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 ИУ, Информатика и системы управления

КАФЕДРА ИУ7, Программное обеспечение ЭВМ и информационные технологии

ЛАБОРАТОРНАЯ РАБОТА №1

ПО ДИСЦИПЛИНЕ

“Анализ алгоритмов”

Студент ИУ7-54Б ________________ **А.А. Андреев**
(Группа) (Подпись, дата) (И.О.Фамилия)

Преподаватель ________________ **Л.Л. Волкова**
(Подпись, дата) (И.О.Фамилия)

2021 г.

Оглавление

Введение.

Данная лабораторная работа посвящена изучению алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Расстояние Левенштейна [1] - это минимальное количество операций вставки/удаления/замены одного символа на другой, необходимых для превращения одной строки в другую.

Вычисление расстояния Левенштейна применяется во многих отраслях для исправления ошибок в слове в компьютерной лингвистике, для сравнения хромосов и белков в биоинформатике.

Цель данной лабораторной работы: Изучение и оценка реализации методов динамического программирования на нахождение расстояния Левенштейна и Дамерау-Левенштейна.

Задачи данной лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна;
2. Получение навыков реализации матричных и рекурсивных версий алгоритмов;
3. Проведение сравнительного анализа линейной и рекурсивной реализации реализаций алгоритмов;
4. Формирование обоснования полученных результатов исследования работы алгоритмов

1 Аналитическая часть

Расстояние Левенштейна [1] - это минимальное количество операций вставки/удаления/замены одного символа на другой, необходимых для превращения одной строки в другую. Каждая операция определяется своей ценой, в общем случае операции определены, как:

- $\omega(\lambda,\ b)$- цена операции вставки
- $\omega(a,\ \lambda)$- цена операции удаления
- $\omega(a,\ b)$- цена операции замены

Расстояние Левенштейна - это минимальная суммарная цена после последовательности замен. Существуют частные случаи нахождения расстояния Левенштейна:

- $\omega(a,\ a)\ =\ 0$
- $\omega(\lambda,\ b)\ =\ 1$
- $\omega(a,\ \lambda)\ =\ 1$
- $\omega(a,\ b)\ =\ 1$ и $a\ \neq\ b$

1.1 Матричный алгоритм Левенштейна

Данный метод реализации алгоритма Левенштейна эффективнее, чем рекурсивный, так как промежуточные значения хранятся в виде матрицы, при этом в любой момент времени мы может обратиться при помощи индексации по строке и столбцу к любой из ранее выполненных операций. В общем виде алгоритм выглядит, как построчное заполнение матрицы:

$$A_{i,j}\ =\ D(i,\ j)$$

1.2 Рекурсивный алгоритм Левенштейна

Формула (1.1) расстояния между двумя строками a и b, где $|a| -$ длина строки $a,\ a[i]$- i-й символ строки a:

$$D(i,j) = \begin{cases} 0 & \mathrm{i} = 0, \mathrm{j} = 0 \\ i & \mathrm{j} = 0, \mathrm{i} > 0 \\ j & \mathrm{i} = 0, \mathrm{j} > 0 \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \mathrm{i} > 0, \mathrm{j} > 0 \\ \qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases}, \qquad (1.1)$$

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

1.3 Рекурсивный алгоритм Левенштейна с заполнением матрицы

Рекурсивный алгоритм Левенштейна с заполнением матрицы - это объединение алгоритмов 1.1. и 1.2.: во время реализации рекурсивного алгоритма Левенштейна происходит заполнение матрицы. Ранее найденные расстояния не рассчитываются заново, они берутся из матрицы.

1.4 Алгоритм Дамерау-Левенштейна [3]

Формула (1.3) нахождения расстояния Дамерау-Левенштейна определяется так, как и (1.1) - формула неэффективна по времени и аналогично для оптимизации используется добавление матрицы для хранения промежуточных значений.

$$
d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1, & \\ \quad d_{a,b}(i-1,j)+1, & \\ \quad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе} \\ \quad \left[\begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \qquad (1.3)
$$

1.5 Выводы из аналитического раздела

В данном разделе были описаны: рекурсивный алгоритм Левенштейна с матрицей и без нее, матричный итерационный алгоритм Левенштейна, итерационный алгоритм Дамерау-Левенштейна.

2 Конструкторская часть.

В данном разделе будет приведены блок-схемы алгоритмов, описанных в аналитическом разделе п.1.

2.1 Схемы алгоритмов

Рекурсивный алгоритм Левенштейна без Кэша

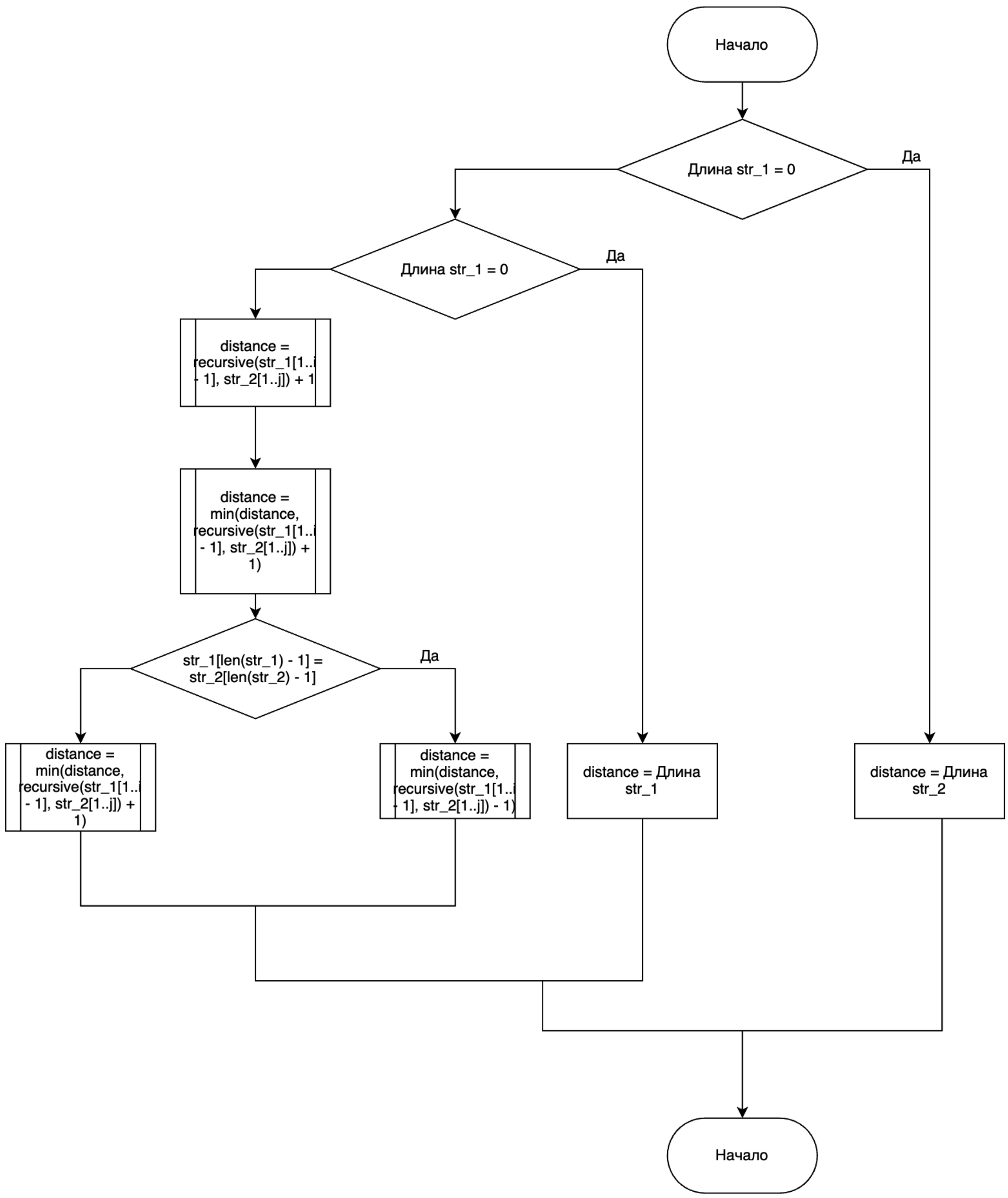

Рисунок 1: Схема рекурсивного алгоритма Левенштейна без Кэша

Рекурсивный алгоритм Левенштейна с кэшем

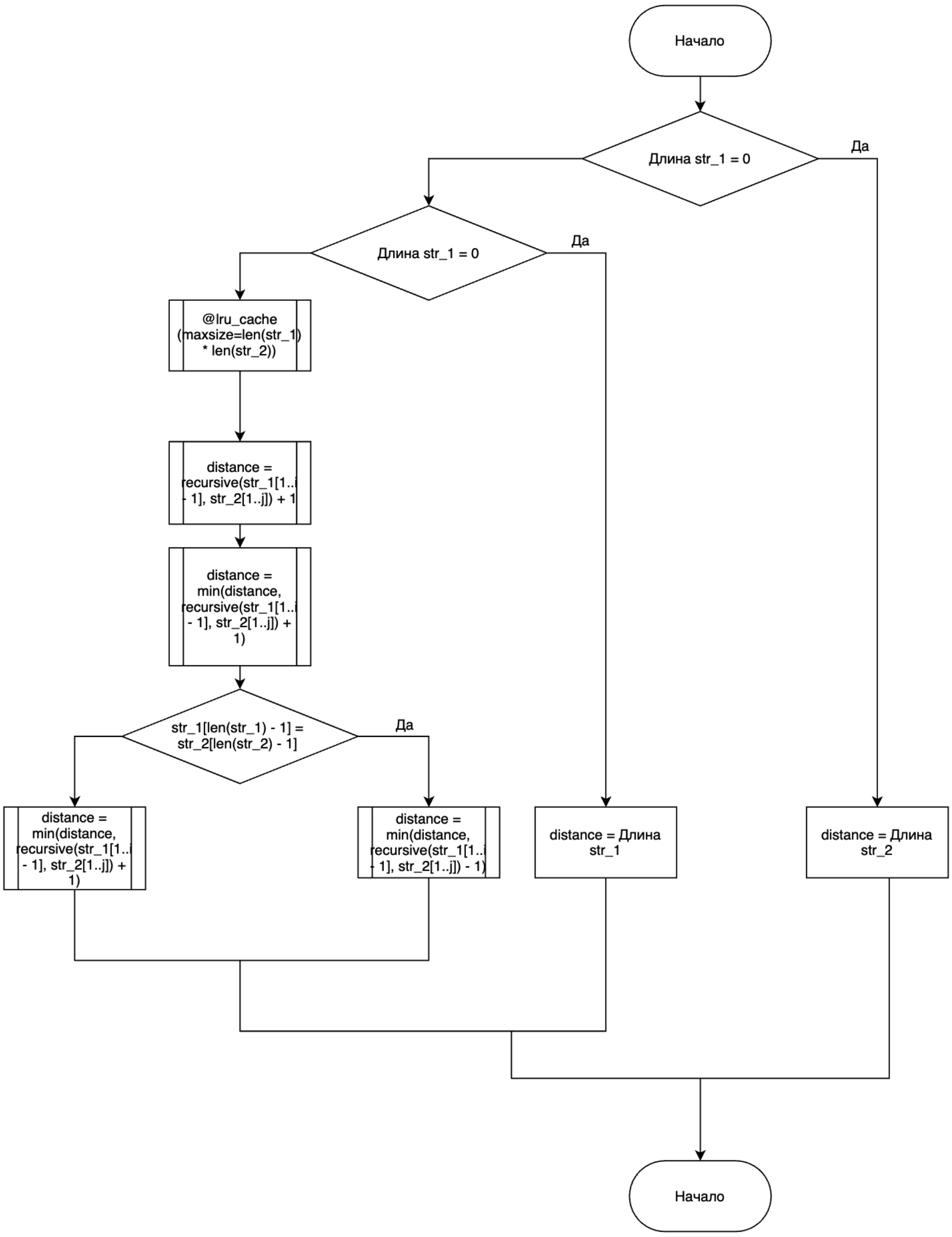

Рисунок 2: Схема рекурсивного алгоритма Левенштейна с Кэшем

Итеративный алгоритм Левенштейна

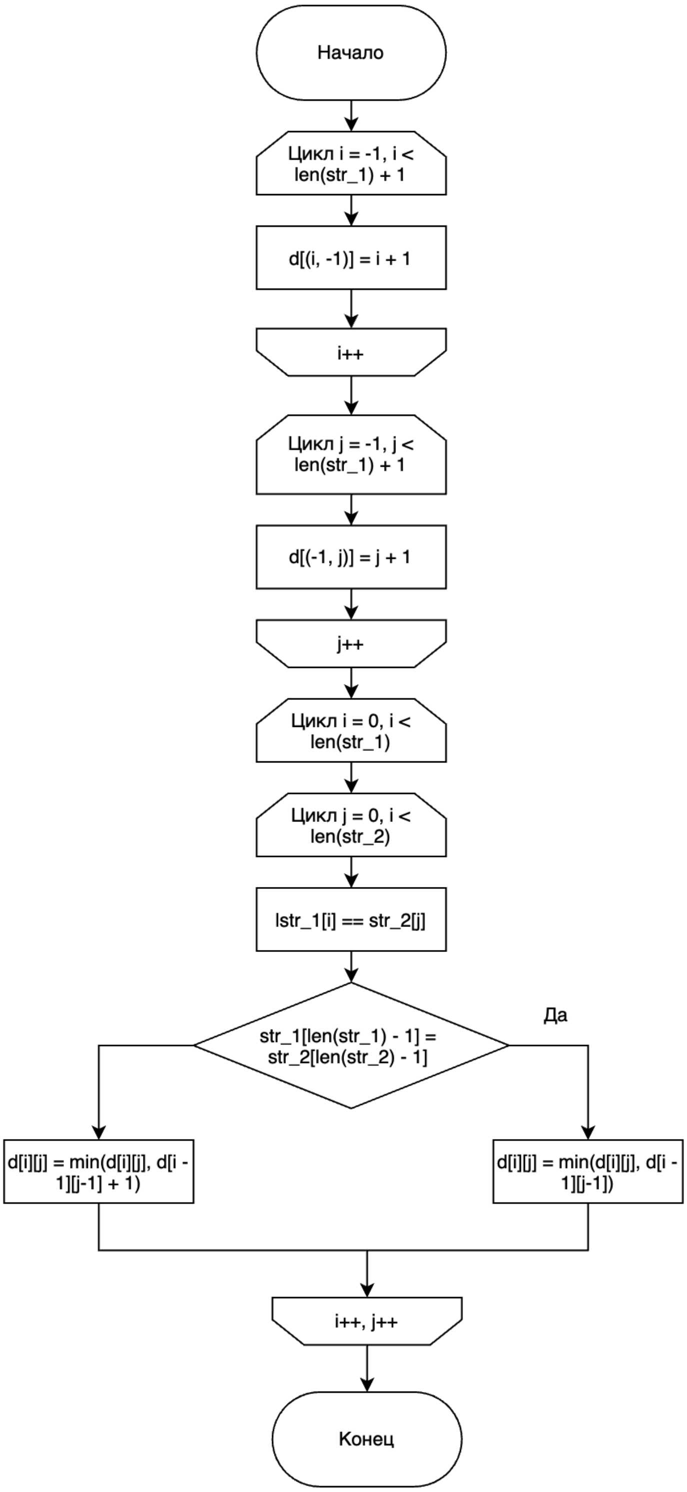

Рисунок 3: Схема итеративного алгоритма Левенштейна

Итеративный алгоритм Дамерау-Левенштейна

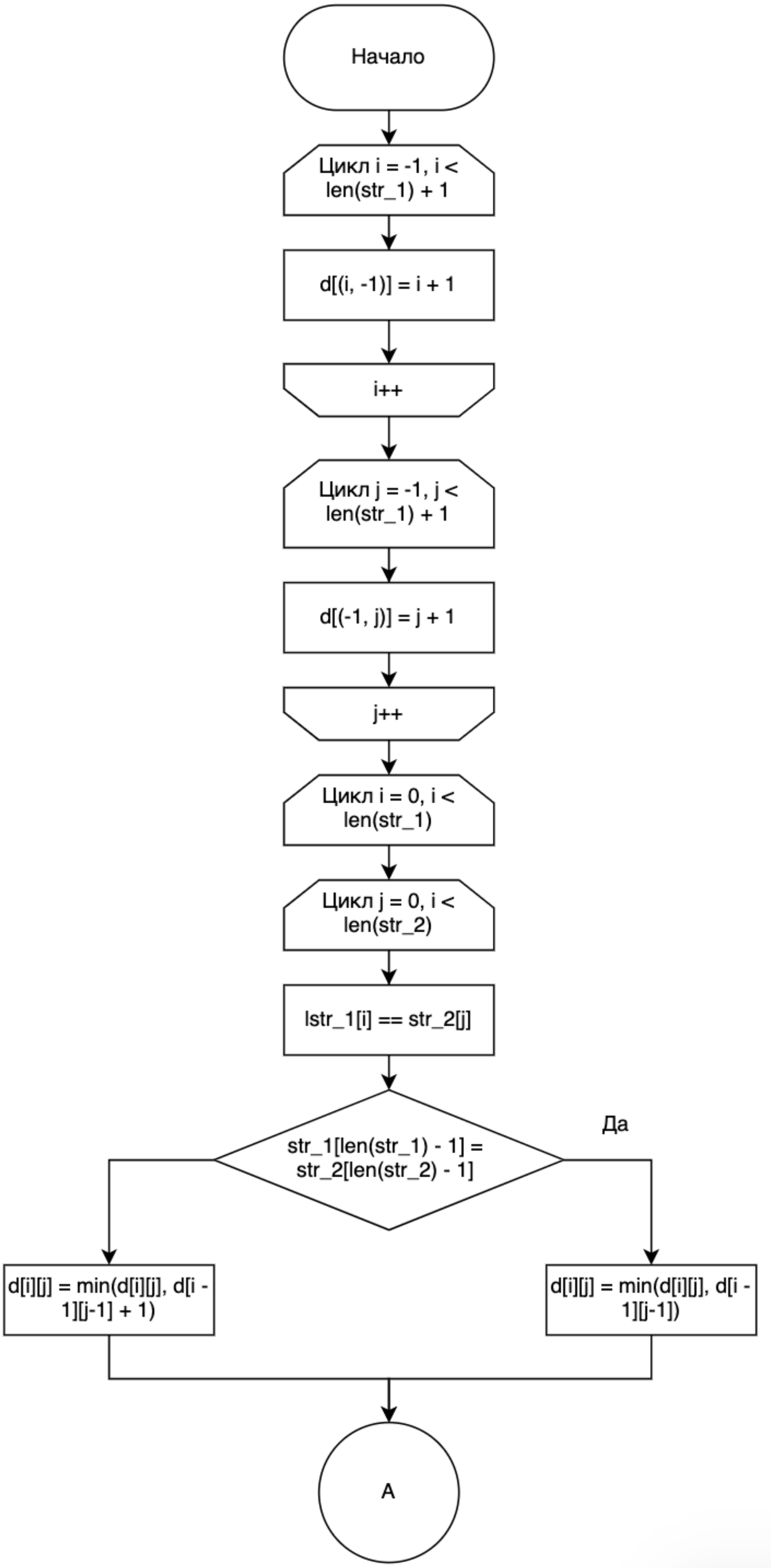

Рисунок 4: Схема итеративного алгоритма Дамерау-Левенштейна, Часть 1

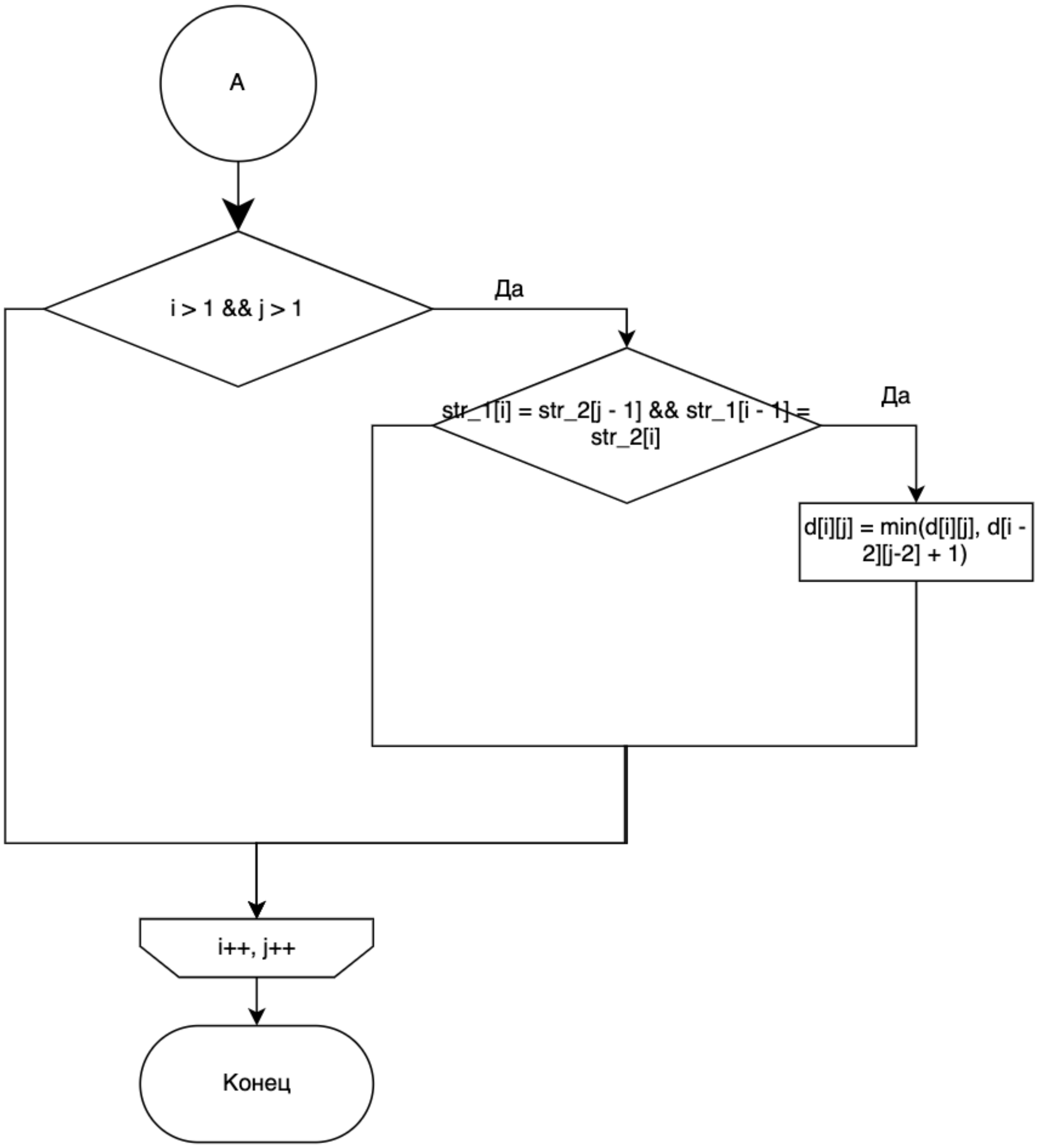

Рисунок 5: Схема итеративного алгоритма Дамерау-Левенштейна, Часть 2

2.2 Вывод

Блок-схемы в данном разделе позволяют перейти к технологической части - непосредственно к программной реализации решения.

Блок-схемы в данном разделе демонстрируют схемы работы рекурсивного алгоритма Левенштейна с кэшем и без, итеративный алгоритм Левенштейна, итеративный алгоритм Дамерау-Левенштейна.

3 Технологическая часть.

В данном разделе будут рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач.

3.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать поставленную на лабораторную работу задачу. Интерфейс для взаимодействия с программой - командная строка. Программа должна выводить полученное расстояние между двумя введенными строками и показывать потраченное на это время.

3.2 Выбор и обоснование языка и среды программирования.

Для разработки данной программы применён язык Python 3 с библиотекой time.clock() [4] для вычисления времени работы процессора, потому что я хочу расширить свои знания в области данного языка программирования.

Для разработки данной программы применён язык Python 3 с библиотекой time.clock() [4] для вычисления времени работы процессора, чтобы расширить знания в области данного языка программирования.

3.3 Реализация алгоритмов

В листингах 1-4 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Программа была реализована в парадигме ООП [2], где в базовый класс был вынесен объект Levenstein (Листинг 1.), внутри него с доступом protected, используемая алгоритмами с меморизацией и без нее, рекурсивная функция получения расстояния между строками.

Наследуемые объекты рекурсивного алгоритма без кэша (Листинг 2), рекурсивного алгоритма с кэшема (Листинг 3), итерационный алгоритм Дамерау-Левенштейна (Листинг 4), итерационный алгоритм Левенштейна (Листинг 5) имеют публичную функцию получения времени работы get_time();

Листинг 1: Базовый класс Levenstein с Рекурсивная функция нахождения расстояния Левенштейна, Часть 1

```
# Объект алгоритма Левенштейна
class Levenshtein:
    # Защищенные наследуемые данные объекта
    _first_string = None
    _second_string = None

    # Ключевое расстояние
```

Листинг 2: Базовый класс Levenstein с Рекурсивная функция нахождения расстояния Левенштейна, Часть 2

```
    _distance = None

    # Ключевое время
    _time = None

    # Создание объекта
    def __init__(self, first_string, second_string):
        # Назначение данных объекта
        self._first_string = first_string
        self._second_string = second_string

        # Установка значения расстояния
        self._distance = config.START_ZERO_VALUE

        # Установка значения времени выполнения
        self._time = config.START_ZERO_VALUE

    # Общая функция получения расстояния между двумя строками
    def get_distance(self):
        return self._distance

    # Общая функция получения времени выполнения
    def get_time(self):
        return self._time

    # Получение расстояния между строками
    def _recursive_get_distance(self, first_string_length, second_string_length):

        # если одна из строк пустая, то расстояние до другой строки - ее длина
        # т.е. n вставок
        if first_string_length == 0 or second_string_length == 0:
            return max(first_string_length, second_string_length)

        # если оба последних символов одинаковые, то съедаем их оба, не меняя расстояние
        elif self._first_string[first_string_length - 1] == self._second_string[second_string_length - 1]:
            return self._recursive_get_distance(first_string_length - 1, second_string_length - 1)

        # выбор минимального значения из трех
        else:
            return 1 + min(
```

Листинг 3: Базовый класс Levenstein с Рекурсивная функция нахождения расстояния Левенштейна, Часть 3

```
            self._recursive_get_distance(first_string_length, second_string_length - 1),  # Удаление
            self._recursive_get_distance(first_string_length - 1, second_string_length),  # Вставка
            self._recursive_get_distance(first_string_length - 1, second_string_length - 1)  # Замена
            )
```

Листинг 3: Наследуемый класс Левенштейна без кэша

```
# Наследуемый объект Рекурсивного алгоритма без кэша
class LevenshteinRecursiveWithoutCache(Levenshtein):

    # Общая функция получения расстояния между двумя строками
    def get_distance(self):
        self._distance = self._recursive_get_distance(len(self._first_string), len(self._second_string))
        return self._distance

    # Получение времени
    def get_time(self):
        t_0 = clock()
        self._recursive_get_distance(len(self._first_string), len(self._second_string))
        t_1 = clock()

        self._time = t_1 - t_0

        return self._time

```

Листинг 4: Наследуемый класс Левенштейна с кэшем, Часть 1

```
# Наследуемый объект Рекурсивного алгоритма с кэшем
class LevenshteinRecursiveWithCache(Levenshtein):
    _first_string_length = None
    _second_string_length = None

    def __init__(self, first_string, second_string):
        super().__init__(first_string, second_string)

        self._first_string_length = len(self._first_string)
        self._second_string_length = len(self._second_string)
```

Листинг 5: Наследуемый класс Левенштейна с кэшем, Часть 2

```

    # Получение времени
    def get_time(self):
        t_0 = clock()
        self.get_distance()
        t_1 = clock()

        self._time = t_1 - t_0

        return self._time

    def get_distance(self):
        # Общая функция получения расстояния между двумя строками
        @lru_cache(maxsize=self._first_string_length * self._second_string_length)
        def get_distance():
            self._distance = self._recursive_get_distance(len(self._first_string), len(self._second_string))
            return self._distance

        # Обновление расстояния
        self._distance = get_distance()

        return self._distance
```

Листинг 6: Наследуемый класс итерационного Дамерау Левенштейна, Часть 1

```
# Объект вычисления Дамерау Левенштейна
class DamerauLevenshtein(Levenshtein):
    # Используемые блины строк
    _first_string_length = None
    _second_string_length = None

    def __init__(self, first_string, second_string):
        super().__init__(first_string, second_string)

        self._first_string_length = len(self._first_string)
        self._second_string_length = len(self._second_string)

    # Получение расстояния между двумя строками
    def get_distance(self):
        d = {}

        for i in range(-1, self._first_string_length + 1):
            d[(i, -1)] = i + 1
        for j in range(-1, self._second_string_length + 1):
            d[(-1, j)] = j + 1

        for i in range(self._first_string_length):
            for j in range(self._second_string_length):
```

Листинг 7: Наследуемый класс итерационного Дамерау Левенштейна, Часть 2

```
                if self._first_string[i] == self._second_string[j]:
                    cost = 0
                else:
                    cost = 1
                d[(i, j)] = min(
                    d[(i - 1, j)] + 1,  # deletion
                    d[(i, j - 1)] + 1,  # insertion
                    d[(i - 1, j - 1)] + cost,  # substitution
                )
                if i and j and self._first_string[i] == self._second_string[j - 1] and \
                        self._first_string[i - 1] == self._second_string[j]:
                    d[(i, j)] = min(d[(i, j)], d[i - 2, j - 2] + cost)  # transposition

        return d[self._first_string_length - 1, self._second_string_length - 1]

    # Получение времени
    def get_time(self):
        t_0 = clock()
        self.get_distance()
        t_1 = clock()

        self._time = t_1 - t_0

        return self._time
```

Листинг 8: Наследуемый класс Левенштейна с кешем, Часть 1

```
# Линейный алгоритм вычисления Левенштейна
class LevenshteinLinear(Levenshtein):
    # Используемые длины строк
    _first_string_length = None
    _second_string_length = None

    def __init__(self, first_string, second_string):
        super().__init__(first_string, second_string)
        self._first_string_length = len(first_string)
        self._second_string_length = len(second_string)

    def _update_distance(self):
        if self._first_string_length > self._second_string_length:
            self._first_string, self._second_string = self._second_string, self._first_string
```

Листинг 9: Наследуемый класс Левенштейна с кэшем, Часть 2

```python
                self._first_string_length,
self._second_string_length = self._second_string_length,
self._first_string_length

            current_row = range(self._first_string_length + 1)
            for i in range(1, self._second_string_length + 1):
                previous_row, current_row = current_row, [i] + [0]
* self._first_string_length
                for j in range(1, self._first_string_length + 1):
                    add, delete, change = previous_row[j] + 1,
current_row[j - 1] + 1, previous_row[j - 1]
                    if self._first_string[j - 1] !=
self._second_string[i - 1]:
                        change += 1
                    current_row[j] = min(add, delete, change)

            return current_row[self._first_string_length]

        # Получение расстояния между двумя строками
        def get_distance(self):
            self._distance = self._update_distance()
            return self._distance

        # Получение времени
        def get_time(self):
            t_0 = clock()
            self.get_distance()
            t_1 = clock()

            self._time = t_1 - t_0

            return self._time
```

3.4 Тестовые данные

Тестовые данные, на которых было протестировано разработанное программное обеспечение, представлено в Таблице 1.

Таблица 1: Тестовые данные, Часть 1

№	Первое слово	Второе слово	Ожидаемый результат, Расстояние, Л. 2	Л. 3	Л. 4	Л.5	Полученный результат, Расстояние, Л. 2	Л. 3	Л. 4	Л. 5
1	увлечение	развлечения	4	4	4	4	4	4	4	4
2	кот	скат	2	2	2	2	2	2	2	2
3	“”	тест	4	4	4	4	4	4	4	4
4	мгту	мгту	2	2	1	2	2	2	1	2

Таблица 2: Тестовые данные, Часть 2

№	Первое слово	Второе слово	Ожидаемый результат				Полученный результат			
			Расстояние				Расстояние			
			Л. 2	Л. 3	Л. 4	Л.5	Л. 2	Л. 3	Л. 4	Л. 5
5	рот	кот	1	1	1	1	1	1	1	1
7	лилия	рим	4	4	4	4	4	4	4	4
8	рим	мир	2	2	2	2	2	2	2	2
9	apple	aplpe	2	2	1	2	2	2	1	2
10	катя	надя	2	2	2	2	2	2	2	2

3.5 Вывод

В аналитическом разделе были представлены разработанный код, демонстрация его работы на тестовых данных Таблицы 1.

В данном разделе были рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач, приведены результаты работы программы на тестовых данных.

4 Исследовательская часть.

4.1. Демонстрация работы программы

Пример работы программы представлен на рисунке 6.

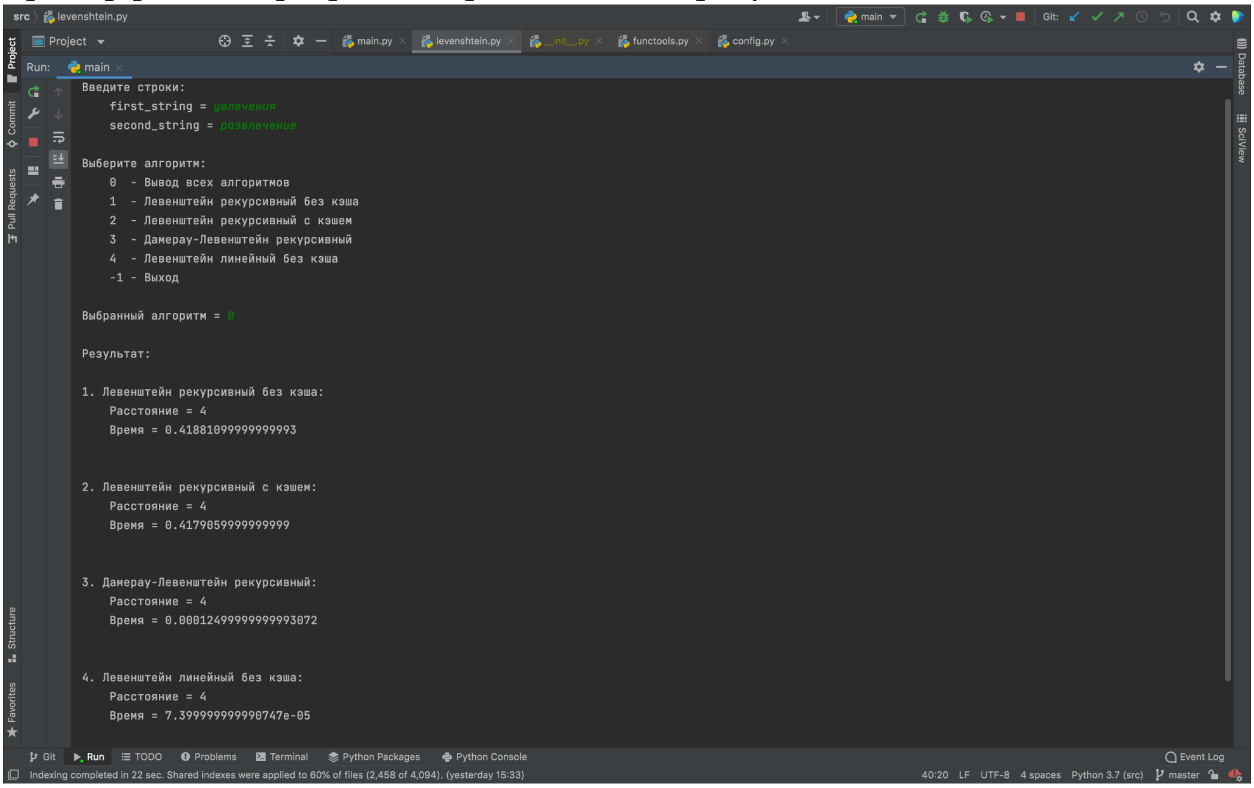

Рисунок 6: Демонстрация работы программы на примере строк Увлечения и Развлечение

4.2. Технические характеристики

В Таблице 3. приведены технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения.

Таблица 3: Технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения

ОС	Mac OS Mojave 64-bit
ОЗУ	8 Gb 2133 MHz LPDDR3
Процессор	2,3 GHz Intel Core i5

4.3. Время выполнения алгоритмов

В Таблице 4. приведена информация о времени выполнения алгоритмов в микросекундах.

Таблица 4: Таблица времени выполнения алгоритмов (в микросекундах)

№	Длина строк	Время			
		Итер. Лев-на	Итер. Дамерау-Лев-на	Рек. Лев-на без кэша	Рек. Лев-на с кэшем
1	10	0,02	0,04	0,50	0,30
2	20	0,04	0,05	0,90	0,40
3	40	0,04	0,05	3,00	1,00
4	100	0,09	0,10	24,00	13,00
5	200	0,10	0,12	29,40	11,10

4.4. Использование памяти

В Таблице 4 представлена информация об использовании памяти во время выполнения разных типов алгоритмов, где string - текстовая строка, integer - целое число.

Таблица 4: Таблица времени выполнения алгоритмов (в наносекундах)

№	Тип вызова	Память
1	Рекур-ый вызов	$(S(STR_1) + S(STR_2)) \cdot (2 \cdot S(str) + 3 \cdot S(int))$
2	Итер-ый вызов	$(S(STR_1) + 1) \cdot (S(STR_2) + 1) \cdot S(int) + 5 \cdot S(int) + 2 \cdot S(str)$

4.5. Вывод

Время работы рекурсивной версии алгоритма увеличивается в геометрической прогрессии, это самый неэффективный способ реализации алгоритма нахождения расстояния Левенштейна по времени и памяти. Наиболее эффектный способ из представленных - итеративный с хранением двух строк.

Время выполнения рекурсивной версии алгоритма увеличивается в геометрической прогрессии, это самый неэффективный способ реализации нахождения расстояния Левенштейна по времени и памяти, использования кэша его позволяет оптимизировать по времени выполнения до 2 раз. Наиболее эффективный способ из представленных оказался итеративный с хранением двух строк: Разница в его выполнении и выполнении рекурсивной версии алгоритма без кэша на размере слова 200 символов достигает 290 раз.

Заключение.

В данной лабораторная работа я изучил изучению алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, провел Изучение и оценку реализации методов динамического программирования на нахождение расстояния Левенштейна и Дамерау-Левенштейна, получил навыки реализации матричных и рекурсивных версий алгоритмов, провел сравнительный анализ линейной и рекурсивной реализации алгоритмов, сформировал обоснования полученных результатов исследования алгоритмов.

В данной были изучены алгоритмы Левенштейна и Дамерау-Левенштейна, получены навыки реализации матричных и рекурсивных версий алгоритмов, проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов, сформированы обоснования полученных результатов исследования работы алгоритмов: алгоритм Левенштейна без кэша оказался менее эффективным по времени выполнения и по затратам памяти, а итеративный алгоритм Левенштейна показал себя значительно лучше остальных. Разница в выполнении алгоритмов на размере слова 200 символов достигает 290 раз. У алгоритма Левенштейна рост скорости времени выполнения растет практически в геометрической прогрессии, но из наблюдений использование кэша для рекурсивной версии алгоритма Левенштейна позволяет сократить время выполнения до 2 раз относительно его версии без кэша.

Список использованной литературы

[1] Расстояние Левенштейна, Jesse Russel и Ronald Cohn [Книга]. Дата обращения: 13.09.2021

[2] Наследование в Python [Электронный ресурс] Режим доступа: https://younglinux.info/oopython/inheritance. Дата обращения: 13.09.2021

[3] Гасфилд. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003. [Книга]. Дата обращения: 13.09.2021

[4] Вычисление процессорного времени выполенения программы [Электронный ресурс] Режим доступа: https://www.tutorialspoint.com/python/time_clock.htm. Дата обращения: 13.09.2021