



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 Нерекursивный алгоритм нахождения расстояния Ле- венштейна . . . . .	5
1.2 Расстояние Дамерау — Левенштейна . . . . .	5
1.2.1 Рекурсивный алгоритм нахождения расстояния Даме- рау — Левенштейна . . . . .	6
1.2.2 Рекурсивный алгоритм нахождения расстояния Даме- рау — Левенштейна с кэшированием . . . . .	7
1.2.3 Нерекursивный алгоритм нахождения расстояния Да- мерау — 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 . . . . .	15
Вывод . . . . .	15
<b>3 Технологическая часть</b>	<b>16</b>
3.1 Средства реализации . . . . .	16
3.2 Сведения о модулях программы . . . . .	16
3.3 Реализация алгоритмов . . . . .	16
3.4 Функциональные тесты . . . . .	22
Вывод . . . . .	22
<b>4 Исследовательская часть</b>	<b>23</b>
4.1 Технические характеристики . . . . .	23
4.2 Демонстрация работы программы . . . . .	23

4.3	Временные характеристики . . . . .	24
4.4	Характеристики по памяти . . . . .	27
4.5	Вывод . . . . .	29
<b>Заключение</b>		<b>31</b>
<b>Список использованных источников</b>		<b>32</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 увеличении значений  $m, n$  алгоритм поиска расстояния Левенштейна, использующий рекурсию, становится малоэффективным по времени за счет того, что в ходе работы алгоритма промежуточные значения  $D(i, j)$  вычисляются неоднократно.

Результаты промежуточных вычислений можно сохранять в матрицу размером  $(n + 1) \times (m + 1)$ , 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 — это расширение расстояния Левенштейна, определяющееся как минимальное количество операций вставки, удаления, замены и транспозиции Т (от англ. transposition).

Расстояние Дамерау — Левенштейна задается следующей рекуррент-

ной формулой:

$$D(m, n) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1), \\ D(i - 2, j - 2) + 1, \end{cases} & \begin{aligned} & \text{если } i, j > 1, \\ & S_1[i] = S_2[j - 1], \\ & S_1[i - 1] = S_2[j], \end{aligned} \\ \min \begin{cases} D(i - 1, j) + 1, \\ D(i, j - 1) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[j]) \end{cases} & \text{иначе.} \end{cases} \quad (1.3)$$

### 1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна реализует формулу (1.3) 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 увеличении  $m$  и  $n$  рекурсивная реализация алгоритма поиска расстояния Дамерау — Левенштейна становится крайне не эффективной по времени, так как промежуточные значения расстояний между подстроками вычисляются неоднократно. Избавиться от повторяющихся вычислений можно с помощью матрицы  $A_{m,n}$ , в которую по ходу работы алгоритма сохраняются соответствующие промежуточные значения  $D(i, j)$  расстояний.

Размер матрицы-кэша равен  $(n + 1) \times (m + 1)$ .

### 1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

При увеличении значений  $m, n$  алгоритм нахождения расстояния Дамерау — Левенштейна, использующий рекурсию, становится менее эффективным по времени, поэтому вместо рекурсивной реализации можно использовать итеративную, для хранения промежуточных значений  $D(i, j)$  применяющую матрицу размером  $(n + 1) \times (m + 1)$ .

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 (1.3).



## Вывод

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

## 2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дameraу — Левенштейна, 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 Дameraу — 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 рисунке 2.2 приведена схема матричной реализации алгоритма поиска расстояния Дамерау — Левенштейна.

На рисунке 2.3 представлена его рекурсивная реализация.

На рисунке 2.4 показана рекурсивная реализация алгоритма нахождения расстояния Дамерау — Левенштейна с использованием матрицы-кэша.

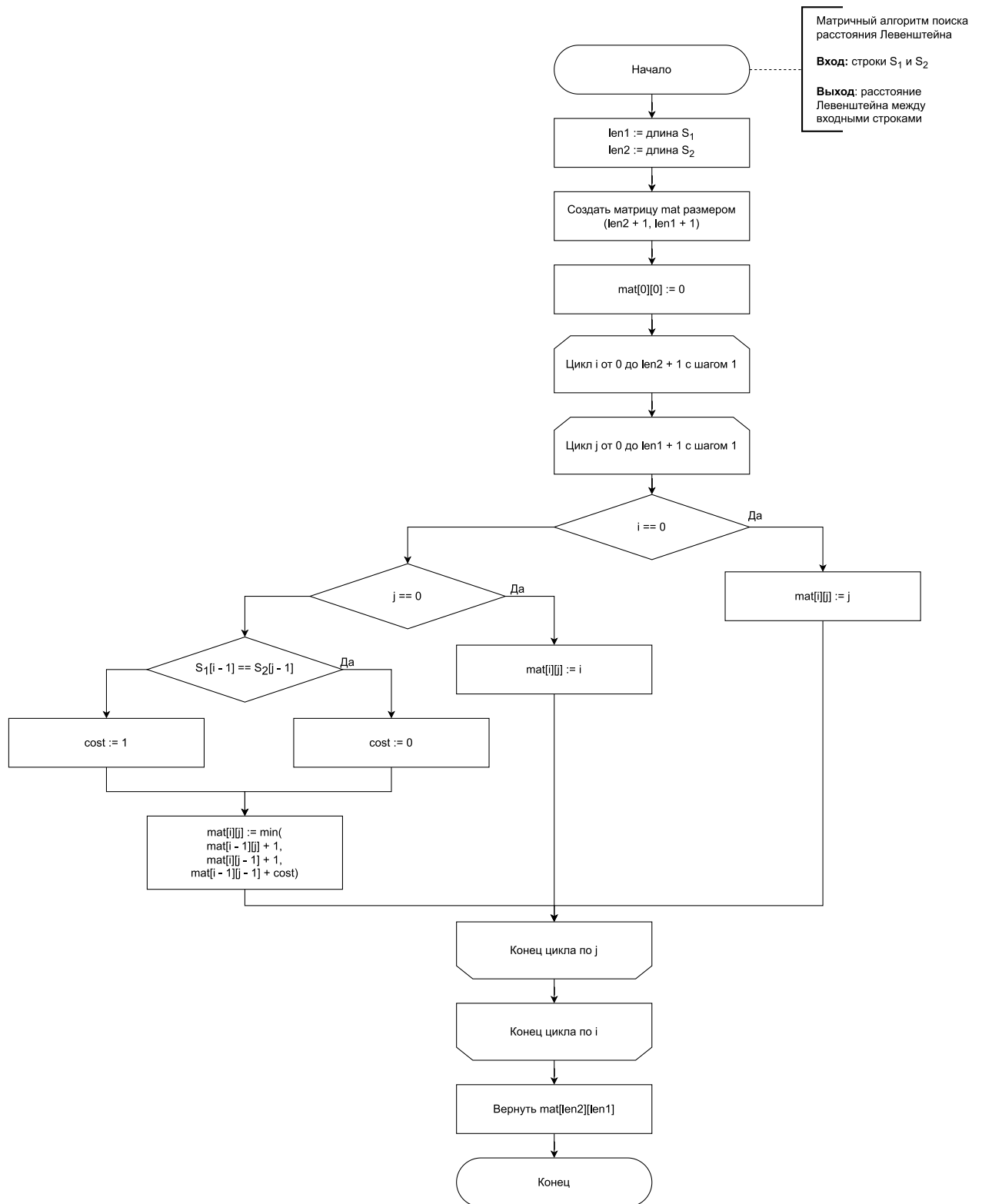


Рисунок 2.1 – Схема матричного алгоритма Левенштейна

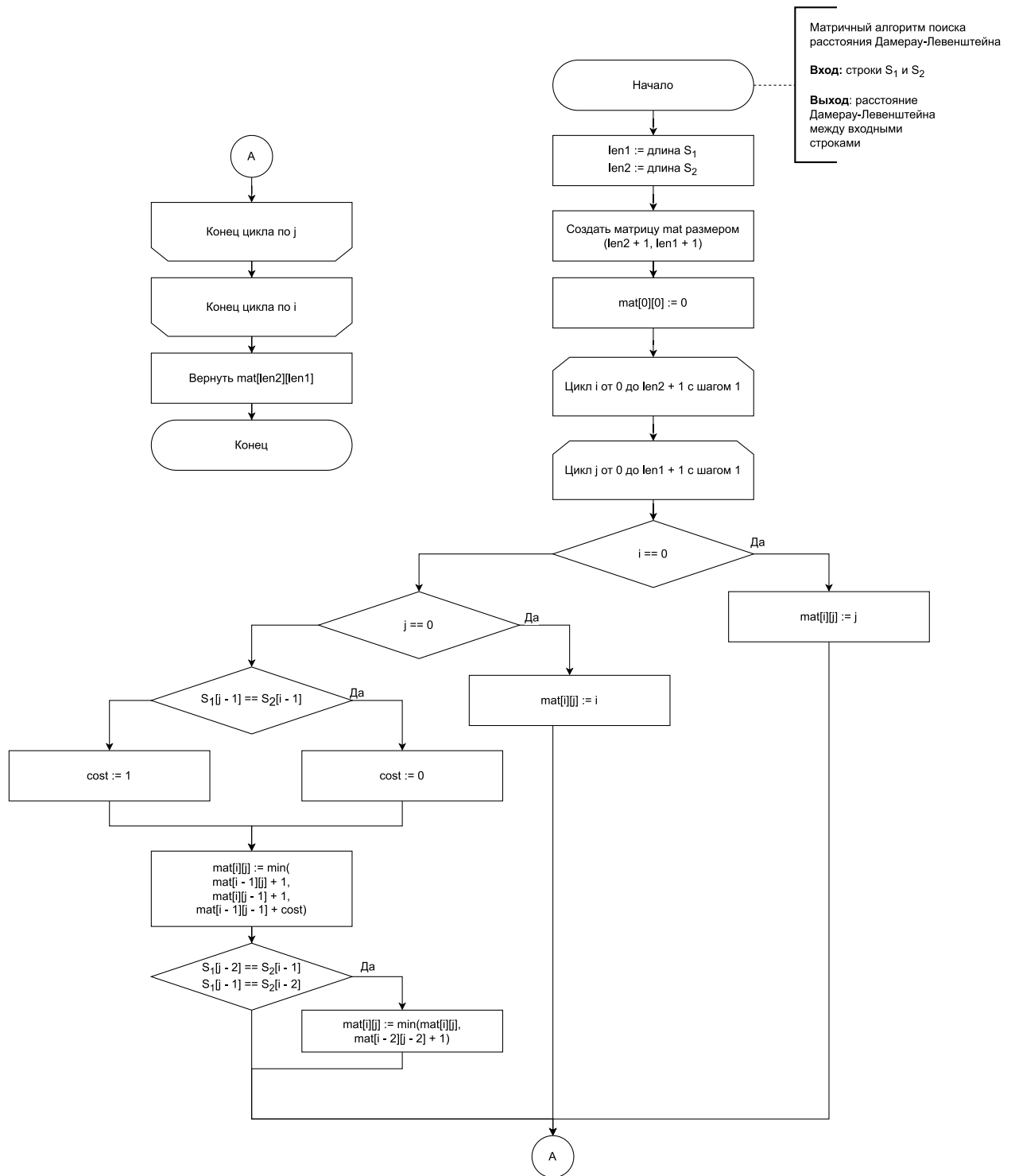


Рисунок 2.2 – Схема матричного алгоритма Дамерау — Левенштейна

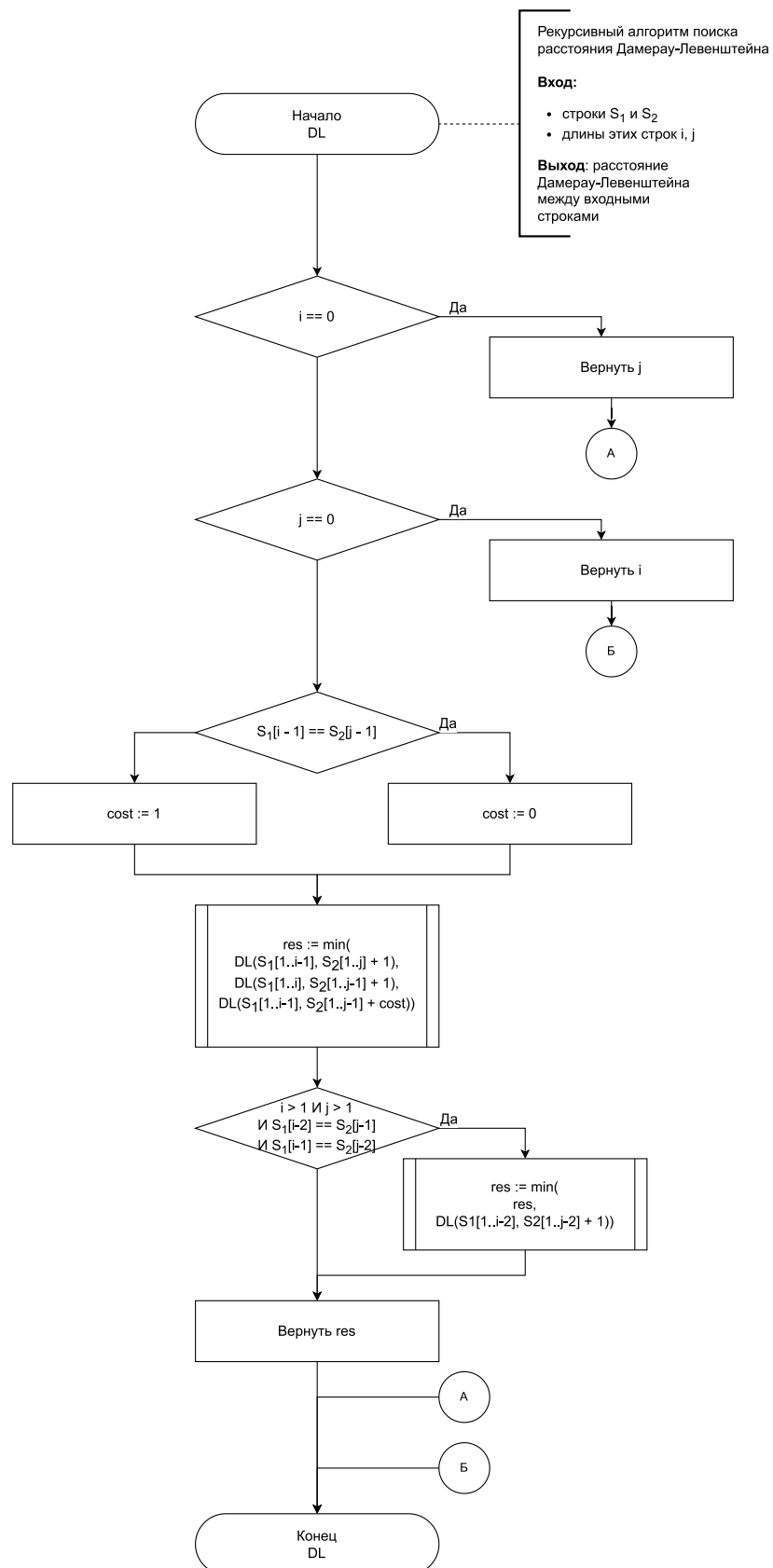


Рисунок 2.3 – Схема рекурсивного алгоритма Дамерау — Левенштейна

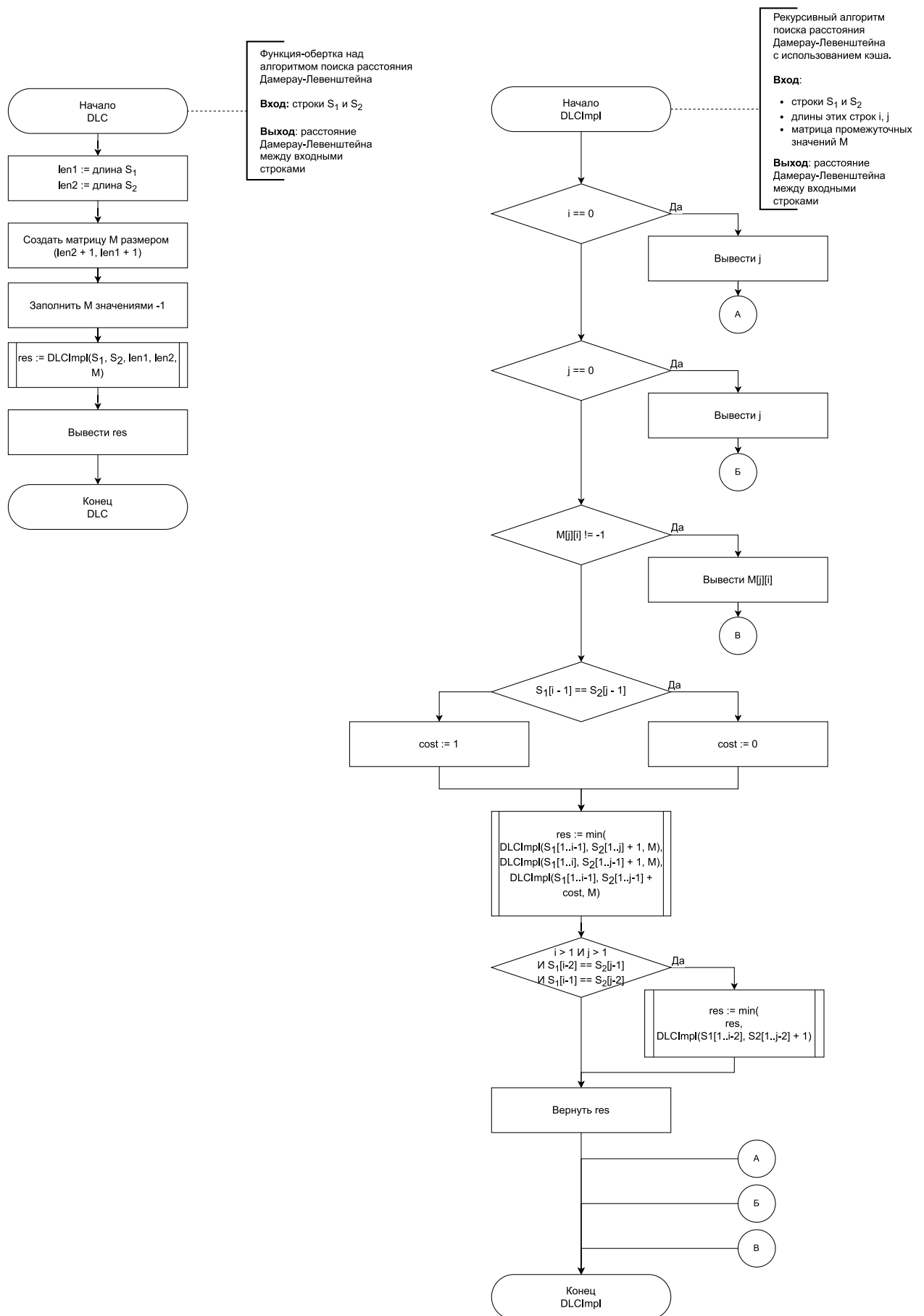


Рисунок 2.4 – Схема рекурсивного алгоритма Дамерау — Левенштейна с использованием кэша

## 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 программирования, используемого при написании данной лабораторной работы, был выбран C++ [3], так как в нем имеется контейнер `std::wstring`, представляющий собой массив символов `std::wchar_t`, и библиотека `<ctime>`, позволяющая производить замеры процессорного времени.

В качестве средства написания кода была выбрана кроссплатформенная среда разработки *Visual Studio Code* за счет того, что она предоставляет 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 Дamerau — Левенштейна;
- `measure.cpp` — файл, содержащий функции, измеряющие процессорное время выполнения реализуемых алгоритмов.

### 3.3 Реализация алгоритмов

Листинг 3.1 – Функция `min`, используемая в реализациях алгоритмов

```
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    }
25 }
```

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

```
1         else
2         {
3             int cost = word1[j - 1] != word2[i - 1];
4
5             mat[i][j] = min(
6                 mat[i - 1][j] + 1,
7                 mat[i][j - 1] + 1,
8                 mat[i - 1][j - 1] + cost);
9         }
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
    &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    {
```

### 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];
14
15             mat[i][j] = min(
16                 mat[i - 1][j] + 1,
17                 mat[i][j - 1] + 1,
18                 mat[i - 1][j - 1] + cost);
19
20             if (word1[j - 2] == word2[i - 1] && word1[j -
21                 1] == word2[i - 2])
22             {
23                 mat[i][j] = std::min(
24                     mat[i][j],
25                     mat[i - 2][j - 2] + 1);
26             }
27         }
28     }
29
30     if (verbose)
31         Matrix::Print(mat, word1, word2);
32
33     int res = mat[len2][len1];
34     Matrix::Free(mat, len2 + 1);
35
36     return res;
37 }
```

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[j - 1] != word2[i - 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 DamLevRecCachempl(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[j - 1] != word2[i - 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
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) @ 3.600 Гц.
- 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 «кошка» и «броненосец».



```

      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 реализованных алгоритмов производилось на строках длинами:

- 1 – 10 с шагом 1 для всех реализаций;
- 10 – 200 с шагом 25 только для нерекursивных реализаций.

В силу того, что время работы алгоритмов может колебаться в связи с различными процессами, происходящими в системе, для обеспечения более

точных результатов измерения для каждого алгоритма повторялись 500 раз, а затем бралось их среднее арифметическое значение.

На рисунке 4.2 показаны зависимости времени выполнения матричных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна от длин входящих строк.

На рисунке 4.3 показаны зависимости времени выполнения рекурсивных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна от длин входящих строк.

На рисунке 4.4 показаны зависимости времени выполнения рекурсивных реализаций алгоритма Дамерау — Левенштейна от длин входящих строк.

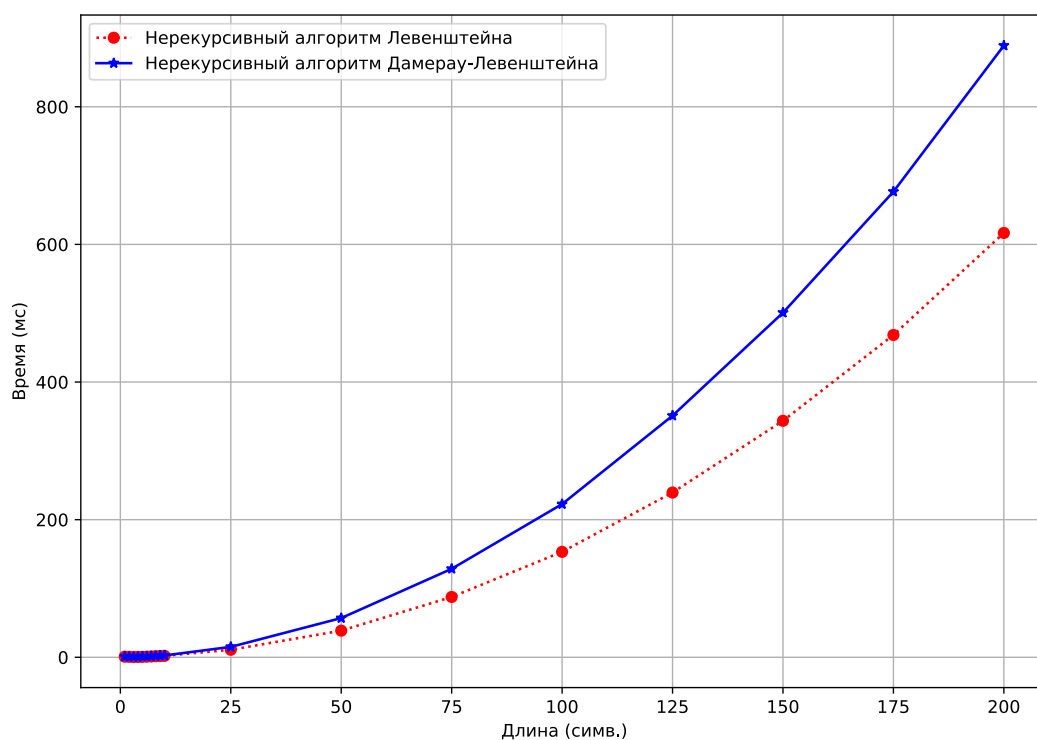


Рисунок 4.2 – Результат измерений времени работы нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна

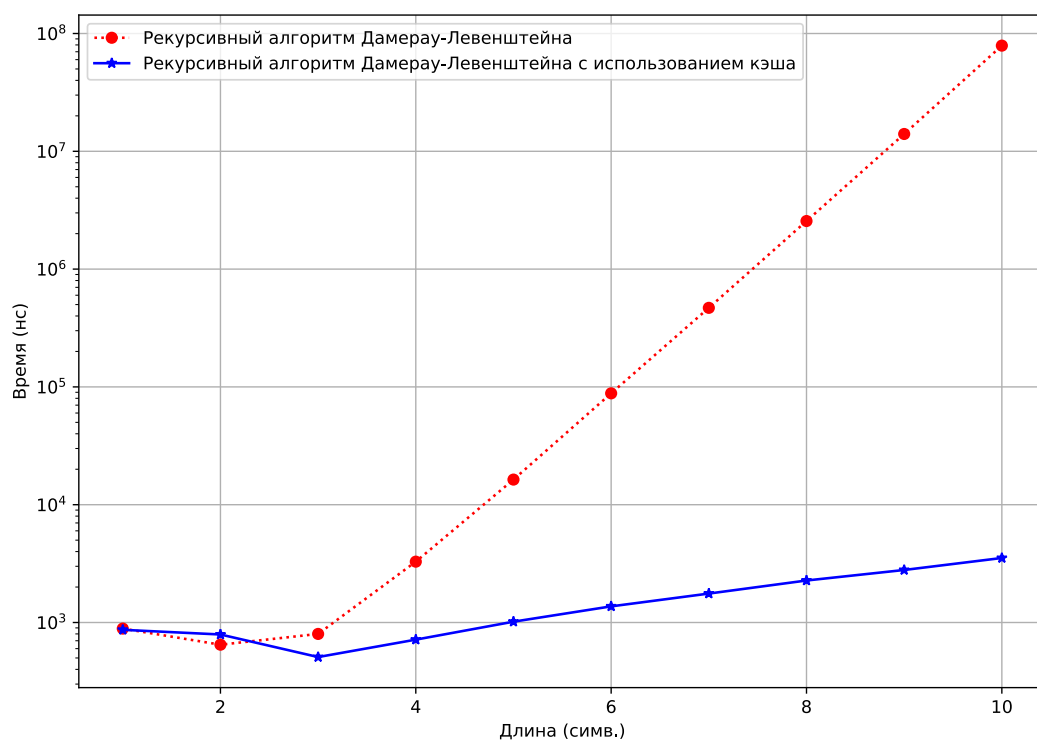


Рисунок 4.3 – Результат измерений времени работы рекурсивных реализаций алгоритма поиска расстояния Дамерау — Левенштейна

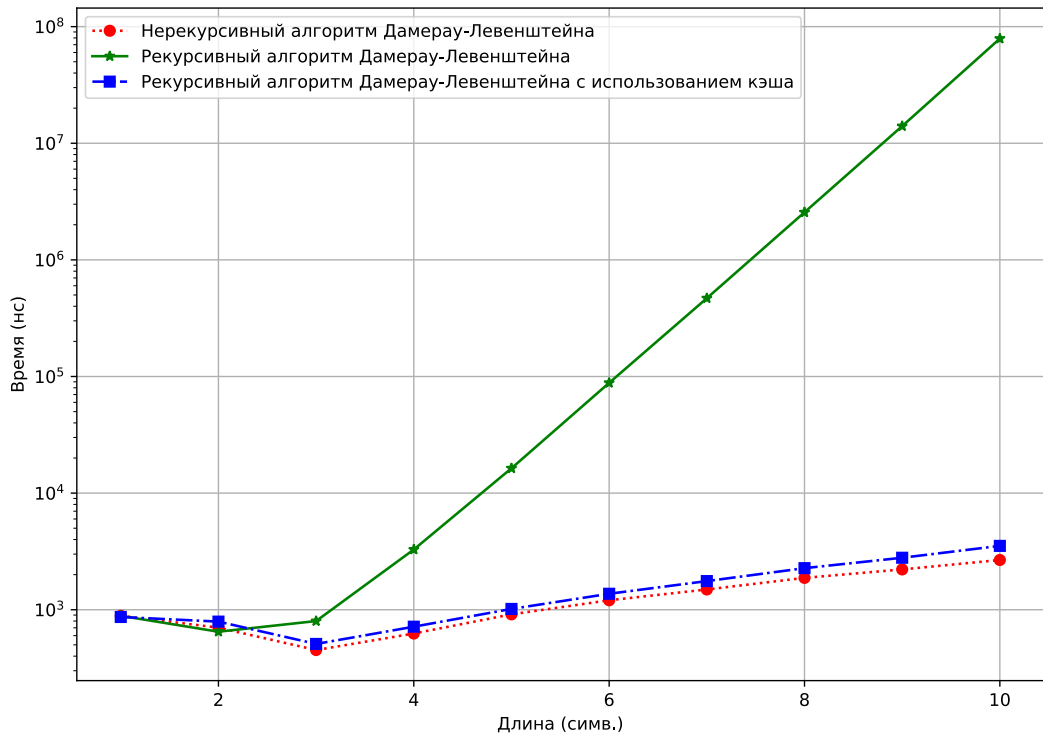


Рисунок 4.4 – Результат измерений времени работы реализаций алгоритмов поиска расстояния Дамерау — Левенштейна

## 4.4 Характеристики по памяти

Введем следующие обозначения:

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

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

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

где  $(m + 1) \cdot (n + 1) \cdot \text{size}(int)$  — размер матрицы,  
 $\text{size}(int ** )$  — размер указателя на матрицу,  
 $(m + 1) \cdot \text{size}(int*)$  — размер указателей на строки матрицы,  
 $(m + n) \cdot \text{size}(char)$  — размер двух входных строк,  
 $2 \cdot \text{size}(int)$  — размер переменных, хранящих длину строк,  
 $3 \cdot \text{size}(int)$  — размер дополнительных переменных.

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.

Сперва рассчитаем объем памяти, используемой каждым вызовом функции поиска расстояния Дамерау — Левенштейна:

$$M_{call} = (m + n) \cdot \text{size}(char) + 2 \cdot \text{size}(int) + 3 \cdot \text{size}(int) + 8 \quad (4.2)$$

где  $(m + n) \cdot \text{size}(char)$  — объем памяти, используемый для хранения двух строк,

$2 \cdot \text{size}(int)$  — размер двух входных строк,

$3 \cdot \text{size}(int)$  — размер дополнительных переменных,

8 байт — адрес возврата.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, поэтому максимальный расход памяти равен

$$M_{DLRec} = (m + n) \cdot M_{call} \quad (4.3)$$

где  $m + n$  — максимальная глубина стека вызовов,

$M_{call}$  — затраты по памяти для одного рекурсивного вызова.

Рекурсивная реализация алгоритма поиска расстояния Дамерау —

Левенштейна с кэшированием для хранения промежуточных значений использует матрицу (кэш), размер которой можно рассчитать следующим образом:

$$M_{cache} = (n + 1) \cdot (m + 1) \cdot \text{size}(int) + \\ + \text{size}(int ** ) + (m + 1) \cdot \text{size}(int*) \quad (4.4)$$

где  $(n + 1) \cdot (m + 1)$  — количество элементов в кэше,  
 $\text{size}(int **)$  — размер указателя на матрицу,  
 $(m + 1) \cdot \text{size}(int*)$  — размер указателя на строки матрицы.

Таким образом, затраты по памяти для рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с использованием кэша:

$$M_{DLRecCache} = M_{DLRec} + M_{cache} \quad (4.5)$$

## 4.5 Вывод

В результате исследования реализуемых алгоритмов по времени выполнения можно сделать следующие выводы:

- 1) При небольших длинах строк (длина  $< 5$  симв.) разница между временем выполнения нерекурсивных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна незначительна. Однако, при увеличении длины обрабатываемых строк алгоритм поиска расстояния Дамерау — Левенштейна выполняется на порядок дольше, что связано с обработкой дополнительного условия о перестановке символов (см. рис. 4.2).
- 2) Рекурсивная реализация алгоритма поиска расстояния Дамерау — Левенштейна с использованием кэша работает на порядок быстрее реализации поиска этого расстояния без кэширования (см. рис. 4.3).
- 3) Время работы матричной и рекурсивной с кэшем реализаций алгоритма поиска расстояния Дамерау — Левенштейна приблизительно

равны и выполняются на порядок быстрее в сравнении с рекурсивной реализацией поиска этого расстояния без кэширования (см. рис. 4.4).

В результате исследования алгоритмов по затрачиваемой памяти можно сделать вывод о том, что итеративные алгоритмы и рекурсивный алгоритм с кэшированием требуют больше памяти по сравнению с рекурсивным без оптимизаций. В реализациях, использующих матрицу, максимальный размер используемой памяти увеличивается пропорционально произведению длин строк, в то время как у рекурсивного алгоритма без кэширования объем затрачиваемой памяти увеличивается пропорционально сумме длин строк.

# Заключение

В результате выполнения лабораторной работы по исследованию алгоритмов поиска расстояния Левенштейна и Дameraу — Левенштейна были решены следующие задачи:

- 1) Описаны алгоритмы поиска расстояний Левенштейна и Дameraу — Левенштейна;
- 2) Разработаны и реализованы соответствующие алгоритмы;
- 3) Создан программный продукт, позволяющий протестировать реализованные алгоритмы;
- 4) Проведен сравнительный анализ процессорного времени выполнения реализованных алгоритмов.

— При небольших длинах строк (длина  $< 5$  симв.) разница между временем выполнения нерекурсивных реализаций алгоритмов Левенштейна и Дameraу — Левенштейна незначительна. При увеличении длин строк время работы матричного алгоритма поиска расстояния Дameraу — Левенштейна становится больше, в связи с обработкой условия о перестановке символов.

— Рекурсивный алгоритм поиска расстояния Дameraу — Левенштейна выполняется на порядок дольше, чем тот же алгоритм, использующий кэширование.

— Время работы матричного и рекурсивного с кэшированием алгоритмов поиска расстояния Дameraу — Левенштейна приблизительно равно.

- 5) Выполнена теоретическая оценка объема затрачиваемой памяти каждым из реализованных алгоритмов: нерекурсивные алгоритмы и рекурсивный алгоритм с кэшированием, требуют больше памяти по сравнению с рекурсивным, не использующим кэширование, так как максимальный размер использования памяти у матричных реализаций увеличивается пропорционально произведению длин входящих строк, а у рекурсивных — пропорционально их сумме.



## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 А. Погорелов Д., М. Таразанов А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна // Синергия Наук. 2019. URL: <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: <https://cyberleninka.ru/article/n/programmная-realizatsiya-nechetkogo-poiska-tekstovoy-informatsii-v-slovare-s-pomoschu-rasstoyaniya-levenshteyna>. (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: 25.09.2023).