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 «Анализы Алгоритмов»

на тему: «Редакционные расстояния между строками»

Студент группы ИУ7-54Б	(Подпись, дата)	Булдаков М. Ю. (Фамилия И.О.)
Преподаватель	(Подпись, дата)	Волкова Л. Л. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

Расстояние Левенштейна – минимальное количество редакционных операций, которое необходимо для преобразования одной строки в другую. Редакционными операциями являются:

— I – вставка одного символа (insert);

— M – удаление (match);

— R – замена (replace).

Также обозначим совпадение как M (match).

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна, отличается от него добавлением операции транспозиции (перестановки).

Редакционные расстояния применяются для решения следующих задач:

— исправление ошибок в словах;

— обучение языковых моделей (расстояние Левенштейна вводится как метрика);

— сравнение геномов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является исследование алгоритмов вычисляющих расстояние Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1) изучить алгоритмы, вычисляющие расстояния Левенштейна и Дамерау-Левенштейна;

2) разработать программное обеспечение, реализующее следующие алгоритмы:

    — нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

    — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна без кеширования;

— рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешированием;

— нерекурсивный алгоритм поиска расстояния Левенштейна.

3) выбрать инструменты для реализации и замера процессорного времени выполнения алгоритмов, описанных выше;

4) проанализировать затраты реализаций алгоритмов по времени и по памяти.

1 Аналитическая часть

Каждая редакционная операция имеет свой штраф, который определяет стоимость данной операции. В общем случае:

— $m(a, b)$ – цена замены символа a на b, при $a \neq b$;

— $m(\lambda, a)$ – цена вставки символа a;

— $m(a, \lambda)$ – цена удаления символа a.

Для решения задачи о редакционном расстоянии, необходимо найти последовательность операций, минимизирующую сумму штрафов.

1.1 Расстояние Левенштейна

При вычислении расстояния Левенштейна будем считать стоимость каждой редакционной операции равной 1:

— $m(a, b) = 1$;

— $m(\lambda, a) = 1$;

— $m(a, \lambda) = 1$.

При этом если символы совпадают, то штраф равен 0, т. е. $m(a, a) = 0$.

Пусть S_1 и S_2 – две строки (длинной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно вычислить по следующей рекуррентной формуле:

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

Значение $m(a,b)$ можно рассчитывать по следующей формуле:

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

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 малоэффективна, поскольку множество промежуточных значений вычисляются несколько раз. Используя матрицу $A_{(M+1)\times(N+1)}$ для хранения промежуточных значений, сведем задачу к итерационному заполнению матрицы $A_{(M+1)\times(N+1)}$ значениями $D(i,j)$. Т. о. значение в ячейке $[i,j]$ равно значению $D(S_1[1...i], S_2[1...j])$.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна модифицирует расстояние Левенштейна, добавляя ко всем перечисленным операциям, операцию перестановки соседних символов. Штраф новой операции также составляет 1.

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

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

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешем

Используя кеш, рекурсивный алгоритм вычисления расстояния по формуле (1.3) можно оптимизировать по времени выполнения. В качестве кеша используется матрица. Суть данной оптимизации заключается в сокращении числа лишних операций, производимых над одними и теми же подстроками несколько раз. В случае, если для текущих подстрок, значение расстояния отсутствует в кеше, то оно вычисляется с помощью рекурсивного алгоритма и заносится в матрицу. Если же значение присутствует в кеше, то алгоритм сразу переходит к следующему шагу.

1.2.2 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием малоэффективна по времени при больших M и N. Можно свести задачу вычисления расстояния Дамерау-Левенштейна к итерационному заполнению матрицы промежуточными значениями $D(i, j)$. При этом матрица будет иметь размер $(M + 1) \times (N + 1)$.

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна, поскольку данные расстояния могут быть вычислены с помощью рекуррентных формул, то алгоритмы могут быть реализованы рекурсивно и итеративно.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

1) На вход подаются две строки, которые могут быть пустыми.

2) Буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

1) Обрабатывать корректно любые входные строки.

2) В результате программа должна вывести число – расстояние Левенштейна (Дамерау-Левенштейна).

3) Возможность обработки строк, включающих буквы как на латинице, так и на кириллице.

4) Наличие функциональности замера процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

2.3 Разработка алгоритмов

На вход алгоритмов подаются строки $S1$ и $S2$.

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

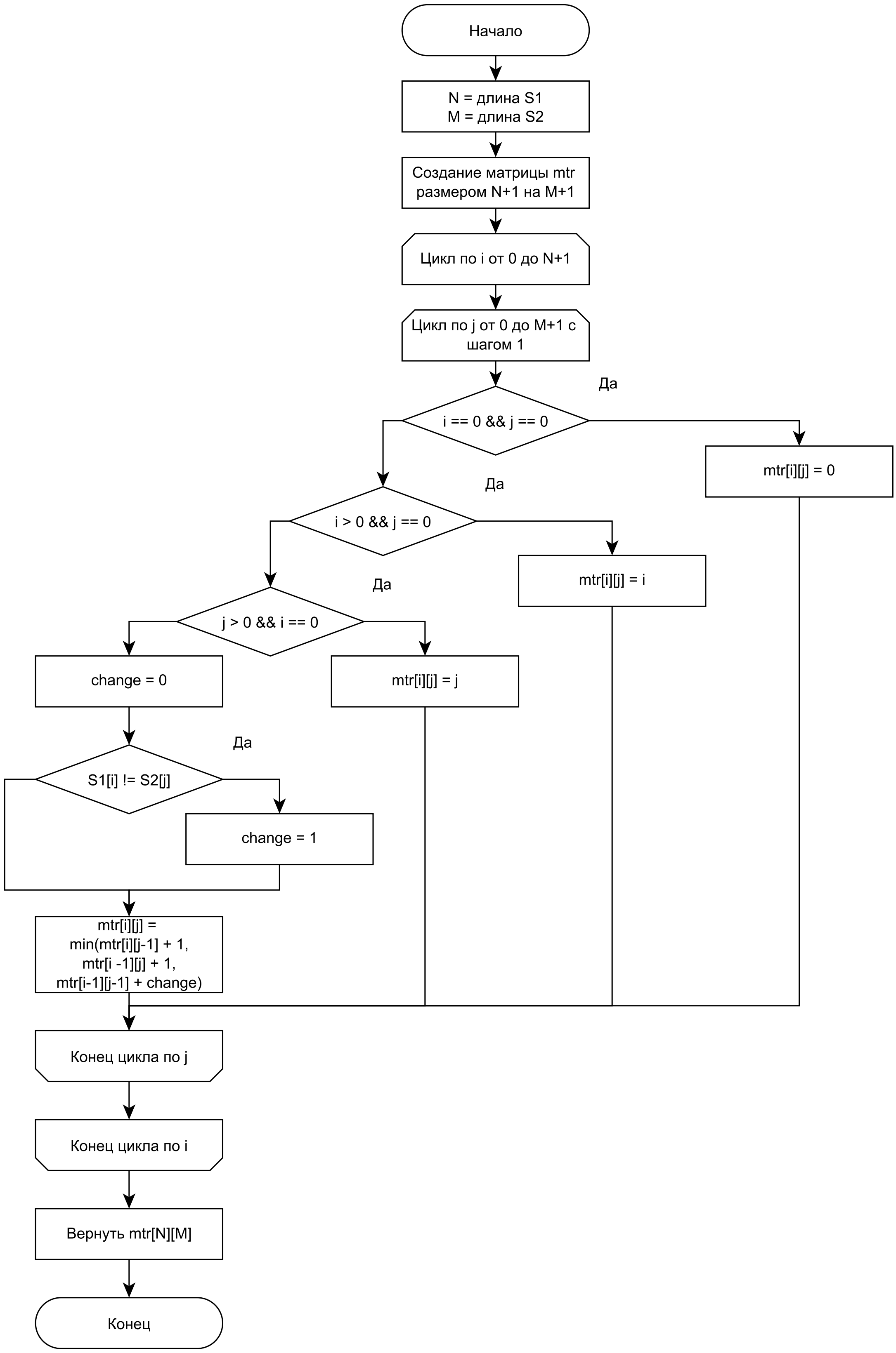

Рисунок 2.1 – Нерекурсивный алгоритм нахождения расстояния Левенштейна

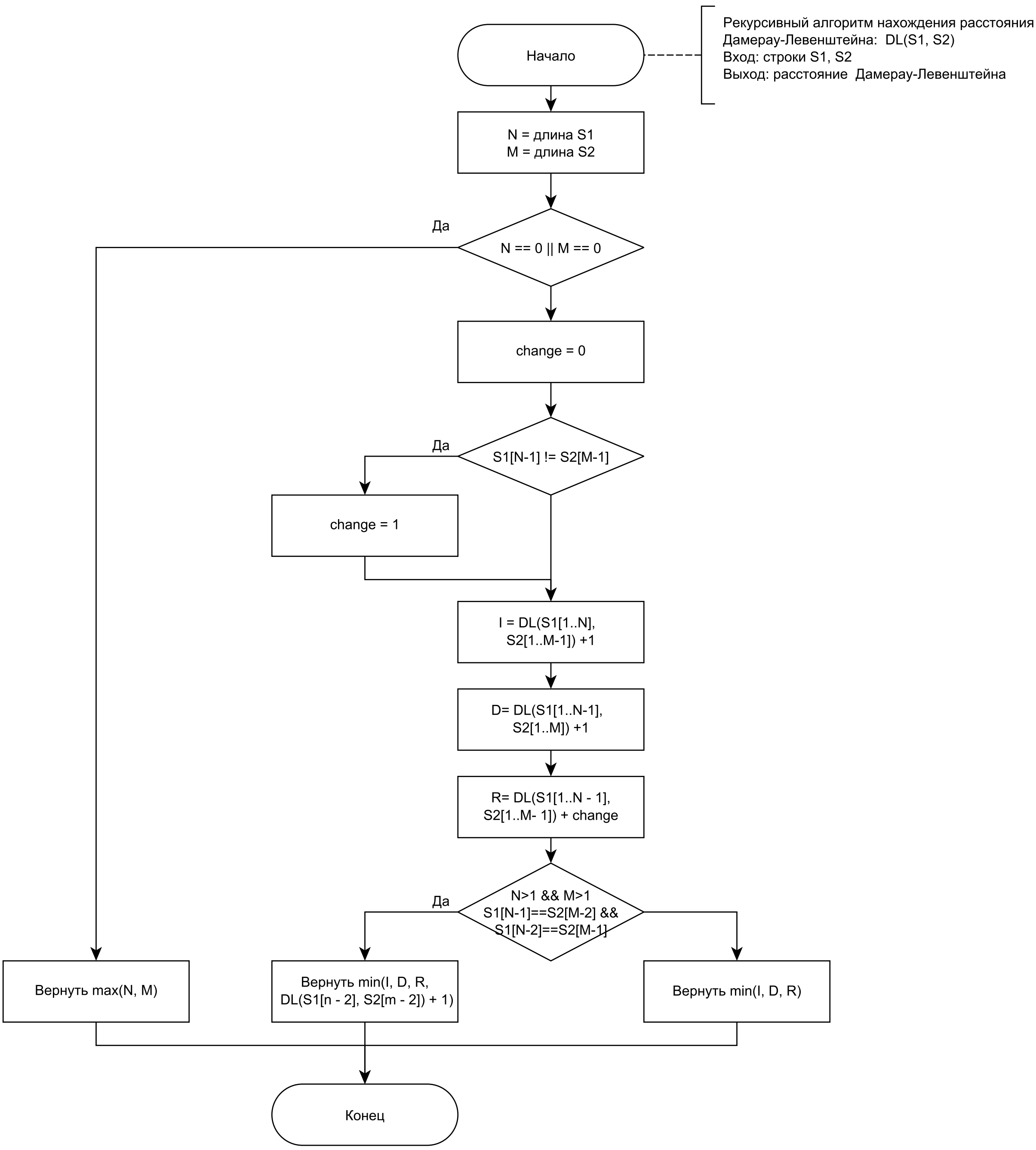

Рисунок 2.2 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

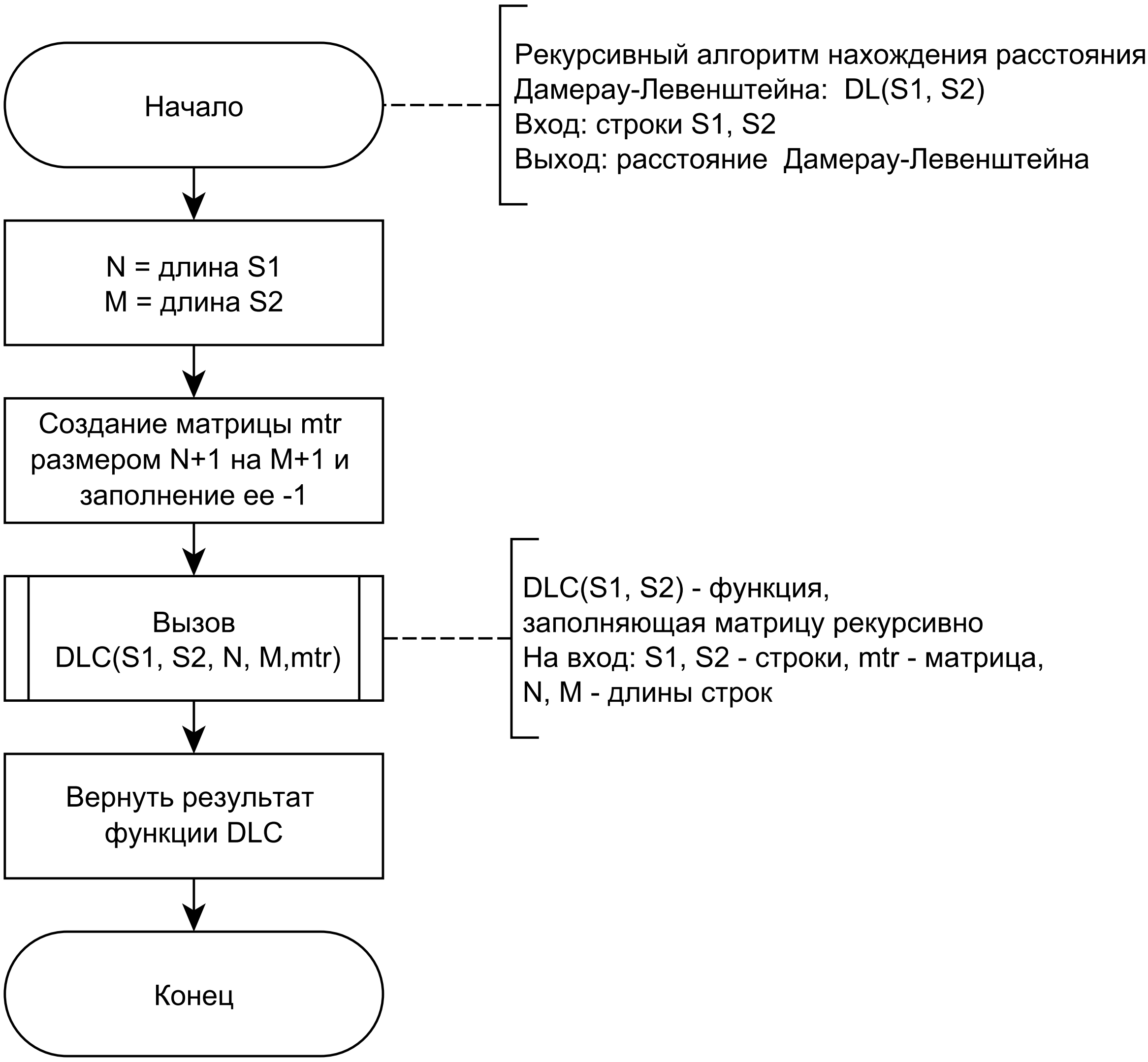

Рисунок 2.3 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешем

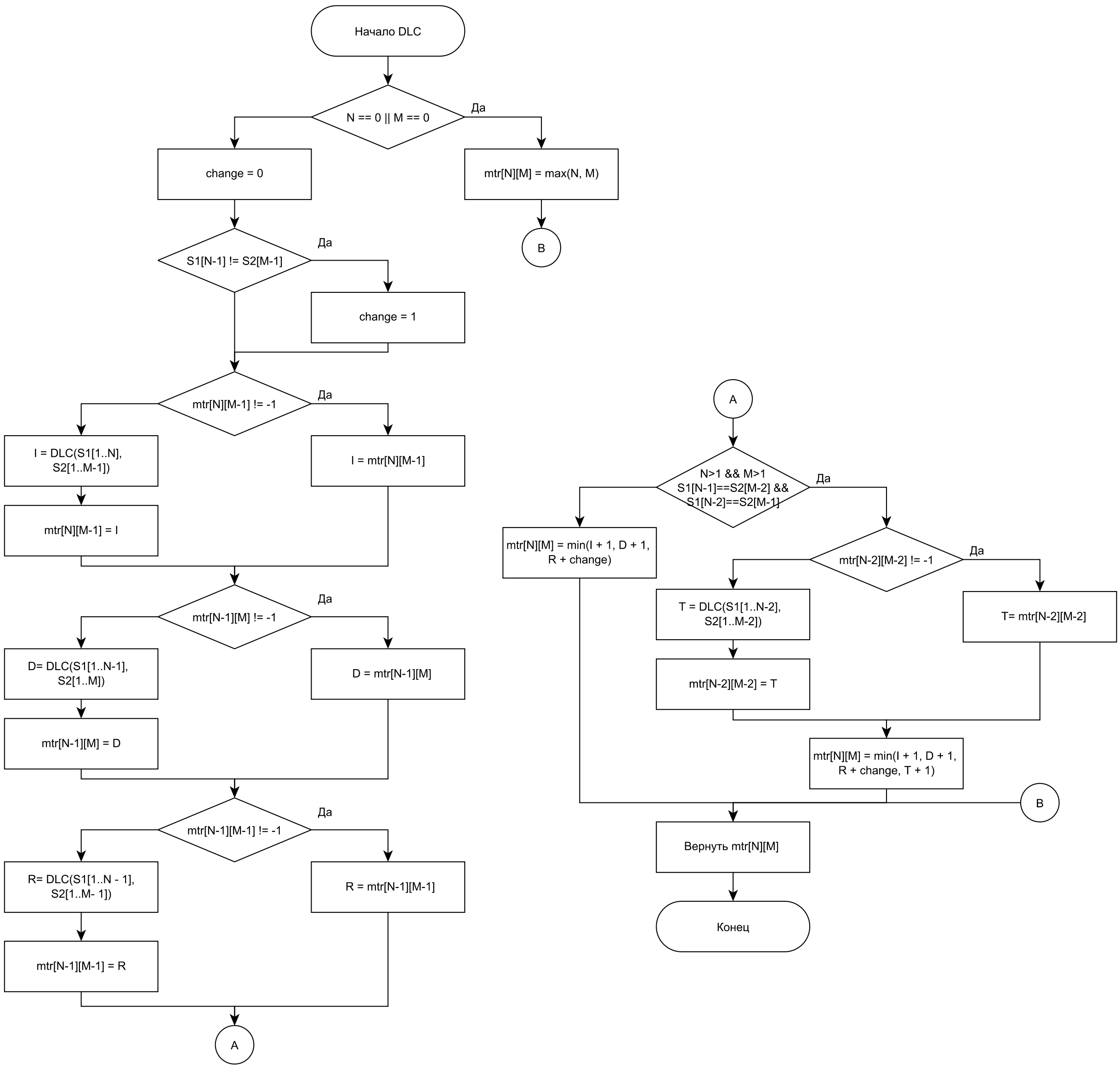

Рисунок 2.4 – Функция заполняющая матрицу расстояний Дамерау-Левенштейна рекурсивно

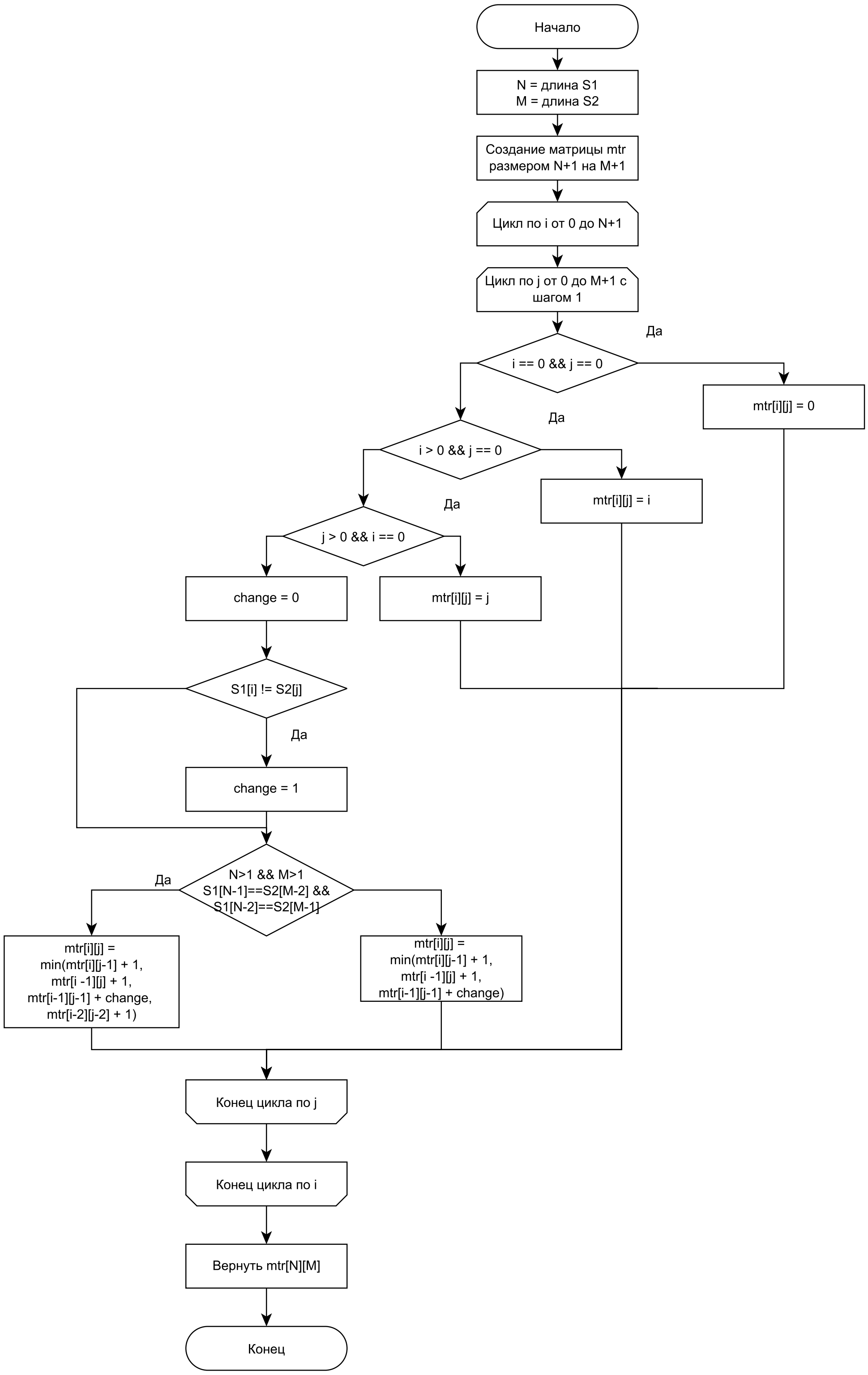

Рисунок 2.5 – Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

2.4 Описание используемых типов и структур данных

Для реализации алгоритмов, будут использованы следующие типы данных:

- *str* – для двух строк, поданных на вход;
- *int* – для длин каждой из строк, поданных на вход, и для возвращаемого значения искомого расстояния.

При реализации алгоритмов будет использована такая структура данных, как матрица, которая является двумерным списком значений типа *int*.

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык *Python* [1]. Данный выбор обусловлен опытом работы с этим языком программирования. Также данный язык позволяет замерять процессорное время с помощью модуля *time*.

Время работы было замерено с помощью функции *process_time()* из модуля *time* [2].

3.2 Реализация алгоритмов

В листингах 3.1 – 3.4 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

```
def m(a, b):
    return 0 if a == b else 1


def levenstein(s1, s2):
    matrix = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if i == 0 and j == 0:
                matrix[i][j] = 0
            elif i > 0 and j == 0:
                matrix[i][j] = i
            elif j > 0 and i == 0:
                matrix[i][j] = j
            else:
                matrix[i][j] = min([matrix[i][j - 1] + 1, matrix[i-1][j] + 1, matrix[i - 1][j - 1] + m(s1[i-1], s2[j-1])])
    return matrix[-1][-1]
```

Листинг 3.2 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы

```
def damerau_levenstein_iter(s1, s2):
    d = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]
    for i in range(len(d)):
        for j in range(len(d[0])):
            if i == 0 and j == 0:
                d[i][j] = 0
            elif i > 0 and j == 0:
                d[i][j] = i
            elif j > 0 and i == 0:
                d[i][j] = j
            elif i > 1 and j > 1 and s1[i-1] == s2[j-2] and
               s1[i-2] == s2[j-1]:
                d[i][j] = min([d[i][j-1] + 1, d[i-1][j] + 1,
                    d[i-1][j-1] + m(s1[i-1], s2[j-1]),
                    d[i-2][j-2] + 1])
            else:
                d[i][j] = min([d[i][j - 1] + 1, d[i-1][j] + 1,
                    d[i - 1][j - 1] + m(s1[i-1], s2[j-1])])
    return d[-1][-1]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def damerau_levenstein_rec(s1, s2):
    if len(s1) == 0 or len(s2) == 0:
        return max(len(s1), len(s2))
    temp = min([damerau_levenstein_rec(s1[:-1], s2) + 1,
       damerau_levenstein_rec(s1, s2[:-1]) + 1,
       damerau_levenstein_rec(s1[:-1], s2[:-1]) + m(s1[-1],
       s2[-1])])

    if len(s1) > 1 and len(s2) > 1 and s1[-1] == s2[-2] and
       s1[-2] == s2[-1]:
        temp = min(temp, damerau_levenstein_rec(s1[:-2],
           s2[:-2]) + 1)

    return temp
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кешированием

```
def damerau_levenstein_rec_cash(s1, s2):
    temp = max(len(s1), len(s2)) + 1
    d = [[-1] * (len(s2) + 1) for _ in range(len(s1) + 1)]

    def dlrc(s1, s2):
        if len(s1) == 0 or len(s2) == 0:
            d[len(s1)][len(s2)] = max(len(s1), len(s2))
            return d[len(s1)][len(s2)]

        if d[len(s1[:-1])][len(s2)] >= 0:
            dlr1 = d[len(s1[:-1])][len(s2)]
        else:
            # Случай если значение не в кеше, то вычисляем и "кешируем"
            dlr1 = dlrc(s1[:-1], s2)
            d[len(s1[:-1])][len(s2)] = dlr1

        if d[len(s1)][len(s2[:-1])] >= 0:
            dlr2 = d[len(s1)][len(s2[:-1])]
        else:
            # Случай если значение не в кеше, то вычисляем и "кешируем"
            dlr2 = dlrc(s1, s2[:-1])
            d[len(s1)][len(s2[:-1])] = dlr2

        if d[len(s1[:-1])][len(s2[:-1])] >= 0:
            dlr3 = d[len(s1[:-1])][len(s2[:-1])]
        else:
            # Случай если значение не в кеше, то вычисляем и "кешируем"
            dlr3 = dlrc(s1[:-1], s2[:-1])
            d[len(s1[:-1])][len(s2[:-1])] = dlr3

        temp = min([dlr1 + 1, dlr2 + 1, dlr3 + m(s1[-1], s2[-1])])

        if len(s1) > 1 and len(s2) > 1 and s1[-1] == s2[-2] and s1[-2] == s2[-1]:
            if d[len(s1[:-2])][len(s2[:-2])] >= 0:
```

```python
                dlr4 = d[len(s1[:-2])][len(s2[:-2])]
            else:
                # Случай если значение не в кеше, то вычисляем и "кешируем"
                dlr4 = dlrc(s1[:-2], s2[:-2])
                d[len(s1[:-2])][len(s2[:-2])] = dlr4
            temp = min(temp, dlr4 + 1)

        d[len(s1)][len(s2)] = temp

        return temp

    return dlrc(s1, s2)
```

3.3 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 кешом**
λ	λ	0	0	0	0
a	b	1	1	1	1
a	a	0	0	0	0
кот	скат	2	2	2	2
ab	ba	2	1	1	1
bba	abba	1	1	1	1
aboba	boba	1	1	1	1
abcdef	gh	6	6	6	6

Вывод

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

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

— Процессор: AMD Ryzen 5 4600H 3 ГГц [3].

— Оперативная память: 16 ГБайт.

— Операционная система: Windows 10 Pro 64-разрядная система версии 22H2 [4].

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 программного обеспечения, а именно показаны результаты вычислений расстояний Левенштейна и Дамерау-Левенштейна для строк «скат», «кот» и «красивый», «карсивый» соответственно.

```
    1 - Расстояние Левенштейна (итеративно)
    2 - Расстояние Дамерау-Левенштейна (итеративно)
    3 - Расстояние Дамерау-Левенштейна (рекурсивно)
    4 - Расстояние Дамерау-Левенштейна (рекурсивно с кешированием)
    5 - Вывести результаты тестов
    6 - Замер времени
    0 - Выход


Выберите опцию: 1
Введите строку 1: скат
Введите строку 2: кот
Расстояние = 2


    1 - Расстояние Левенштейна (итеративно)
    2 - Расстояние Дамерау-Левенштейна (итеративно)
    3 - Расстояние Дамерау-Левенштейна (рекурсивно)
    4 - Расстояние Дамерау-Левенштейна (рекурсивно с кешированием)
    5 - Вывести результаты тестов
    6 - Замер времени
    0 - Выход


Выберите опцию: 4
Введите строку 1: красивый
Введите строку 2: карсивый
Расстояние = 1


    1 - Расстояние Левенштейна (итеративно)
    2 - Расстояние Дамерау-Левенштейна (итеративно)
    3 - Расстояние Дамерау-Левенштейна (рекурсивно)
    4 - Расстояние Дамерау-Левенштейна (рекурсивно с кешированием)
    5 - Вывести результаты тестов
    6 - Замер времени
    0 - Выход


Выберите опцию:
```

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

4.3 Время выполнения алгоритмов

Результаты замеров времени работы алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна приведены в таблице 4.1. Замеры времени проводились на строках одинаковой длины и усреднялись для каждого набора одинаковых экспериментов.

Таблица 4.1 – Время работы алгоритмов (в секундах)

Длина строк	Л (и)	Д-Л (и)	Д-Л (р)	Д-Л (рк)
1	0	0	0	0
2	0	1.56e-05	1.56e-05	0
3	0.85e-06	1.56e-05	3.13e-05	1.56e-05
4	1.56e-05	1.56e-05	1.72e-04	3.13e-05
5	1.56e-05	1.56e-05	9.69e-04	4.69e-05
6	1.56e-05	3.13e-05	5.11e-03	6.25e-05
7	3.13e-05	3.13e-05	2.76e-02	9.38e-05
8	4.69e-05	4.69e-05	1.52e-01	1.09e-04
9	4.69e-05	6.25e-05	8.33e-01	1.25e-04

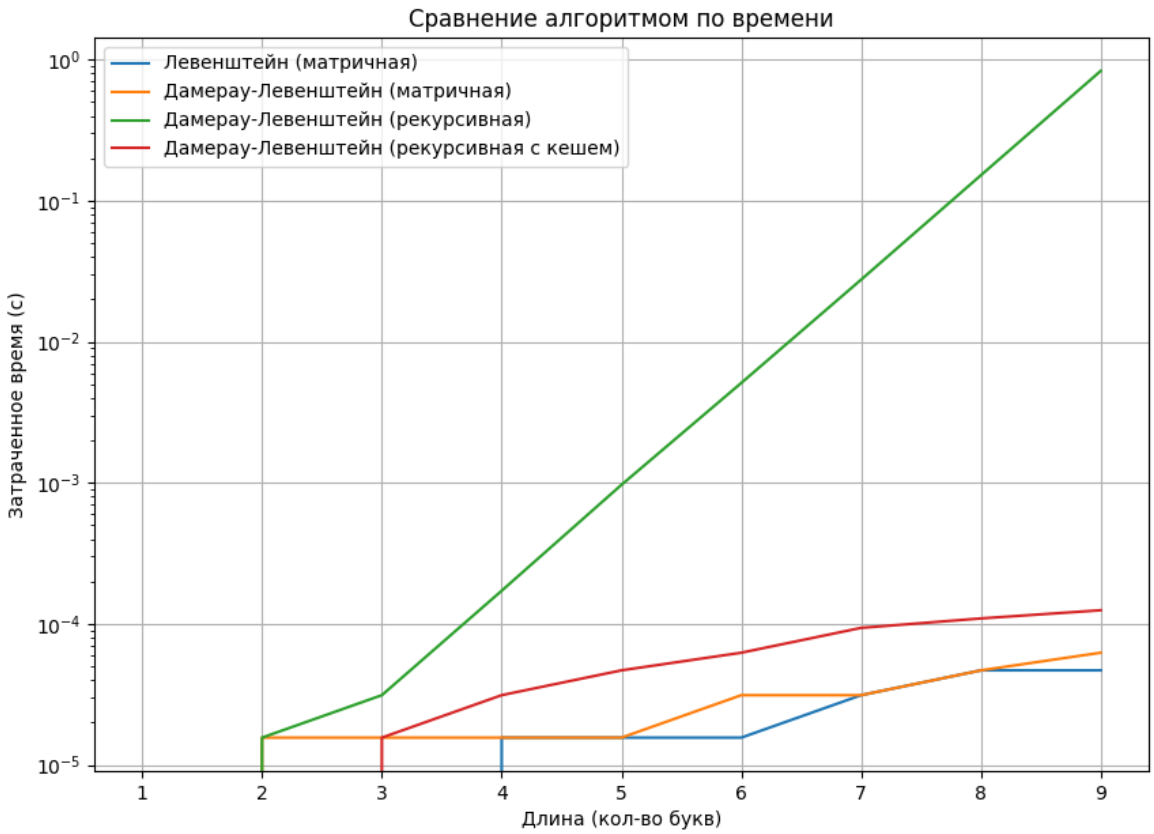

Рисунок 4.2 – Сравнение алгоритмов по времени

Наиболее эффективными являются алгоритмы, использующие матрицы, после них по скорости работы идет алгоритм использующий кеш, это обусловлено тем, что по сравнению с обычной рекурсией, мы не вычисляем повторно одни и те же значения, но все равно тратим время на рекурсивные вызовы.

4.4 Характеристики по памяти

Введем следующие обозначения:

— n – длина строки S_1;

— m – длина строки S_2;

— $size()$ – функция, вычисляющая размер в байтах;

— int – целочисленный тип данных;

— $string$ – строковый тип данных.

Т. к. алгоритмы, вычисляющие расстояния Левенштейна и Дамерау-Левенштейна, не отличаются по использованию памяти, то достаточно рассмотреть итеративную реализацию одного из этих алгоритмов, рекурсивную и рекурсивную с кешированием реализации алгоритмов вычисления расстояния Дамерау-Левенштейна.

Использование памяти при итеративной реализации теоритически равно:

$$(n+1) * (m+1) * size(int) + 2 * size(string) + 2 * size(int), \tag{4.1}$$

где

— $(n+1) * (m+1) * size(int)$ – хранение матрицы;

— $2 * size(string)$ – хранение двух строк;

— $2 * size(int)$ – адрес возврата и возвращаемое значение.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, соответственно, максимальный расход памяти равен:

$$(n+m) * (2 * size(string) + 3 * size(int)), \tag{4.2}$$

где

- $(n+m)$ – максимальная глубина стека вызовов;
- $2 * size(string)$ – хранение двух строк;
- $2 * size(int)$ – адрес возврата и возвращаемое значение;
- $size(int)$ – временная переменная.

Для алгоритма, использующего кеширование требуется дополнительно память под кеш и 4 временных переменных:

$$(n+m)*(2*size(string)+6*size(int))+(n+1)*(m+1)*size(int), \tag{4.3}$$

где

- $(n+m)$ – максимальная глубина стека вызовов;
- $2 * size(string)$ – хранение двух строк;
- $2 * size(int)$ – адрес возврата и возвращаемое значение;
- $4 * size(int)$ – временные переменные;
- $(n+1) * (m+1) * size(int)$ – хранение кеша.

По расходу памяти итеративные алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в итеративном растет как произведение длин строк, в то время как у рекурсивного алгоритма – как сумма длин строк.

Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна.

По таблице 4.1 видно, что рекурсивный алгоритм в n раз проигрывает итеративному при длине строк 10. Поэтому рекурсивные алгоритмы следует использовать лишь при малых длинах строк.

При этом как было замечено в пункте 4.4, рекурсивные алгоритмы занимают меньше памяти, чем итеративные алгоритмы.

Рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейн будет более затратным по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау-Левенштейна, но менее затратным по памяти по отношению к итеративному алгоритму Дамерау-Левенштейна. При этом рекурсивные алгоритм с кешированием проигрывает по памяти и по времени итеративному.

Заключение

В результате исследования было определено, что время алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна растет в геометрической прогрессии при увеличении длин строк. Лучшие показатели по времени дает матричная реализация алгоритма нахождения расстояния Дамерау-Левенштейна и его рекурсивная реализация с кешем, использование которых приводит к 21-кратному превосходству по времени работы уже на длине строки в 4 символа за счет сохранения необходимых промежуточных вычислений. При этом итеративная реализации с использованием матрицы занимают довольно много памяти при большой длине строк.

Цель данной лабораторной работы были достигнуты, а именно описание и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной целей были выполнены следующие задачи.

1) Описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;

2) Создано программное обеспечение, реализующее следующие алгоритмы.

   — нерекурсивный метод поиска расстояния Левенштейна;

   — нерекурсивный метод поиска расстояния Дамерау-Левенштейна;

   — рекурсивный метод поиска расстояния Дамерау-Левенштейна;

   — рекурсивный с кешированием метод поиска расстояния Дамерау-Левенштейна.

3) Выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов.

4) Проведены анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.

Список использованных источников

1 The official home of the Python Programming Language [Электронный ресурс]. — Режим доступа: `https://www.python.org/` (дата обращения: 19.09.2023).

2 time — Time access and conversions [Электронный ресурс]. — Режим доступа: `https://docs.python.org/3/library/time.html#time.process_time` (дата обращения: 19.09.2023).

3 Amd [Электронный ресурс]. — Режим доступа: `https://www.amd.com/en.html` (дата обращения: 28.09.2023).

4 Windows 10 Pro 22h2 64-bit [Электронный ресурс]. — Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows10` (дата обращения: 28.09.2023).