



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 «Анализ Алгоритмов»  
на тему: «Редакционное расстояние»

Студент группы ИУ7-51Б

\_\_\_\_\_  
(Подпись, дата)

Шубенина Д. 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

<b>Введение</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Расстояние Левенштейна . . . . .	4
1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна . . . . .	5
1.2 Расстояние Дамерау – Левенштейна . . . . .	5
1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна . . . . .	6
1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна с кэшированием . . . . .	7
1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна . . . . .	7
Вывод . . . . .	8
<b>2 Конструкторская часть</b>	<b>9</b>
2.1 Требования к программному обеспечению . . . . .	9
2.2 Требования вводу . . . . .	9
2.3 Разработка алгоритмов . . . . .	9
2.4 Описание используемых типов данных . . . . .	14
Вывод . . . . .	14
<b>3 Технологическая часть</b>	<b>15</b>
3.1 Средства реализации . . . . .	15
3.2 Сведения о модулях программы . . . . .	15
3.3 Реализация алгоритмов . . . . .	16
3.4 Функциональные тесты . . . . .	21
Вывод . . . . .	21
<b>4 Исследовательская часть</b>	<b>22</b>
4.1 Технические характеристики . . . . .	22
4.2 Демонстрация работы программы . . . . .	22

4.3	Временные характеристики . . . . .	23
4.4	Характеристики по памяти . . . . .	25
4.5	Вывод . . . . .	26
<b>Список использованных источников</b>		<b>27</b>

# Введение

**Расстояние Левенштейна** (также называемое редакционным расстоянием или дистанцией редактирования) — это метрика, которая измеряет разницу между двумя строками. Определяет минимальное количество операций вставки, удаления и замены символов, необходимых для преобразования одной строки в другую.

**Расстояние Дамерау – Левенштейна** является расширением расстояния Левенштейна, которое включает дополнительную операцию — транспозицию, чтобы обработать случаи, когда символы меняются местами или переупорядочиваются.

Расстояния Левенштейна и Дамерау – Левенштейна используются при решении следующих задач:

- 1) корректировка поискового запроса;
- 2) классификация текстов;
- 3) распознавание речи;
- 4) определение сходства между текстами;

**Целью** данной лабораторной работы является изучение, реализация и исследование алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна.

Необходимо выполнить следующие **задачи**:

- 1) изучить алгоритмы Левенштейна и Дамерау – Левенштейна для нахождения редакционного расстояния между строками;
- 2) реализовать данные алгоритмы;
- 3) выполнить сравнительный анализ алгоритмов по затрачиваемым ресурсам (времени, памяти);
- 4) описать и обосновать полученные результаты в отчете.

# 1 Аналитическая часть

## 1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения строки в другую [1].

Введем следующие обозначения операций:

- $w(a, b)$  — цена замены символа  $a$  на символ  $b$ ;
- $w(\varepsilon, b)$  — цена вставки символа  $b$ ;
- $w(a, \varepsilon)$  — цена удаления символа  $a$ ;

Каждая операция имеет определенную цену:

- **M** (от англ. match):  $w(a, a) = 0$
- **R** (от англ. replace):  $w(a, b) = 1, a \neq b$
- **I** (от англ. insert):  $w(\varepsilon, b) = 1$
- **D** (от англ. delete):  $w(a, \varepsilon) = 1$

Пусть имеется две строки  $S_1$  и  $S_2$  длиной  $m$  и  $n$  соответственно. Расстояние Левенштейна  $d(S_1, S_2) = D(m, n)$  рассчитывается по следующей рекуррентной формуле [2]:

$$D(m, n) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[j]), \end{cases} & j > 0, i > 0 \end{cases} \quad (1.1)$$

где сравнение символов строк  $S_1$  и  $S_2$  производится следующим образом:

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

### 1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

При больших значениях  $m$  и  $n$  рекурсивная реализация алгоритма поиска расстояния Левенштейна малоэффективна по времени выполнения, так как промежуточные значения  $D(i, j)$  вычисляются несколько раз. В таком случае можно воспользоваться итеративной реализацией данного алгоритма, использующего матрицу для хранения промежуточных значений.

Матрица имеет размеры

$$(n + 1) \times (m + 1) \quad (1.3)$$

где  $m$  — длина строки  $S_1$ ,  $n$  — длина строки  $S_2$ .

В ячейке  $[i, j]$  матрицы хранится значение  $D(S_1[1..i], S_2[1..j])$ . Первому элементу матрицы присвоено значение 0. Вся матрица заполняется в соответствии с соотношением (1.1).

## 1.2 Расстояние Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна является метрикой для измерения различий между двумя строками. Оно определяется как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для превращения одной строки в другую. Это расширение расстояния Левенштейна, так как, помимо трех базовых операций, оно также включает операцию транспозиции Т (от англ. transposition).

Расстояние Дамерау – Левенштейна определяется по следующей рекуррентной формуле:

$$D(m, n) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1), \\ D(i - 2, j - 2) + 1, \end{cases} & \begin{array}{l} \text{если } i, j > 1, \\ S_1[i] = S_2[j - 1], \\ S_1[i - 1] = S_2[j], \end{array} \\ \min \begin{cases} D(i - 1, j) + 1, \\ D(i, j - 1) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[j]) \end{cases} & \text{иначе.} \end{cases} \quad (1.4)$$

### 1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна

Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна реализует формулу (1.4) следующим образом:

- 1) Если одна из строк пустая, возвращается длина другой строки.
- 2) Если последние символы двух строк совпадают, рекурсивно вызывается функция для остатков строк (без последних символов).
- 3) Иначе рекурсивно вызываются четыре варианта преобразования строки:

— **Вставка:** к результату рекурсивного вызова для остатка первой строки добавляется 1.

- **Удаление:** к результату рекурсивного вызова для остатка второй строки добавляется 1.
- **Замена:** к результату рекурсивного вызова для остатков строк добавляется 1.
- **Транспозиция:** если последние и предпоследние символы двух строк совпадают, к результату рекурсивного вызова для остатка строк добавляется 1.

4) Возвращается минимальное из четырех вариантов значение.

### 1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна с кэшированием

При больших  $m$  и  $n$  рекурсивная реализация алгоритма поиска расстояния Дамерау – Левенштейна малоэффективна по времени, так как промежуточные значения расстояний между подстроками вычисляются неоднократно. Для оптимизации рекурсивного алгоритма по времени можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы  $A_{m,n}$  промежуточными значениями  $D(i, j)$ .

### 1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна

При больших значениях  $m$ ,  $n$  алгоритм нахождения расстояния Дамерау – Левенштейна, использующий рекурсию, не является эффективным по времени. Вместо рекурсивной реализации можно использовать итерационную реализацию. В таком случае в качестве структуры для хранения промежуточных значений  $D(i, j)$  используется



матрица, имеющая размеры

$$(n + 1) \times (m + 1) \tag{1.5}$$

В ячейке  $[i, j]$  матрицы хранится значение  $D(S_1[1..i], S_2[1..j])$ . Первому элементу матрицы присвоено значение 0. Вся матрица заполняется в соответствии с соотношением (1.4).

## Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна — их рекурсивные и итеративные реализации. Также была рассмотрена оптимизация алгоритма нахождения расстояния Дамерау–Левенштейна с помощью кэширования.

## 2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна, приведены описание используемых типов данных и структуры программного обеспечения.

### 2.1 Требования к программному обеспечению

К программе предъявлен ряд функциональных требований:

- наличие интерфейса для выбора действий;
- возможность ввода строк;
- возможность обработки строк, состоящих как из латинских символов, так и из кириллических;
- возможность произвести замеры процессорного времени работы реализованных алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна.

### 2.2 Требования вводу

- 1) На вход реализованным алгоритмам подаются две строки.
- 2) Строки могут включать как латинские, так и кириллические символы.
- 3) Буквы нижнего и верхнего регистра считаются разными символами.

### 2.3 Разработка алгоритмов

На рисунке 2.1 представлена схема матричного алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.4 приведены схемы матрич-

ной, рекурсивной и рекурсивной с кэшированием реализаций алгоритма нахождения расстояния Дамерау – Левенштейна.

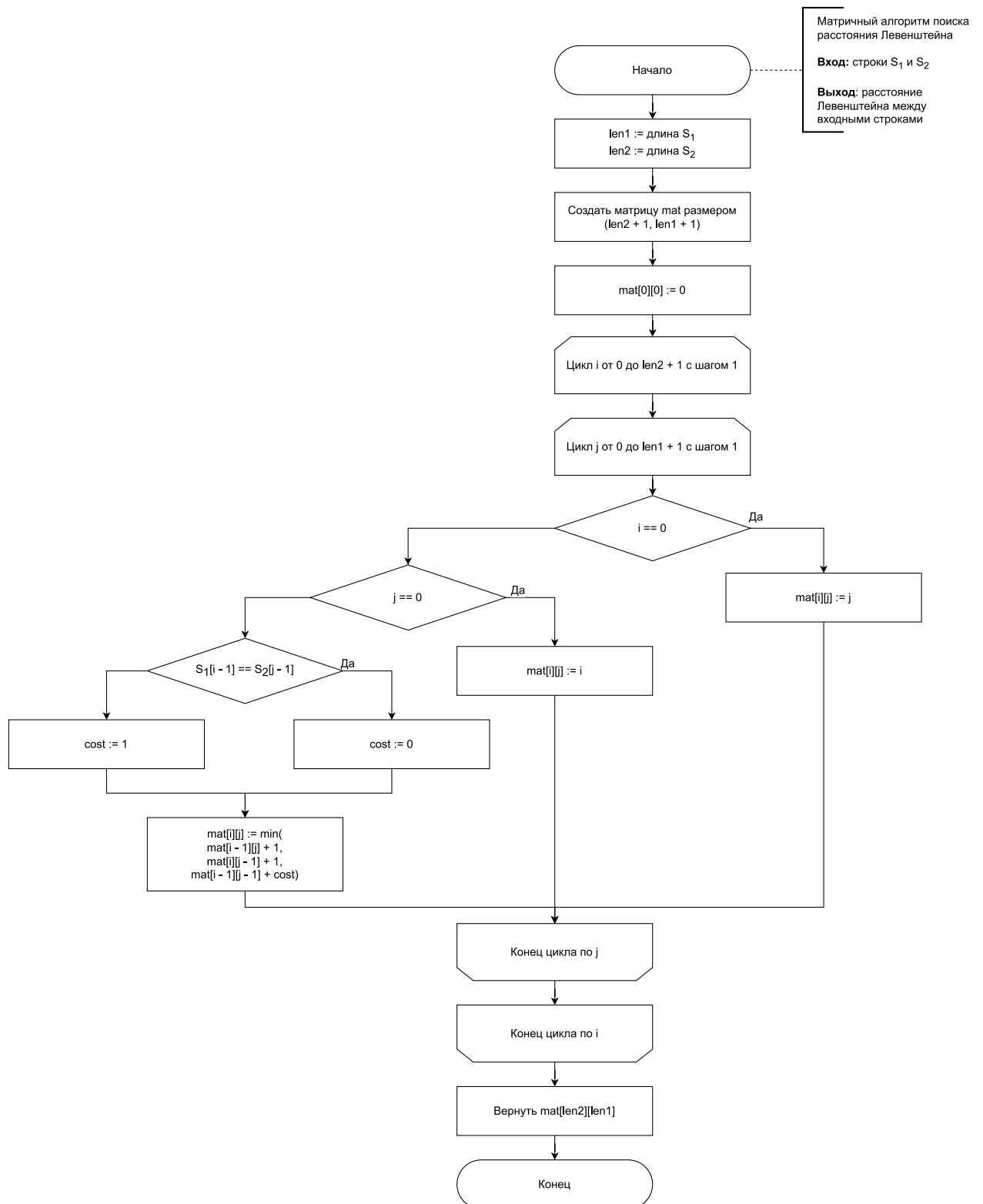


Рисунок 2.1 – Схема матричного алгоритма Левенштейна

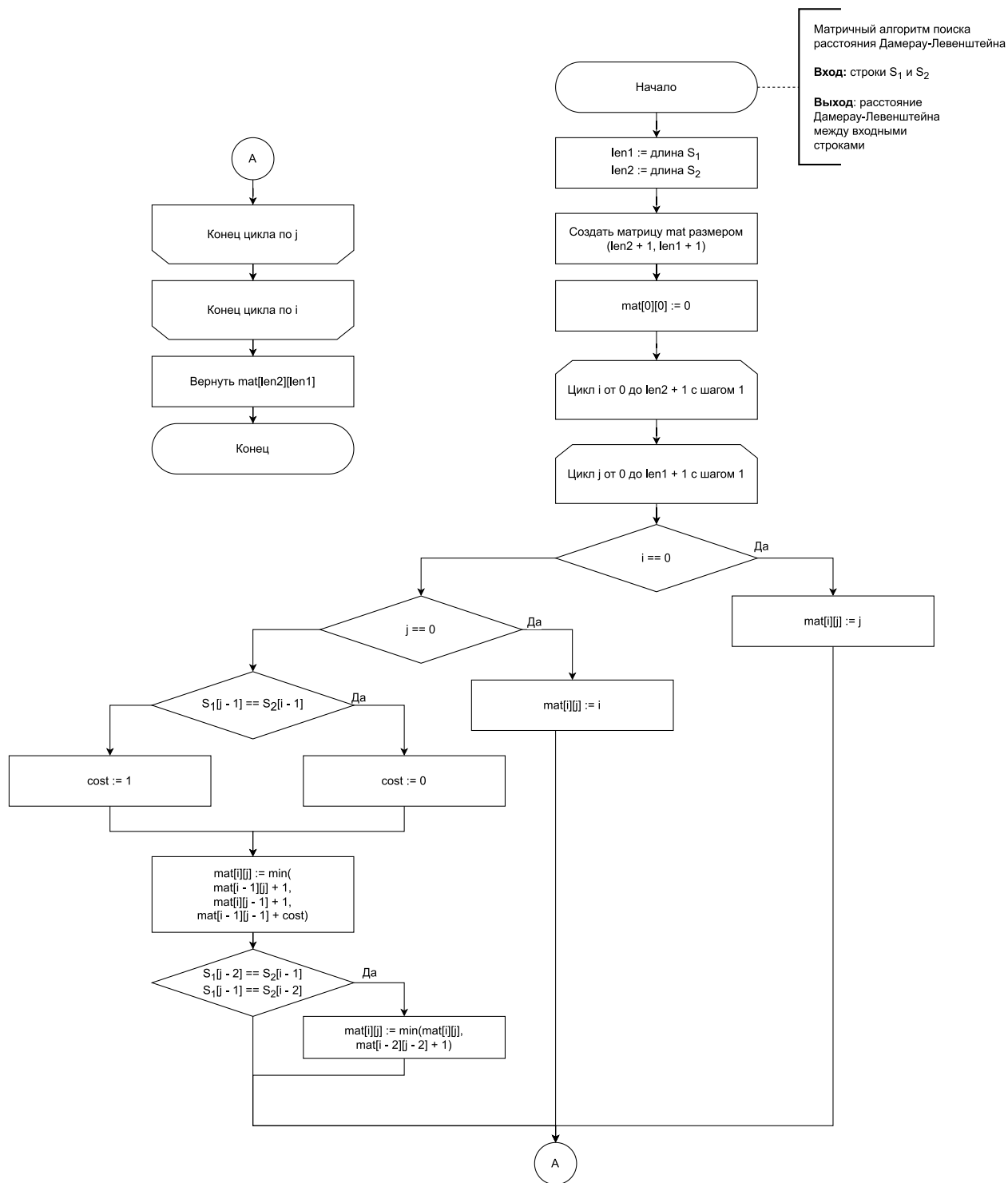


Рисунок 2.2 – Схема матричного алгоритма Дамерау – Левенштейна

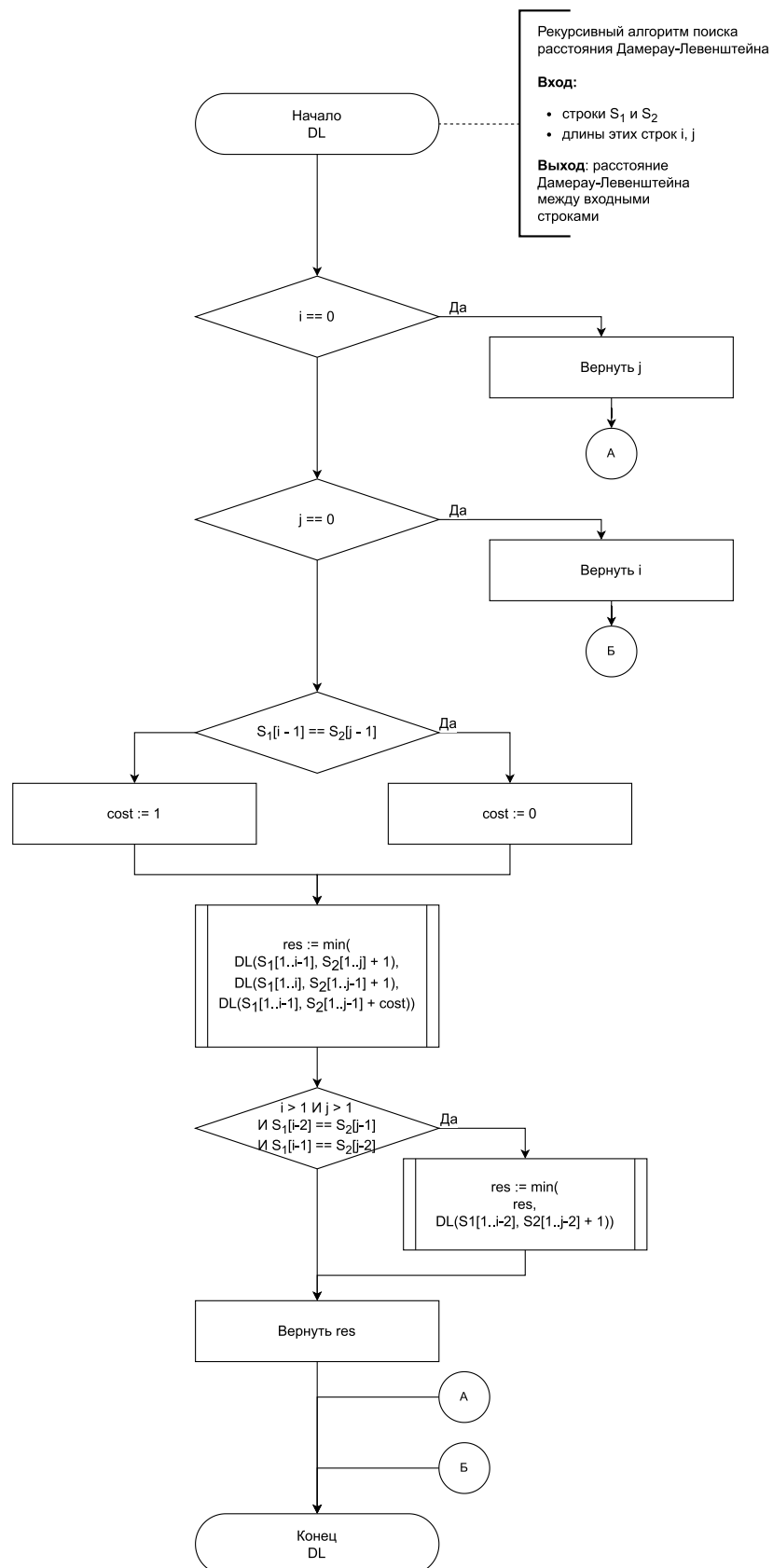


Рисунок 2.3 – Схема рекурсивного алгоритма Дамерау – Левенштейна

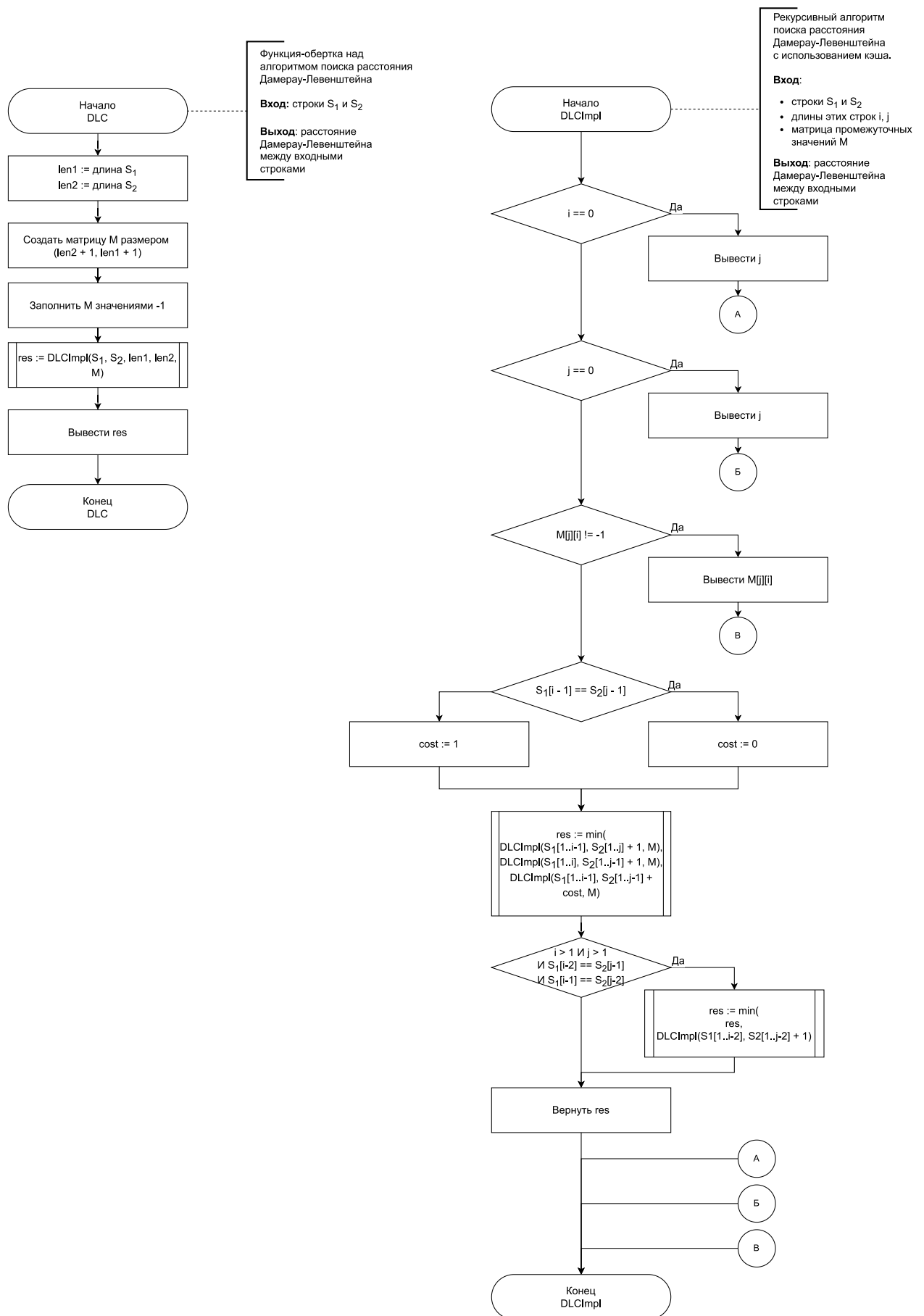


Рисунок 2.4 – Схема рекурсивного алгоритма Дамерау – Левенштейна с использованием кэша

## 2.4 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

- *строка* — массив символов типа `wchar_t`;
- *длина строки* — значение длины строки типа `int`;
- *матрица* — двумерный массив значений типа `int`.

## Вывод

В данном разделе на основе теоретических данных были перечислены требования к ПО. Также были построены схемы реализуемых алгоритмов на основе данных, полученных на этапе анализа.

## 3 Технологическая часть

В данном разделе приведены средства реализации программного обеспечения, сведения о модулях программы, листинг кода и функциональные тесты.

### 3.1 Средства реализации

В качестве языка, используемого при написании данной лабораторной работы, был выбран язык C++ [3]. Этот выбор обусловлен тем, что в данном языке программирования имеется контейнер `std::wstring`, представляющий собой массив символов типа `std::wchar_t`. Также в языке C++ имеется библиотека `<ctime>`, позволяющая выполнять замеры процессорного времени.

В качестве средства написания кода была выбрана кроссплатформенная среда разработки *Visual Studio Code*, т.к. она предоставляет широкий функционал для проектирования, разработки и отладки ПО.

### 3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

- `main.cpp` — файл, содержащий точку входа в программу;
- `matrix.cpp` — файл, содержащий функции создания матрицы, ее освобождения и вывода на экран;
- `algorithms.cpp` — файл, содержащий реализации алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна;
- `measure.cpp` — файл, содержащий функции, измеряющие процессорное время выполнения реализуемых алгоритмов.



### 3.3 Реализация алгоритмов

Листинг 3.1 – Функция `max`, используемая в алгоритмах

```
1 template <typename T>
2 static T min(T x, T y, T z)
3 {
4     return std::min(x, std::min(y, z));
5 }
```

Листинг 3.2 – Матричный алгоритм поиска расстояния Левенштейна  
(часть 1)

```
1 int LevNonRec(const std::wstring &word1, const std::wstring
    &word2, bool verbose)
2 {
3     int len1 = word1.length();
4     int len2 = word2.length();
5
6     int **mat = Matrix::Allocate(len2 + 1, len1 + 1);
7     if (!mat)
8         return -1;
9
10    mat[0][0] = 0;
11    for (int i = 0; i <= len2; ++i)
12    {
13        for (int j = 0; j <= len1; ++j)
14        {
15            if (i == 0)
16            {
17                mat[i][j] = j;
18            }
19            else if (j == 0)
20            {
21                mat[i][j] = i;
22            }
23        }
24    }
```

Листинг 3.3 – Матричный алгоритм поиска расстояния Левенштейна  
(часть 2)

```
1         else
2         {
3             int cost = (word1[j - 1] == word2[i - 1]) ? 0 :
4                 1;
5
6             mat[i][j] = min(
7                 mat[i - 1][j] + 1,
8                 mat[i][j - 1] + 1,
9                 mat[i - 1][j - 1] + cost);
10        }
11    }
12
13    if (verbose)
14        Matrix::Print(mat, word1, word2);
15
16    int res = mat[len2][len1];
17    Matrix::Free(mat, len2 + 1);
18
19    return res;
20 }
```

Листинг 3.4 – Матричный алгоритм поиска расстояния  
Дамерау–Левенштейна (часть 1)

```
1 int DamLevNonRec(const std::wstring &word1, const std::wstring
2     &word2, bool verbose)
3 {
4     int len1 = word1.length();
5     int len2 = word2.length();
6
7     int **mat = Matrix::Allocate(len2 + 1, len1 + 1);
8     if (!mat)
9         return -1;
10
11     mat[0][0] = 0;
12     for (int i = 0; i <= len2; ++i)
13     {
```

### Листинг 3.5 – Матричный алгоритм поиска расстояния

Дамерау – Левенштейна (часть 2)

```
1      for (int j = 0; j <= len1; ++j)
2      {
3          if (i == 0)
4          {
5              mat[i][j] = j;
6          }
7          else if (j == 0)
8          {
9              mat[i][j] = i;
10         }
11         else
12         {
13             int cost = (word1[j - 1] == word2[i - 1]) ? 0 :
14                         1;
15
16             mat[i][j] = min(
17                 mat[i - 1][j] + 1,
18                 mat[i][j - 1] + 1,
19                 mat[i - 1][j - 1] + cost);
20
21             if (word1[j - 2] == word2[i - 1] && word1[j -
22                 1] == word2[i - 2])
23             {
24                 mat[i][j] = std::min(
25                     mat[i][j],
26                     mat[i - 2][j - 2] + 1);
27             }
28         }
29     }
30
31     if (verbose)
32         Matrix::Print(mat, word1, word2);
33
34     int res = mat[len2][len1];
35     Matrix::Free(mat, len2 + 1);
36
37     return res;
38 }
```

### Листинг 3.6 – Рекурсивный алгоритм поиска расстояния

Дамерау – Левенштейна

```
1 int DamLevRec(const std::wstring &word1, const std::wstring
    &word2, int ind1, int ind2)
2 {
3     if (ind1 == 0)
4         return ind2;
5     if (ind2 == 0)
6         return ind1;
7
8     int cost = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;
9
10    int res = min(
11        DamLevRec(word1, word2, ind1, ind2 - 1) + 1,
12        DamLevRec(word1, word2, ind1 - 1, ind2) + 1,
13        DamLevRec(word1, word2, ind1 - 1, ind2 - 1) + cost);
14
15    if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -
        2] && word1[ind1 - 2] == word2[ind2 - 1])
16        res = std::min(res, DamLevRec(word1, word2, ind1 - 2,
            ind2 - 2) + 1);
17
18    return res;
19 }
```

### Листинг 3.7 – Рекурсивный алгоритм поиска расстояния

Дамерау – Левенштейна с кэшированием (реализация) (часть 1)

```
1 static int DamLevRecCacheImpl(const std::wstring &word1, const
    std::wstring &word2, int ind1, int ind2, int **memo)
2 {
3     if (ind1 == 0)
4         return ind2;
5
6     if (ind2 == 0)
7         return ind1;
8
9     if (memo[ind2][ind1] != -1)
10        return memo[ind2][ind1];
11
12    int cost = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;
```

### Листинг 3.8 – Рекурсивный алгоритм поиска расстояния

Дамерау – Левенштейна с кэшированием (реализация) (часть 2)

```
1  int res = min(  
2      DamLevRecCacheImpl(word1, word2, ind1, ind2 - 1, memo)  
3          + 1,  
4      DamLevRecCacheImpl(word1, word2, ind1 - 1, ind2, memo)  
5          + 1,  
6      DamLevRecCacheImpl(word1, word2, ind1 - 1, ind2 - 1,  
7          memo) + cost  
8  );  
9  
10 if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -  
11     2] && word1[ind1 - 2] == word2[ind2 - 1])  
12     res = std::min(res, DamLevRecCacheImpl(word1, word2,  
13         ind1 - 2, ind2 - 2, memo) + 1);  
14  
15 memo[ind2][ind1] = res;  
16 return res;  
17 }
```

### Листинг 3.9 – Рекурсивный алгоритм поиска расстояния

Дамерау – Левенштейна с кэшированием (оберточная функция)

```
1  int DamLevRecCache(const std::wstring &word1, const  
2      std::wstring &word2)  
3  {  
4      int len1 = word1.length();  
5      int len2 = word2.length();  
6  
7      int **memo = Matrix::Allocate(len2 + 1, len1 + 1, -1);  
8      if (!memo)  
9          return -1;  
10  
11     int res = DamLevRecCacheImpl(word1, word2, len1, len2,  
12         memo);  
13  
14     Matrix::Free(memo, len2 + 1);  
15     return res;  
16 }
```

### 3.4 Функциональные тесты

Таблица 3.1 – Функциональные тесты

Входные данные		Расстояние и алгоритм			
Строка 1	Строка 2	Левенштейна	Дамерау – Левенштейна		
		Итеративный	Итеративный	Рекурсивный	
				Без кэша	С кэшем
а	б	1	1	1	1
а	в	1	1	1	1
г	г	0	0	0	0
сердце	солнце	3	3	3	3
стол	стул	1	1	1	1
кот	ток	2	2	2	2
кто	кот	2	1	1	1
Вениамин	Венгрия	4	4	4	4
стол	столы	1	1	1	1

### Вывод

Были реализованы алгоритмы Левенштейна (итеративно) и Дамерау – Левенштейна (итеративно, рекурсивно, рекурсивно с кэшированием). Проведено тестирование реализованных алгоритмов.

## 4 Исследовательская часть

### 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

- Процессор: Intel i5-1035G1 (8) @ 3.600GHz.
- Оперативная память: 16 ГБайт.
- Операционная система: Manjaro Linux x86\_64 (версия ядра Linux 5.15.131-1-MANJARO).

Во время проведения измерений времени ноутбук был подключен к сети электропитания и был нагружен только системными приложениями.

### 4.2 Демонстрация работы программы

На рисунке 4.1 показан пример работы разработанной программы для случая, когда пользователь выбирает действие «Запуск алгоритмов поиска расстояния Левенштейна» и вводит строки «кот» и «кошка».

```

      Меню
1. Запуск алгоритмов поиска расстояния Левенштейна:
  1) Нерекursивный Левенштейна.
  2) Нерекursивный Дамерау–Левенштейна.
  3) Рекурсивный Дамерау–Левенштейна без кэша.
  4) Рекурсивный Дамерау–Левенштейна с кэшем.
2. Замерить время для реализованных алгоритмов.
0. Выход

Выберите опцию (0–2): 1

Введите 1–е слово: кошка
Введите 2–е слово: броненосец

Выводить матрицы для итеративных реализаций? [y/N]:

Минимальное кол–во операций:
  1) Нерекursивный Левенштейна:                9
  2) Нерекursивный Дамерау–Левенштейна:        9
  3) Рекурсивный Дамерау–Левенштейна без кэша:  9
  4) Рекурсивный Дамерау–Левенштейна с кэшем:   9

      Меню
1. Запуск алгоритмов поиска расстояния Левенштейна:
  1) Нерекursивный Левенштейна.
  2) Нерекursивный Дамерау–Левенштейна.
  3) Рекурсивный Дамерау–Левенштейна без кэша.
  4) Рекурсивный Дамерау–Левенштейна с кэшем.
2. Замерить время для реализованных алгоритмов.
0. Выход

Выберите опцию (0–2): █

```

Рисунок 4.1 – Демонстрация работы программы

### 4.3 Временные характеристики

Исследование временных характеристик алгоритмов производилось на случайно сгенерированных строках длинами от 1 до 10, длины изменяются с шагом 1. Для нерекursивных алгоритмов отдельно производилось сравнение на строках длинами от 20 до 100 с шагом изменения длины 10. Во избежание погрешности измерения для каждой строки производились 50 раз, затем вычислялось среднее арифметическое всех полученных значений времени.



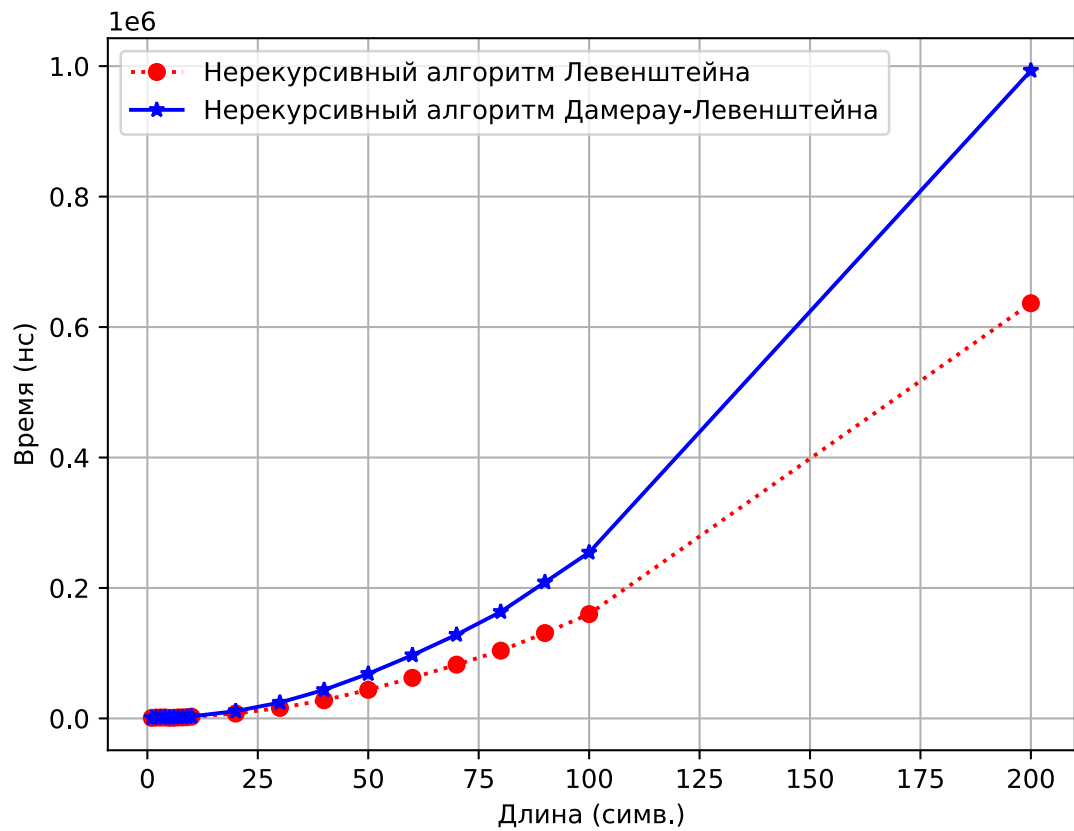


Рисунок 4.2 – Результат измерений времени работы нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна

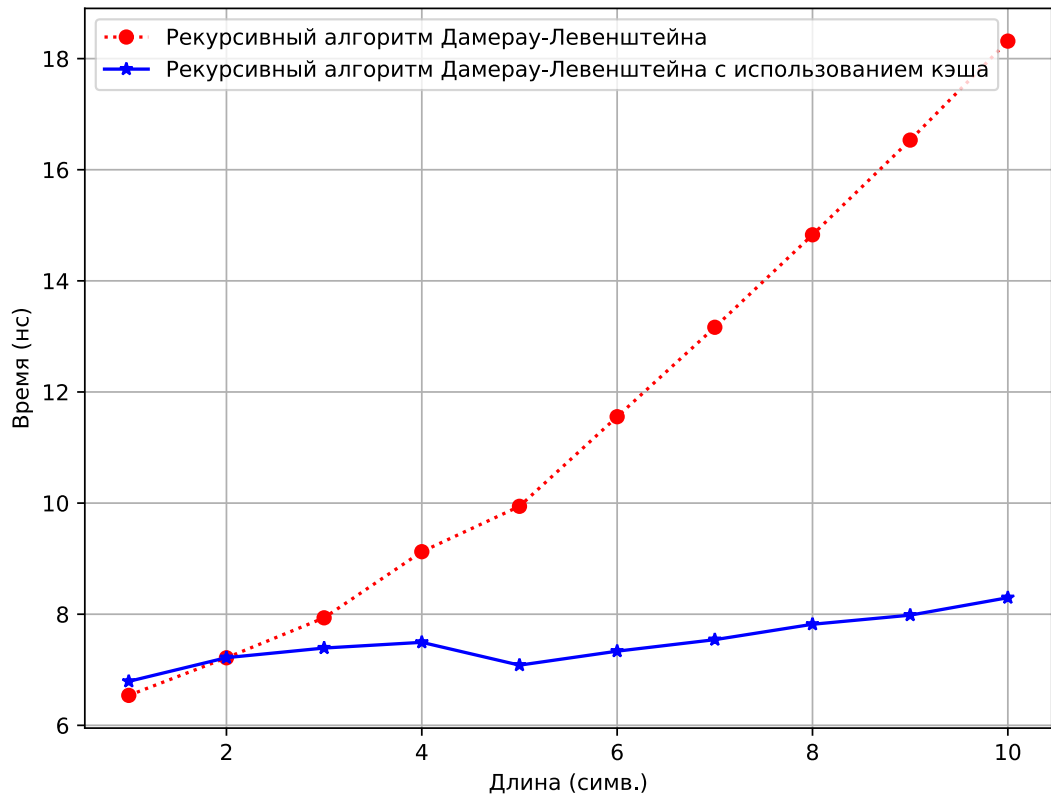


Рисунок 4.3 – Результат измерений времени работы рекурсивный реализаций алгоритма поиска расстояния Дамерау – Левенштейна

## 4.4 Характеристики по памяти

Введем следующие обозначения:

- $m$  — длина строки  $S_1$ ;
- $n$  — длина строки  $S_2$ ;
- $size(v)$  — функция, вычисляющая размер входного параметра  $v$  в байтах;
- $char$  — тип данных, используемый для хранения символа строки;
- $int$  — целочисленный тип данных.

Теоретически оценим объем используемой памяти итеративной реализацией алгоритма поиска расстояния Левенштейна:

$$\begin{aligned}
M_{LevIter} = & (m + 1) \cdot (n + 1) \cdot size(int) + (m + n) \cdot size(char) + \\
& + size(int **) + (m + 1) \cdot size(int*) + \\
& + 3 \cdot size(int) + 2 \cdot size(int), \quad (4.1)
\end{aligned}$$

где:

- $(m + 1) \cdot (n + 1) \cdot size(int)$  — размер матрицы;
- $size(int **)$  — размер указателя на матрицу;
- $(m + 1) \cdot size(int*)$  — размер указателей на строки матрицы;
- $(m + n) \cdot size(char)$  — размер двух входных строк;
- $2 \cdot size(int)$  — размер переменных, хранящих длину строк;
- $3 \cdot size(int)$  — размер переменных цикла  $i$ ,  $j$  и промежуточной переменной  $cost$ .

Для алгоритма поиска расстояния Дамерау–Левенштейна теоретическая оценка объема используемой памяти идентична.

Произведем оценку затрат по памяти для рекурсивных реализаций алгоритма нахождения расстояния Дамерау–Левенштейна.

Для

## 4.5 Вывод

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 А. Погорелов Д., М. Таразанов А., Л. Волкова Л. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна // Синергия Наук. 2019. URL: — Режим доступа: <https://elibrary.ru/item.asp?id=36907767> (дата обращения 10.10.2023).
- 2 В. Траулько М. Программная реализация нечеткого поиска текстовой информации в словаер с помощью расстояния Левенштейна // Форум молодых ученых. 2017. URL: — Режим доступа: <https://cyberleninka.ru/article/n/programmная-realizatsiya-nechetkogo-poiska-tekstovoy-informatsii-v-slovaer-s-pomoschyu-rasstoyniya-Levenstejna> (дата обращения 14.10.2023).
- 3 Документация по Microsoft C++ [Электронный ресурс]. — Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017> (дата обращения: 25.09.2022).