



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)

Савинова М. Г.

(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

Введение	3
1 Аналитическая часть	4
1.1 Расстояние Левенштейна	4
1.2 Расстояние Дамерау – Левенштейна	5
2 Конструкторская часть	6
2.1 Требования к программному обеспечению	6
2.2 Требования к вводу	6
2.3 Разработка алгоритмов	6
2.4 Описание используемых типов данных	12
3 Технологическая часть	13
3.1 Средства реализации	13
3.2 Сведения о модулях программы	13
3.3 Реализация алгоритмов	14
3.4 Функциональные тесты	20
4 Исследовательская часть	21
4.1 Технические характеристики	21
4.2 Демонстрация работы программы	21
4.3 Временные характеристики	23
4.4 Характеристики по памяти	23
4.5 Вывод	27
Заключение	28
Список использованных источников	29

Введение

Расстояние Левенштейна — это метрика, используемая для измерения разницы между двумя строками. Она вычисляет минимальное количество односимвольных изменений (вставок, удалений или замен), необходимых для преобразования одной строки в другую.

Расстояние Левенштейна обычно используется в различных областях [1], в том числе:

- 1) **Проверка орфографии:** выявление и исправление ошибок для слов на основе их расстояния Левенштейна.
- 2) **Анализ последовательности ДНК:** измерение сходства между последовательностями ДНК позволяет исследователям сравнивать и анализировать генетические данные.
- 3) **Обработка естественного языка (НЛП):** используется в таких задачах, как классификация текстов, поиск информации и машинный перевод, для определения сходства между текстами.

Расстояние Дамерау – Левенштейна является расширением расстояния Левенштейна, которое не учитывает транспозиции. Дополнительная операция позволяет обрабатывать случаи, когда символы меняются местами или переупорядочиваются.

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 [2] — это минимальное количество редакторских операций вставки, замены и удаления, которые необходимо выполнить для преобразования одной строки в другую.

Каждая операция имеет свою цену(w). Редакционное предписание — это минимальная последовательность действий, которую необходимо выполнить для получения из первой строки вторую. Эта цена и есть искомое расстояние Левенштейна.

Введем следующие обозначения:

- 1) **I** (от англ. insert) — вставка ($w(\lambda, b) = 1$);
- 2) **R** (от англ. replace) — замена ($w(a, b) = 1, a \neq b$);
- 3) **D** (от англ. delete) — удаление ($w(a, \lambda) = 1$).

Также рассмотрим функцию $D(i, j)$: ее значением является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 (длиной M и N соответственно) рассчитывается по следующей рекуррентной формуле:

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

где сравнение символов строк S_1 и S_2 рассчитывается таким образом:

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

1.2 Расстояние Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна [1] — это мера разницы двух строк, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является расширением расстояния Левенштейна, поскольку помимо трех базовых операций содержит еще операцию транспозиции T (от англ. transposition).

Расстояние Дамерау – Левенштейна определяется следующей рекуррентной формуле:

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

Вывод

Формулы для вычисления расстояний Левенштейна и Дамерау – Левенштейна задаются **рекуррентно**, поэтому алгоритмы для нахождения их расстояний можно реализовать как *итеративно*, так и *рекурсивно*.

2 Конструкторская часть

В данном разделе будут реализованы схемы алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна, приведено описание используемых типов данных, а также описана структура программного обеспечения.

2.1 Требования к программному обеспечению

К программе предъявлен ряд требований:

- наличие интерфейса для выбора действий;
- возможность ввода строк;
- возможность обработки строк, включающих буквы как на латинице, так и на кириллице;
- возможность произвести замеры процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна.

2.2 Требования к вводу

- 1) На вход подаются две строки;
- 2) Буква нижнего и верхнего регистра считаются разными символами;
- 3) Строки могут включать как символы латиницы, так и кириллицы.

2.3 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2 .

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2–2.5 представлены схемы алгоритмов поиска Дameraу – Левенштейна.

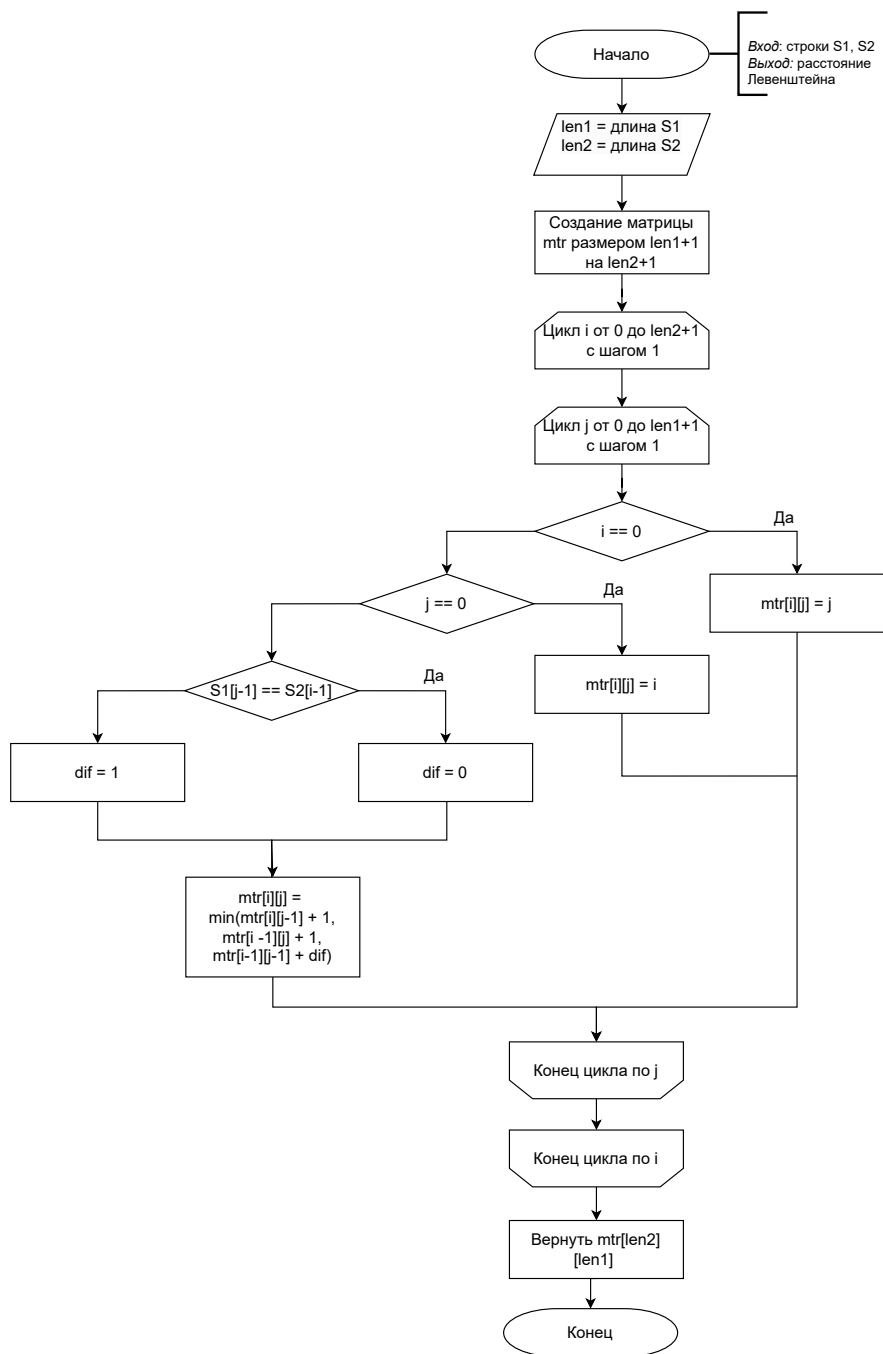


Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

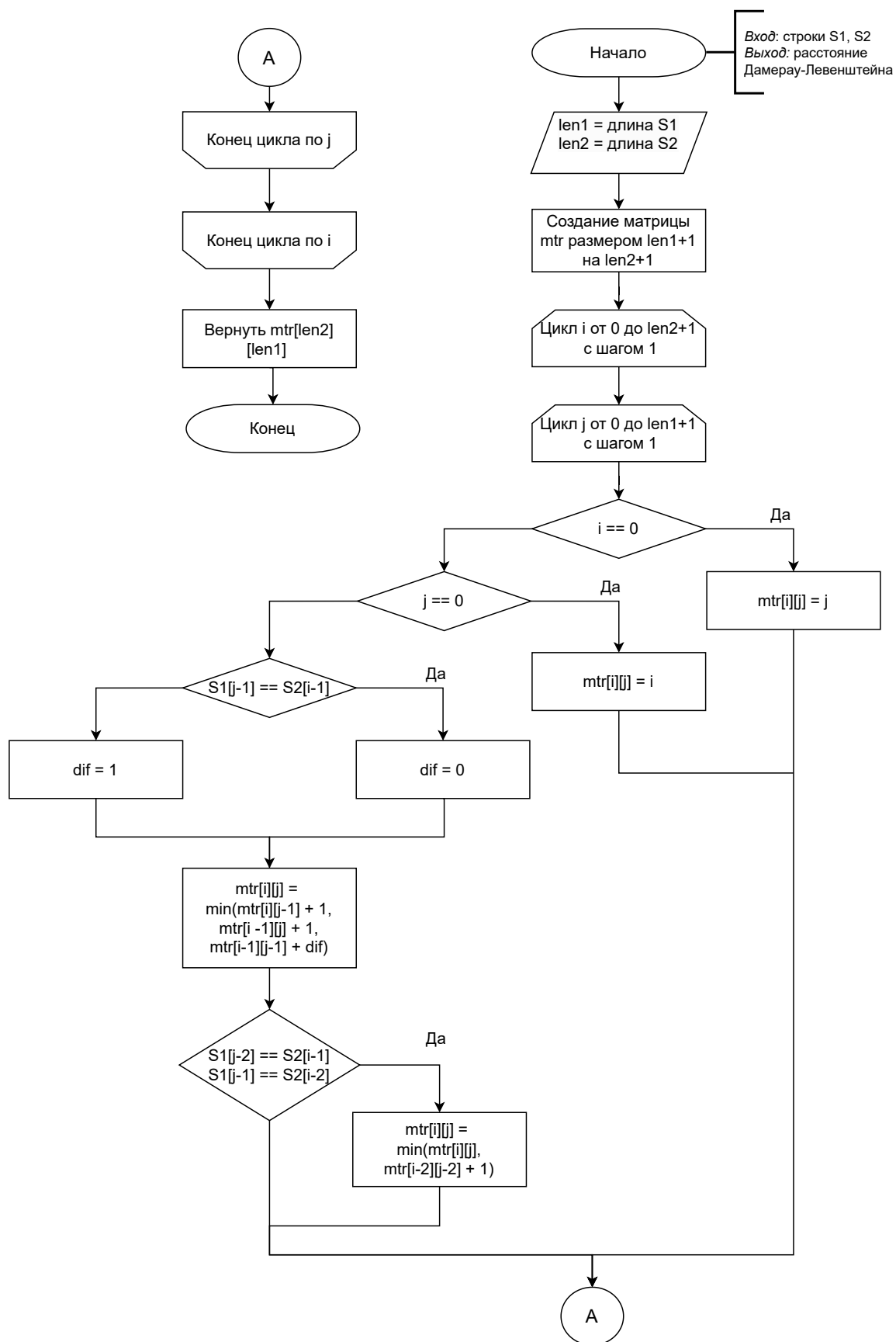


Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна

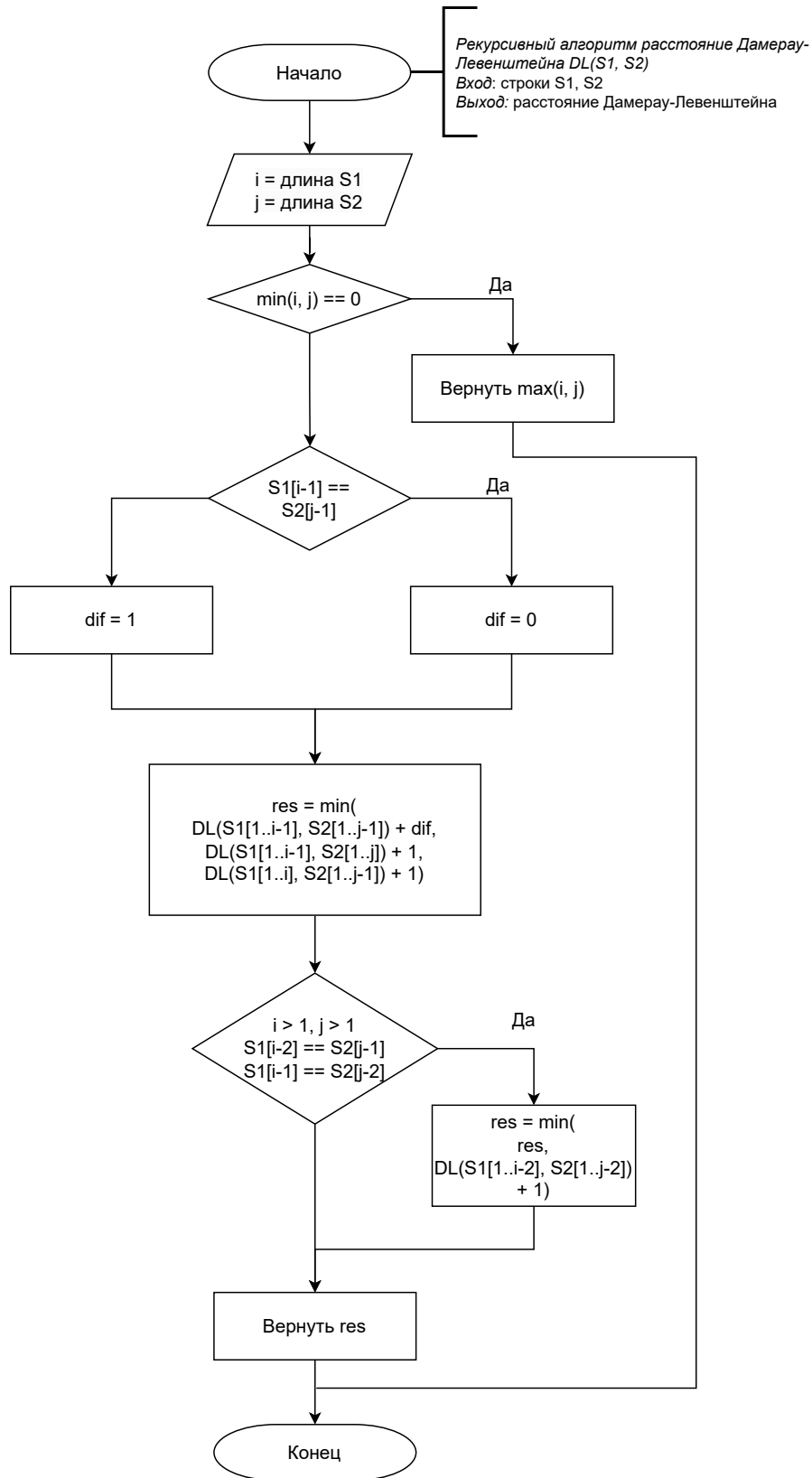


Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна

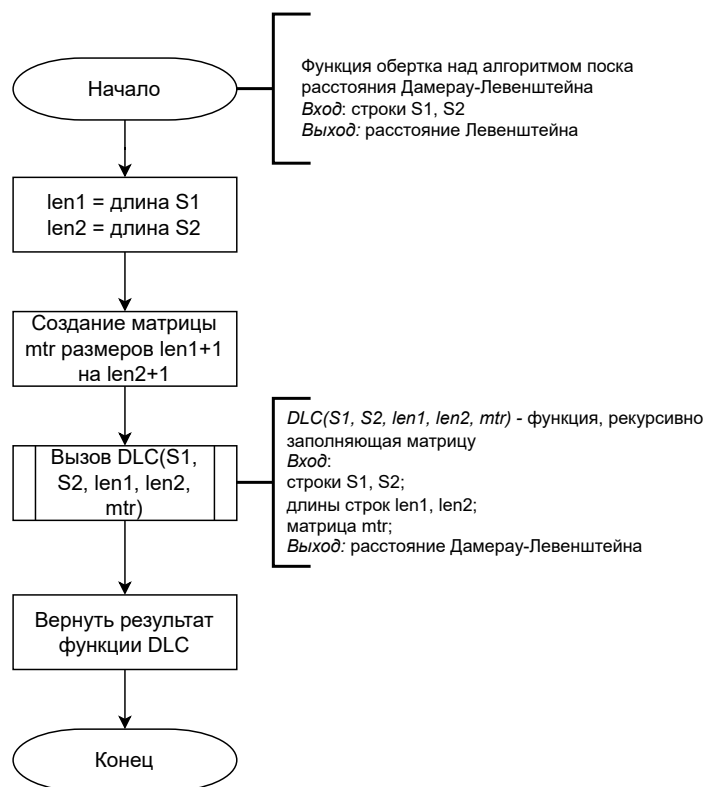


Рисунок 2.4 – Схема вызова рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с кешированием

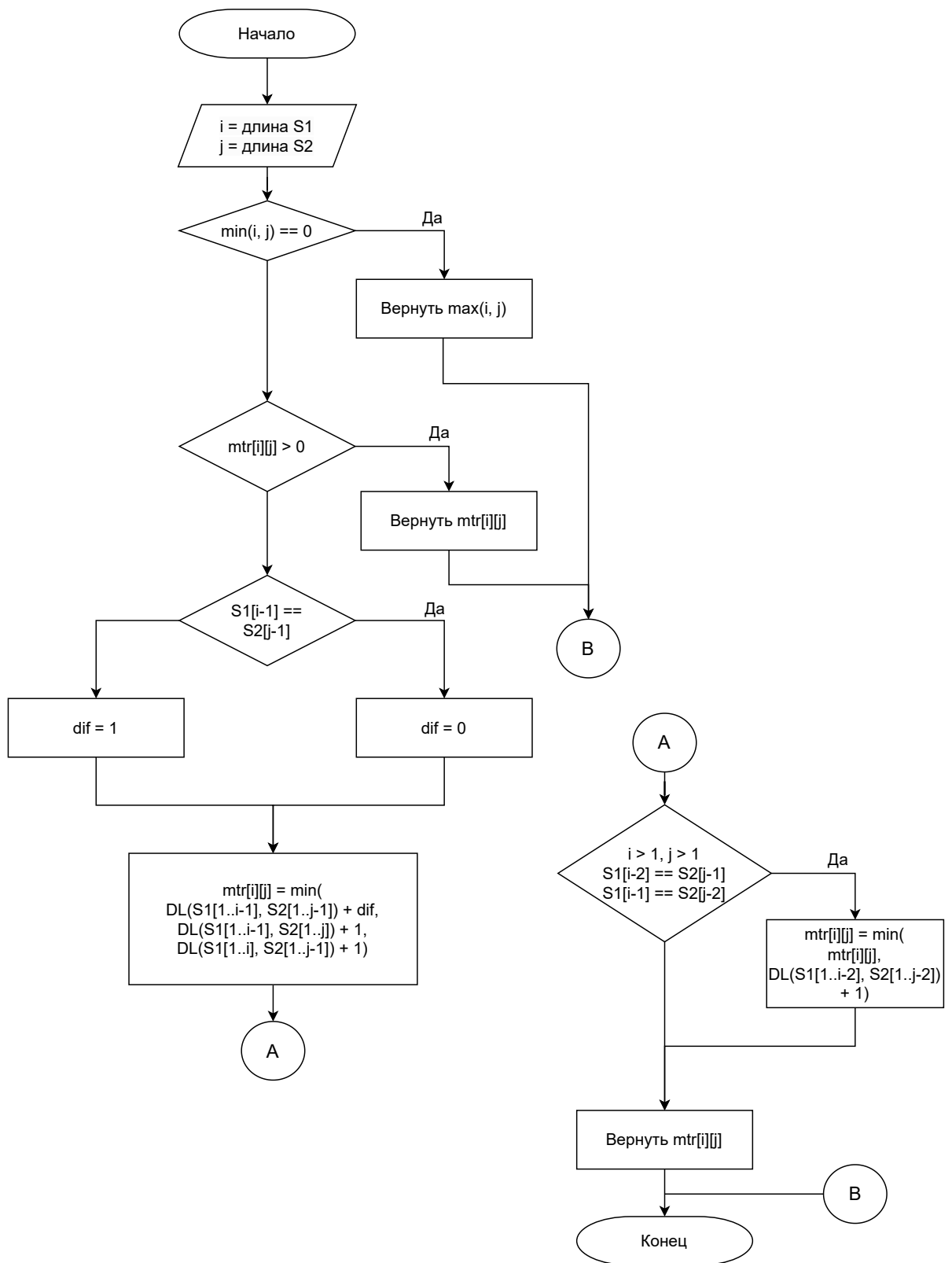


Рисунок 2.5 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с кешированием

2.4 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- *строка* — массив типа *wchar_t* размером длины строки;
- *длина строки* — целое число типа *int*;
- *матрица* — двумерный массив значений типа *int*.

Вывод

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

3 Технологическая часть

В данном разделе будут приведены средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык `C++` [3], так как в нем есть стандартная библиотека `ctime` [4], которая позволяет производить замеры процессорного времени выполнения программы; тип данных `std::wstring`, позволяющий хранить как кириллические символы, так и латинские; В качестве среды разработки был выбран `VisualStudioCode`: он является кроссплатформенным и предоставляет полный функционал для проектирования и отладки кода.

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

- `main.cpp` — файл, содержащий точку входа в программу, из которой происходит вызов алгоритмов по разработанному интерфейсу;
- `algorithms.cpp` — файл содержит функции поиска расстояния Левенштейна и
Дамерау – Левенштейна;
- `matrix.cpp` — файл содержит функции динамического выделения и очищения памяти для матрицы, а так же ее вывод на экран;
- `measure.cpp` — файл содержит функции, замеряющее процессорное время для алгоритмов поиска расстояния Левенштейна и Дамерау – Левенштейна;

3.3 Реализация алгоритмов

В листингах 3.2–3.7 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау – Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы (начало)

```
1 int Algs::notRecursiveLev(wstring &word1, wstring &word2, bool  
   print) {  
2  
3     int len1 = word1.length();  
4     int len2 = word2.length();  
5  
6     int** mtr = Matrix::allocate(len2 + 1, len1 + 1);  
7  
8     if (!mtr)  
9         return 0;  
10  
11    for (int i = 0; i <= len2; ++i) {  
12  
13        for (int j = 0; j <= len1; ++j) {  
14  
15            if (i == 0)  
16                mtr[i][j] = j;  
17            else if (j == 0)  
18                mtr[i][j] = i;  
19            else {  
20                int dif = (word1[j - 1] == word2[i - 1]) ? 0 :  
21                    1;  
22                mtr[i][j] = min(mtr[i - 1][j] + 1,  
23                               min(mtr[i][j - 1] + 1, mtr[i -  
24                                   1][j - 1] + dif));  
25            }  
26        }  
    }  
}
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна с использованием матрицы (конец)

```
1     if (print)
2         Matrix::print(mtr, word1, word2);
3
4     int res = mtr[len2][len1];
5     Matrix::release(mtr, len2 + 1);
6
7     return res;
8 }
```

Листинг 3.3 – Функция нахождения расстояния Дameraу – Левенштейна с использованием матрицы (начало)

```
1 int Algs::notRecursiveDamLev(wstring &word1, wstring &word2,
2     bool print) {
3
4     int len1 = word1.length();
5     int len2 = word2.length();
6
7     int** mtr = Matrix::allocate(len2 + 1, len1 + 1);
8
9     if (!mtr)
10         return 0;
11
12     for (int i = 0; i <= len2; ++i) {
13
14         for (int j = 0; j <= len1; ++j) {
15
16             if (i == 0)
17                 mtr[i][j] = j;
18             else if (j == 0)
19                 mtr[i][j] = i;
20             else {
21                 int dif = (word1[j - 1] == word2[i - 1]) ? 0 :
22                     1;
23                 mtr[i][j] = min(mtr[i - 1][j] + 1,
24                     min(mtr[i][j - 1] + 1, mtr[i -
25                         1][j - 1] + dif));
```

Листинг 3.4 – Функция нахождения расстояния Дамерау – Левенштейна с использованием матрицы (конец)

```
1         if (word1[j - 2] == word2[i - 1] && word1[j -  
2             1] == word2[i - 2])  
3             mtr[i][j] = min(mtr[i][j], mtr[i - 2][j -  
4                 2] + 1);  
5         }  
6     }  
7 }  
8  
9     if (print)  
10         Matrix::print(mtr, word1, word2);  
11  
12     int res = mtr[len2][len1];  
13     Matrix::release(mtr, len2 + 1);  
14  
15     return res;  
16 }
```


Листинг 3.5 – Функция нахождения расстояния Дameraу – Левенштейна рекурсивно

```
1 int Algs::recursive(wstring &word1, wstring &word2, int ind1,
2   int ind2) {
3     if (min(ind1, ind2) == 0)
4       return max(ind1, ind2);
5
6     int dif = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;
7
8     int res = min(recursive(word1, word2, ind1 - 1, ind2 - 1) +
9       dif,
10      min(recursive(word1, word2, ind1 - 1, ind2) +
11        1,
12        recursive(word1, word2, ind1, ind2 - 1) +
13        1));
14
15     if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -
16       2] && word1[ind1 - 2] == word2[ind2 - 1])
17       res = min(res, recursive(word1, word2, ind1 - 2, ind2 -
18       2) + 1);
19
20     return res;
21 }
```

Листинг 3.6 – Функция вызова рекурсивного алгоритма с кешированием для поиска расстояния Дамерау – Левенштейна

```
1 int Algs::recursiveCash_Decor(wstring& word1, wstring& word2,  
    bool print) {  
2  
3     int len1 = word1.length();  
4     int len2 = word2.length();  
5  
6     int** cash = Matrix::allocate(len2 + 1, len1 + 1, true);  
7  
8     if (!cash)  
9         return 0;  
10  
11     int res = recursiveCash(word1, word2, len1, len2, cash);  
12  
13     if (print)  
14         Matrix::print(cash, word1, word2);  
15  
16     Matrix::release(cash, len2 + 1);  
17  
18     return res;  
19 }
```

Листинг 3.7 – Функция нахождения расстояния Дameraу – Левенштейна рекурсивно с кешированием

```
1 int Algs::recursiveCash(wstring &word1, wstring &word2, int
  ind1, int ind2, int** cash) {
2
3     if (cash[ind2][ind1])
4         return cash[ind2][ind1];
5
6     if (min(ind1, ind2) == 0)
7         return cash[ind2][ind1] = max(ind1, ind2);
8
9     int dif = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;
10
11    int res = min(recursiveCash(word1, word2, ind1 - 1, ind2 -
      1, cash) + dif,
12                  min(recursiveCash(word1, word2, ind1 - 1,
      ind2, cash) + 1,
13                      recursiveCash(word1, word2, ind1, ind2 -
      1, cash) + 1));
14
15    if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -
      2] && word1[ind1 - 2] == word2[ind2 - 1])
16        res = min(res, recursiveCash(word1, word2, ind1 - 2,
      ind2 - 2, cash) + 1);
17
18    cash[ind2][ind1] = res;
19
20    return res;
21 }
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау–Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные		Расстояние и алгоритм			
Строка 1	Строка 2	Левенштейна	Дамерау – Левенштейна		
		Итеративный	Итеративный	Рекурсивный	
				Без кеша	С кешем
а	Ь	1	1	1	1
а	а	0	0	0	0
кот	скат	2	2	2	2
кот	кто	2	1	1	1
Австралия	Австрия	2	2	2	2
кот	ток	2	2	2	2
слон	слоны	1	1	1	1

Вывод

Были реализованы и протестированы алгоритмы поиска расстояния Левенштейна итеративно, а также поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивного с кешированием. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

- Процессор: AMD Ryzen 5 5500U – 2.10 GHz;
- Оперативная память: 16 ГБайт;
- Операционная система: Windows 10 Pro 64-разрядная система версии 22H2.

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного ПО, а именно показаны результаты работы алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна на примере двух строк *«секста»* и *«септима»*.

```

        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 кэшом;
2. Замерить время и память для реализованных алгоритмов;
0. Выход

Выберете пункт (0-2): 1

Введите 1е слово: секста
Введите 2е слово: септима

Минимальное кол-во операций:

      с  е  к  с  т  а
    0  1  2  3  4  5  6
с  1  0  1  2  3  4  5
е  2  1  0  1  2  3  4
п  3  2  1  1  2  3  4
т  4  3  2  2  2  2  3
и  5  4  3  3  3  3  3
м  6  5  4  4  4  4  4
а  7  6  5  5  5  5  4
1) Нерекursивный Левенштейна: 4

      с  е  к  с  т  а
    0  1  2  3  4  5  6
с  1  0  1  2  3  4  5
е  2  1  0  1  2  3  4
п  3  2  1  1  2  3  4
т  4  3  2  2  2  2  3
и  5  4  3  3  3  3  3
м  6  5  4  4  4  4  4
а  7  6  5  5  5  5  4
2) Нерекursивный Дамерау-Левенштейна: 4
3) Рекурсивный Дамерау-Левенштейна без кэша: 4
4) Рекурсивный Дамерау-Левенштейна с кэшом: 4

```

Рисунок 4.1 – Демонстрация работы программы

4.3 Временные характеристики

Все реализованный алгоритмы сравнивались на случайно сгенерированных строках длиной:

- 0–10 с шагом 1 для всех алгоритмов;
- 10–200 с шагом 10 для нерекурсивных и рекурсивного с кешированием.

Поскольку замеры по времени имеют некоторую погрешность, для каждой строки и каждой реализации алгоритма замеры производились 1000 раз, а затем вычислялось среднее арифметическое значение.

На рисунке 4.2 представлен график, иллюстрирующий зависимость времени работы от длины строк для рекурсивных реализаций алгоритмов поиска расстояния Дамерау – Левенштейна с кешем и без.

На рисунке 4.3 представлен график, иллюстрирующий зависимость времени работы от длины строк для матричных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна.

На рисунке 4.4 представлен график, иллюстрирующий зависимость времени работы от длины строк для матричных реализаций. На графике видно, что матричные реализации работают значительно быстрее рекурсивной реализации с кешированием.

4.4 Характеристики по памяти

Введем следующие обозначения:

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

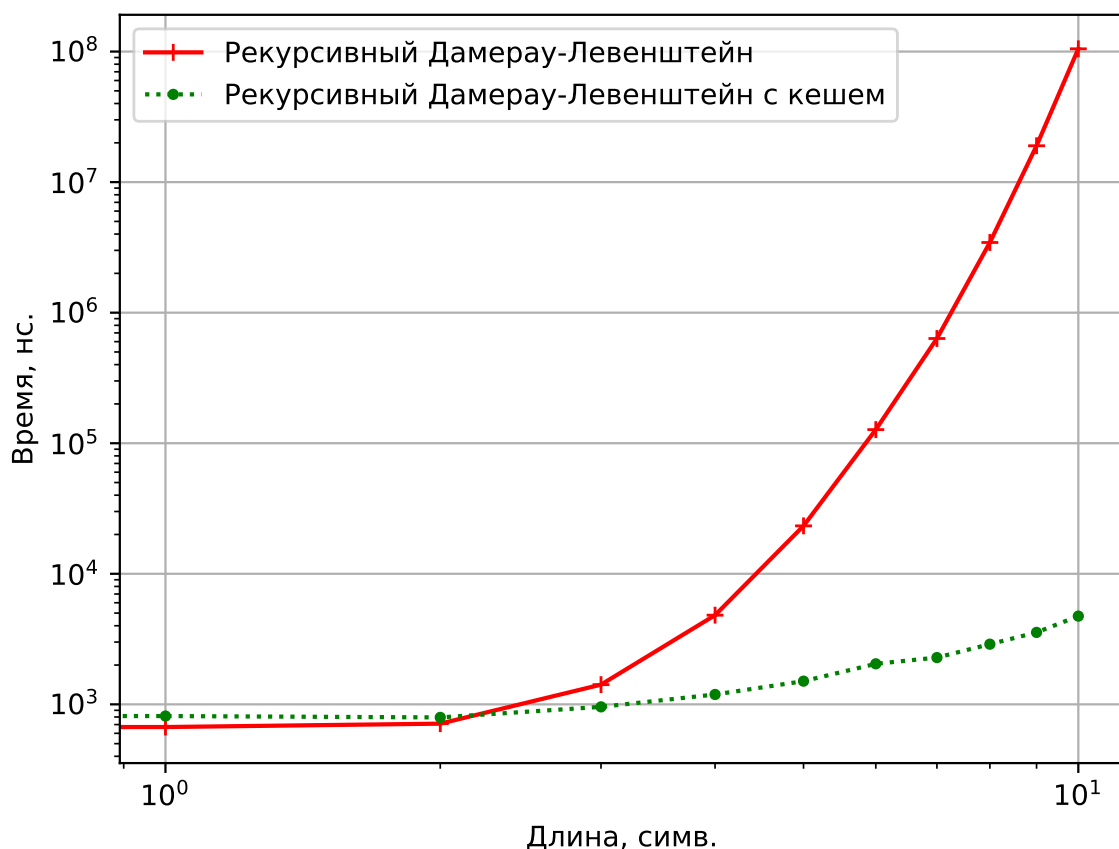


Рисунок 4.2 – Сравнение по времени рекурсивных реализаций алгоритмов поиска расстояния Дамерау – Левенштейна с кешем и без

Использование памяти при **итеративной реализации** алгоритма поиска расстояния Левенштейна теоретически равно:

$$M_{iter} = (m + 1) \cdot (n + 1) \cdot size(int) + (n + m) \cdot size(char) + 5 \cdot size(int) + size(int **) + (n + 1) \cdot size(int*), \quad (4.1)$$

где:

- $(n + 1) \cdot (m + 1) \cdot size(int)$ — хранение матрицы;
- $(n + m) \cdot size(char)$ — хранение двух строк;
- $2 \cdot size(int)$ — хранение размеров строк;
- $3 \cdot size(int)$ — дополнительные переменные;
- $size(int **) + (n + 1) \cdot size(int*)$ — указатель на матрицу;

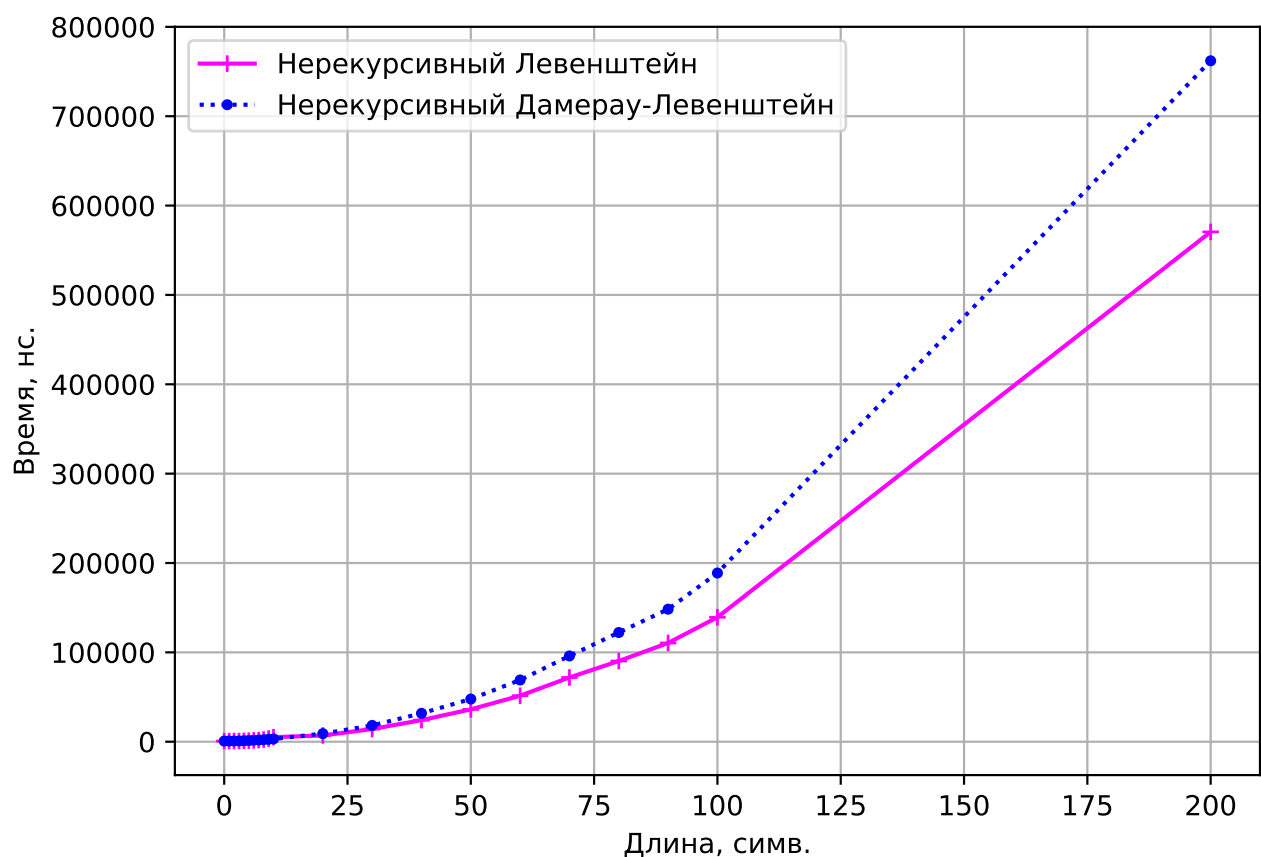


Рисунок 4.3 – Сравнение по времени матричных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна

Использование памяти при **итеративной реализации** алгоритма поиска расстояния Дамерау – Левенштейна идентично формуле 4.1.

Рассчитаем затраты по памяти для **рекурсивного** алгоритма поиска расстояния Дамерау – Левенштейна (*для каждого вызова*):

$$M_{call} = (m + n) \cdot size(char) + 4 \cdot size(int) + 8 \quad (4.2)$$

где:

- $(n + m) \cdot size(char)$ — хранение двух строк;
- $2 \cdot size(int)$ — хранение размеров строк;
- $2 \cdot size(int)$ — дополнительные переменные;
- 8 байт — адрес возврата.

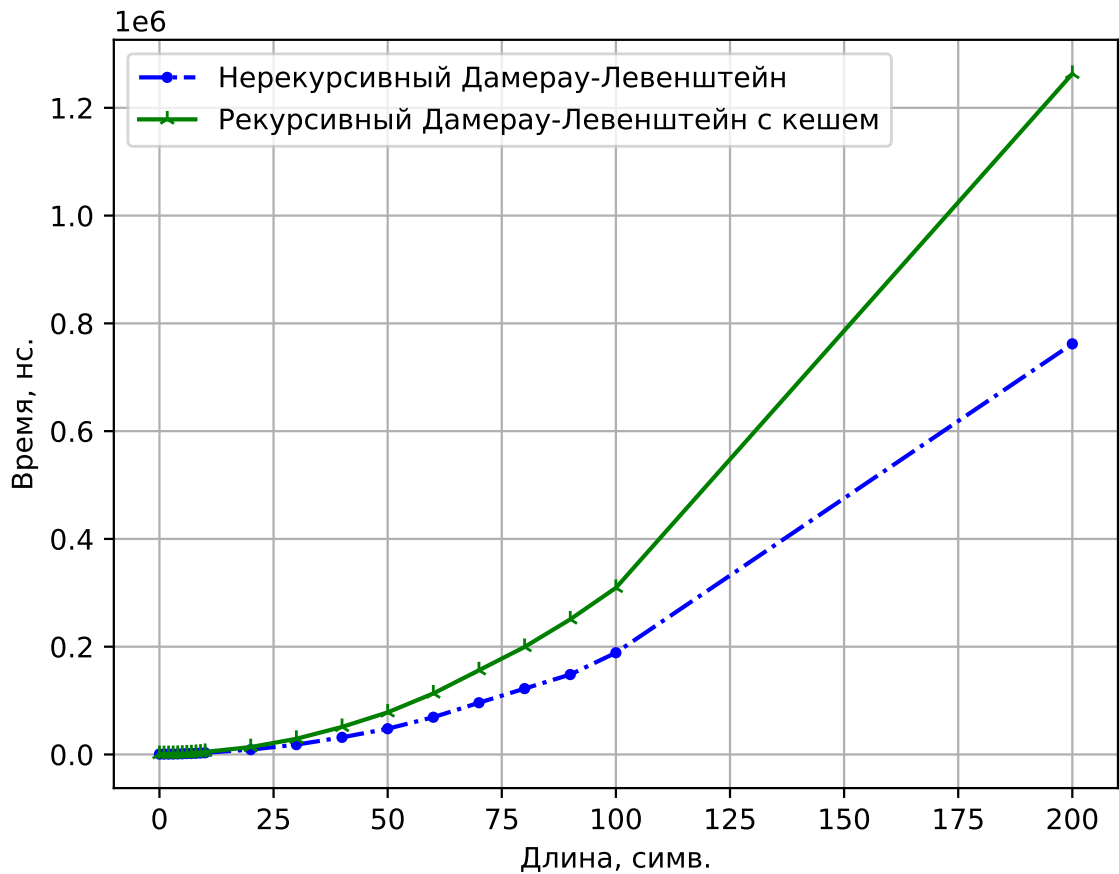


Рисунок 4.4 – Сравнение по времени матричной реализации алгоритма и рекурсивной реализации с кешем поиска расстояния Дамерау – Левенштейна

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

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

где:

- $n + m$ — максимальная глубина стека;
- M_{call} — затраты по памяти для одного рекурсивного вызова.

Для рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с использованием кеша необходимо подсчитать размер самого кеша:

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

где:

- $(m + 1) \cdot (n + 1)$ — количество элементов в кеше;
- $size(int **) + (n + 1) \cdot size(int*)$ — хранение указателей.

Затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с учетом кеша:

$$M_{recCash} = M_{rec} + M_{cash} \quad (4.5)$$

4.5 Вывод

По времени выполнения:

- 1) При малых длинах строк (< 5) рекурсивные реализации с кешем и без для поиска расстояния Дамерау – Левенштейна имеют приблизительно одинаковое время работы, но с увеличением длины строки реализация без кеша выполняется на порядок дольше, поскольку не происходит повторное вычисление значений (см рис. 4.2);
- 2) Разница между итеративными реализациями алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна незначительна, и обусловлена она дополнительным условием на проверку равенства соседних символов для расстояния Дамерау – Левенштейна (см рис. 4.3);
- 3) Итеративная реализация работает на порядок быстрее рекурсивной с кешем для поиска расстояния Дамерау – Левенштейна (см рис. 4.4).

По затрачиваемой памяти итеративные алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в итеративной реализации растет как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

В результате выполнения лабораторной работы при исследовании алгоритмов нахождения расстояний Левенштейна и Дamerau – Левенштейна были выполнены следующие задачи:

- 1) Описаны алгоритмы поиска расстояния Левенштейна и Дamerau – Левенштейна;
- 2) Разработаны и реализованы соответствующие алгоритмы;
- 3) Создан программный продукт, позволяющий протестировать реализованные алгоритмы;
- 4) Проведен сравнительный анализ процессорного времени выполнения реализованных алгоритмов;
- 5) Проведен сравнительный анализ затрачиваемой алгоритмами памяти.

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 И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163.
- 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).
- 4 C library function clock() [Электронный ресурс]. — Режим доступа: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm (дата обращения: 25.09.2023).