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 №3

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 исследованию и сравнению алгоритмов сортировки.

Сортировка - процесс перегруппировки заданной последовательности в некотором определенном порядке, он необходим для удобной работы с этим объектом: В отсортированной последовательности данных поиск элемента происходит значительно быстрее, например при помощи алгоритма бинарного поиска будет затрачено время до логарифма количества элементов последовательности.

Наиболее важная характеристика алгоритма сортировки - это скорость его работы, она определяется функциональной зависимостью среднего времени сортировки последовательности данных, заданной длины, от этой длины. Время сортировки будет пропорционально количеству сравнений и перестановки элементов данных в процессе их сортировки.

Цель данной лабораторной работы: исследование и сравнение трех нерекурсивных алгоритмов сортировки.

Задачи данной лабораторной работы:

1. Изучение и реализация трех нерекурсивных алгоритмов сортировки: пузырек, выбором, вставками;
2. Провести сравнительный анализ трудоемкости алгоритмов;
3. Провести сравнительный анализ алгоритмов на основе экспериментальных данных;

1 Аналитическая часть

В данном разделе будут описаны нерекурсивные алгоритмы сортировки пузырьком, выбором, вставками.

1.1 Сортировка пузырьком

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется перестановка элементов. Проходы по массиву повторяются
N - 1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).

1.2 Сортировка выбором

Алгоритм состоит из повторяющихся операций сравнения и перемещения элементом из рассматриваемого массива: сначала необходимо найти номер минимального значения в текущем списке, после чего произвести обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции), и только потом отсортировать хвост списка, исключив из рассмотрения уже отсортированные элементы

1.3 Сортировка вставками

Алгоритм состоит из операций последовательного просмотра элементов не рассмотренной и уже отсортированной последовательности: на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем.

Для оптимизации работы алгоритма используют методы разделения последовательности на несколько частей.

1.4 Вывод

В аналитической части были описаны: нерекурсивные алгоритмы сортировки пузырьком, выбором, вставками.

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

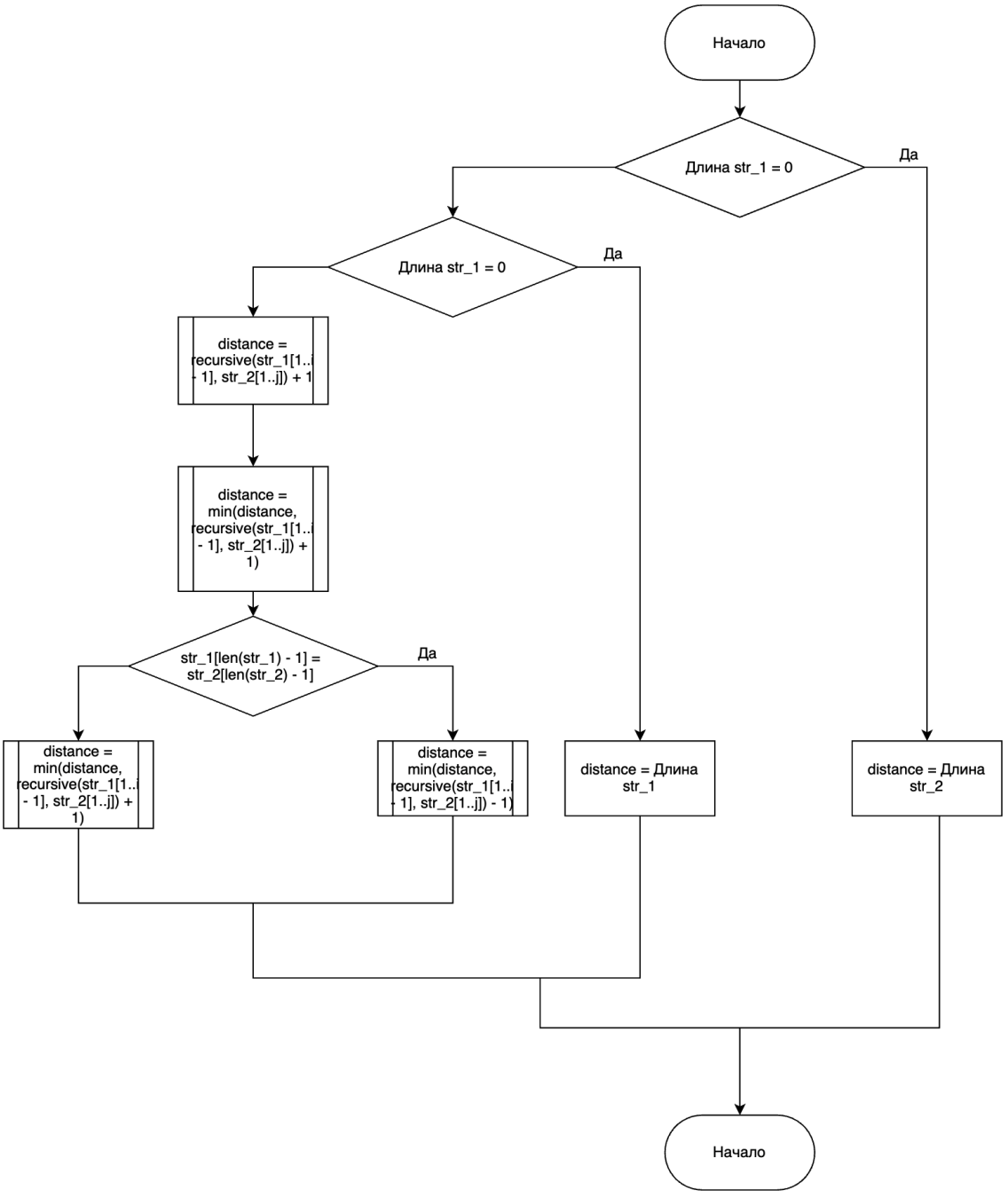

Рисунок 1: Схема рекурсивного алгоритма Левенштейна без Кэша

Рекурсивный алгоритм Левенштейна с кэшем

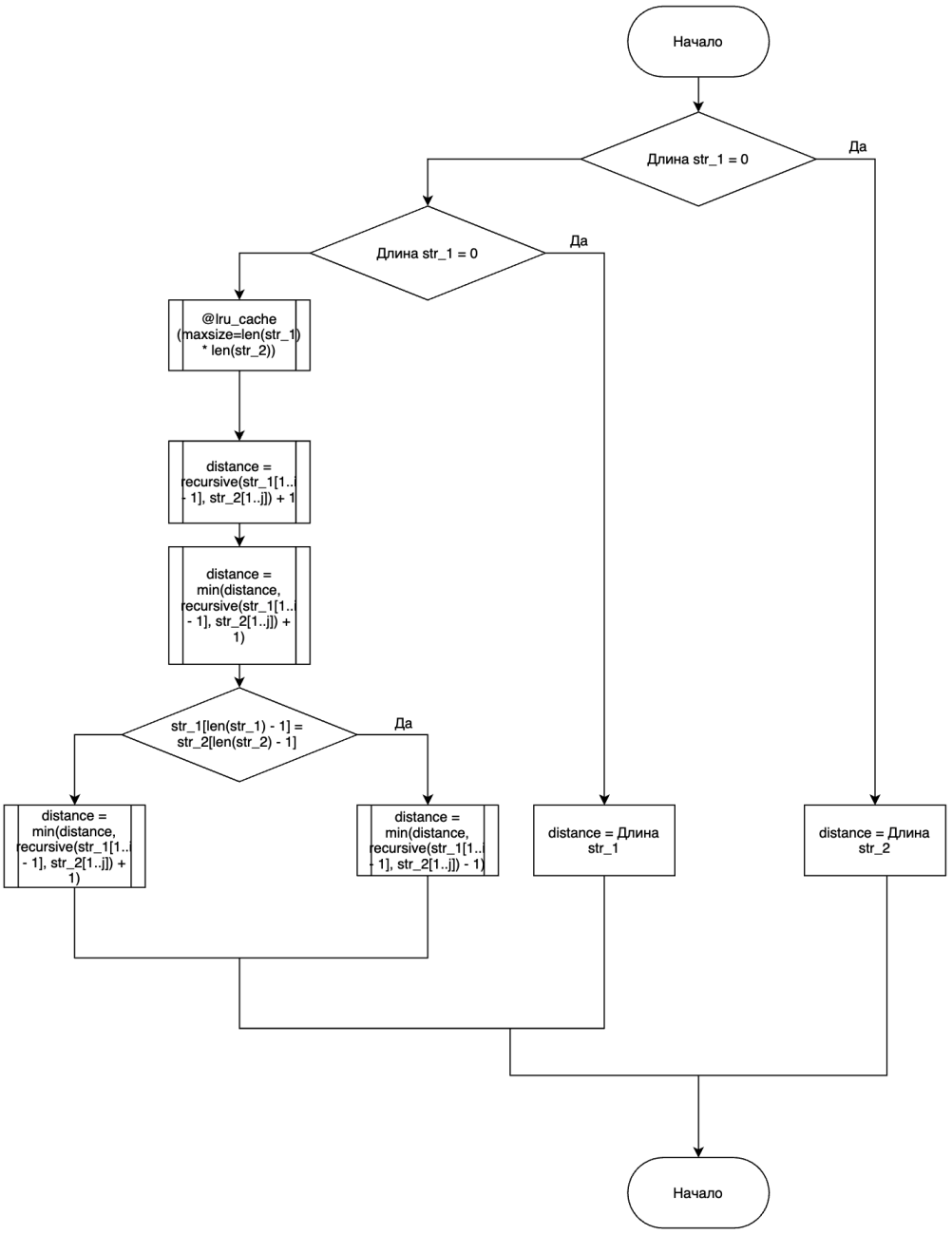

Рисунок 2: Схема рекурсивного алгоритма Левенштейна с Кэшем

Итеративный алгоритм Левенштейна

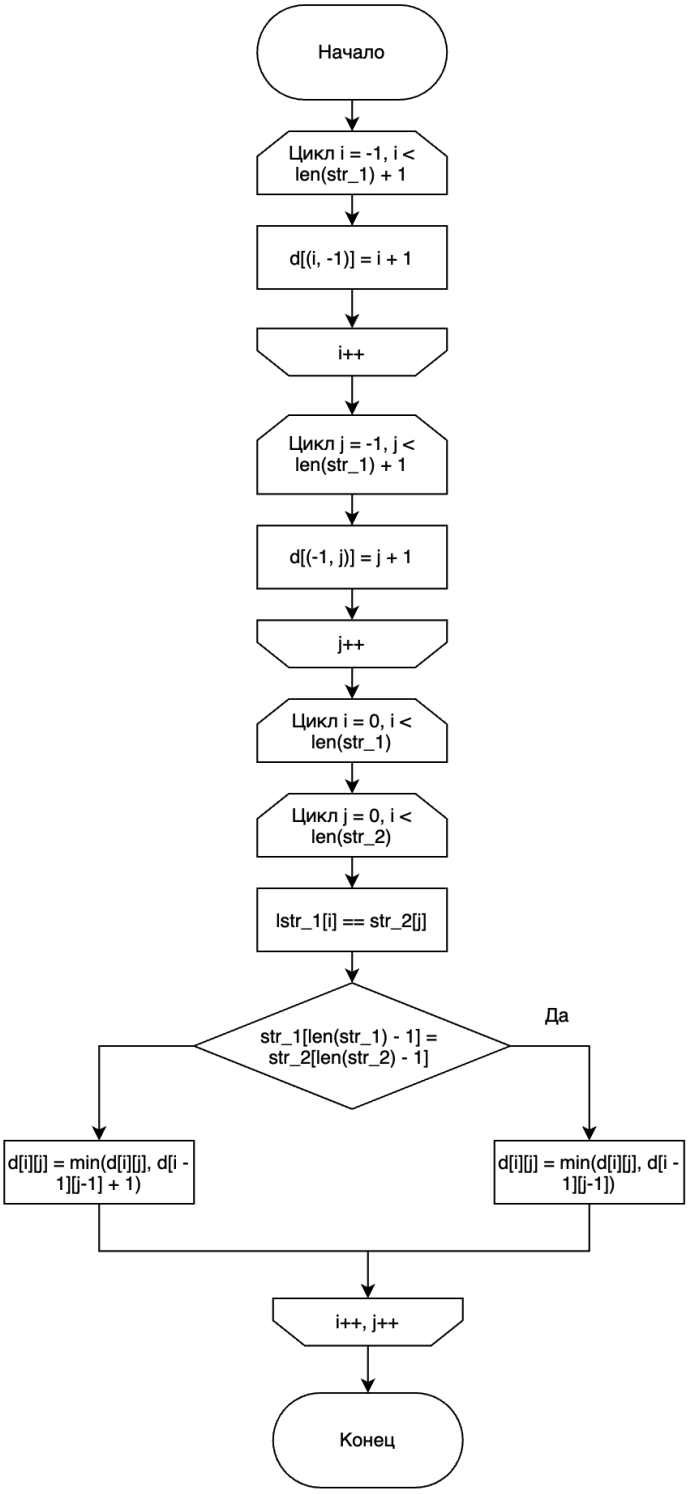

Рисунок 3: Схема итеративного алгоритма Левенштейна

Итеративный алгоритм Дамерау-Левенштейна

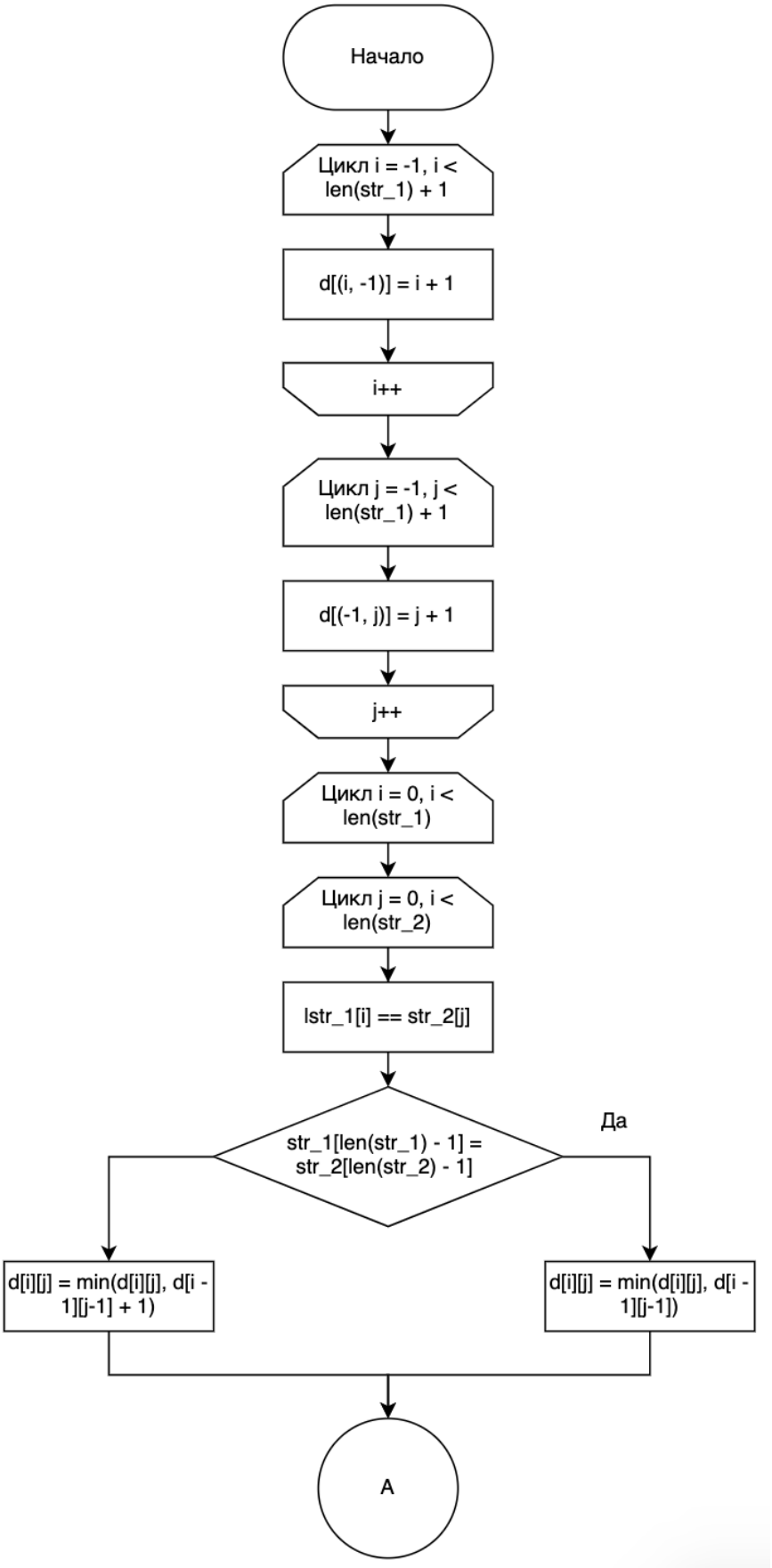

Рисунок 4: Схема итеративного алгоритма Дамерау-Левенштейна, Часть 1

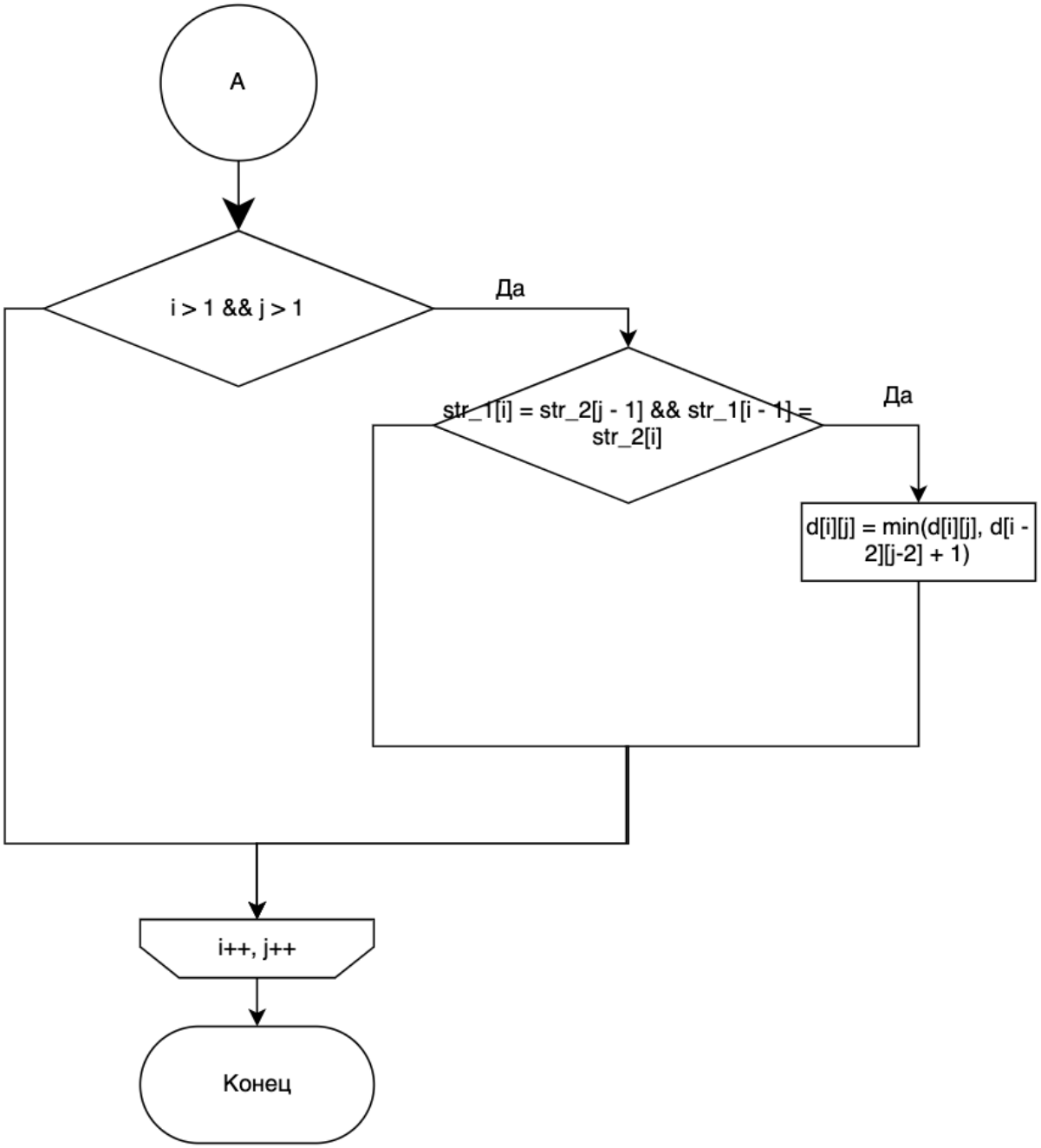

Рисунок 5: Схема итеративного алгоритма Дамерау-Левенштейна, Часть 2

2.2 Модель вычислений

2.3 Трудоемкость алгоритмов

2.3.1 Сортировка пузырьком

2.3.2 Сортировка выбором

2.3.3 Сортировка вставками

2.4 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

```
1. # Объект алгоритма Левенштейна
2. class Levenshtein:
3.     # Защищенные наследуемые данные объекта
4.     _first_string = None
5.     _second_string = None
6.
7.     # Ключевое расстояние
```

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

```
8.       _distance = None
9.
10.      # Ключевое время
11.      _time = None
12.
13.      # Создание объекта
14.      def __init__(self, first_string, second_string):
15.          # Назначение данных объекта
16.          self._first_string = first_string
17.          self._second_string = second_string
18.
19.          # Установка значения расстояния
20.          self._distance = config.START_ZERO_VALUE
21.
22.          # Установка значения времени выполнения
23.          self._time = config.START_ZERO_VALUE
24.
25.      # Общая функция получения расстояния между двумя строками
26.      def get_distance(self):
27.          return self._distance
28.
29.      # Общая функция получения времени выполнения
30.      def get_time(self):
31.          return self._time
32.
33.      # Получение расстояния между строками
34.      def _recursive_get_distance(self, first_string_length,
   second_string_length):
35.
36.          # если одна из строк пустая, то расстояние до другой
   строки - ее длина
37.          # т.е. n вставок
38.          if first_string_length == 0 or second_string_length
   == 0:
39.              return max(first_string_length,
   second_string_length)
```

```
40.
41.             # если оба последних символов одинаковые, то съедаем
    их оба, не меняя расстояние
42.             elif self._first_string[first_string_length - 1] ==
    self._second_string[second_string_length - 1]:
43.                 return
    self._recursive_get_distance(first_string_length - 1,
    second_string_length - 1)
44.
45.             # выбор минимального значения из трех
46.             else:
47.                 return 1 + min(
```

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

```
48.
    self._recursive_get_distance(first_string_length,
    second_string_length - 1),   # Удаление
49.
    self._recursive_get_distance(first_string_length - 1,
    second_string_length),   # Вставка
50.
    self._recursive_get_distance(first_string_length - 1,
    second_string_length - 1)   # Замена
51.                 )
```

Листинг 3: Наследуемый класс Левенштейна без кэша

```
1. # Наследуемый объект Рекурсивного алгоритма без кэша
2. class LevenshteinRecursiveWithoutCache(Levenshtein):
3.
4.     # Общая функция получения расстояния между двумя строками
5.     def get_distance(self):
6.         self._distance =
    self._recursive_get_distance(len(self._first_string),
    len(self._second_string))
7.         return self._distance
8.
9.     # Получение времени
10.        def get_time(self):
11.            t_0 = clock()
12.            self._recursive_get_distance(len(self._first_string),
    len(self._second_string))
13.            t_1 = clock()
14.
15.            self._time = t_1 - t_0
16.
17.            return self._time
18.
```

Листинг 4: Наследуемый класс Левенштейна с кэшем, Часть 1

```
1. # Наследуемый объект Рекурсивного алгоритма с кэшем
2. class LevenshteinRecursiveWithCache(Levenshtein):
3.     _first_string_length = None
4.     _second_string_length = None
5.
6.     def __init__(self, first_string, second_string):
7.         super().__init__(first_string, second_string)
8.
9.         self._first_string_length = len(self._first_string)
10.            self._second_string_length = len(self._second_string)
```

Листинг 5: Наследуемый класс Левенштейна с кэшем, Часть 2

```
11.
12.        # Получение времени
13.        def get_time(self):
14.            t_0 = clock()
15.            self.get_distance()
16.            t_1 = clock()
17.
18.            self._time = t_1 - t_0
19.
20.            return self._time
21.
22.        def get_distance(self):
23.            # Общая функция получения расстояния между двумя
   строками
24.            @lru_cache(maxsize=self._first_string_length *
   self._second_string_length)
25.            def get_distance():
26.                self._distance =
   self._recursive_get_distance(len(self._first_string),
   len(self._second_string))
27.                return self._distance
28.
29.            # Обновление расстояния
30.            self._distance = get_distance()
31.
32.            return self._distance
```

Листинг 6: Наследуемый класс итерационного Дамерау Левенштейна, Часть 1

```
1. # Объект вычисления Дамерау Левенштейна
2. class DamerauLevenshtein(Levenshtein):
3.     # Используемые блины строк
4.     _first_string_length = None
5.     _second_string_length = None
6.
7.     def __init__(self, first_string, second_string):
8.         super().__init__(first_string, second_string)
9.
```

```
10.           self._first_string_length = len(self._first_string)
11.           self._second_string_length = len(self._second_string)
12.
13.       # Получение расстояния между двумя строками
14.       def get_distance(self):
15.           d = {}
16.
17.           for i in range(-1, self._first_string_length + 1):
18.               d[(i, -1)] = i + 1
19.           for j in range(-1, self._second_string_length + 1):
20.               d[(-1, j)] = j + 1
21.
22.           for i in range(self._first_string_length):
23.               for j in range(self._second_string_length):
```

Листинг 7: Наследуемый класс итерационного Дамерау Левенштейна, Часть 2

```
24.                   if self._first_string[i] ==
    self._second_string[j]:
25.                       cost = 0
26.                   else:
27.                       cost = 1
28.                   d[(i, j)] = min(
29.                       d[(i - 1, j)] + 1,  # deletion
30.                       d[(i, j - 1)] + 1,  # insertion
31.                       d[(i - 1, j - 1)] + cost,  # substitution
32.                   )
33.                   if i and j and self._first_string[i] ==
    self._second_string[j - 1] and \
34.                           self._first_string[i - 1] ==
    self._second_string[j]:
35.                       d[(i, j)] = min(d[(i, j)], d[i - 2, j - 2]
    + cost)  # transposition
36.
37.           return d[self._first_string_length - 1,
    self._second_string_length - 1]
38.
39.       # Получение времени
40.       def get_time(self):
41.           t_0 = clock()
42.           self.get_distance()
43.           t_1 = clock()
44.
45.           self._time = t_1 - t_0
46.
47.           return self._time
```

Листинг 8: Наследуемый класс Левенштейна с кешем, Часть 1

```
1. # Линейный алгоритм вычисления Левенштейна
2. class LevenshteinLinear(Levenshtein):
3.     # Используемые длины строк
4.     _first_string_length = None
```

```
5.     _second_string_length = None
6.
7.     def __init__(self, first_string, second_string):
8.         super().__init__(first_string, second_string)
9.         self._first_string_length = len(first_string)
10.            self._second_string_length = len(second_string)
11.
12.        def _update_distance(self):
13.            if self._first_string_length >
    self._second_string_length:
14.                self._first_string, self._second_string =
    self._second_string, self._first_string
```

Листинг 9: Наследуемый класс Левенштейна с кэшем, Часть 2

```
15.                self._first_string_length,
    self._second_string_length = self._second_string_length,
    self._first_string_length
16.
17.            current_row = range(self._first_string_length + 1)
18.            for i in range(1, self._second_string_length + 1):
19.                previous_row, current_row = current_row, [i] + [0]
    * self._first_string_length
20.                for j in range(1, self._first_string_length + 1):
21.                    add, delete, change = previous_row[j] + 1,
    current_row[j - 1] + 1, previous_row[j - 1]
22.                    if self._first_string[j - 1] !=
    self._second_string[i - 1]:
23.                        change += 1
24.                    current_row[j] = min(add, delete, change)
25.
26.            return current_row[self._first_string_length]
27.
28.        # Получение расстояния между двумя строками
29.        def get_distance(self):
30.            self._distance = self._update_distance()
31.            return self._distance
32.
33.        # Получение времени
34.        def get_time(self):
35.            t_0 = clock()
36.            self.get_distance()
37.            t_1 = clock()
38.
39.            self._time = t_1 - t_0
40.
41.            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				Полученный результат			
			Расстояние				Расстояние			
			Л. 2	Л. 3	Л. 4	Л.5	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	мтгу	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.

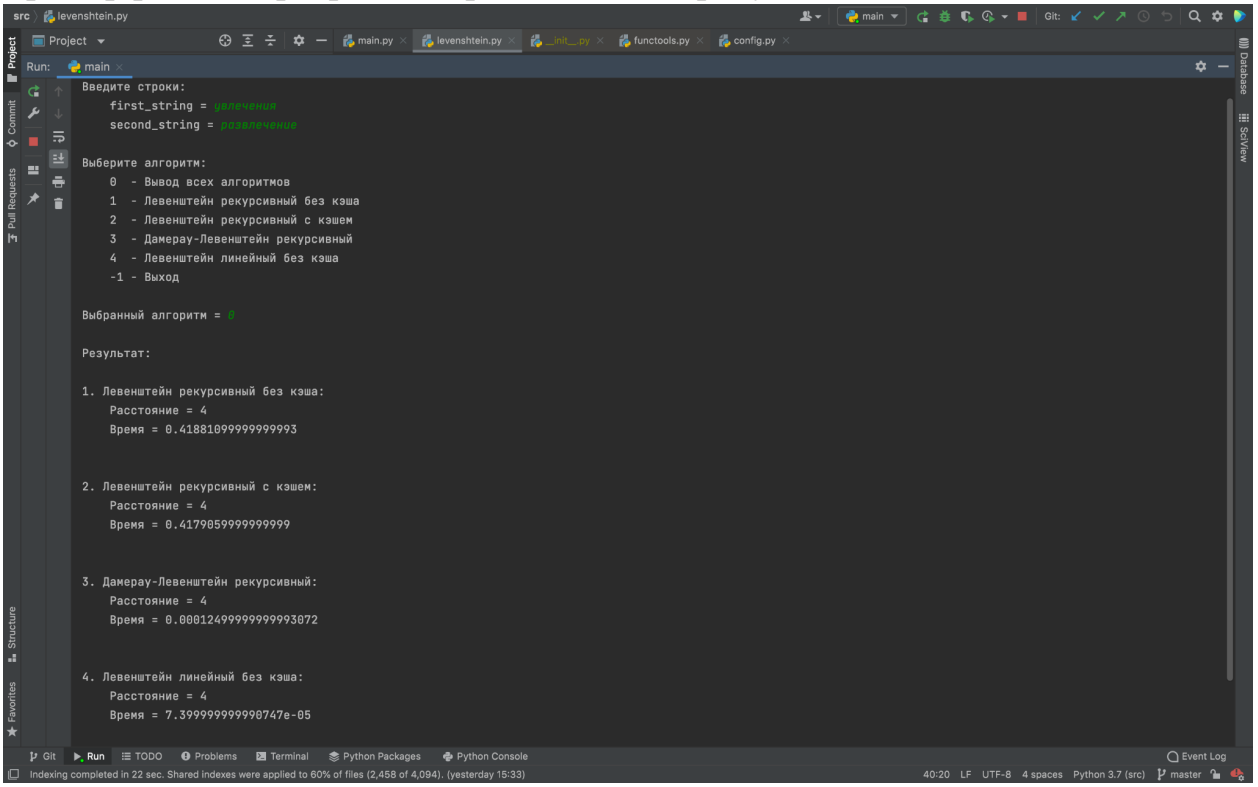

Рисунок 6: Демонстрация работы программы на примере строк Увлечения и Развлечение

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