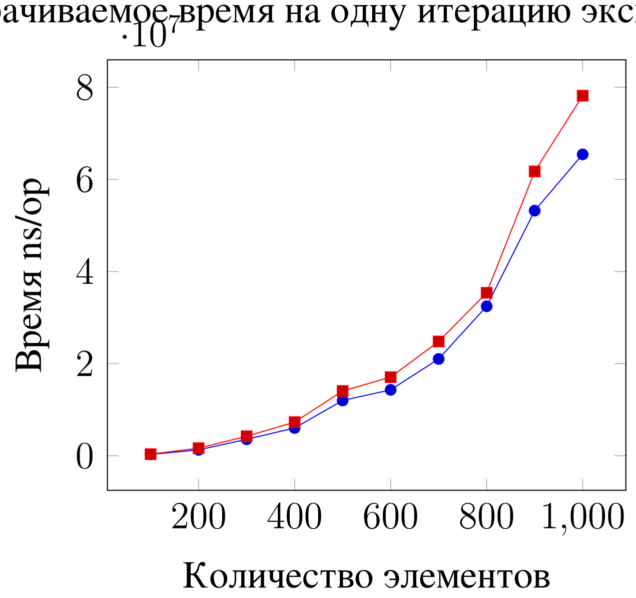
Для рекурсивного алгоритма время на одну итерацию достигает 1.75 мс, при длине строки в 12 символов. Аналогичный эксперимент для матричных реализаций занимает 4 мкс. Поэтому рекурсивная реализация была исключена из дальнейшего исследования.

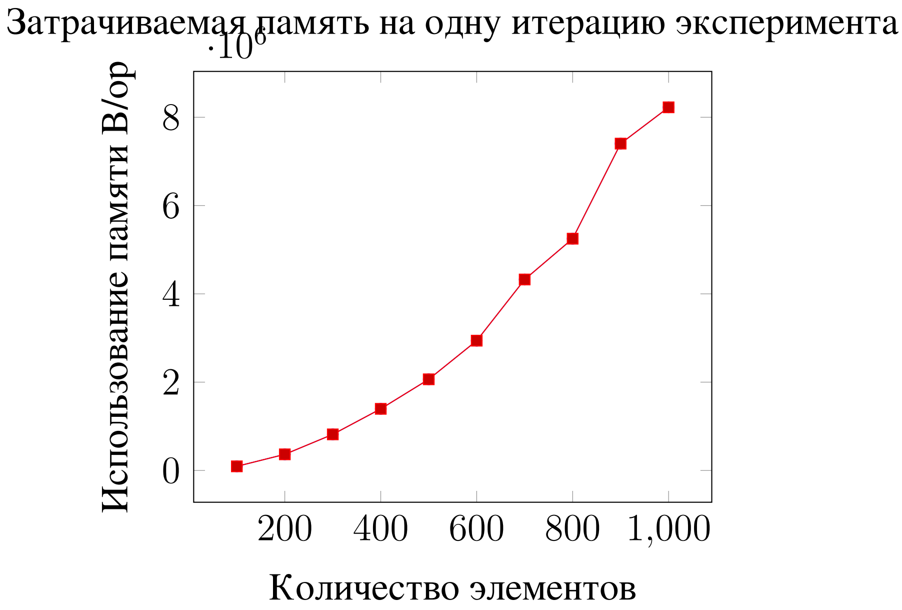
4.4 Сравнительный анализ на материале экспериментальных данных

В данном разделе будут приведены результаты экспериментов.

Далее приведены сравнительные характеристики матричных реализаций.

Затрачиваемое время на одну итерацию эксперимента





4.5 Выводы

Использование памяти матричными реализациями алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна примерно одинаково, поскольку размерность матрицы одна и та же.

По времени матричный алгоритм поиска расстояния Дамерау-Левенштейна медленнее матричного алгоритма Левенштейна за счёт дополнительного условия на штраф за транспозицию.

ЗАКЛЮЧЕНИЕ

В результате работы был изучен метод динамического программирования на материала алгоритмов Левенштейна и Дамерау-Левенштейна.

Была разработана программа реализующая алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна и получены экспериментальные данные сравнения этих алгоритмов по времени и памяти.