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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

# ОТЧЕТ

## ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Название: Расстояние Левенштейна и Дамерау – Левенштейна

Дисциплина: Анализ алгоритмов

| | | | |
|---|---|---|---|
| Студент | ИУ7-56Б | | Ковель А.Д. |
| | Группа | Подпись, дата | И. О. Фамилия |
| Преподаватель | | | Волкова Л.Л. |
| | | Подпись, дата | И. О. Фамилия |

Москва, 2022 г.

# Оглавление

# Введение

Нахождение редакционного расстояния одна из задач компьютерной лингвистики, которая находит применение в огромном количестве областей, начиная от предиктивных систем набора текста и заканчивая разработкой искусственного интеллекта. Впервые задачу поставил советский ученый В. И. Левенштейн [1], впоследствии её связали с его именем. В данной работе будут рассмотрены алгоритмы редакционного расстояния Левенштейна и расстояние Дамерау – Левенштейна [2].

Расстояния Левенштейна — метрика, измеряющая разность двух строк символов, определяемая в количестве редакторских операций(а именно удаления, вставки и замены), требуемых для преобразования одной последовательности в другую. Расстояние Дамерау – Левенштейна модификация, добавляющая к редакторским операциям транспозицию, или обмен двух соседних символов местами. Алгоритмы имеют некоторое количество модификаций, позволяющих эффективнее решать поставленную задачу. В данной работе будут предложены реализации алгоритмов, использующие парадигмы динамического программирования.

Цель лабораторной работы – получить навыки динамического программирования. Задачами лабораторной работы являются изучение и реализация алгоритмов Левенштейна и Дамерау –Левенштейна, применение парадигм динамического программирования при реализации алгоритмов и сравнительный анализ алгоритмов на основе экспериментальных данных.

В данной лабораторной работе будут рассмотрены разные реализации данных алгоритмов нахождения редакторских расстояний. Такие как: итеративный, рекурсивный и рекурсивный с кэшем.

Также будут приведены сравнения реализации по времени и памяти.

# 1 Аналитический раздел

## 1.1 Расстояние Левенштейна

Редакторское расстояние (расстояние Левенштейна) — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую. Каждая редакторская операция имеет цену (штраф). В общем случае, имея на входе строку, $X = x_1x_2 \dots x_n$, и, $Y = y_1y_2 \dots y_n$, расстояние между ними можно вычислить с помощью операций:

- $\text{delete}(u, \varepsilon) = \delta$;
- $\text{insert}(\varepsilon, v) = \delta$;
- $\text{replace}(u, v) = \alpha(u, v) \leq 0$ (здесь, $\alpha(u, u) = 0$ для всех $u$).;

Необходимо найти последовательность замен с минимальным суммарным штрафом. Далее, цена вставки и удаления будет считаться равной 1. Пусть даны строки: s1 = s1[1..L1], s2 = s2[1..L2], s1[1..i] — подстрока s1 длинной i, начиная с 1-го символа, s2[1..j] - подстрока s2 длинной j, начиная с 1-го символа. Расстояние Левентштейна посчитывается формулой 1.1:

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

## 1.2 Расстояние Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна — модификация расстояния Левенштейна, добавляющая транспозицию к редакторским операциям, предложенными Левенштейном (см. 1.1). изначально алгоритм разрабатывался для сравнения текстов, набранных человеком (Дамерау показал, что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе. Поэтому метрика Дамерау – Левенштейна часто используется в редакторских программах для проверки правописания).

Используя условные обозначения, описанные в разделе 1.1, рекурсивная формула для нахождения расстояния Дамерау – Левенштейна, $f(i,j)$ ,между подстроками, $x_1 \dots x_i$ и, $y_1 \dots y_j$, имеет следующий вид:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i,\ j = 0, \\ \delta_j,\ i = 0, \\ \min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \left[ \begin{array}{l} \delta + f_{X,Y}(i-2, j-2),\ i,j > 1 x_i = y_{j-1} x_{i-1} = y_j \\ \infty,\ \text{иначе}; \end{array} \right. \end{cases} \end{cases} \tag{1.2}$$

## 1.3 Рекурсивная формула

Используя условные обозначения, описанные в разделе 1.2, рекурсивная формула для нахождения расстояния Дамерау – Левенштейна $f(i,j)$ между подстроками, $x_1 \ldots x_i$, и, $y_1 \ldots y_j$, имеет вид 1.3:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i, \ j = 0, \\ \delta_j, \ i = 0, \\ \min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \left[ \begin{array}{l} \delta + f_{X,Y}(i-2, j-2) \quad i, j > 1 x_i = y_{j-1} x_{i-1} = y_j \\ \infty, \ \text{иначе}; \end{array} \right. \end{cases} \end{cases} \tag{1.3}$$

$f_{X,Y}$ — редакционное расстояние между двумя подстроками — первыми $i$ символами строки $X$ и первыми $j$ символами строки $Y$. Можно вывести следующие утверждения:

- Если редакционное расстояние нулевое, то строки равны:
  $f_{X,Y} = 0 \Rightarrow X = Y$
- Редакционное расстояние симметрично:
  $f_{X,Y} = f_{Y,X}$
- Максимальное значение $f_{X,Y}$ — размерность более длинной строки:
  $f_{X,Y} \leq max(|X|, |Y|)$
- Минимальное значение $f_{X,Y}$ — разность длин обрабатываемых строк:
  $f_{X,Y} \geq abs(|X| - |Y|)$
- Аналогично свойству треугольника, редакционное расстояние между двумя строками не может быть больше чем редакционные расстояния каждой из этих строк с третьей:
  $f_{X,Y} \leq f_{X,Z} + f_{Z,Y}$

## 1.4 Матрица расстояний

В алгоритме нахождения редакторского расстояния Дамерау – Левенштейна возможно использование матрицы расстояний.

Пусть, $C_{0\dots|X|,0\dots|Y|}$, — матрица расстояний, где, $C_{i,j}$ — минимальное количество редакторских операций, необходимое для преобразования подстроки, $x_1 \dots x_i$, в подстроку, $y_1 \dots y_j$. Матрица заполняется следующим образом 1.4:

$$
Ci, j = \begin{cases} i & j = 0, \\ j & i = 0, \\ \min \begin{cases} C_{i-1,j-1} + \alpha(x_i, y_i), \\ C_{i-1,j} + 1, \\ C_{i,j-1} + 1) \end{cases} & \end{cases} \tag{1.4}
$$

При решении данной задачи используется ключевая идея динамического программирования — чтобы решить поставленную задачу, требуется разбить на отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Здесь небольшие подзадачи — это заполнение ячеек таблицы с индексами, $i < |X|, j < |Y|$. После заполнения всех ячеек матрицы в ячейке, $C_{|X|,|Y|}$, будет записано искомое расстояние.

## 1.5 Рекурсивный алгоритм расстояния Дамерау – Левенштейна с мемоизацией

При реализации рекурсивного алгоритма используется мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Отличие от формулы 1.4 состоит лишь в начальной инициализации матрицы флагом $\infty$, который сигнализирует о том, была ли обработана ячейка. В случае если ячейка была обработана, алгоритм переходит к следующему шагу.

## 1.6 Вывод

Были рассмотрены обе вариации алгоритма редакторского расстояния (Левенштейна и Дамерау – Левенштейна). Также были приведены разные способы реализации этих алгоритмов такие как: рекурсивный, итеративный и рекурсивный с мемоизацией. Итеративный может быть реализован с помощью парадигм динамического программирования или матрицей расстояния. Рекурсивный с мемоизацией позволяет ускорить обычный рекурсивный алгоритм за счет матрицы, в которой промежуточные подсчеты.

# 2 Конструкторский раздел

В данном разделе представлены схемы реализуемых алгоритмов и их модификации.

## 2.1 Матричные итерационные алгоритмы

На рисунке 2.1 изображена схема алгоритма нахождения расстояния Дамерау – Левенштейна итеративно с использованием матрицы расстояний.

## 2.2 Модификация матричных алгоритмов

Мемоизация - это прием сохранения промежуточных результатов, которые могут еще раз понадобиться в ближайшее время, чтобы избежать их повторного вычисления. Матричный алгоритм нахождения расстояния Дамерау – Левенштейна может быть модифицирован, используя мемоизацию — достаточно инициализировать матрицу значением $\infty$, которое будет рассмотрено в качестве флага. На рисунке 2.3 изображена схема алгоритма, использующая этот прием.

## 2.3 Рекурсивные алгоритмы

На рисунке 2.4 изображена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

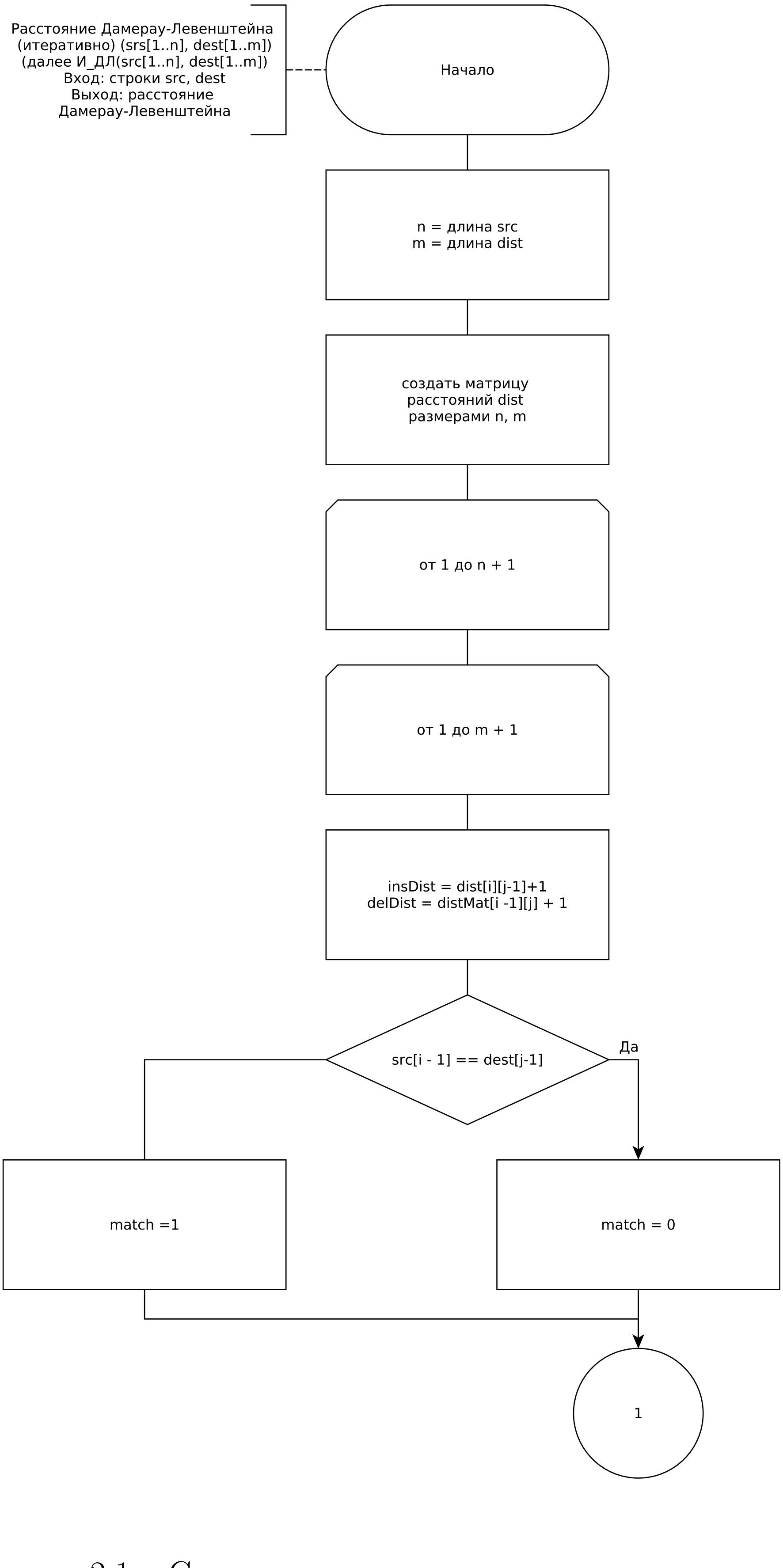


Рисунок 2.1 – Схема итерационного алгоритма расстояния Дамерау–Левенштейна с заполнением матрицы расстояний

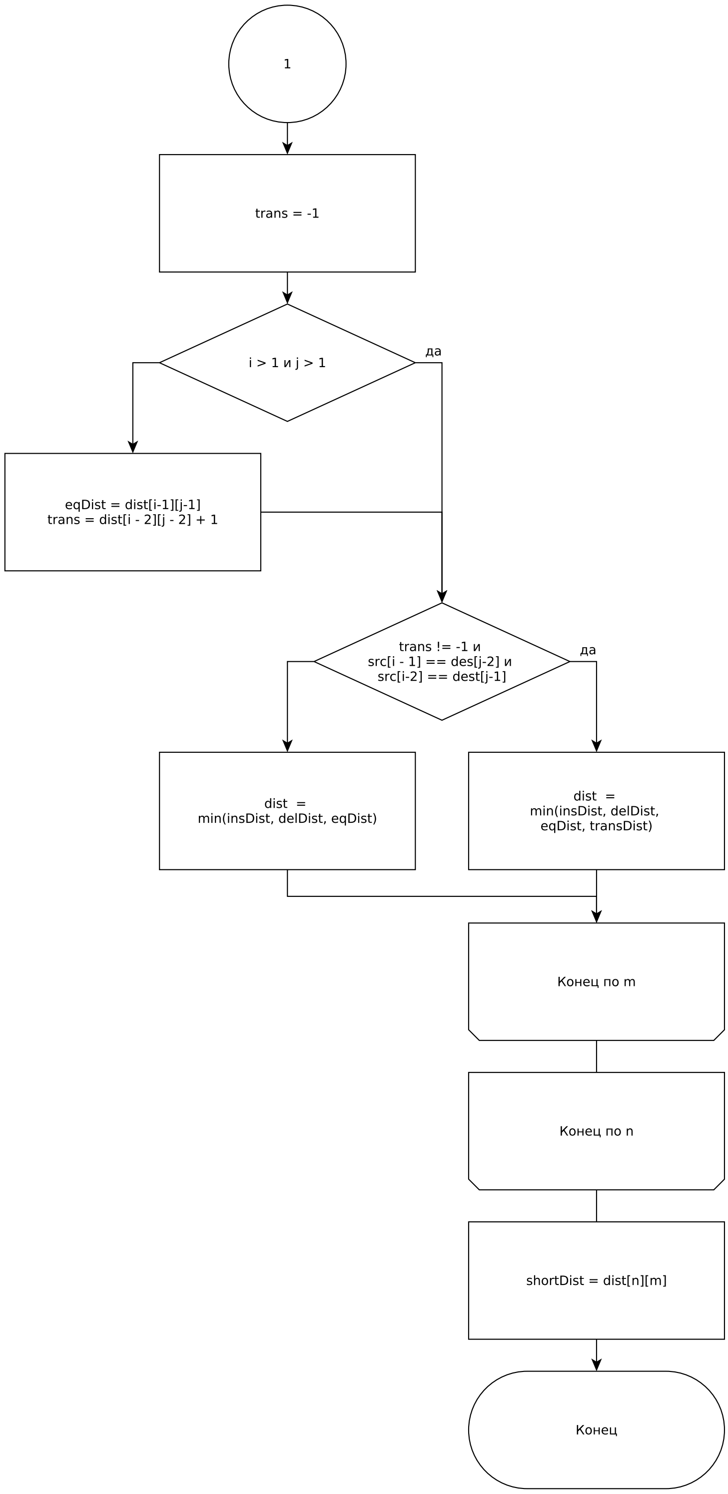


Рисунок 2.2 – Схема итерационного алгоритма расстояния Дамерау–Левенштейна с заполнением матрицы расстояний

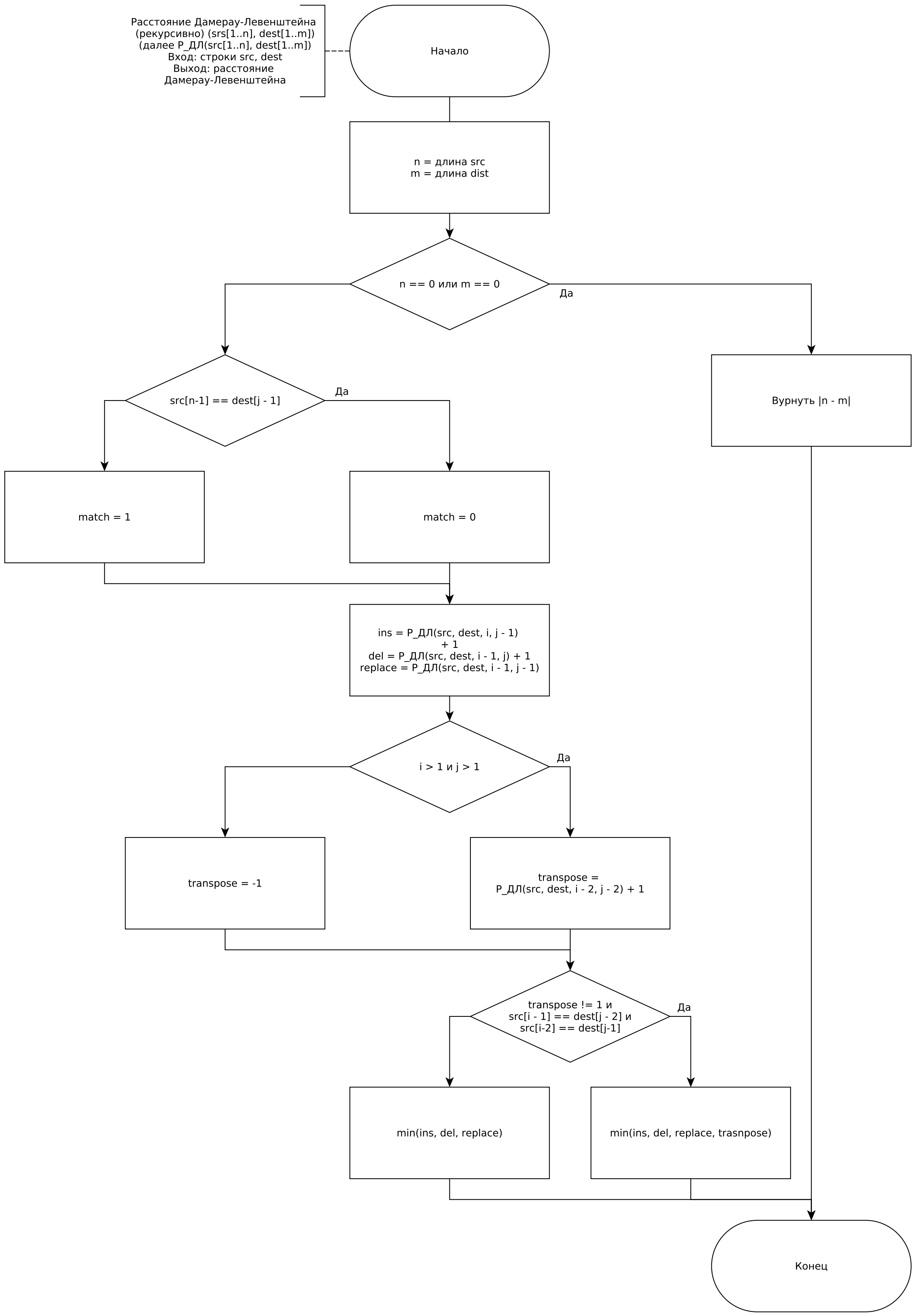


Рисунок 2.3 – Схема рекурсивного алгоритма расстояния Дамерау–Левенштейна с мемоизацией

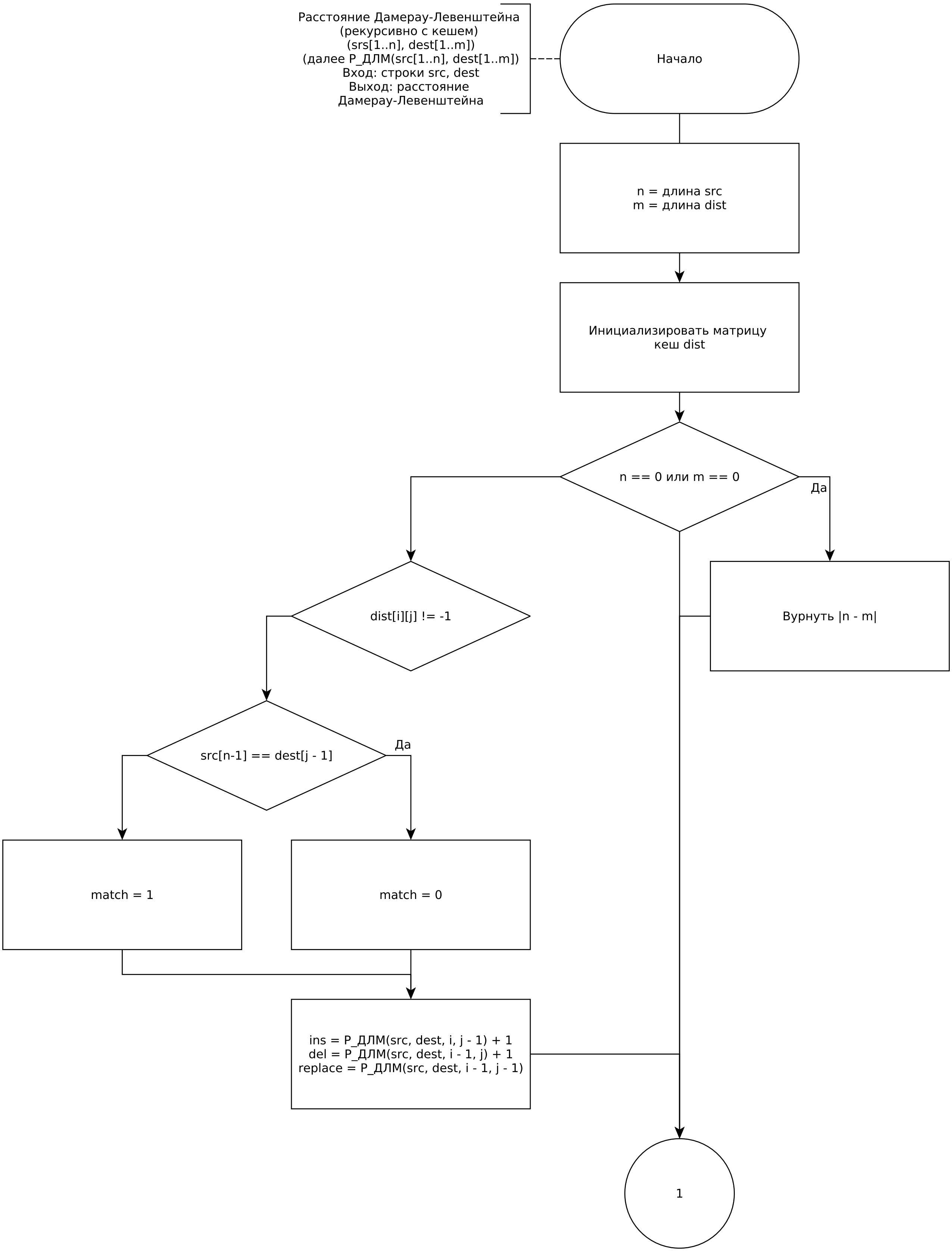


Рисунок 2.4 – Схема рекурсивного алгоритма расстояния Дамерау–Левенштейна

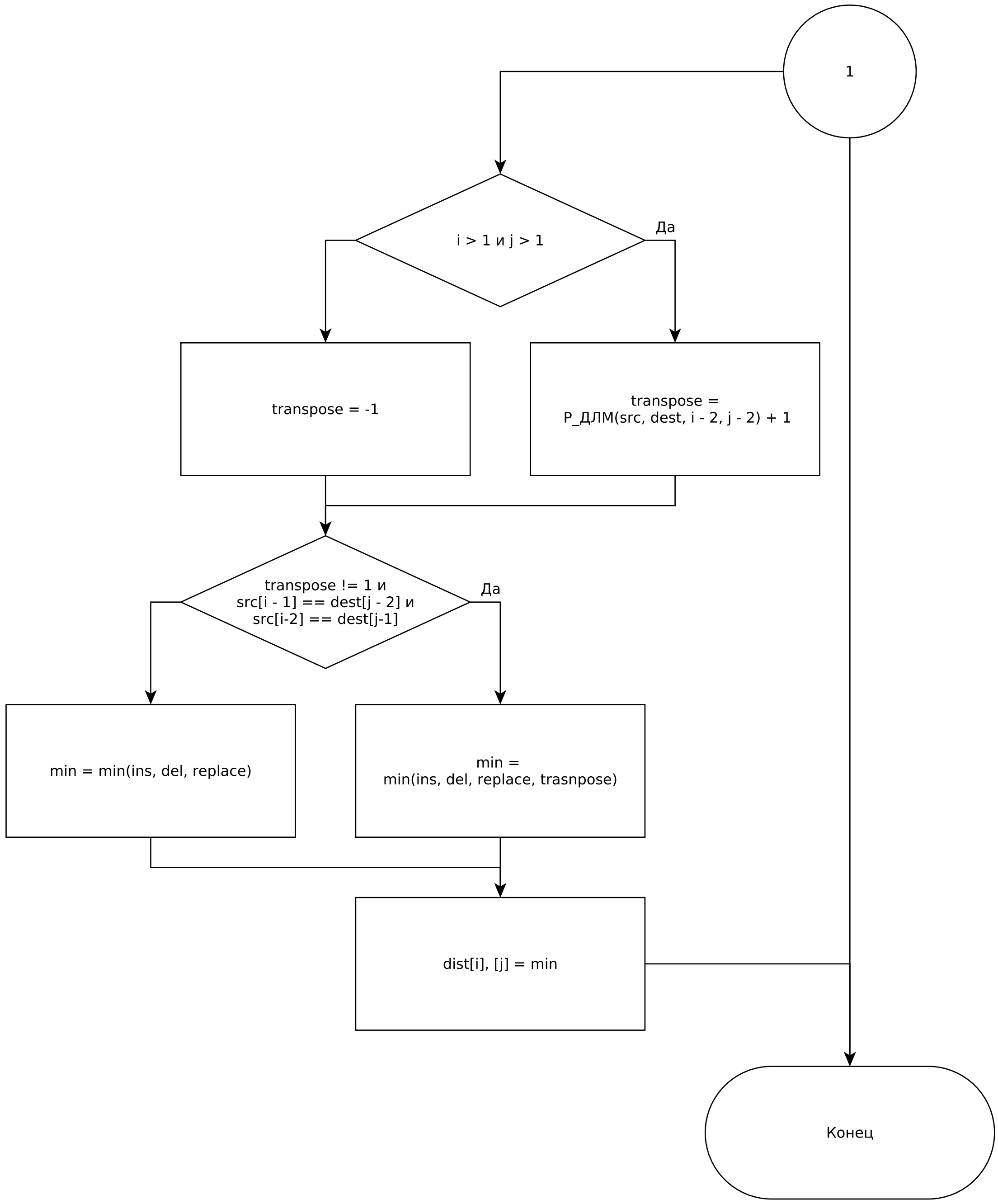


Рисунок 2.5 – Схема рекурсивного алгоритма расстояния Дамерау–Левенштейна

## 2.4 Вывод

На основе формул и теоретических данных, полученных в аналитическом разделе, были спроектированы схемы алгоритмов.

# 3 Технологический раздел

## 3.1 Требования к ПО

Программа должна отвечать следующим требованиям:

- ПО корректно реагирует на любые действия пользователя;
- ПО возвращает полученное расстояние;
- ПО принимает текстовые данные в любой раскладке.
- Время отклика программы на любое действие пользователя должно быть приемлемым.
- ПО корректно высчитывает время своей работы

## 3.2 Средства реализации

Для реализации ПО был выбран язык программирования Golang[3]. Данный язык предоставляет следующие возможности:

- Язык компилируемый, то есть на выходе будет получен исполняемый файл, что позволит точно измерить время работы программы.
- Средства объектно-ориентированного программирования ограничиваются интерфейсами. Это позволяет создавать абстракции, при этом оставаясь высокоскоростным языком.
- Статическая типизация. Позволяет избежать ошибок, допущенных по невнимательности, упрощает чтение и понимание кода.
- Обширная стандартная библиотека. Позволяет использовать уже написанные шаблоны, что уменьшает количество кода, которое необходимо написать. Поддерживает библиотеки C и C++

В качестве среды разработки была выбрана среда VS Code[4], запуск происходил через команду go run main.go.

## 3.3 Листинги кода

### 3.3.1 Реализация алгоритмов

В качестве представления строковых данных был выбран тип rune[4] – псевдоним для типа int32.

В листингах 3.1 - 3.3 приведены реализации алгоритмов, описанных в разделе 1.

Листинг 3.1 – Программный код нахождения расстояния Дамерау – Левенштейна итеративно

```
func CountDamNoRec(src, dest string) (int, MInt) {
  var (n, m, dist, shortDist, transDist int)

  srcRune, destRune := []rune(src), []rune(dest)

  n, m = len(srcRune), len(destRune)

  distMat := makeMatrix(n, m)

  for i := 1; i < n + 1; i++ {
    for j := 1; j < m + 1; j++ {
      insDist := distMat[i][j - 1] + 1
      delDist := distMat[i - 1][j] + 1

      match := 1
      if src[i - 1] == dest[j - 1] {
        match = 0
      }
      eqDist := distMat[i - 1][j - 1] + match

      transDist = -1
      if i > 1 && j > 1 {
        transDist = distMat[i - 2][j - 2] + 1
      }

      if transDist != -1 && srcRune[i - 1] == destRune[j - 2] &&
      srcRune[i - 2] == destRune[j - 1] {
        dist = getMinOfValues(insDist, delDist, eqDist, transDist)
```

```
        } else {
            dist = getMinOfValues(insDist, delDist, eqDist)
        }
        distMat[i][j] = dist
    }
  }

  shortDist = distMat[n][m]

  return shortDist, distMat
}
```

Листинг 3.2 – Программный код нахождения расстояния Дамерау – Левенштейна рекурсивно

```
func _countDamRecElem(src, dest []rune, i, j int) int {
  if (getMinOfValues(i, j) == 0) {
    return getMaxOf2Values(i, j)
  }

  match := 1
  if (src[i - 1] == dest[j - 1]) {
    match = 0
  }

  insert := _countDamRecElem(src, dest, i, j - 1) + 1
  delete := _countDamRecElem(src, dest, i - 1, j) + 1
  replace := match+_countDamRecElem(src, dest, i - 1, j - 1)

  transpose := -1

  if i > 1 && j > 1 {
    transpose = _countDamRecElem(src, dest, i - 2, j - 2) + 1
  }

  min := 0
  if transpose != -1 && src[i - 1] == dest[j - 2]
 && src[i - 2] == dest[j - 1] {
    min = getMinOfValues(insert, delete, replace, transpose)
  } else {
    min = getMinOfValues(insert, delete, replace)
  }
```

```
  return min
}

func CountDamRecNoCache(src, dest string) int {

  srcRune, destRune := []rune(src), []rune(dest)
  n, m := len(srcRune), len(destRune)

  return _countDamRecElem(srcRune, destRune, n, m)
}
```

Листинг 3.3 – Программный код нахождения расстояния Дамерау – Левенштейна рекурсивно с кэшем

```
func _countDamRecElemCache(src, dest []rune, i, j int, distMat
    MInt) int {
  if (getMinOfValues(i, j) == 0) {
    return getMaxOf2Values(i, j)
  }

  if distMat[i][j] != -1 {
    return distMat[i][j]
  }

  match := 1
  if (src[i - 1] == dest[j - 1]) {
    match = 0
  }

  insert := _countDamRecElemCache(src, dest, i, j - 1, distMat) +
     1
  delete := _countDamRecElemCache(src, dest, i - 1, j, distMat) +
     1
  replace := match + _countDamRecElemCache(src, dest, i - 1, j -
     1, distMat)

  transpose := -1

  if i > 1 && j > 1 {
    transpose = _countDamRecElemCache(src, dest, i - 2, j - 2,
       distMat) + 1
  }
```

```

	min := 0
	if transpose != -1 && src[i - 1] == dest[j - 2]
	&& src[i - 2] == dest[j - 1] {
		min = getMinOfValues(insert, delete, replace, transpose)
	} else {
		min = getMinOfValues(insert, delete, replace)
	}

	distMat[i][j] = min
	return distMat[i][j]

}

func CountDamRecCache(src, dest string) int {
	srcRune, destRune := []rune(src), []rune(dest)
	n, m := len(srcRune), len(destRune)

	distMat := makeMatrixInf(n, m)

	return _countDamRecElemCache(srcRune, destRune, n, m, distMat)
}
```

### 3.3.2 Подпрограммы

В листингах 3.4 - 3.6 приведены используемые подпрограммы.

Листинг 3.4 – Функция нахождения минимума из N целых чисел

```
	func getMinOfValues(values ...int) int {
		min := values[0]

		for _, i := range values {
			if min > i {
				min = i
			}
		}

		return min
	}
```

Листинг 3.5 – Функция нахождения максимума из двух целых чисел

```go
func getMaxOf2Values(v1, v2 int) int {
  if v1 < v2 {
    return v2
  }
  return v1
}
```

Листинг 3.6 – Определение типа целочисленной матрицы; его инициализация и вывод

```go
type MInt [][]int

func makeMatrix(n, m int) MInt {
  matrix := make(MInt, n + 1)

  for i := range matrix {
    matrix[i] = make([]int, m + 1)
  }

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

  for i := 0; i < n + 1; i++ {
    matrix[i][0] = i
  }
  return matrix
}


func (mat MInt) PrintMatrix() {
  for i := 0; i < len(mat); i++ {
    for j := 0; j < len(mat[0]); j++ {
      fmt.Printf("%3d ", mat[i][j])
    }
    fmt.Printf("\n")
  }
}
```

## 3.4 Тестовые данные

| № | $S_1$ | $S_2$ | DLIter | DLRec | DLRecCache |
|---|---|---|---|---|---|
| 1 | « » | « » | 0 | 0 | 0 |
| 2 | «book» | «bosk» | 1 | 1 | 1 |
| 3 | «book» | «back» | 2 | 2 | 2 |
| 4 | «book» | «bacc» | 3 | 3 | 3 |
| 5 | «aboba» | «acacb» | 4 | 4 | 4 |
| 6 | «дверь» | «деврь» | 1 | 1 | 1 |
| 6 | «дверь» | «дверь» | 1 | 1 | 1 |

## 3.5 Вывод

На основе схем из конструкторского раздела были разработаны программные реализации требуемых алгоритмов.

# 4 Исследовательская часть

## 4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Pop!_OS 22.04 LTS [5] Linux [6];
- Оперативная память 16 GiB;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [7];

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования.

## 4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи профилирования – сбора характеристик работы программы: времени выполнения и затрат по памяти. Для каждой функции были написаны тесты оценки эффективности "бенчмарки"[8], представленные встроенными в Golang средствами. тесты эффективности, реализованные стандартными средствами Golang автоматически делают некоторое количество замеров, предоставляя результат с некоторой погрешностью.

Листинг 4.1 – Пример теста эффективности

```
func BenchmarkCountDamNoRec10(b testing.B) src := "abaoboaobj"dest :=
    "da;ldfjalj"for i := 0; i < b.N; i++ CountDamNoRec(src, dest)
```

Результаты тестирования приведены в таблице. Прочерк в таблице означает что тестирование для этого набора данных не выполнялось.

Таблица 4.1 – Время выполнения алгоритмов

| Длина строк | Время выполнения() | | |
|---|---|---|---|
| | DRecMem | DLIter | DLRec |
| 5 | 2344 | 1114 | 17228 |
| 10 | 6747 | 3142 | 109170295 |
| 40 | 92218 | 36281 | - |
| 80 | 402839 | 142910 | - |
| 160 | 1582974 | 646498 | - |
| 240 | 3505394 | 1348110 | - |

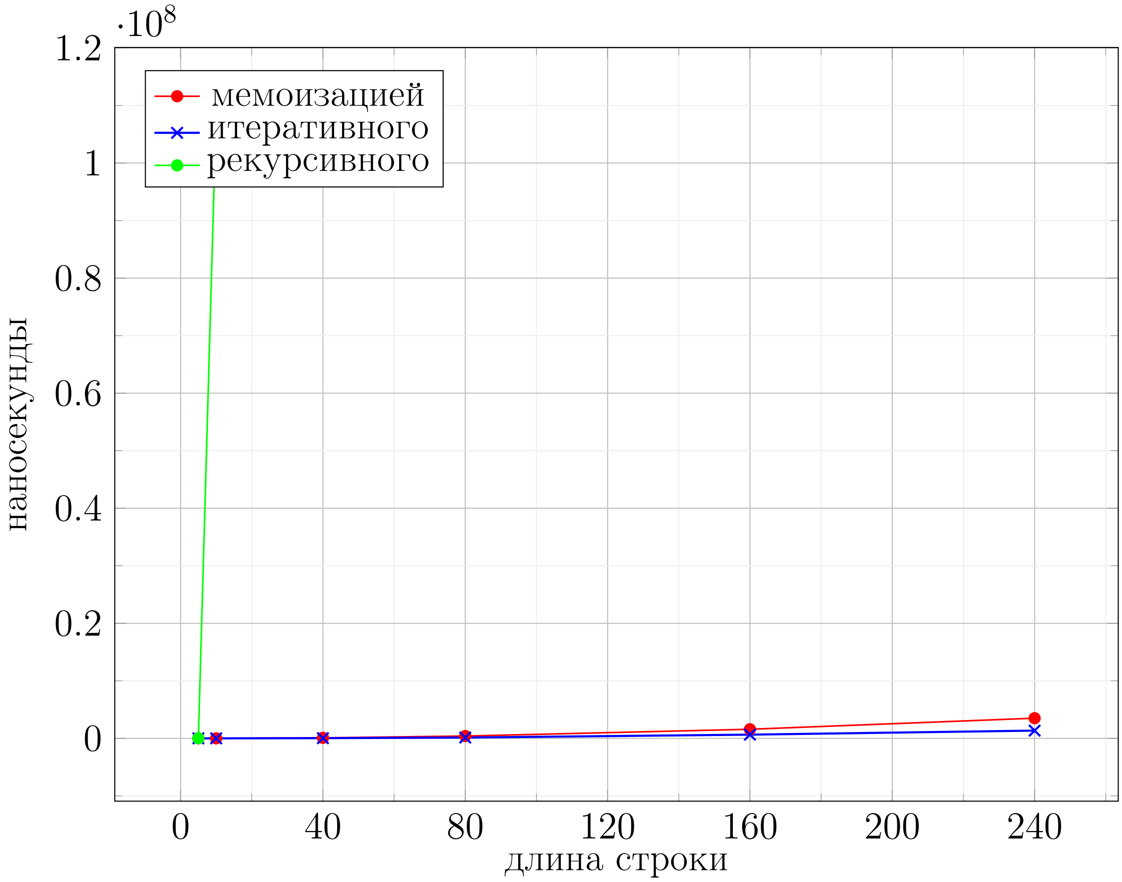


Рисунок 4.1 – Сравнение рекурсивного с мемоизацией, итеративного и рекурсивного расстояния Дамерау – Левенштейна

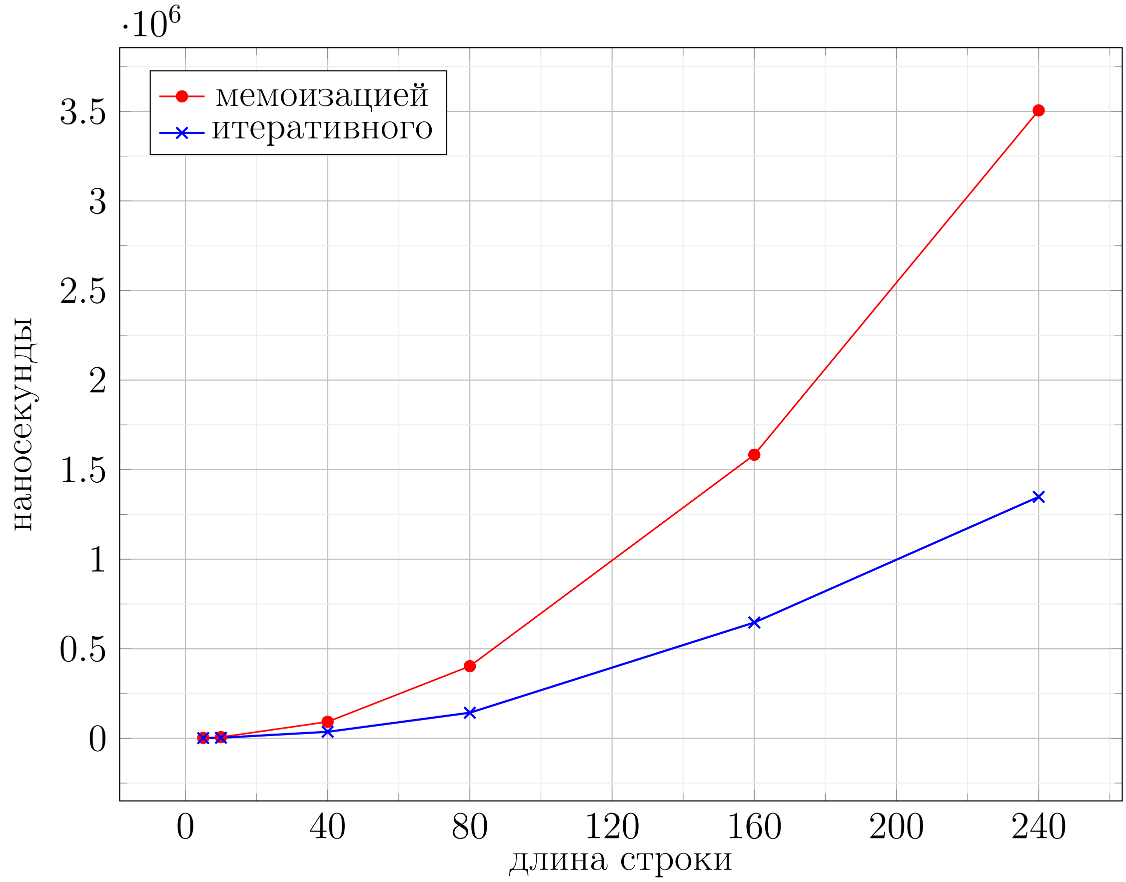


Рисунок 4.2 – Сравнение рекурсивного с мемоизацией, итеративного расстояния Дамерау – Левенштейна

## 4.3 Использование памяти

Максимальная глубина стека при вызове рекурсивных функций имеет следующий вид:

$$M_{recursive} = (n \cdot lvar + ret + ret_{int}) \cdot depth \tag{4.1}$$

Где:

$n$ — количество аллоцированных локальных переменных;
$lvar$ — размер переменной типа int
$ret$ — адрес возврата;
$ret_{int}$ — возвращаемое значение;
$depth$ — максимальная глубина стека вызова, которая равна $|S_1| + |S_2|$.

Использование памяти при итеративных реализациях:

$$M_{iter} = |S_1| + |S_2| + (|S_1| + 1 \cdot |S_2| + 1) \cdot lvar + n \cdot lvar + ret + ret_{int} \quad (4.2)$$

Где $(|S_1| + 1 \cdot |S_2| + 1) \cdot lvar$ – место в памяти под матрицу расстояний.

## 4.4 Вывод

В данном разделе были сравнены алгоритмы по памяти и по времени. Рекурсивный алгоритм Дамерау – Левенштейна работает дольше итеративных реализаций – время этого алгоритма увеличивается в геометрической прогрессии с ростом размера строк. Рекурсивный алгоритм с мемоизацией превосходит простой рекурсивный алгоритм по времени. По расходу памяти все реализации проигрывают рекурсивной за счет большого количества выделенной памяти под матрицу расстояний.

То есть самым эффективный по памяти: рекурсивный алгоритм. Самый эффективный по времени: итеративный алгоритм (исходя из сделанных тестов.)

Стоит отметить, что для языков, где возможна передача указателя на массивы, самым эффективным и по времени, и по памяти будет алгоритм, использующий мемоизацию.

# Заключение

В рамках лабораторной работы были:

- Рассмотрены три алгоритма нахождения редакторского расстояния Дамерау – Левенштейна.
- В аналитическом разделе были изучены смысловые различия между алгоритмами и их формульное представление.
- В рамках конструкторского раздела были получены схемы алгоритмов.
- В технологическом разделе был выбран язык программирования и представлена реализация на нем, также были приведены тестовые данные.
- В исследовательской части были сравнены алгоритмы по скорости и по памяти. Самым эффективным по времени оказался итеративный алгоритм. Самым эффективным по памяти — рекурсивный алгоритм.

В ходе лабораторной работы получены навыки динамического программирования, реализованы и изученные алгоритмы нахождения редакторского расстояния.

# Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.

[3] Golang Документация[Электронный ресурс]. Режим доступа: https://go.dev/doc/ (дата обращения: 24.09.2022).

[4] Go rune. Режим доступа: https://golangdocs.com/rune-in-golang (дата обращения: 04.09.2022).

[5] Pop OS 22.04 LTS [Электронный ресурс]. Режим доступа: https://pop.system76.com (дата обращения: 04.09.2022).

[6] Linux – Документация [Электронный ресурс]. Режим доступа: https://docs.kernel.org (дата обращения: 24.09.2022).

[7] Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [Электронный ресурс]. Режим доступа: https://www.amd.com/en/products/cpu/amd-ryzen-7-2700 (дата обращения: 04.09.2022).

[8] Go 1 Release Notes. Режим доступа: https://pkg.go.dev/testing (дата обращения: 04.09.2022).