МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Кривозубов Влад, ИУ7-53Б

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

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове(поисковая строка браузера)
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

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

Задачами данной лабораторной 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 двух 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

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

1. D (англ. delete) — удалить,
2. I (англ. insert) — вставить,
3. R (replace) — заменить,
4. M(match) - совпадение.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна 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:

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

1.0.1 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2 Конструкторская часть

Требования к программе:

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными
3. Две пустые строки - корректный ввод, программа не должна аварийно завершаться
4. Для всех алгоритмов выводиться процессорное время исполнения
5. Для всех алгоритмов кроме Левенштейна с рекурсивной реализацией должна выводиться матрица

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

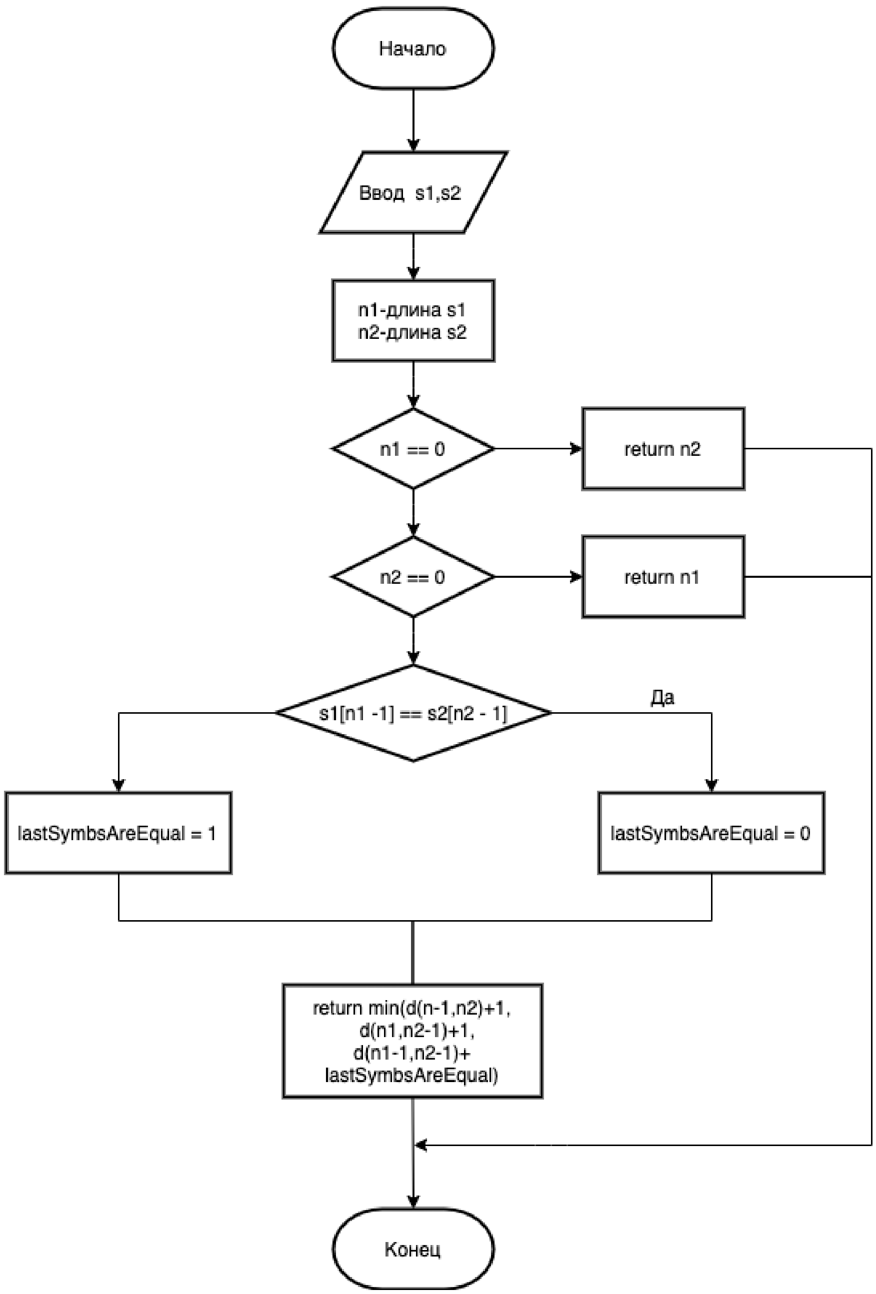


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

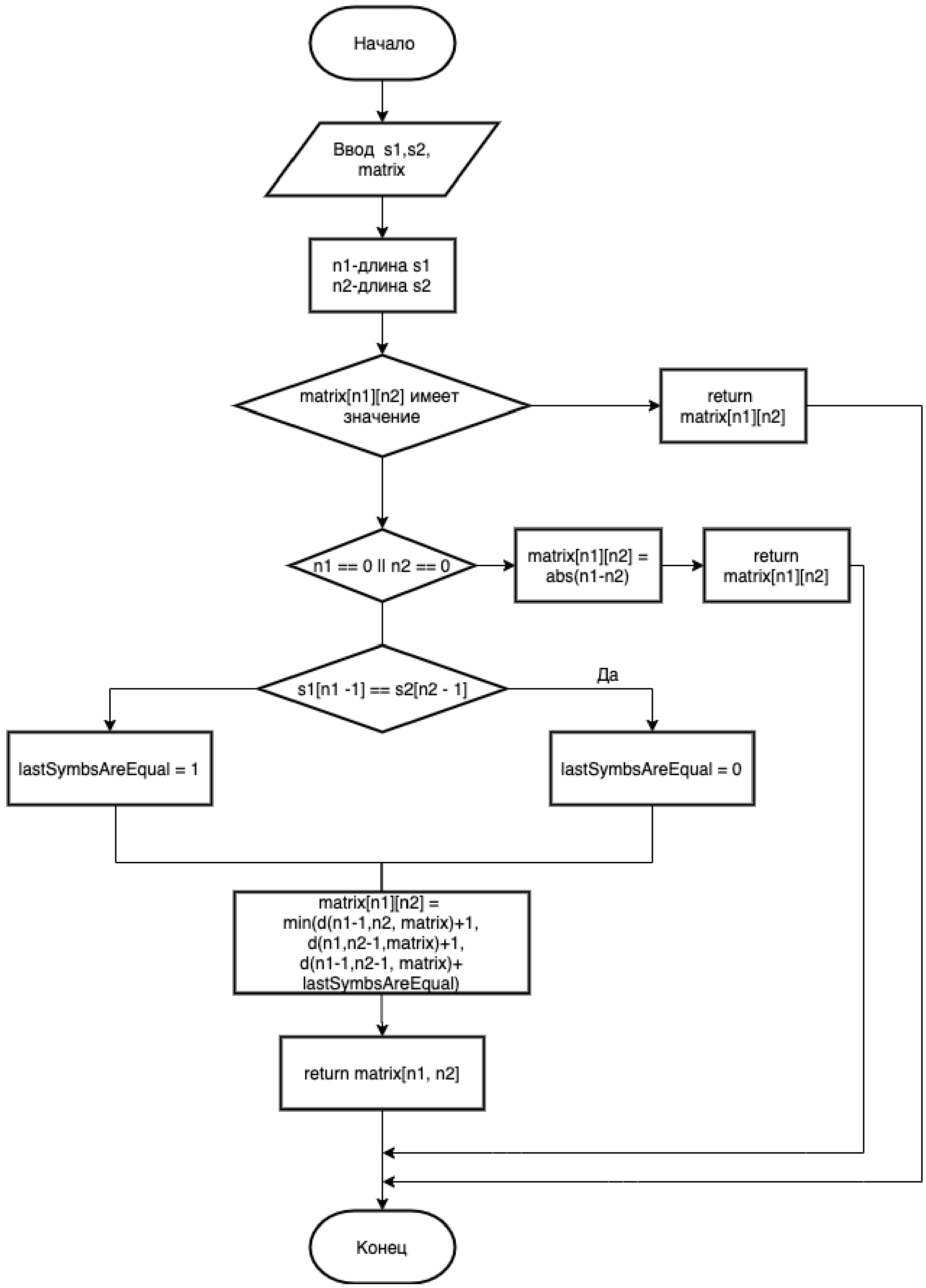


Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с матричной оптимизацией

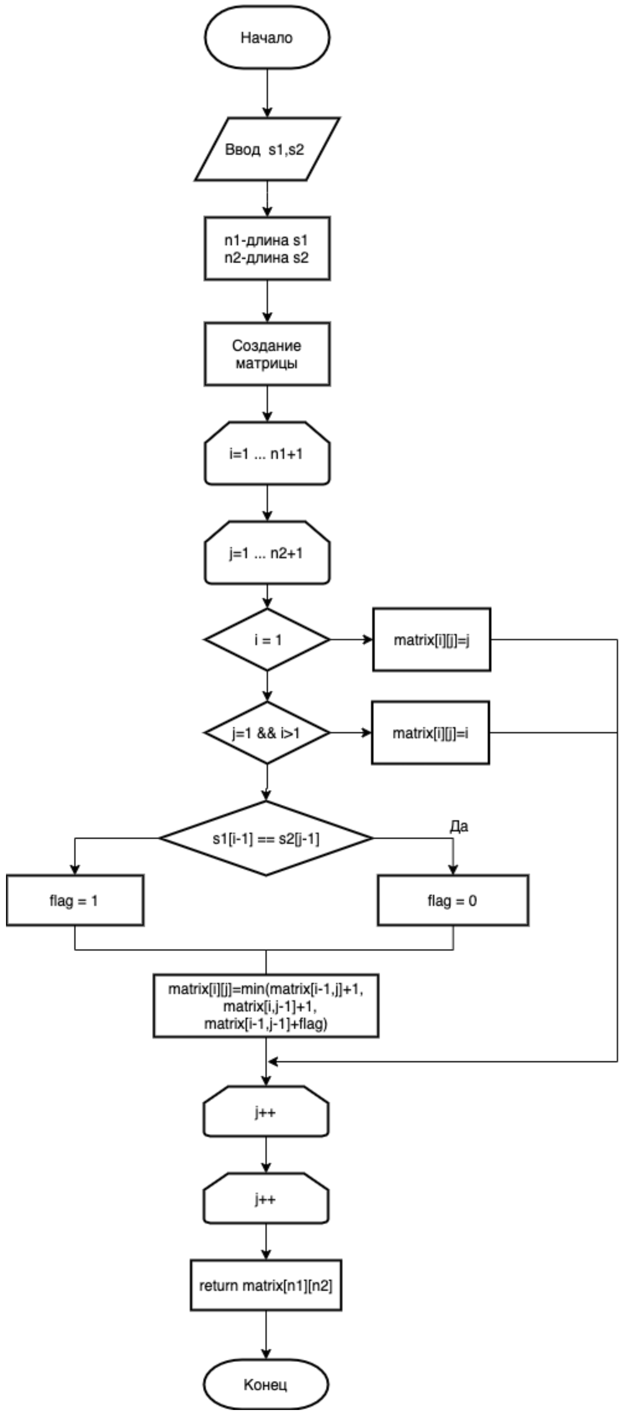


Рис. 2.3: Схема матричного алгоритма нахождения расстояния Левенштейна

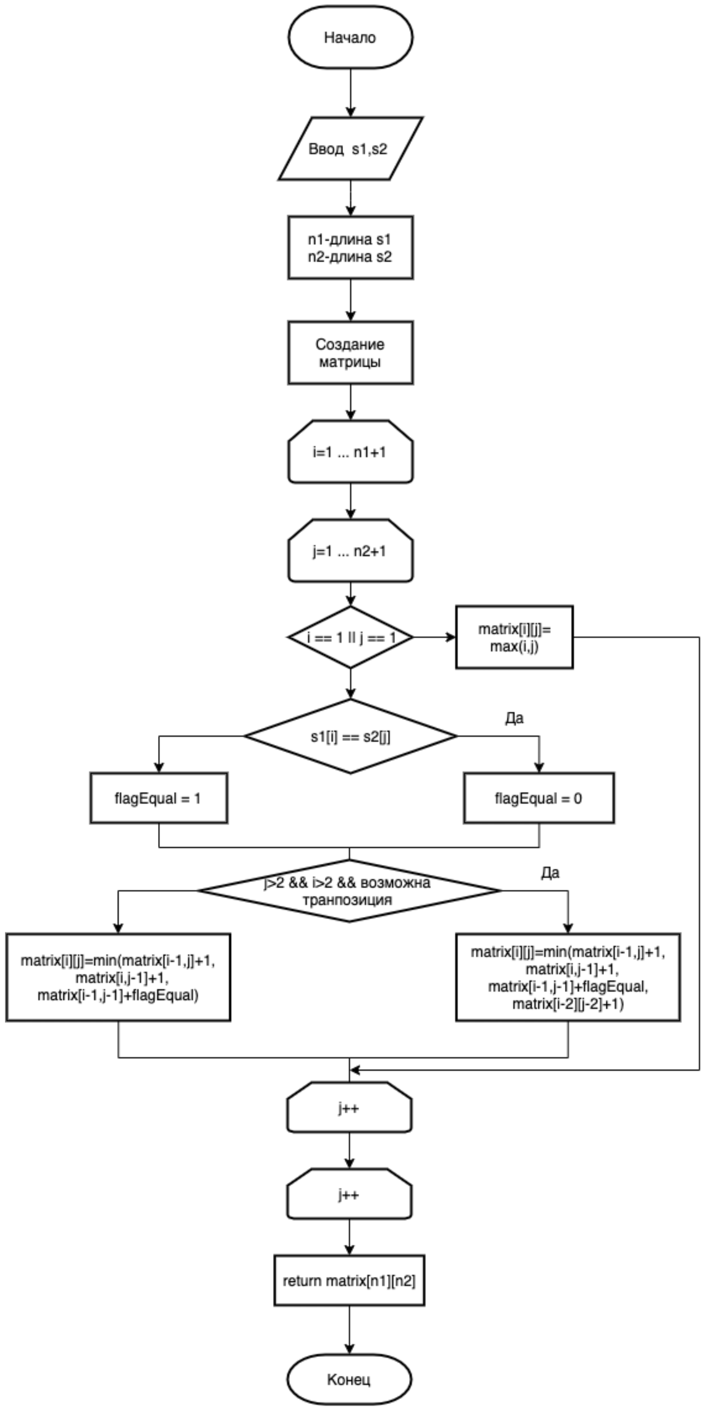


Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран язык программирования Swift в связи с потребностью практики разработки на нем. Среда разработки - VS Code.

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
func recursiveLevenstein(_ strA: String, _ strB: String) ->
    Int {
  if strA.count == 0 || strB.count == 0 {
    return abs(strA.count - strB.count)
  }

  let lastSymbsAreEqual = (strA.last == strB.last) ? 0 : 1

  return min(recursiveLevenstein(strA, removeLastSymb(strB)
     ) + 1,
  recursiveLevenstein(removeLastSymb(strA), strB) + 1,
  recursiveLevenstein(removeLastSymb(strA), removeLastSymb(
     strB)) + lastSymbsAreEqual)
}
```

Листинг 3.2: Функция удаления последнего символа в строке

```
func removeLastSymb(_ str: String) -> String {
```

```
  return String(str[str.startIndex..<str.index(before: str.
      endIndex)])
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна рекурсивно с матрицей

```
func recursiveOptimizedLevenstein(_ strA: String, _ strB:
    String, _ matrix: inout Array<Array<Int?>>) -> Int {
  guard matrix[strA.count][strB.count] == nil else { return
      matrix[strA.count][strB.count]! }

  if strA.count == 0 || strB.count == 0 {
    matrix[strA.count][strB.count] = abs(strA.count - strB.
        count)
    return matrix[strA.count][strB.count]!
  }

  let lastSymbsAreEqual = (strA.last == strB.last) ? 0 : 1

  matrix[strA.count][strB.count] = min(
      recursiveOptimizedLevenstein(strA, removeLastSymb(strB
      ), &matrix) + 1,
  recursiveOptimizedLevenstein(removeLastSymb(strA), strB,
      &matrix) + 1,
  recursiveOptimizedLevenstein(removeLastSymb(strA),
      removeLastSymb(strB), &matrix) + lastSymbsAreEqual)

  return matrix[strA.count][strB.count]!
}
```

Листинг 3.4: Функция нахождения расстояния Левенштейна матрично

```
func matrixLevenstein(_ strA: String, _ strB: String, _
    matrix: inout Array<Array<Int?>>) -> Int {
  for i in 0...strA.count {
    for j in 0...strB.count {
      if i == 0 {
        matrix[i][j] = j
      }
      else if j == 0 && i > 0 {
```

```
        matrix[i][j] = i
      }
      else {
        let flag = (strA[strA.index(strA.startIndex,
           offsetBy: i - 1)] == strB[strB.index(strB.
           startIndex, offsetBy: j - 1)]) ? 0 : 1
        matrix[i][j] = min(matrix[i][j - 1]! + 1,
        matrix[i - 1][j]! + 1,
        matrix[i - 1][j - 1]! + flag)
      }
    }
  }

  return matrix[strA.count][strB.count]!
}
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
func matrixLamerauLevenstein(_ strA: String, _ strB: String
   , _ matrix: inout Array<Array<Int?>>) -> Int {
  for i in 0...strA.count {
    for j in 0...strB.count {
      if i == 0 || j == 0 {
        matrix[i][j] = max(i, j)
      }
      else {
        let flagEqual = (strA[strA.index(strA.startIndex,
           offsetBy: i - 1)] == strB[strB.index(strB.
           startIndex, offsetBy: j - 1)]) ? 0 : 1

        if j > 1 && i > 1 && isTransposition(strA, strB, i,
            j) {
          matrix[i][j] = min(matrix[i][j - 1]! + 1,
          matrix[i - 1][j]! + 1,
          matrix[i - 1][j - 1]! + flagEqual,
          matrix[i - 2][j - 2]! + 1)
        }
        else {
          matrix[i][j] = min(matrix[i][j - 1]! + 1,
          matrix[i - 1][j]! + 1,
```

```
                matrix[i - 1][j - 1]! + flagEqual)
            }
        }
    }
}

return matrix[strA.count][strB.count]!
}
```

Листинг 3.6: Функция проверки траннозиции

```
func isTransposition(_ strA: String, _ strB: String, _ i: Int, _ j: Int) -> Bool {
    return (strA[strA.index(strA.startIndex, offsetBy: i - 1)] == strB[strB.index(strB.startIndex, offsetBy: j - 2)] &&
    strA[strA.index(strA.startIndex, offsetBy: i - 2)] == strB[strB.index(strB.startIndex, offsetBy: j - 1)])
}
```

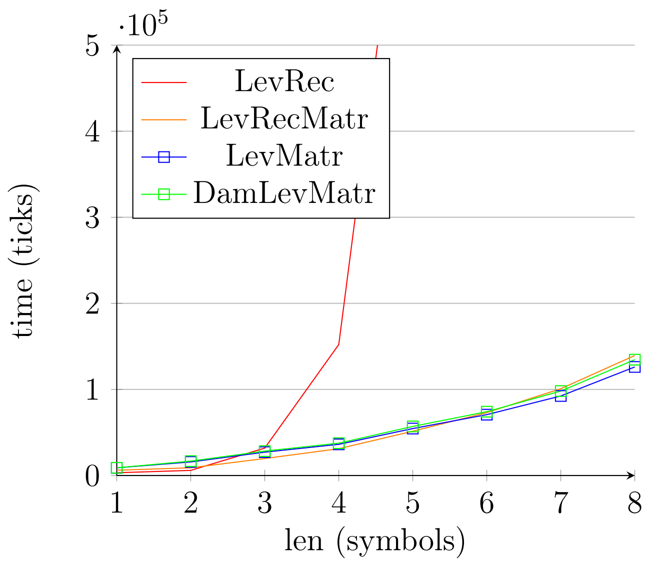
4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.

Таблица 4.1: Процессорное время исполнения

len	Lev(R)	Lev(RM)	Lev(M)	DamLev(M)
1	3202	6136	8920	8938
2	5967	8967	15781	16726
3	31863	19746	27140	28338
4	152154	31130	36210	37542
5	817255	51472	54552	57048
6	4384089	73043	70704	74147
7	22989380	100836	92401	98123
8	124300785	139443	126136	134602



Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов. Показатели рекурсивного алгоритма Левенштейна при увеличении размера слов резко ухудшаются. Это связано с большим количеством повторов. Рекурсивная версия с матрицей не теряет своей производительности так как там исключаются повторы. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также видно, что алгоритм Дамерау-Левенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы примерно одинаковы по временной эффективности.

4.2 Тесты

Проведем тестирование программы. В столбцах "Ожидание"и "Результат"4 числа соответсвуют рекурсивному алгоритму нахождения расстояния Левенштейна, рекурсивно-матричному алгоритму нахождения расстояния Левенштейна, матричному алгоритму расстояния Левенштейна, матричному алгоритму нахождения расстояния Дамерау-Левенштейна.

Таблица 4.2: Таблица тестовых данных

№	Слово №1	Слово №2	Ожидание	Результат
1			0 0 0 0	0 0 0 0
2	1234	2143	3 3 3 2	3 3 3 2
3	amm	add	2 2 2 2	2 2 2 2
4	error	dog	4 4 4 4	4 4 4 4
5		submission	10 10 10 10	10 10 10 10
6	mochombo	0	8 8 8 8	8 8 8 8
7	dfufdfd	fddfdffdd	5 5 5 4	5 5 5 4

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных 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 исследований делается вывод о том, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.