**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 дисциплине "Анализ алгоритмов"

**Тема** Расстояние Левенштейна и Дамерау-Левенштейна

**Студент** Малышев И. А.

**Группа** ИУ7-51Б

**Оценка (баллы)** _________________

**Преподаватель:** Волкова Л. Л.

Москва — 2021 г.

# Оглавление


**Введение** **2**

**1 Аналитическая часть** **3**

1.1 Расстояние Левенштейна . . . 3

1.1.1 Рекусивный алгоритм нахождения расстояния Левенштейна . . . 3

1.1.2 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы 3

1.1.3 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы . . . 4

1.2 Расстояние Дамерау-Левенштейна . . . 4

1.2.1 Рекусивный алгоритм нахождения расстояния Дамерау-Левенштейна . . . 4

1.2.2 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы 4

1.3 Вывод . . . 4

**2 Конструкторская часть** **5**

2.1 Схемы алгоритмов . . . 5

2.2 Вывод . . . 10

**3 Технологическая часть** **11**

3.1 Средства реализации . . . 11

3.2 Реализация алгоритмов . . . 11

3.3 Тестирование . . . 14

3.4 Вывод . . . 15

**4 Исследовательская часть** **16**

4.1 Технические характеристики . . . 16

4.2 Время выполнения реализаций алгоритмов . . . 16

4.3 Оценка затрачиваемой памяти . . . 17

**Заключение** **18**

**Литература** **19**

# Введение

Динамическое программирование - это форма вычислений, при которой следующий член вычисляется на основе предыдущего. Простейшим примером применения является вычисление чисел Фибоначчи. Кроме того, динамическое программирование может применяться и в более сложных задачах таких, как проебразование строк из одной в другую. В этом случае задача сводится к вычислению расстояния Левенштейна (редакционного расстояния) - минимального количества операций вставки, удаления символа или замены символа один на другой, необходимых для преобразования одной строки в другую. Расстояние Левенштейна применяется в следующих областях:

- компьютерной лингвистике для устранения ошибок в набираемом тексте;
- в бионформатике для сравнения генов.

Поэтому **целью** данной работы является получение навыка динамического программирования на примере реализации алгоритмов редакционного расстояния.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Изучить алгоритмы расчета редакционного расстояния;
2. Реализовать алгоритмы подсчета редакционного расстояния;
3. Протестировать реализованные алгоритмы;
4. Провести сравнительный анализ реализаций алгоритмов по затраченному процессорному времени и памяти.

# 1 Аналитическая часть

В данном разделе определяются расстояния Левенштейна и Дамерау-Левенштейна, а также рассматриваются различные алгоритмы и их модификации для вычисления указанных расстояний.

## 1.1 Расстояние Левенштейна

Для вычисления редакционного расстояния вводятся следующие цены операций:

- замена одного символа на другой - 1;
- вставка символа - 1;
- удаление символа - 1;
- совпадение - 0.

С учётом этого вводится рекурсивная формула для вычисления расстояния Левенштейна:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{i = 0, j = 0} \\ i, & \text{i > 0, j = 0} \\ j, & \text{i = 0, j > 0} \\ \min\{ & \\ \quad D(s1[1..i], s2[1..j-1]) + 1 & \\ \quad D(s1[1..i-1], s2[1..j]) + 1 & \text{i > 0, j > 0} \\ \quad D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, & \text{s1[i] = s2[j]} \\ 1, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.1}$$

### 1.1.1 Рекусивный алгоритм нахождения расстояния Левенштейна

Данный алгоритм использует для решения формулу 1.1, однако в отличие от прерыдущих является рекурсивным, а значит, для хранения промежуточных результатов используется стек. Кроме того, при этом подходе возникает проблема повторных вычислений, так как функция $D(s1[1..i], s2[1..j])$ будет выполняться несколько раз в разных ветвях дерева вызовов.

### 1.1.2 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы

Данный алгоритм использует для решения задачи матрицу размером $(m+1)*(n+1)$, где $m$ и $n$ - длины двух строк, одну из которых необходимо преобразовать к другой. На каждом шаге работы алгоритма заполняется одна клетка матрицы в соответствии с формулой 1.1. По окончании алгоритма результат будет находиться в последней заполненной клетке.

### 1.1.3 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы

Данный алгоритм является модификацией предыдущего. Очеивдно, что на кажом шаге алгоритма используются значения из текущей и предыдущей строки матрицы, поэтому достаточно хранить только их, а не всю матрицу.

## 1.2 Расстояние Дамерау-Левенштейна

Дамерау дополнил определение расстояния Левенштейна еще одной операцией, а именно операцией перестановки двух букв местами, стоимость которой тоже равна 1. Расстояние Дамерау-Левенштейна вычисляется по следующей формуле:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{i = 0, j = 0} \\ i, & \text{i > 0, j = 0} \\ j, & \text{i = 0, j > 0} \\ \min\{ & \\ \quad D(s1[1..i], s2[1..j-1]) + 1 & \\ \quad D(s1[1..i-1], s2[1..j]) + 1 & \text{i > 0, j > 0} \\ \quad D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, & \text{s1[i] = s2[j]} \\ 1, & \text{иначе} \end{cases} & \\ \left[ \begin{array}{ll} D(s1[1..i-2], s2[1..j-2]) + 1, & \text{если i>1, j>1, s1[i] = s2[j-1], s1[i-1] = s2[j]} \\ \text{inf}, & \text{иначе} \end{array} \right. & \\ \} & \end{cases} \tag{1.2}$$

### 1.2.1 Рекусивный алгоритм нахождения расстояния Дамерау-Левенштейна

Данный алгоритм использует для решения формулу 1.2 и является рекурсивным, а значит, для хранения промежуточных результатов используется стек. Кроме того, при этом подходе возникает проблема повторных вычислений, так как функция $D(s1[1..i], s2[1..j])$ будет выполняться несколько раз в разных ветвях дерева.

### 1.2.2 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы

Данный алгоритм решает проблему повторных вычислений простого рекурсивного алгоритма. При данном подходе вводится матрица размером $(m + 1) * (n + 1)$, содержащая уже вычисленные промежуточные результаты. На каждом шаге работы алгоритма заполняется одна клетка матрицы в соответствии с формулой 1.2. По окончании алгоритма результат будет находиться в последней заполненной клетке.

## 1.3 Вывод

В данном разделе были даны определения расстояний Левенштейна и Дамерау-Левенштейна, а также рассмотрены 5 алгоритмов вычисления указанных расстояний.

# 2 Конструкторская часть

В данном разделе разрабатываются схемы алгоритмов на основе их описания, приведённого в аналитическом разделе.

## 2.1 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3, 2.4 и 2.5 показаны схемы алгоритма рекурсивного Левенштейна, нерекурсивного алгоритма Левенштейна с кэшем в виде матрицы и двух строк матрицы, рекурсивного алгоритма Дамерау-Левенштейна и нерекурсивного алгоритма с кэшем в виде матрицы соответственно.

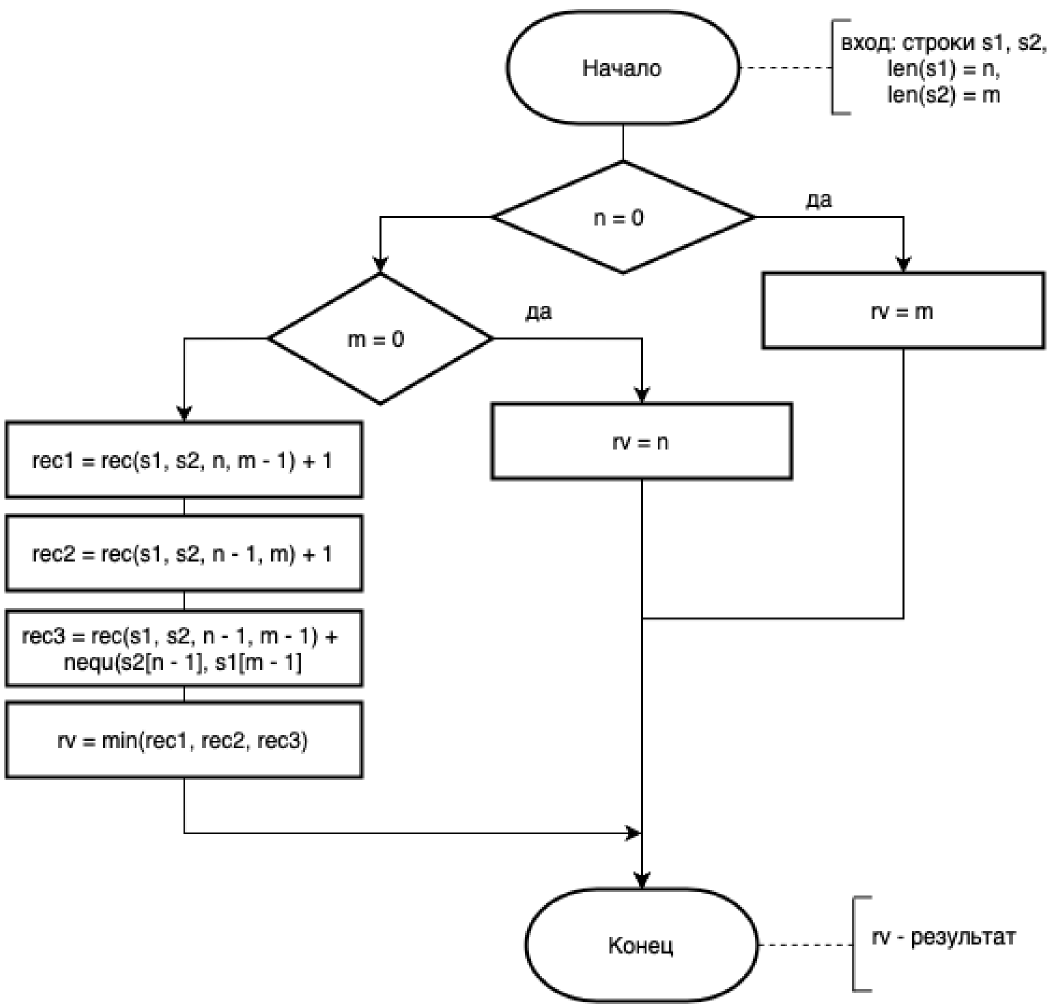


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

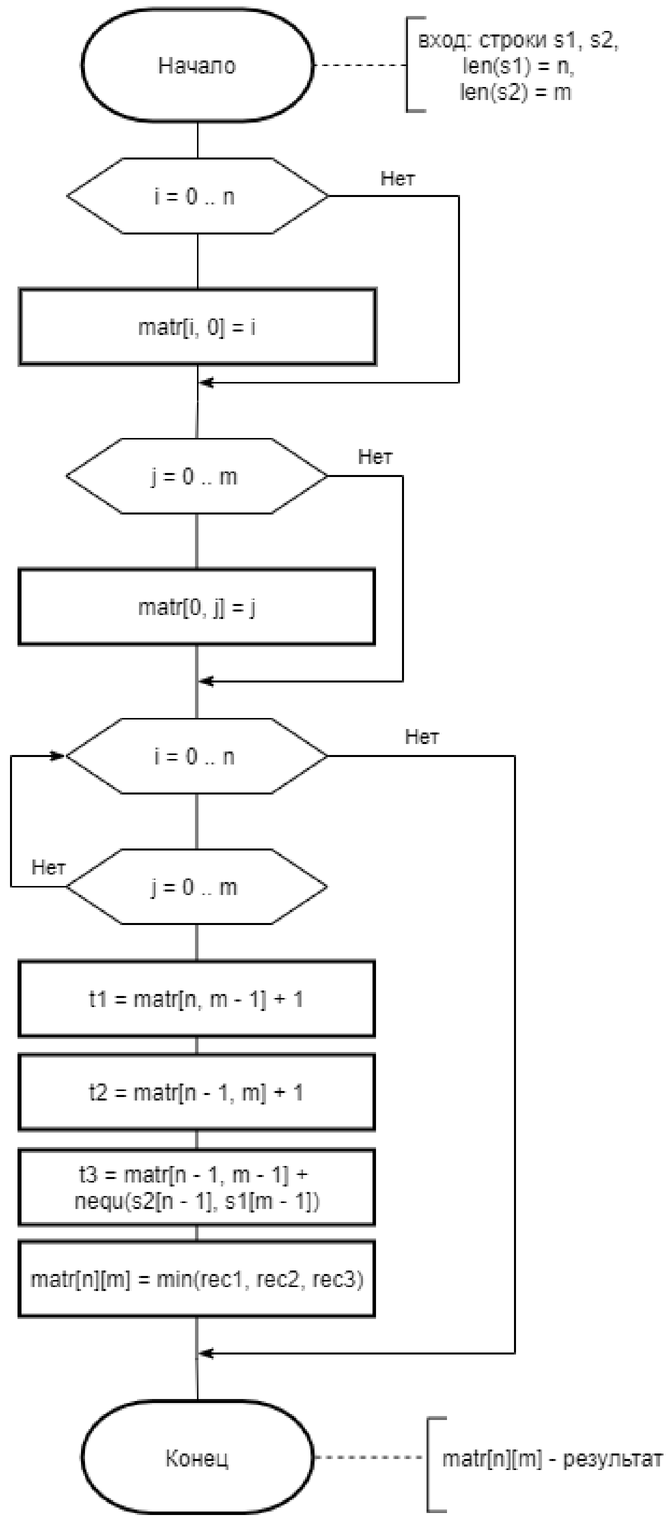


Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшем в виде матрицы

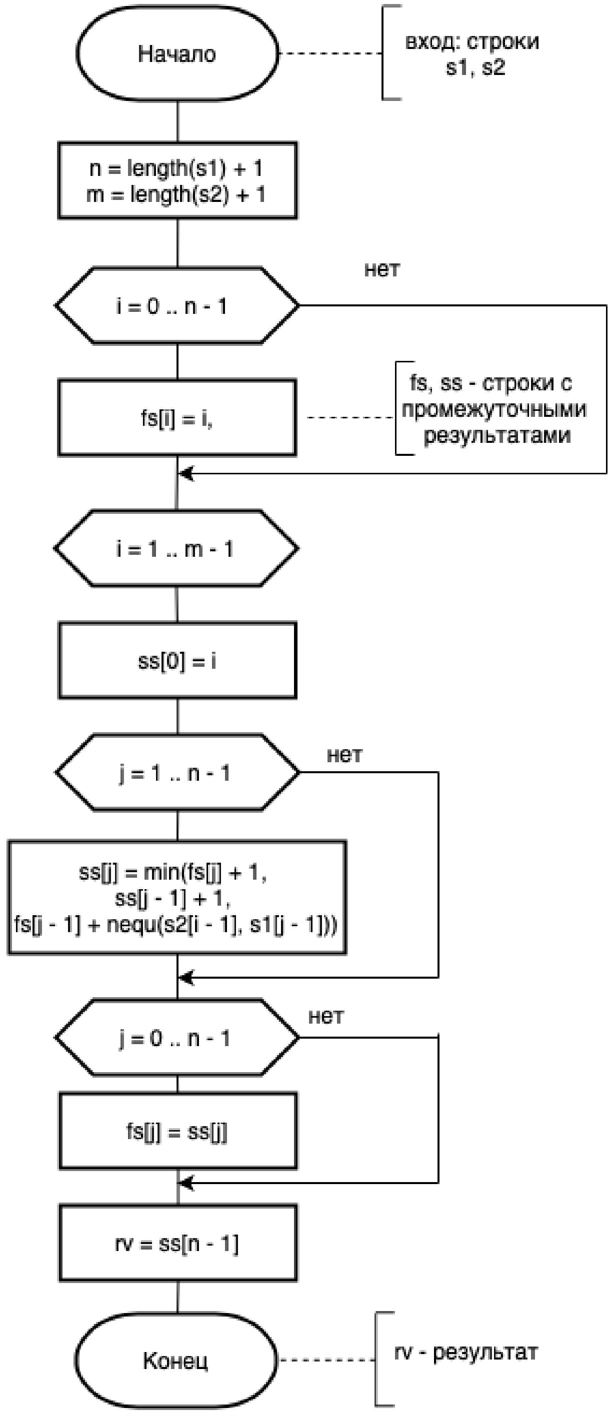


Рис. 2.3: Схема нерекурсивного алгоритма нахождения расстояния Левенштейна с кэшем в виде двух строк матрицы

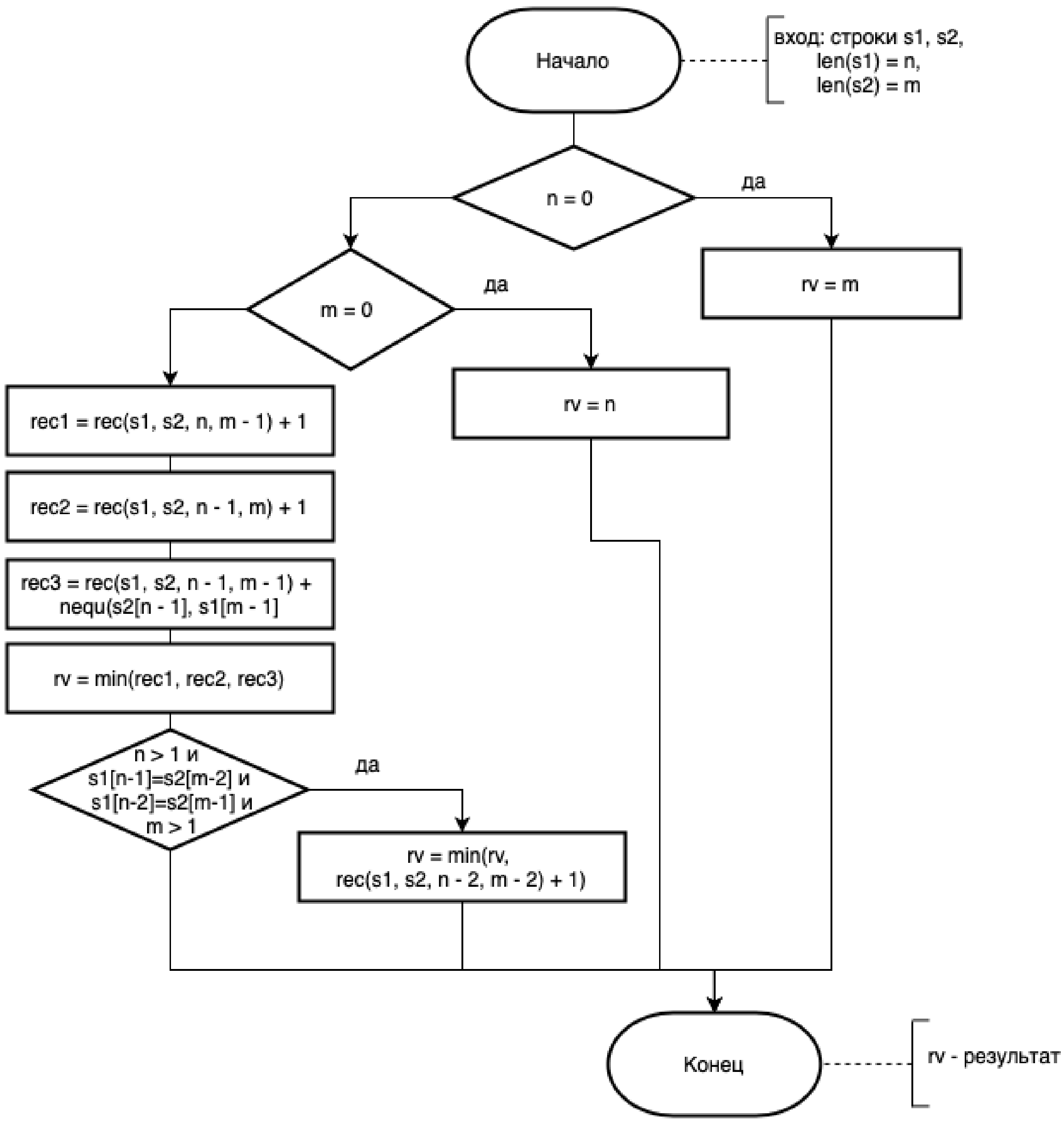


Рис. 2.4: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

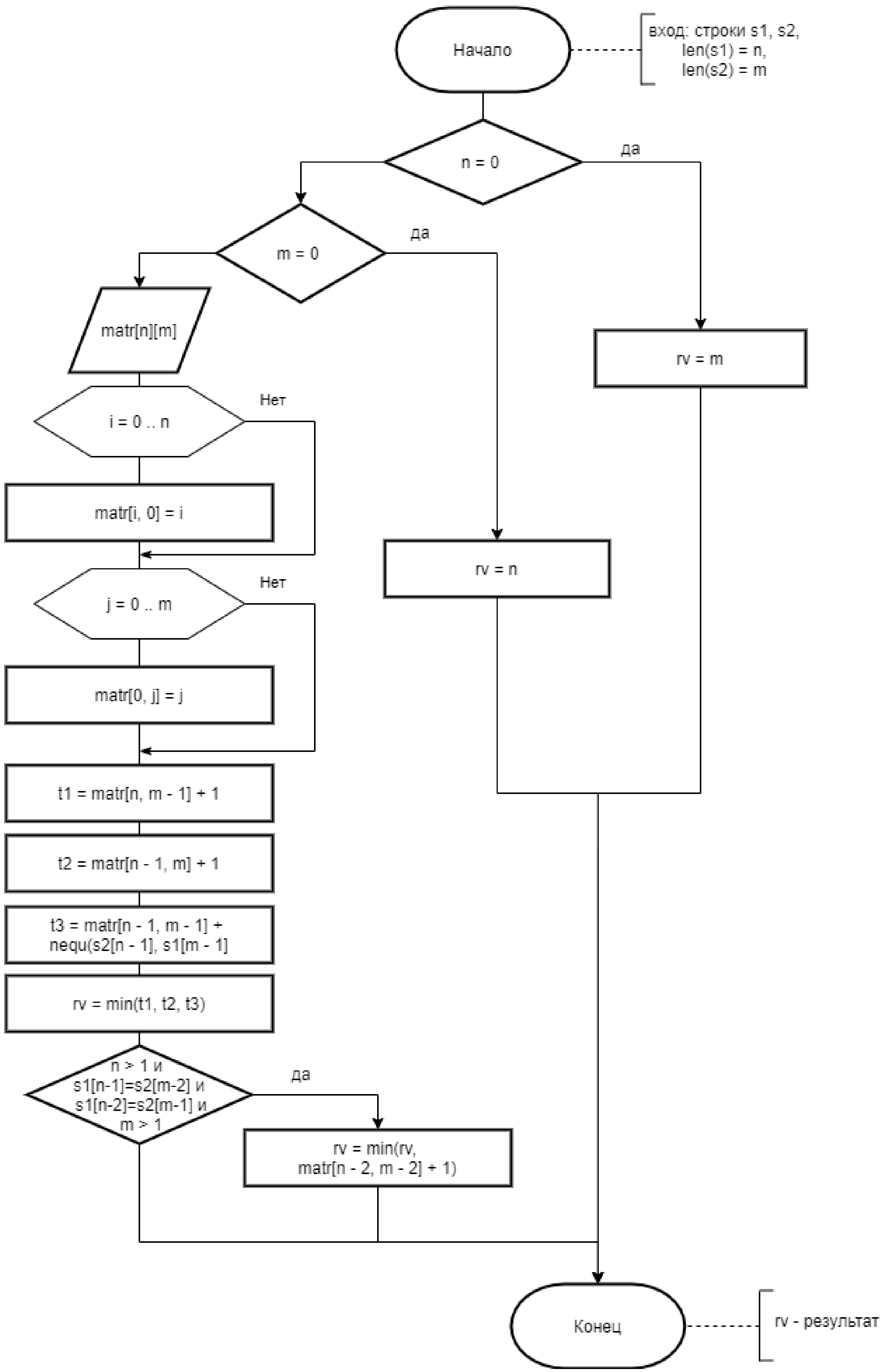


Рис. 2.5: Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшем в виде матрицы

## 2.2 Вывод

В данном разделе были построены схемы пяти алгоритмов нахождения редакционного расстояния на основе их описания, приведённого в аналитической части.

# 3 Технологическая часть

В данном разделе приводится реализация алгоритмов, схемы которых были разработаны в конструкторской части. Кроме того, обосновывается выбор технологического стека и проводится тестирование реализованных алгоритмов.

## 3.1 Средства реализации

В качестве языка программирования был выбран C#, а среду разработки – Visual Studio, т. к. я знаком с данным языком и имею представление о тестировании программ в данном языке. Время работы алгоритмов было замерено с помощью библиотеки System.Diagnostics, класса Stopwatch, который имеет методы для расчёта процессорного времени [1].

## 3.2 Реализация алгоритмов

В листингах 3.1 - 3.5 приведена реализации алгоритмов, описанных в 2.1.

Листинг 3.1: Функция для рекурсивного нахождения расстояния Левенштейна

```
static int _LevDist(string source, int srclen, string target, int trglen)
{
  if (srclen * trglen == 0)
    return Math.Max(srclen, trglen);

  int substitutionCost = 0;
  if (source[srclen - 1] != target[trglen - 1])
    substitutionCost = 1;

  int deletion = _LevDist(source, srclen - 1, target, trglen) + 1;
  int insertion = _LevDist(source, srclen, target, trglen - 1) + 1;
  int substitution = _LevDist(source, srclen - 1, target, trglen - 1) +
     substitutionCost;

  return Minimum(deletion, insertion, substitution);
}

static int LevDistRec(string source, string target) =>
  _LevDist(source, source.Length, target, target.Length);
```

Листинг 3.2: Функция для нерекурсивного нахождения расстояния Левенштейна с кэшем в виде матрицы

```
static int LevDistMatr(string source, string target)
{
  if (source.Length * target.Length == 0)
    return Math.Max(target.Length, source.Length);

  int n = source.Length + 1;
  int m = target.Length + 1;
  int[,] matrixD = new int[n, m];

  const int deletionCost = 1;
  const int insertionCost = 1;

  for (int i = 0; i < n; i++)
    matrixD[i, 0] = i;

  for (int j = 0; j < m; j++)
    matrixD[0, j] = j;

  for (int i = 1; i < n; i++)
  {
    for (int j = 1; j < m; j++)
    {
      int substitutionCost = source[i - 1] == target[j - 1] ? 0 : 1;

      matrixD[i, j] = Minimum(matrixD[i - 1, j] + deletionCost,
      matrixD[i, j - 1] + insertionCost,
      matrixD[i - 1, j - 1] + substitutionCost);
    }
  }

  return matrixD[n - 1, m - 1];
}
```

Листинг 3.3: Функция для нерекурсивного нахождения расстояния Левенштейна с кэшем в виде двух строк матрицы

```
static int LevDistTwoRows(string source, string target)
{
  if (source.Length * target.Length == 0)
    return Math.Max(target.Length, source.Length);

  int m = target.Length;
  int n = source.Length;
  int[,] distance = new int[2, m + 1];

  for (int j = 1; j <= m; j++)
    distance[0, j] = j;

  int currentRow = 0;
  for (int i = 1; i <= n; ++i)
  {
    currentRow = i % 2;
    distance[currentRow, 0] = i;
    int previousRow = (currentRow + 1) % 2;
    for (int j = 1; j <= m; j++)
    {
      int cost = target[j - 1] == source[i - 1] ? 0 : 1;
      distance[currentRow, j] = Minimum(distance[previousRow, j] + 1,
      distance[currentRow, j - 1] + 1,
      distance[previousRow, j - 1] + cost);
    }
  }
  return distance[currentRow, m];
}
```

Листинг 3.4: Функция для рекурсивного нахождения расстояния Дамерау-Левенштейна

```
static int _DamLevDist(string source, int srclen, string target, int trglen)
{
  if (srclen * trglen == 0)
    return Math.Max(srclen, trglen);

  int deletion = _DamLevDist(source, srclen - 1, target, trglen) + 1;
  int insertion = _DamLevDist(source, srclen, target, trglen - 1) + 1;
  int substitution = _DamLevDist(source, srclen - 1, target, trglen - 1) +
     (source[srclen - 1] != target[trglen - 1] ? 1 : 0);

  int min = Minimum(deletion, insertion, substitution);

  if (srclen > 1 && trglen > 1 && source[srclen - 1] == target[trglen - 2] &&
     source[srclen - 2] == target[trglen - 1])
    min = Math.Min(min, _DamLevDist(source, srclen - 2, target, trglen - 2) + 1);

  return min;
}

static int DamLevDistRec(string source, string target) =>
  _DamLevDist(source, source.Length, target, target.Length);
```

Листинг 3.5: Функция для нерекурсивного нахождения расстояния Дамерау-Левенштейна с кэшем в виде матрицы

```
static int DamLevDistMatr(string source, string target)
{
  if (source.Length * target.Length == 0)
    return Math.Max(target.Length, source.Length);

  int n = source.Length + 1;
  int m = target.Length + 1;
  int[,] arrayD = new int[n, m];

  for (int i = 0; i < n; i++)
    arrayD[i, 0] = i;

  for (int j = 0; j < m; j++)
    arrayD[0, j] = j;

  for (int i = 1; i < n; i++)
  {
    for (int j = 1; j < m; j++)
    {
      int cost = source[i - 1] == target[j - 1] ? 0 : 1;

      arrayD[i, j] = Minimum(arrayD[i - 1, j] + 1,
      arrayD[i, j - 1] + 1,
      arrayD[i - 1, j - 1] + cost);

      if (i > 1 && j > 1 && source[i - 1] == target[j - 2] && source[i - 2] == target[j
         - 1])
        arrayD[i, j] = Math.Min(arrayD[i, j], arrayD[i - 2, j - 2] + cost);
    }
  }

  return arrayD[n - 1, m - 1];
}
```

## 3.3 Тестирование

В таблицах 3.1 и 3.2 приведены тесты для функций нахождения редакционного расстояния.

Таблица 3.1: Тестирование алгоритмов нахождения расстояния Левенштейна

| Входные строки | Результат | Ожидаемый результат |
|---|---|---|
| $ckat, kot$ | 2 | 2 |
| $abc, defg$ | 4 | 4 |
| $abcd, abcd$ | 0 | 0 |

Таблица 3.2: Тестирование алгоритмов нахождения расстояния Дамерау-Левенштейна

| Входные строки | Результат | Ожидаемый результат |
|---|---|---|
| $ckat, kot$ | 2 | 2 |
| $abc, defg$ | 4 | 4 |
| $abcd, abcd$ | 0 | 0 |
| $abcd, badc$ | 2 | 2 |

Все тесты пройдены успешно.

## 3.4 Вывод

В данном разделе были реализованы 5 алгоритмов нахождения редакционного расстояния с помощью выбранных средств разработки. Кроме того, реализованные алгоритмы были протестированы.

# 4 Исследовательская часть

В данном разделе проводится сравнительный анализ реализованных алгоритмов по процессорному времени и по затрачиваемой памяти.

## 4.1 Технические характеристики

Все нижепреведенные замеры времени проведены на процессоре: Intel Core i7, 4 GHz, 4-ядерный.

## 4.2 Время выполнения реализаций алгоритмов

Для сравнительного анализа времени выполнения реализаций алгоритмов проведен эксперимент. Для замеров были сформированы строки, с суммарной длиной, варьирующейся от 6 до 24 включительно с шагом 2.

Время измерялось 1000 раз для каждой пары строк, после усреднялось. Время на графике (рис. 4.1) представлено в микросекундах.

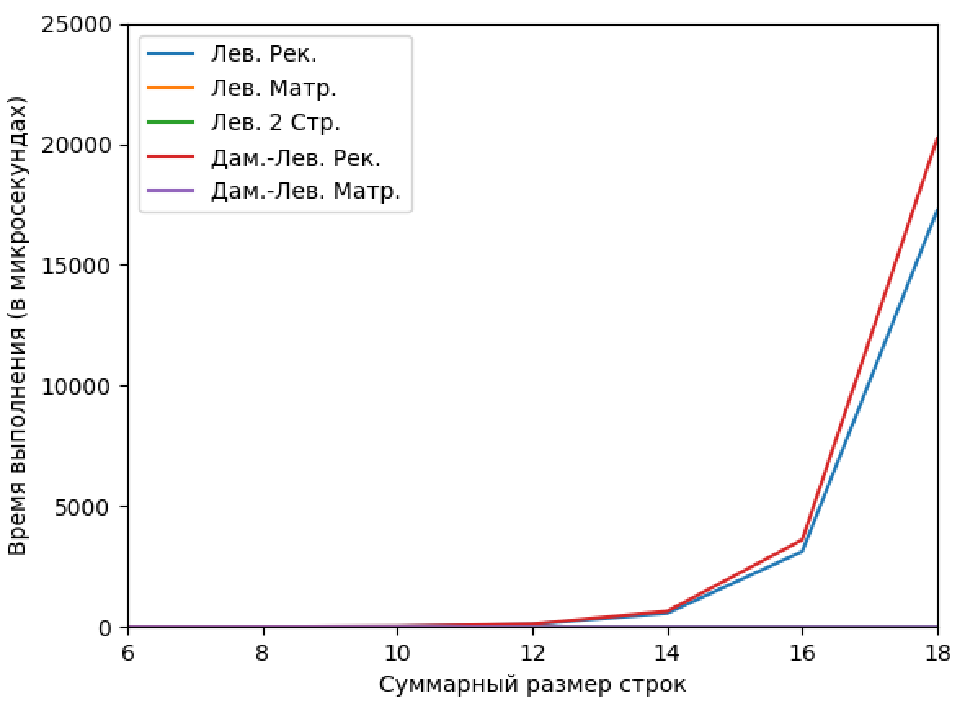


Рис. 4.1: Зависимость времени от суммарной длины строк

Время в микросекундах для всех реализаций представлено в таблице 4.1.

Таблица 4.1: Зависимость затрачиваемого процессорного времени от суммарной длины строк для алгоритмов

| Суммарная длина строк | Лев. Рек. | Лев. Матр. | Лев. 2 Стр. | Дам.-Лев. Рек. | Дам.-Лев. Матр. |
|---|---|---|---|---|---|
| 6 | 1.31 | 0.30 | 0.28 | 0.79 | 0.31 |
| 8 | 3.79 | 0.41 | 0.38 | 4.34 | 0.47 |
| 10 | 19.92 | 0.66 | 0.63 | 23.21 | 0.66 |
| 12 | 106.72 | 0.82 | 0.81 | 122.97 | 1.02 |
| 14 | 571.67 | 1.13 | 1.06 | 654.56 | 1.27 |
| 16 | 3128.63 | 1.50 | 1.30 | 3609.66 | 1.76 |
| 18 | 17262.44 | 1.73 | 1.59 | 20235.26 | 2.43 |

## 4.3 Оценка затрачиваемой памяти

Алгоритмы вычисления расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти вычисляется по формуле (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 2 \cdot \mathcal{C}(\text{int}) + \mathcal{C}(\text{bool})), \tag{4.1}$$

где $\mathcal{C}$ — оператор вычисления размера, $S_1$, $S_2$ — строки, int — целочисленный тип, string — строковый тип, bool - логический тип.

Использование памяти при итеративной реализации теоретически вычисляется по формуле (4.2).

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 5 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string})) \tag{4.2}$$

## Вывод

Рекурсивный алгоритм вычисления расстояния Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 9 символов, матричная реализация алгоритма вычисления расстояния Левенштейна превосходит по времени работы рекурсивную почти в 10000 раз. Версия с двумя строками работает немного быстрее матричной реализации.

Алгоритм вычисления расстояния Дамерау — Левенштейна используется для решения других задач, поэтому говорить о его отставании от алгоритма вычисления расстояния Левенштейна, исходя из временных затрат, некорректно.

По расходу памяти алгоритмы с использованием матрицы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

# Заключение

В рамках данной лабораторной работы были решены следующие задачи:

- изучены и реализованны 5 алгоритмов расчета редакционного расстояния;
- протестированы реализованные алгоритмы;
- проведён сравнительный анализ алгоритмов по затраченному процессорному времени и памяти.

Поставленная цель, состоящая в получении навыка динамического программирования на примере реализации алгоритмов редакционного расстояния, достигнута.

# Литература

1. Свойство Process.UserProcessorTime [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/api/system.diagnostics.stopwatch?view=net-5.0. Дата обращения: 01.10.2021