1 Цель работы

Научиться реализовывать алгоритмы сортировки.

2 Задание

Вариант 4

Задание на реализацию:

Задание 2:

Реализуйте структуру данных «Массив», элементами которого выступают экземпляры класса Car (минимум 10 элементов), содержащие следующие поля (марка, VIN, объем двигателя, стоимость, средняя скорость). Добавьте методы для сортировки вставками (по возрастанию) по полю «VIN» и сортировка расческой (по убыванию) по полю «средняя скорость».

Задание 5:

Реализуйте структуру данных «Массив», элементами которого выступают экземпляры класса Book (минимум 10 элементов), содержащие следующие поля (автор, издательство, кол-во страниц, стоимость, ISBN). Добавьте методы для быстрой сортировки (по возрастанию) по полю «кол-во страниц» и сортировки по основанию (по убыванию) по полю «стоимость»

3 Теория по заданиям

Сортировка вставками (Insertion sort) - работает путем последовательной вставки каждого элемента несортированного списка в его правильное положение в отсортированной части списка.

Этапы алгоритма:

- Начинается с первого элемента массива, первый элемент массива считается уже отсортированным;
- Сравнивается второй элемент с первым и проверяет, если второй элемент меньше, то поменяет их местами;
- Переход к третьему элементу, сравнивание его с первыми двумя элементами и располагается в правильное положение;
- Повторяется до тех пор, пока не будет отсортирован весь массив.

Пример работы алгоритма сортировки вставками по возрастанию на рисунке 3.1.

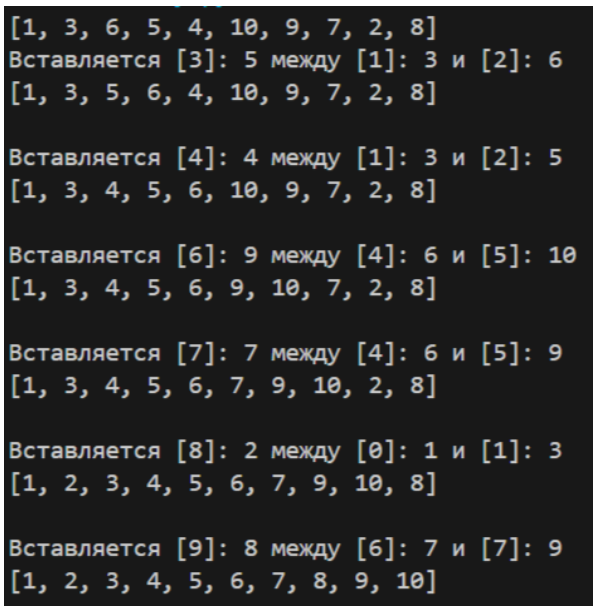

Рисунок 3.1 – Пример работы сортировки вставками массива из чисел

Сортировка расчёской (Comp sort):

Этапы алгоритма:

- Берётся разрыв между сравниваемыми элементами берётся максимальный;
- На каждой итерации разрыв уменьшается путём деления расстояния на коэффициент k = 1,3;
- Процедура продолжается до тех пор, пока разность индексов сравниваемых компонентов не станет равна единице;
- Достигнув единицы, алгоритм будет сравнивать соседние элементы — как в пузырьковом методе. Эта итерация окажется последней.

Пример работы алгоритма сортировки расчёской по убыванию на рисунке 3.2.

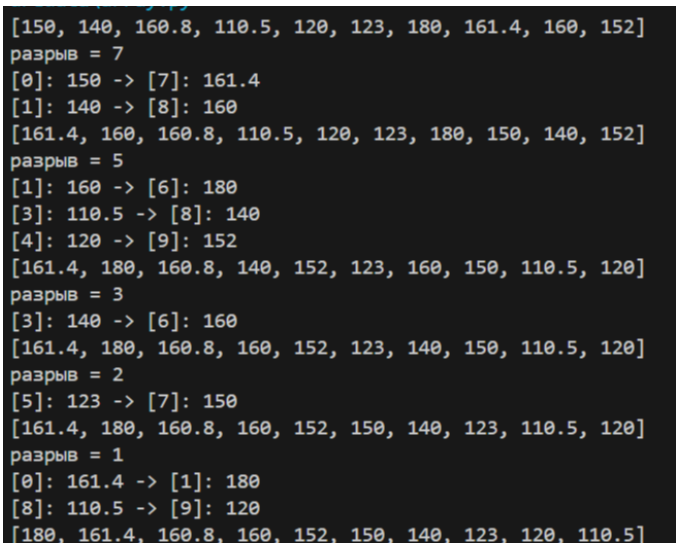

Рисунок 3.2 - Пример работы сортировки расчёской массива из чисел

Быстрая сортировка (Quack sort):

Этапы алгоритма:

- Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. В задание 5 в качестве опорной точки выбирается самый последний из каждого нового заданного диапазона.
- Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на два непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного» и «равные» и «большие».
- Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Пример работы алгоритма быстрой сортировки по возрастанию на рисунке 3.3.

```
[1, 3, 6, 5, 4, 10, 9, 7, 5, 2, 8]

Диапазон от 0 до 10 по левой стороне от опорной - опорная точка 8
опорная точка 8 сдвинусаль влево на 2 шага
[1, 3, 6, 5, 4, 7, 5, 2, 8, 10, 9]

Диапазон от 0 до 7 по левой стороне от опорной - опорная точка 2
опорная точка 2 сдвинусаль влево на 6 шага
[1, 2, 3, 6, 5, 4, 7, 5, 8, 10, 9]

Диапазон от 2 до 7 по правой стороне от опорной - опорная точка 5
опорная точка 5 сдвинусаль влево на 2 шага
[1, 2, 3, 5, 4, 5, 6, 7, 8, 10, 9]

Диапазон от 2 до 4 по левой стороне от опорной - опорная точка 4
опорная точка 4 сдвинусаль влево на 1 шага
[1, 2, 3, 4, 5, 5, 6, 7, 8, 10, 9]

Диапазон от 6 до 7 по правой стороне от опорной - опорная точка 7
опорная точка 7 сдвинусаль влево на 0 шага
[1, 2, 3, 4, 5, 5, 6, 7, 8, 10, 9]

Диапазон от 9 до 10 по правой стороне от опорной - опорная точка 9
опорная точка 9 сдвинусаль влево на 1 шага
[1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10]
```

Рисунок 3.3 - Пример работы быстрой сортировки массива из целых чисел

Сортировка по основанию (Radix sort) - это алгоритм сортировки, который сортирует не числа целиком, а значения разрядов.

Он разбирается с числами на уровне единиц, десятков, сотен и т. д. и только потом делает общую сортировку. Это позволяет алгоритму не бегать по всем сравниваемым числам и не делать миллион сравнений.

Принцип работы поразрядной сортировки заключается в том, что данные сначала делятся по разрядам, а потом сортируются внутри каждого разряда.

Пример работы алгоритма сортировки по основанию по убыванию на рисунке 3.4.

```
[472, 473, 401, 277, 370, 320, 250, 501, 399, 410]
10^0 [[399], [], [277], [], [], [], [473], [472], [401, 501], [370, 320, 250, 410]]
10^1 [[399], [], [277, 473, 472, 370], [], [250], [], [], [320], [410], [401, 501]]
10^2 [[], [], [], [], [501], [473, 472, 410, 401], [399, 370, 320], [277, 250], [], []]
[501, 473, 472, 410, 401, 399, 370, 320, 277, 250]
```

Рисунок 3.4 – Пример работы сортировки по основанию массива из целых чисел

Массив до сортировки равен [472, 473, 401, 277, 370, 320, 250, 501, 399, 410]. Дальше показаны этапы сортировки с каждой новой строкой по разрядам единиц (10^0), десяток (10^1) и сотен (10^2).

Последняя строка – итог алгоритма после сортировки по основанию равен [501, 473, 472, 410, 401, 399, 370, 320, 277, 250].

4 Код программы для задания 2

Структура данных Массив – Array для задания 2 и задания 5 со всеми методами.

Путь файла с Массивом– structuredata/array.py

```python
from temp.car import Car
from temp.book import Book
import ctypes
from typing import Generic, TypeVar, Iterator

T = TypeVar("T")

class Array(Generic[T]):
    # основная переменная - вместимость массива:
    def __init__(self, capacity: int) -> None:
        self.length: int = 0
        self.capacity: int = capacity
        # создаём массив на основе Си
        self.arr: ctypes.Array[T] = (capacity * ctypes.py_object)()

    # возвращает длину
    def get_length(self) -> int:
        return self.length

    # возвращает вместимость
    def get_capacity(self) -> int:
        return self.capacity

    # получить значение по индексу
    def get(self, index: int) -> T:
        # если индекс вне диапазона длины массива, то ошибка
        if index >= self.get_length() or index < 0:
            raise "Index is out of range"
        return self.arr[index]
```

```
    # магический метод - получение значения по индексу
    def __getitem__(self, index: int) -> T:
        # если индекс вне диапазона длины массива, то ошибка
        if index >= self.get_length() or index < 0:
            raise "Index is out of range"
        return self.arr[index]

    # выделяем больше места в массиве (увеличивает вместимость)
    def expand_capacity(self, cap: int) -> None:
        size = cap
        if size <= 0:
            size = 1
        # создаёт новый массив с новой вместимостью
        new_arr: ctypes.Array[T] = (size * ctypes.py_object)()
        # передаются все значения прошлого массива
        for i in range(self.get_length()):
            new_arr[i] = self.get(i)
        # передаём новые данные массив и вместимость
        self.arr: ctypes.Array[T] = new_arr
        self.capacity = size

    # добавляет в конец массива элемент
    def append(self, elem: T) -> None:
        # если вместимость кончается, то увеличиваем
        if self.length == self.capacity:
            self.expand_capacity(self.capacity * 2)
        # добавляем элемент
        self.arr[self.length] = elem
        self.length += 1

    def insert(self, elem: T, index: int) -> None:
        if index > self.length or index < 0:
            raise "Index is out of range"
        # если вместимость кончается, то увеличиваем
        if self.length == self.capacity:
            self.expand_capacity(self.capacity * 2)
        for i in range(self.length, index - 1, -1):
            if index == 0 and i - 1 < 0:
                self.arr[0] = elem
                continue
            self.arr[i] = self.arr[i - 1]
        self.arr[index] = elem
        self.length += 1

    # Магический метод замены значения по индексу
    def __setitem__(self, index: int, elem: T) -> None:
        # если индекс вне диапазона длины массива, то ошибка
        if index >= self.get_length() or index < 0:
            raise "Index is out of range"
        self.arr[index] = elem
```

```
    # удаление по индексу - возвращает удалённый элемент
    def remove(self, index: int) -> T:
        # проверка индекса
        if index >= self.get_length() or index < 0:
            raise "Index is out of range"

        elem: T = self.get(index)
        # сдвиг значений по индексам плево на 1 шаг от индекса до
длины - 1
        for i in range(index, self.get_length() - 1):
            self.arr[i] = self.arr[i + 1]

        self.length -= 1
        return elem

    # очистка (пустой массив)
    def clear(self) -> None:
        self.length = 0
        self.arr: ctypes.Array[T] = (self.capacity *
ctypes.py_object)()

    # Сортировка вставками - по возрастанию для Car - VIN
    def sort_insertion(self) -> None:
        # если массив пуст, то ничего не делается
        if self.length <= 0:
            return
        # проверка на тип - если класс Машина, то задаём основное
значение - vin
        if type(self.arr[0]) == Car:
           for i in range(self.length):
                self.arr[i].set_flag("vin")
        # от 0 до длинны массива, i - индекс значения, который будем
вставлять
        for i in range(self.length):
            # вставка self.arr[i] за j - индексом значения, отсчёт
идет справа налево
            for j in range(i - 1, -1, -1):
                # Вставка происходит если значение индекса i будет
меньше значение индекса j вместе с (вставка в начало j == 0 или если
соседнее значение меньше чем вставляемое значение)
                if self.arr[i] < self.arr[j] and (j == 0 or
self.arr[i] > self.arr[j - 1]):
                    # Вставка значения - копия в нужную позицию
                    self.insert(self.arr[i], j)
                    # Удаление значения со старой позиции
                    self.remove(i + 1)
                    # выходим из цикла, так как вставка уже произошла
                    break

    # Сортировка расчёской - по убыванию для Car - средняя скорость
```

```
    def sort_comb(self) -> None:
        # если массив пуст, то ничего не делается
        if self.length <= 0:
            return
        # проверка на тип - если класс Машина, то задаём основное
значение - средняя скорость
        if type(self.arr[0]) == Car:
            for i in range(self.length):
                self.arr[i].set_flag("speed")

        # Функция - меняет местами значения по индексам
        def swap(ind1: int, ind2: int) -> None:
            self.arr[ind1], self.arr[ind2] = self.arr[ind2],
self.arr[ind1]

        # коэффициент разрыва
        k = 1.3
        # разрыв
        size = self.length
        # пока разрыв больше одного, то сортировка продолжается
(разрыв = 1 в цикле присутствует)
        while size > 1:
            # задаём разрыв
            size = int(size // k)
            # идёт сортировка с диапазоном разрыва от 0 до длина
массива - разрыв
            for i in range(self.length - size):
                # если значение меньше, то меняем местами
                if self.arr[i] < self.arr[i + size]:
                    swap(i, i + size)

    # Рекурсивная Сортировка быстрая - по возрастанию для Book - кол-
во страниц
    def rec_sort_quick(self, start: int, end: int) -> None:
        # рекурсия идёт, пока конец меньше начала диапазона сортировки
        if start < end:
            # опорная точка - конец (от отобранного диапазона)
            pivot = self.arr[end]
            # less на сколько сдвинулась опорная точка влево
            less = 0
            for i in range(start, end):
                # если опорная точка меньше элемента то элемент ставим
в конец диапазона end
                if pivot < self.arr[i - less]:
                    # удаляем элемент для переноса в конец диапазона
                    elem = self.remove(i - less)
                    # вставка элемента вправо в конец диапазона
                    self.insert(elem, end)
                    # сдвиг опорной точки влево + 1
                    less += 1
```

```
            # end - less = индекс опорной точки (она стоит на месте),
рекурсия продолжается с левой и правой сторонной опорной точки
            self.rec_sort_quick(start, end - less - 1)
            self.rec_sort_quick(end - less + 1, end)

    # Сортировка быстрая - по возрастанию для Book - кол-во страниц
    def sort_quick(self) -> None:
        # если массив пуст, то ничего не делается
        if self.length <= 0:
            return
        # проверка на тип - если класс Книга, то задаём основное
значение - кол-во страниц
        if type(self.arr[0]) == Book:
            for i in range(self.length):
                self.arr[i].set_flag("pages")
        # рекурсия быстрой сортировки
        self.rec_sort_quick(0, self.get_length() - 1)

    # Сортировка по основанию - по убыванию для Book - стоимость
    def sort_radix(self) -> None:
        # если массив пуст, то ничего не делаем
        if self.length <= 0:
            return
        # проверка на тип - если класс Книга, то задаём основное
значение - стоимость
        if type(self.arr[0]) == Book:
            for i in range(self.length):
                self.arr[i].set_flag("cost")
        # сортировка по десятичной системе от 0 до 9 (включительно) -
список из списков
        nums = [[] for _ in range(10)]
        # новый массив для сортировки - копия
        new_arr: ctypes.Array[T] = (self.length * ctypes.py_object)()
        for i in range(self.get_length()):
            new_arr[i] = self.get(i)
        # максимальная длина целого числа в массиве
        countmx = len(str(max(new_arr, key=lambda x: len(str(x)))))
        # от 0 до максимальной длины - сколько разрядов должен пройти
для сортировки
        for round in range(countmx):
            # сортировка одного из разрядов (round) целых чисел
            for i in new_arr:
                # значение - целое число
                flag = str(i)
                # если разряда нету (привышен), то считаем его за ноль
                if len(flag) >= round + 1:
                    # индекс разряда (начиная с конца целого числа)
                    index = (len(flag) - 1) - round
                    # добавляем в список десятичной системы
                    nums[(len(nums) - 1) - int(flag[index])] += [i]
                else:
```

```
                    nums[len(nums) - 1] += [i]
            # счётчик индекса массива
            index = 0
            # заменяем отсортированные значения
            for i in nums:
                for j in i:
                    new_arr[index] = j
                    index += 1
            # очищяем список сортировки
            nums.clear()
            nums = [[] for _ in range(10)]
        # добавляем в основной массив отсортированные значения
        for i in range(self.length):
            self.arr[i] = new_arr[i]

    # магический метод итерирования - возвращает значение
    def __iter__(self) -> Iterator[T]:
        for i in range(self.get_length()):
            yield self.get(i)

    # магический метод вывода класса массива в формате [знач1, знач2,
... знач N]
    def __str__(self) -> str:
        text = ""
        for i in range(self.get_length()):
            text += str(self.get(i)) + ", "
        if text == "":
            return "[]"
        return f"[{text[:-2]}]"

    # магический метод предназначен для машинно-ориентированного
вывода
    def __repr__(self) -> str:
        return self.__str__()
```

Для аннотации типа берём из библиотеки **typing** функции - Generic, Optional, TypeVar, Iterator:

- **TypeVar** – любой тип переменной(если задаётся int то всегда будет только int если str то str и т.д.); (универсальный тип);
- **Generic** – Универсальный класс (Абстрактный базовый класс для универсальных типов.) чтобы использовать нужны TypeVar, только аргументы этого типа. (показывает, что за тип без него(Generic) тип не показывает);
- **Optional** – может хранить либо None либо узел списка (T);
- **Iterator** – указывает на то, что объект имеет реализацию метода iter.

Методы класса Массив:

- **__init__(capacity)** – принимает в себя вместимость, создаёт Массив и заполняет его пустотой;
- **get_leght()** – возвращает длину;
- **get_capacity()** – возвращает вместимость;
- **get(index)** – получить значение по индексу;
- **__getitem__(index)** – магический метод - получение значения по индексу;
- **expand_capacity(cap)** – выделяем больше места в массиве (увеличивает вместимость);
- **append(elem)** – добавляет в конец массива элемент;
- **insert(elem, index)** – вставляет элемент на заданную позицию в массив;
- **__setitem__(index, elem)** – Магический метод замены значения по индексу;
- **remove(index)** – удаление по индексу - возвращает удалённый элемент;
- **clear()** – очистка (пустой массив);
- **sort_insertion()** – Сортировка вставками - по возрастанию для Car – VIN;
- **sort_comb()** – Сортировка расчёской - по убыванию для Car - средняя скорость;
- **rec_sort_quick(start, end)** – Рекурсивная Сортировка быстрая - по возрастанию для Book - кол-во страниц;
- **sort_quick()** – Сортировка быстрая - по возрастанию для Book - кол-во страниц;
- **sort_radix()** – сортировка по основанию - по убыванию для Book – стоимость;
- **__iter__()** – магический метод итерирования - при итерировании будет возвращать значение по индексу;
- **__str__()** – магический метод вывода класса массива в формате [знач1, знач2, ... знач N];
- **__repr__()** – магический метод предназначен для машинно-ориентированного вывода.

Класс Car(Машины) для 2 задания:

Путь файла класса Car и примерами элементов этого класса (element) – temp/car.py

```
from typing import Self, Optional
from dataclasses import dataclass
from functools import total_ordering
```

```python
# дополняет недостающие методы за счёт других (изменятся >=, <=, !=,
>)
@total_ordering
@dataclass
class Car:
    mark: str
    vin: str
    engine_capacity: int
    cost: int
    average_speed: float

    # флаг - переключатель основного параметра - по умолчанию cost
    def __post_init__(self) -> None:
        self.flag = "cost"

    # задать основной параметр
    def set_flag(self, text: str) -> None:
        if text in ["mark", "vin", "capacity", "cost", "speed"]:
            self.flag = text

    # получить основной параметр или получить параметр
    def get_flag(self, text: Optional[str] = None) -> str | float:
        tx = self.flag
        if text is not None:
            tx = text
        if tx == "mark":
            return self.mark
        elif tx == "vin":
            return self.vin
        elif tx == "capacity":
            return self.engine_capacity
        elif tx == "cost":
            return self.cost
        elif tx == "speed":
            return self.average_speed
        return self.cost

    # магический метод - меньше <
    def __lt__(self, other: Self) -> bool:
        other_any = other.get_flag(self.flag)
        return self.get_flag() < other_any

    # магический метод - равно ==
    def __eq__(self, other: Self) -> bool:
        other_any = other.get_flag(self.flag)
        return self.get_flag() == other_any

    # магический метод - сложение +
    def __add__(self, other: Self) -> str | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() + other_any
```

```
    # магический метод - вычитание -
    def __sub__(self, other: Self) -> float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() - other_any

    # магический метод - умножение *
    def __mul__(self, other: Self) -> str | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() * other_any

    # магический метод - деление /
    def __truediv__(self, other: Self) -> float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() / other_any

    # магический метод - деление на цело //
    def __floordiv__(self, other: Self) -> int | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() // other_any

    # магический метод - вывода
    def __str__(self) -> str:
        return str(self.get_flag())

    # магический метод предназначен для машинно-ориентированного
вывода
    def __repr__(self) -> str:
        return str(self.get_flag())


# 10 раличных модель машин (10 элементов для дерева)
element = [
    Car("Toyota", "JTJHY00W004036549", 10, 2_000_000, 150),
    Car("Lexus", "JTJHT00W604011511", 20, 2_104_000, 140),
    Car("Hyundai", "KMFGA17PPDC227020", 15, 3_003_000, 160.8),
    Car("Haval", "LGWFF4A59HF701310", 10, 2_500_000, 110.5),
    Car("Jeep", "1J8GR48K25C547741", 12, 2_111_000, 120),
    Car("Chery", "LVVDB11B6ED069797", 8, 1_8000_500, 123),
    Car("Ford", "1FACP42D3PF141464", 30, 4_003_900, 180),
    Car("Tesla", "5YJ3E1EA8KF331791", 24, 3_060_000, 161.4),
    Car("Lada", "XTAKS0Y5LC6599289", 16, 2_044_000, 160),
    Car("Ravon", "XWBJA69V9JA004308", 18, 3_900_000, 152)
]
```

Декоратор **@dataclass** – убирает лишние методы (автоматизирует) __init__, __repr__, __str__ и __eq__ вместо этого можно сразу аннотацию писать

Например было __init__(self, name) -> self.name = name, станет name: str

__post_init__(self) - метод, который срабатывает после __init__, здесь можно инициализировать переменные и всё, что можно было в __init__.

Декоратор **@total_ordering** заполняет не достающие методы по уже существующем например по __lt__ и __eq__, задаст остальные методы сравнения.

Методы set_flag(self, text) задаёт основное значение по названию (по умолчанию – стоимость), get_flag(self) даёт основное значение.

Метод массива сортировки вставками (sort_insertion) по возрастанию для Класса Car (Машины) по полю vin (VIN):

```
    # Сортировка вставками - по возрастанию для Car - VIN
    def sort_insertion(self) -> None:
        # если массив пуст, то ничего не делается
        if self.length <= 0:
            return
        # проверка на тип - если класс Машина, то задаём основное
значение - vin
        if type(self.arr[0]) == Car:
           for i in range(self.length):
                self.arr[i].set_flag("vin")
        # от 0 до длинны массива, i - индекс значения, который будем
вставлять
        for i in range(self.length):
            # вставка self.arr[i] за j - индексом значения, отсчёт
идет справа налево
            for j in range(i - 1, -1, -1):
                # Вставка происходит если значение индекса i будет
меньше значение индекса j вместе с (вставка в начало j == 0 или если
соседнее значение меньше чем вставляемое значение)
                if self.arr[i] < self.arr[j] and (j == 0 or
self.arr[i] > self.arr[j - 1]):
                    # Вставка значения - копия в нужную позицию
                    self.insert(self.arr[i], j)
                    # Удаление значения со старой позиции
                    self.remove(i + 1)
                    # выходим из цикла, так как вставка уже произошла
                    break
```

Метод сортировки вставками сначала проверяет на размер массива, после сам тип элемента в массиве, если это Car, то задаём основное поле – vin (VIN).

Сортировка идёт последовательно от начала до длины массива. Начальный элемент проходит без изменений. Дальше с каждым новой итерацией, значение которое будет сравнивается и вставится между меньшим соседом и значением индекса j. Происходит вставка копии значения и удаляется старое значение на прошлой позиции. И так пока все элементы в массиве не кончатся.

Метод массива сортировки расчёской (sort_comp) по убыванию для Класса Car (Машины) по полю average_speed (средняя скорость):

```
    # Сортировка расчёской - по убыванию для Car - средняя скорость
    def sort_comb(self) -> None:
        # если массив пуст, то ничего не делается
        if self.length <= 0:
            return
        # проверка на тип - если класс Машина, то задаём основное
значение - средняя скорость
        if type(self.arr[0]) == Car:
            for elem in self.arr:
                elem.set_flag("speed")

        # Функция - меняет местами значения по индексам
        def swap(ind1: int, ind2: int) -> None:
            self.arr[ind1], self.arr[ind2] = self.arr[ind2],
self.arr[ind1]

        # коэффициент разрыва
        k = 1.3
        # разрыв
        size = self.length
        # пока разрыв больше одного, то сортировка продолжается
(разрыв = 1 в цикле присутствует)
        while size > 1:
            # задаём разрыв
            size = int(size // k)
            # идёт сортировка с диапазоном разрыва от 0 до длина
массива - разрыв
            for i in range(self.length - size):
                # если значение меньше, то меняем местами
                if self.arr[i] < self.arr[i + size]:
                    swap(i, i + size)
```

Метод сортировки расчёской сначала проверяет на размер массива, после сам тип элемента в массиве, если это Car, то задаём основное поле – average_speed (средняя скорость).

Имеется функция swap, меняющий местами позиции значений по индексам, чтобы менять места значений при сортировки.

Задаётся коэффициент k=1.3 и разрыв size = длина массива. Сортировка продолжается пока разрыв не достигнет единицы. А до тех тот выполняет сортировку, где сравнивает индексы с разрывом от 0 до длины – разрыв. Если значение меньше значения разрыва, то меняет местами до конца цикла.

4.1 Тесты

Код теста:

```
from structuredata.array import Array
# element - содержит 10 различных элементов машин (10 различных Car)
from temp.car import Car, element

ex_car1 = Car("Haval 0.2v", "LGUFF4A59HF507891", 7, 3_890_000, 88)
ex_car2 = Car("Haval 0.3v", "JJJF4A59HF567892", 6, 3_000_100, 58)
same_cost = Car("Haval 0.4v", "LHHFF4G67HF777001", 5, 3_890_000, 180)

cap = 10
arr = Array(cap)
ind = 4

sorted_vin = sorted([i.vin for i in element])
sorted_average_speed = sorted([i.average_speed for i in element])
sorted_average_speed.reverse()

# добавление в конец
def test_append():
    for elem in element:
        arr.append(elem)

    for i in range(len(element)):
        assert element[i] == arr[i]

# проверка функций на get_ дающие значение
def test_gets():
    assert arr.get_capacity() == cap
    assert arr.get_length() == len(element)

    for i in range(len(element)):
        assert element[i] == arr.get(i)

# вставка с проверка размера
def test_resize_capacity():
    new_cap = cap * 2
    arr.append(element[2])
    assert arr.get_capacity() == new_cap

# вставка
def test_insert():
    arr.insert(ex_car1, ind)
    assert arr[ind] == ex_car1
    assert arr[ind - 1] == element[ind - 1]
    assert arr[ind + 1] == element[ind]
    arr.insert(ex_car2, 0)
    assert arr[0] == ex_car2
    assert arr[1] == element[0]
    arr.insert(same_cost, arr.get_length())
```

```
    assert arr[arr.get_length() - 1] == same_cost

# задаёт элемент по индексу
def test_set_elem():
    arr[0] = element[-1]
    assert arr[0] == element[-1]

# удаление и очистка
def test_remove_and_clear():
    assert arr.remove(0) == element[-1]
    assert arr.remove(arr.get_length() - 1) == same_cost
    assert arr.remove(arr.get_length() - 1) == element[2]
    assert arr.remove(ind) == ex_car1
    assert arr.get_length() == len(element)

# проверка сортировки вставками
def test_sort_insertion():
    arr.sort_insertion()
    for i in range(len(element)):
        assert arr[i].get_flag() == sorted_vin[i]

# проверка сортировки расчёской
def test_sort_comb():
    arr.sort_comb()
    for i in range(len(element)):
        assert arr[i].get_flag() == sorted_average_speed[i]

# проверка вывода
def test_str():
    list_str = "[" + ", ".join(map(str, sorted_average_speed)) + "]"
    assert list_str == str(arr)

# проверка работы итерации
def test_iteration():
    n = 0
    for elem in arr:
        assert elem.get_flag() in sorted_average_speed
        assert elem.get_flag() == sorted_average_speed[n]
        n += 1
```

Результат пройденных тестов (рисунок 4.1).

```
PS D:\Saves\GUAP\SEM3\AuSD\laba4> pytest test_2.py
========================== test session starts ==========================
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba4
collected 10 items

test_2.py ..........                                              [100%]

=========================== 10 passed in 0.02s ==========================
```

Рисунок 4.1 – результат проверки теста 2 задания

Все 10 тестов были пройдены успешно.

4.2 Бенчмарк

Был произведён бенчмарк сортировки на количество значений.

Код программы:

```
from structuredata.array import Array
from temp.car import Car, element
from timeit import timeit
from random import choice, randint

times1 = ""
times2 = ""

for su in range(1, 6):
    # количество элементов
    k1 = 10 ** su
    table = Array(k1)
    dit = []
    marks = [i.mark for i in element]
    for i in range(k1):
        dit += [Car(
            mark=choice(marks),
            vin="".join(choice("0123456789ABCDEFGHJKLMNPRSTUVWXYZ")
for i in range(17)),
            engine_capacity=randint(50, 400),
            cost=randint(900_000, 6_000_000),
            average_speed=randint(70_00, 250_00)/100
        )]

    for x in dit:
        table.append(x)
    code1 = """table.sort_insertion()"""

    time1 = timeit(code1, number=1, globals={"dit": dit, "table":
table})
    table.clear()

    for x in dit:
        table.append(x)
    code2 = """table.sort_comb()"""

    time2 = timeit(code2, number=1, globals={"dit": dit, "table":
table})
    table.clear()

    times1 += f"        {k1}            |          {time1}\n"
    times2 += f"        {k1}            |        {time2}\n"
```

```
print("Сортировка вставками: по возрастанию - по полю vin")
print(f"Кол-во элементов    |   Время")
print(times1)

print("Сортировка расчёской: по убыванию - по полю средняя скорость")
print(f"Кол-во элементов    |   Время")
print(times2)
```

Вывод в консоли изображено на рисунке 4.2.

```
Сортировка вставками: по возврастанию - по полю vin
Кол-во елементов    |   Время
        10          |          6.450001092161983e-05
        100          |          0.0032325999927707016
        1000          |          0.32316699999501
        10000          |          33.734448799994425
        100000          |          4119.782863

Сортировка расчёской: по убыванию - по полю средняя скорость
Кол-во елементов    |   Время
        10          |        2.2599997464567423e-05
        100          |        0.0003536000003805384
        1000          |        0.006622399989282712
        10000          |        0.10357370000565425
        100000          |        1.7854930000030436
```

Рисунок 4.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов влияет на время каждого алгоритма сортировки.

Кол-во элементов	Время в секундах
10	0.0000645
100	0.0032326
1000	0.3231670
10000	33.7344488
100000	4119.782863

Таблица 4.1 – Сортировка вставками

Кол-во элементов	Время в секундах
10	0.0000226
100	0.0003536
1000	0.0066224
10000	0.1035737

100000	1.7854930

Таблица 4.2 – Сортировка расчёской

Сортировка расчёской сильнее выигрывает сортировки вставками для массива. Так как в сортировки вставками постоянно перебираются для каждой позиции заново массив. То в сортировки расчёской перебирается только определённое количество с определённым диапазоном, благодаря коэффициенту k=1.3, что сокращает время сортировки во многом.

5 Код программы для задания 5

Класс Book(Книги) для 5 задания:

Путь файла класса Book и примерами элементов этого класса (element_book) – temp/book.py

```
from typing import Self, Optional
from dataclasses import dataclass
from functools import total_ordering

# дополняет недостающие методы за счёт других (изменятся >=, <=, !=,
>)
@total_ordering
@dataclass
class Book:
    author: str
    publisher: str
    pages: int
    cost: int
    isbn: int

    # флаг - переключатель основного параметра - по умолчанию cost
    def __post_init__(self) -> None:
        self.flag = "cost"

    # задать основной параметр
    def set_flag(self, text: str) -> None:
        if text in ["author", "publisher", "pages", "cost", "isbn"]:
            self.flag = text

    # получить основной параметр или получить параметр
    def get_flag(self, text: Optional[str] = None) -> str | float:
        tx = self.flag
        if text is not None:
            tx = text
        if tx == "author":
            return self.author
        elif tx == "publisher":
```

```
            return self.publisher
        elif tx == "pages":
            return self.pages
        elif tx == "cost":
            return self.cost
        elif tx == "isbn":
            return self.isbn
        return self.cost

    # магический метод - меньше <
    def __lt__(self, other: Self) -> bool:
        other_any = other.get_flag(self.flag)
        return self.get_flag() < other_any

    # магический метод - равно ==
    def __eq__(self, other: Self) -> bool:
        other_any = other.get_flag(self.flag)
        return self.get_flag() == other_any

    # магический метод - сложение +
    def __add__(self, other: Self) -> str | int | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() + other_any

    # магический метод - вычитание -
    def __sub__(self, other: Self) -> int | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() - other_any

    # магический метод - умножение *
    def __mul__(self, other: Self) -> str | int | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() * other_any

    # магический метод - деление /
    def __truediv__(self, other: Self) -> float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() / other_any

    # магический метод - деление на цело //
    def __floordiv__(self, other: Self) -> int | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() // other_any

    # магический метод - вывода
    def __str__(self) -> str:
        return str(self.get_flag())

    # магический метод предназначен для машинно-ориентированного
вывода
    def __repr__(self) -> str:
```

```
        return str(self.get_flag())


# 10 раличных модель машин (10 элементов для дерева)
element_book = [
    Book("Л. Н. Толстой", "Эксмо", 1000, 472, 9785699120147),
    Book("О. Л. Ершова", "Эксмо", 520, 473, 9785699867935),
    Book("Роберт Джордан", "Азбука", 33, 401, 9785677520647),
    Book("В. Д. Афиногенов", "Вече", 45, 277, 9785697425819),
    Book("Люси Мод Монтгомери", "Азбука", 145, 370, 9785696450122),
    Book("М. Ю. Лермонтов", "Азбука", 76, 320, 9785699224456),
    Book("Н. Н. Телепова", "Эксмо", 80, 250, 9785693322112),
    Book("С. Н. Зигуненко", "Вече", 60, 501, 9785664550121),
    Book("И. Е. Забелин", "Азбука", 120, 399, 9785693124755),
    Book("В. Д. Афиногенов", "Вече", 230, 410, 9785699673146)
]
```

Методы set_flag(self, text) задаёт основное значение по названию (по умолчанию – стоимость), get_flag(self) даёт основное значение.

Метод массива быстрой сортировки (sort_quick) по возрастанию для Класса Book (Книги) по полю pages (кол-во страниц):

```
    # Рекурсивная Сортировка быстрая - по возрастанию для Book - кол-
во страниц
    def rec_sort_quick(self, start: int, end: int) -> None:
        # рекурсия идёт, пока конец меньше начала диапазона сортировки
        if start < end:
            # опорная точка - конец (от отобранного диапазона)
            pivot = self.arr[end]
            # less на сколько сдвинулась опорная точка влево
            less = 0
            for i in range(start, end):
                # если опорная точка меньше элемента то элемент ставим
в конец диапазона end
                if pivot < self.arr[i - less]:
                    # удаляем элемент для переноса в конец диапазона
                    elem = self.remove(i - less)
                    # вставка элемента вправо в конец диапазона
                    self.insert(elem, end)
                    # сдвиг опорной точки влево + 1
                    less += 1
            # end - less = индекс опорной точки (она стоит на месте),
рекурсия продолжается с левой и правой сторонной опорной точки
            self.rec_sort_quick(start, end - less - 1)
            self.rec_sort_quick(end - less + 1, end)

    # Сортировка быстрая - по возрастанию для Book - кол-во страниц
    def sort_quick(self) -> None:
        # если массив пуст, то ничего не делается
```

```
        if self.length <= 0:
            return
        # проверка на тип - если класс Книга, то задаём основное
значение - кол-во страниц
        if type(self.arr[0]) == Book:
            for i in range(self.length):
                self.arr[i].set_flag("pages")
        # рекурсия быстрой сортировки
        self.rec_sort_quick(0, self.get_length() - 1)
```

Метод быстрой сортировки сначала проверяет на размер массива, после сам тип элемента в массиве, если это Book, то задаём основное поле – pages (кол-во страниц).

После вызывает рекурсию с начальным полным диапазоном от 0 до длинны массива.

Рекурсия продолжает пока конец больше начала диапазона. Определяется опорная точка pivot – это всегда конец заданного диапазона. И less – на сколько сдвинулась опорная точка влево.

Дальше идёт сортировка, сравнение всех левых элементов опорной точки с самой точкой, если элемент больше опорной точки, то элемент перемещается в право к концу заданного диапазона. Это засчитывается за сдвиг, поэтому less += 1.

Далее происходит рекурсия только уже с диапазонами правой и левой части от опорной точки и так, до того как диапазоны не закончатся.

Метод массива сортировки по основанию (sort_radix) по убыванию для Класса Book (Книги) по полю cost (стоимость):

```
    # Сортировка по основанию - по убыванию для Book - стоимость
    def sort_radix(self) -> None:
        # если массив пуст, то ничего не делаем
        if self.length <= 0:
            return
        # проверка на тип - если класс Книга, то задаём основное
значение - стоимость
        if type(self.arr[0]) == Book:
            for elem in self.arr:
                elem.set_flag("cost")
        # сортировка по десятичной системе от 0 до 9 (включительно) -
список из списков
        nums = [[] for _ in range(10)]
        # новый массив для сортировки - копия
        new_arr: ctypes.Array[T] = (self.length * ctypes.py_object)()
        for i in range(self.get_length()):
            new_arr[i] = self.get(i)
        # максимальная длина целого числа в массиве
        countmx = len(str(max(new_arr, key=lambda x: len(str(x)))))
```

```
        # от 0 до максимальной длины - сколько разрядов должен пройти
для сортировки
        for round in range(countmx):
            # сортировка одного из разрядов (round) целых чисел
            for i in new_arr:
                # значение - целое число
                flag = str(i)
                # если разряда нету (привышен), то считаем его за ноль
                if len(flag) >= round + 1:
                    # индекс разряда (начиная с конца целого числа)
                    index = (len(flag) - 1) - round
                    # добавляем в список десятичной системы
                    nums[(len(nums) - 1) - int(flag[index])] += [i]
                else:
                    nums[len(nums) - 1] += [i]
            # счётчик индекса массива
            index = 0
            # заменяем отсортированные значения
            for i in nums:
                for j in i:
                    new_arr[index] = j
                    index += 1
            # очищяем список сортировки
            nums.clear()
            nums = [[] for _ in range(10)]
        # добавляем в основной массив отсортированные значения
        for i in range(self.length):
            self.arr[i] = new_arr[i]
```

Метод сортировки по основанию сначала проверяет на размер массива, после сам тип элемента в массиве, если это Book, то задаём основное поле – cost (стоимость).

Создаётся список nums с размером 10 и заполняется справа налево для убывания. Если разряд будет равно нулю то будет в самом конце списка в индексе 9.

Создаётся копия массива для редактирования и сортировки. Ищется самое длинное целое число, у которого больше разрядов. Далее идёт счётчик по разрядам и сортировка в список nums от 9 до 0 (включительно). Если у целого числа нет разряда, то идёт в конец списка nums. После каждого разряда массив заполняется с полученными значениями после сортировки, а список nums очищается и так пока все разряды не кончатся.

После вносим в основной массив изменения после сортировки.

5.1 Тесты

Код теста:

```
from structuredata.array import Array
# element_book - содержит 10 различных элементов книг (10 различных
Book)
```

```
from temp.book import Book, element_book

ex_book1 = Book("П. Л. Нимский", "Эксмо", 520, 555, 9780000000000)
ex_book2 = Book("Шиткин", "Эксмо", 120, 55, 9780000000001)
same_cost = Book("И. Д. Тимухин", "Азбука", 520, 255, 9780000000002)

cap = 10
arr = Array(cap)
ind = 4

sorted_pages = sorted([i.pages for i in element_book])
sorted_cost = sorted([i.cost for i in element_book])
sorted_cost.reverse()

# добавление в конец
def test_append():
    for elem in element_book:
        arr.append(elem)

    for i in range(len(element_book)):
        assert element_book[i] == arr[i]

# проверка функций на get_ дающие значение
def test_gets():
    assert arr.get_capacity() == cap
    assert arr.get_length() == len(element_book)

    for i in range(len(element_book)):
        assert element_book[i] == arr.get(i)

# вставка с проверка размера
def test_resize_capacity():
    new_cap = cap * 2
    arr.append(element_book[2])
    assert arr.get_capacity() == new_cap

# вставка
def test_insert():
    arr.insert(ex_book1, ind)
    assert arr[ind] == ex_book1
    assert arr[ind - 1] == element_book[ind - 1]
    assert arr[ind + 1] == element_book[ind]
    arr.insert(ex_book2, 0)
    assert arr[0] == ex_book2
    assert arr[1] == element_book[0]
    arr.insert(same_cost, arr.get_length())
    assert arr[arr.get_length() - 1] == same_cost

# задаёт элемент по индексу
def test_set_elem():
    arr[0] = element_book[-1]
```

```
    assert arr[0] == element_book[-1]

# удаление и очистка
def test_remove_and_clear():
    assert arr.remove(0) == element_book[-1]
    assert arr.remove(arr.get_length() - 1) == same_cost
    assert arr.remove(arr.get_length() - 1) == element_book[2]
    assert arr.remove(ind) == ex_book1
    assert arr.get_length() == len(element_book)

# проверка быстрой сортировки
def test_sort_quick():
    arr.sort_quick()
    for i in range(len(element_book)):
        assert arr[i].get_flag() == sorted_pages[i]

# проверка сортировки по основанию
def test_sort_radix():
    arr.sort_radix()
    for i in range(len(element_book)):
        assert arr[i].get_flag() == sorted_cost[i]

# проверка вывода
def test_str():
    list_str = "[" + ", ".join(map(str, sorted_cost)) + "]"
    assert list_str == str(arr)

# проверка работы итерации
def test_iteration():
    n = 0
    for elem in arr:
        assert elem.get_flag() in sorted_cost
        assert elem.get_flag() == sorted_cost[n]
        n += 1
```

Результат пройденных тестов (рисунок 5.1).

```
PS D:\Saves\GUAP\SEM3\AuSD\laba4> pytest test_5.py
========================= test session starts =========================
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba4
collected 10 items

test_5.py ..........                                             [100%]

========================== 10 passed in 0.02s ==========================
```

Рисунок 5.1 – результат проверки теста 5 задания

Все 10 тестов были пройдены успешно.

5.2 Бенчмарк

Был произведён бенчмарк сортировки на количество значений.

Код программы:

```
from structuredata.array import Array
from temp.book import Book
from timeit import timeit
from random import choice, randint
from string import ascii_letters

times1 = ""
times2 = ""

for su in range(1, 5):
    # количество элементов
    k1 = 10 ** su
    table = Array(k1)
    dit = []
    for i in range(k1):
        dit += [Book(
            author="".join(choice(ascii_letters) for i in
range(randint(5, 30))),
            publisher="".join(choice(ascii_letters) for i in
range(randint(5, 30))),
            pages=randint(10, 1000),
            cost=randint(100, 10000),
            isbn=int("".join(str(choice(range(10))) for i in
range(13)))
        )]

    for x in dit:
        table.append(x)
    code1 = """table.sort_quick()"""

    time1 = timeit(code1, number=1, globals={"dit": dit, "table":
table})
    table.clear()

    for x in dit:
        table.append(x)
    code2 = """table.sort_radix()"""

    time2 = timeit(code2, number=1, globals={"dit": dit, "table":
table})
    table.clear()
    times1 += f"        {k1}            |          {time1}\n"
    times2 += f"        {k1}            |        {time2}\n"

print("Быстрая сортировка: по возрастанию - по полю кол-во страниц")
```

```
print(f"Кол-во элементов    |   Время")
print(times1)

print("Сортировка по основанию: по убыванию - по полю стоимость")
print(f"Кол-во элементов    |   Время")
print(times2)
```

Вывод в консоли изображено на рисунке 5.2.

```
Быстрая сортировка: по возврастанию - по полю кол-во страниц
Кол-во елементов    |   Время
        10          |          4.020000051241368e-05
        100         |          0.004172100001596846
        1000        |          0.8088707999995677
        10000       |          114.38220509998791

Сортировка по основанию: по убыванию - по полю стоимость
Кол-во елементов    |   Время
        10          |        4.780000017490238e-05
        100         |        0.00028929999098181725
        1000        |        0.002827200005413033
        10000       |        0.029547900005127303
```

Рисунок 5.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов влияет на время каждого алгоритма сортировки.

Кол-во элементов	Время в секундах
10	0.0000402
100	0.0041721
1000	0.8088708
10000	114.3822051

Таблица 5.1 – Быстрая сортировка

Кол-во элементов	Время в секундах
10	0.0000478
100	0.0002893
1000	0.0028272
10000	0.0295479

Таблица 5.2 – Сортировка по основанию

Быстрая сортировка оказалась самой медленной сортировкой для работы с массивом, но сортировка по основанию стала самой эффективной сортировкой из всех.

Выводы

В данной лабораторной работе я создала структуру данную Массив с применением таких алгоритмов сортировок как:

- Сортировка вставками;
- Сортировка расчёской;
- Быстрая сортировка;
- Сортировка по основанию.

Научилась применять аннотацию типов с помощью библиотеки typing. А также провела тест своих структур данных вместе с бенчмарком. Бенчмарк показал, что для массива хорошо подходят сортировка расчёской и сортировка по основанию.