

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №1
по курсу «Алгоритмы и структуры данных»
Тема: Сортировка вставками, выбором, пузырьковая
Вариант 15

Выполнил:
Левахин Лев Александрович
К3140

Проверил:



Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Сортировка вставкой	
Задача №2. Сортировка вставкой +	
Задача №3. Сортировка вставкой по убыванию	
Задача №4. Линейный поиск	
Задача №5. Сортировка выбором	
Дополнительные задачи	4
Задача №6. Пузырьковая сортировка	
Задача №7. Знакомство с жителями Сортлэнда	
Вывод	5

Задачи по варианту

Задача №1. Сортировка вставкой

Листинг кода.

```
"""
Сортировка вставкой базовая
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

n = int(file_in.readline()) #Количество элементов
lst = list(map(int, file_in.readline().split())) #Список с элементами

def insertion_sort(n:int, lst:list) ->str:
    for i in range(1, n):
        key = lst[i]
        j = i-1
        while (j>=0) and (lst[j]>key):
            lst[j+1] = lst[j]
            j -=1
        lst[j+1] = key
    lst = [str(el) for el in lst]
    return " ".join(lst)

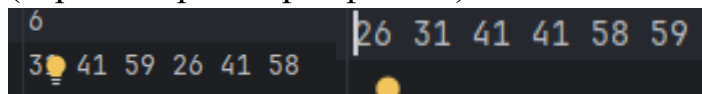
file_out.write(insertion_sort(n, lst))
```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию сортировки вставкой, которая принимает на вход количество элементов массива и сам список. Проходимся по элементам с 1 до количества элементов. Переменной ключу задаём значение текущего элемента, переменной j задаём значение предыдущего. Пока предыдущий элемент больше или равен 0 и этот элемент больше, чем ключ(текущий), будем выполнять условие. Следующий элемент списка от j приравниваем к предыдущему и убавляем j. Таким образом сортируя массив. Приравниваем элемент от j к исходному элементу – ключу. Преобразовываем каждый элемент списка в строку, чтобы вернуть строку значений через пробел, с помощью метода join. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:

(скрины input output файлов)



Тесты к задаче:

Листинг кода:

```
from lab1.task1.src.task1 import insertion_sort
import datetime
```

```
import tracemalloc

file_in = open("../txtfiles/input.txt")

n = int(file_in.readline()) #Количество элементов
lst = list(map(int, file_in.readline().split())) #Список с элементами

tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

print(insertion_sort(n, lst)) # Выводим результат отработанной функции

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ") #Выводим время работы в мегабайтах
```

Скриншоты работы тестов:

```
26 31 41 41 58 59
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.000442 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0ч:00м:00с	0.000442 МБ

Вывод по задаче:

- 1) Сортировка вставкой не очень быстро работает, но довольна проста и понятна.

Задача №2. Сортировка вставкой + Листинг кода.

```
"""
Сортировка вставкой +, дополнительно выводит номер, на который был
поставлен элемент при обработке
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

n = int(file_in.readline()) # Количество элементов
lst = list(map(int, file_in.readline().split())) # Список с элементами

def insertion_sort(n:int, lst:list) -> str:
    indexes = "1 "
    for i in range(1, n):
        key = lst[i]
        j = i-1
        while (j>=0) and (lst[j]>key):
            lst[j+1] = lst[j]
```

```

        j -= 1
        lst[j+1] = key
        indexes+=str(j+1+1)+" "

    lst = [str(el) for el in lst]
    res = indexes+"\n"+ " ".join(lst)
    return res

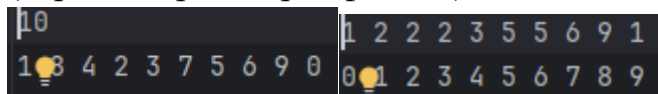
file_out.write(insertion_sort(n, lst))

```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию сортировки вставкой, которая принимает на вход количество элементов массива и сам список. Создаём строку куда будем записывать индексы. Так как первый элемент остаётся на своём месте, сразу запишем в эту строку единицу. Проходимся по элементам с 1 до количества элементов. Переменной ключу задаём значение текущего элемента, переменной j задаём значение предыдущего. Пока предыдущий элемент больше или равен 0 и этот элемент больше, чем ключ(текущий), будем выполнять условие. Следующий элемент списка от j приравниваем к предыдущему и убавляем j. Таким образом сортируя массив. Добавляем к строке индексов число, на которое поставили элемент списка. Приравниваем элемент от j к исходному элементу – ключу. Создаём переменную рез, в которую записываем строку индексы, переходим на новую строку, преобразовываем каждый элемент списка в строку, чтобы вернуть строку значений через пробел, с помощью метода join. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)



Тесты к задаче:

Листинг кода:

```

from lab1.task2.src.task2 import insertion_sort
import datetime
import tracemalloc

file_in = open("../txtfiles/input.txt")

n = int(file_in.readline()) #Количество элементов
lst = list(map(int, file_in.readline().split())) #Список с элементами

tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

```

```
print(insertion_sort(n, lst)) # Выводим результат отработанной функции

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ") # Выводим время работы в мегабайтах
```

Скрины работы тестов:

```
1 2 2 2 3 5 5 6 9 1
0 1 2 3 4 5 6 7 8 9
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.000852 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000852 Мб

Вывод по задаче:

1) На основе сортировки вставкой можно реализовать её более сложный и интересный прототип.

Задача №3. Сортировка вставкой по убыванию

Листинг кода.

```
"""
Сортировка вставкой в невозрастающем(убывающем) порядке
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

n = int(file_in.readline()) # Количество элементов
lst = list(map(int, file_in.readline().split())) # Список с элементами

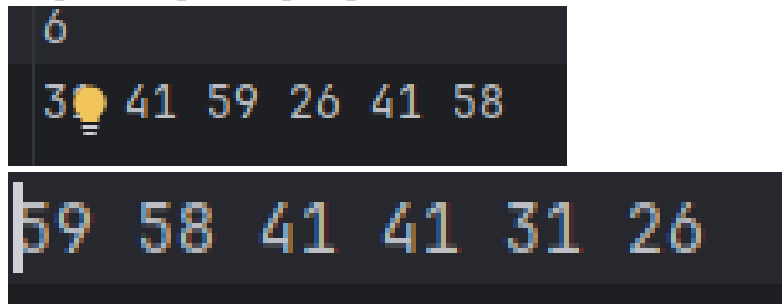
def insertion_sort(n:int, lst:list) ->str:
    for i in range(1, n):
        key = lst[i]
        j = i-1
        while (j>=0) and (lst[j]<key): # Поменялся знак неравенства, теперь
нам подходит случай, когда предыдущий элемент меньше чем ключ
            lst[j+1] = lst[j]
            j -=1
        lst[j+1] = key
    lst = [str(el) for el in lst]
    return " ".join(lst)

file_out.write(insertion_sort(n, lst))
```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию сортировки вставкой, которая принимает на вход количество элементов массива и сам список. Проходимся по элементам с 1 до количества элементов. Переменной ключу задаём значение текущего элемента, переменной *j* задаём значение предыдущего. Пока предыдущий элемент больше или равен 0 и этот элемент меньше, чем ключ(текущий), будем выполнять условие. Следующий элемент списка от *j* приравниваем к предыдущему и убавляем *j*. Таким образом сортируя массив. Приравниваем элемент от *j* к исходному элементу – ключу. Преобразовываем каждый элемент списка в строку, чтобы вернуть строку значений через пробел, с помощью метода `join`. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)



Тесты к задаче:

Листинг кода:

```
from lab1.task3.src.task3 import insertion_sort
import datetime
import tracemalloc

file_in = open("../txtfiles/input.txt")

n = int(file_in.readline()) #Количество элементов
lst = list(map(int, file_in.readline().split())) #Список с элементами

tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

print(insertion_sort(n, lst)) # Выводим результат отработанной функции

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ") #Выводим время работы в мегабайтах
```

Скрины работы тестов:

```
59 58 41 41 31 26
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.000442 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000442 МБ

Вывод по задаче:

- 1) Поменяв всего лишь один символ, можно из сортировки вставкой по возрастанию сделать сортировку вставкой по убыванию.

Задача №4. Линейный поиск

Листинг кода.

```
"""
Линейный поиск
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

a = list(map(int, file_in.readline().split()))
v = int(file_in.readline())

def line_search(lst: list, find_el: int):
    len_lst = len(lst)
    cnt_el = 0
    ind_list = []
    for i in range(len_lst):
        if lst[i] == find_el:
            cnt_el += 1
            ind_list.append(i)
    if len(ind_list) > 1:
        return f"Cnt v: {len(ind_list)}\nIndex's v: {", ".join([str(el) for el in ind_list])}"
    elif len(ind_list) == 1:
        return ind_list[0]
    return -1

file_out.write(str(line_search(a, v)))
```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию линейного поиска, которая принимает на вход список с элементами и значение элемента, индекс которого нужно найти. Создаём переменную с длиной списка, счётчик элементов, список с индексом значений. Проходимся по числам с 0 до количества элементов.

Если находим нужный элемент – то добавляем к счётчику 1 и добавляем аргумент в список. Если длина этого списка больше одного, будем записывать в файл все элементы и их индексы. Если же длина равна 1, то возвращаем только сам элемент. Если такого элемента нет в строке, возвращаем -1. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:

(скрины input output файлов)

```
128 128 917 1628731 287 12840 3 7371 0927309 71
1
-1
```

Результат работы кода на собственном примере:

1)

```
10 10 30 40 50 60 70
1💡
Cnt v: 2
Index's v: 0, 1
```

2)

```
4 8 15 16 23 42
23
4
```

Тесты к задаче:

Листинг кода:

```
from lab1.task4.src.task4 import line_search
import datetime
import tracemalloc

file_in = open("../txtfiles/input.txt")

a = list(map(int, file_in.readline().split()))
v = int(file_in.readline())

tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

print(line_search(a, v)) # Выводим результат отработанной функции
```

```

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ") # Выводим время работы в мегабайтах

```

Скрины работы тестов:

0) Из задачи

```

Cnt v: 2
Index's v: 0, 1
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.000358 МБ

```

1)

```

-1
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.00035 МБ

```

2)

```

4
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.00035 МБ

```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000358 Мб
Пример 1	0:00:00	0.00035 Мб
Пример 2	0:00:00	0.00035 Мб

Вывод по задаче:

- 1) С помощью условных операторов и разных условий задачи можно сильно её усложнить.

- 2) Чтобы легко узнать индекс элемента, можно перебирать не сами элементы списка, а только числа, а проверять уже список от индекса.

Задача №5. Сортировка выбором

Листинг кода.

```
"""
Сортировка выбором
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

n = int(file_in.readline()) # Количество элементов
lst = list(map(int, file_in.readline().split())) # Список с элементами

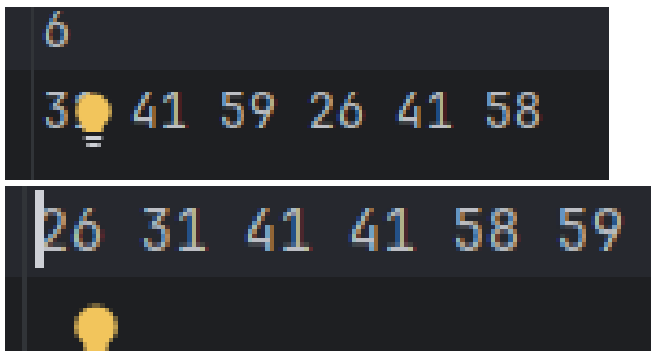
def selection_sort(n: int, lst: list) -> str:
    lst_copy = lst[:]
    res = [min(lst)]
    lst_copy.remove(min(lst))
    for i in range(n-1):
        res.append(min(lst_copy))
        lst_copy.remove(min(lst_copy))
    res += lst_copy
    res = [str(el) for el in res]
    return " ".join(res)

file_out.write(selection_sort(n, lst))
```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию сортировки выбором, которая принимает на вход количество элементов массива и сам список. Создаём копию списка, записываем в список результатов минимальный из списка, сразу удаляем его. Проходимся по длине списка – 1. В результат добавляем минимальный элемент из копии списка и сразу удаляем его оттуда. После этого добавляем оставшийся элемент к результату. Преобразовываем каждый элемент списка в строку, чтобы вернуть строку значений через пробел, с помощью метода join. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)



```
6
31 41 59 26 41 58

26 31 41 41 58 59
```

Тесты к задаче:

Листинг кода:

```
from lab1.task1.src.task1 import insertion_sort
from lab1.task5.src.task5 import selection_sort
import datetime
import tracemalloc
import random

lst_hud = [random.randint(1, 100_000) for i in range(6_000)]
n_hud = 6_000

lst_sr = [random.randint(1, 10_000) for j in range(2_000)]
n_sr = 2_000

print("Просчитаем время и память работы Сортировки вставкой в худшем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

insertion_sort(n_hud, lst_hud)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах

print("Просчитаем время и память работы Сортировки выбором в худшем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

selection_sort(n_hud, lst_hud)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах
```

```

print("Просчитаем время и память работы Сортировки вставкой в среднем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

insertion_sort(n_sr, lst_sr)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах

print("Просчитаем время и память работы Сортировки выбором в среднем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

selection_sort(n_sr, lst_sr)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах

```

Скрины работы тестов(Сравнение работы двух сортировок на средних и максимальных значениях):

```

Просчитаем время и память работы Сортировки вставкой в худшем случае
Итоговое время: 0:00:04.809014
Используемая память: 8e-05 МБ
Память на пике: 0.363668 МБ

Просчитаем время и память работы Сортировки выбором в худшем случае
Итоговое время: 0:00:00.244055
Используемая память: 0.000876 МБ
Память на пике: 0.382286 МБ

Просчитаем время и память работы Сортировки вставкой в среднем случае
Итоговое время: 0:00:00.494728
Используемая память: 0.000904 МБ
Память на пике: 0.382286 МБ

Просчитаем время и память работы Сортировки выбором в среднем случае
Итоговое время: 0:00:00.028000
Используемая память: 0.000904 МБ
Память на пике: 0.382286 МБ

```

	Время выполнения	Затраты памяти (Мб)
Сортировка вставкой средние значения	5мс	0.38
Сортировка выбором средние значения	02 мс	0.38
Сортировка вставкой максимальные значения	4с	0.36
Сортировка выбором максимальные значения	2мс	0.38

Вывод по задаче:

- 1) Интересно сравнивать две сортировки, которые работают разными методами.
- 2) Одну и ту же задачу отсортировать массив можно решить множеством разных способов.

Дополнительные задачи

Задача №6. Пузырьковая сортировка

Листинг кода.

```
"""
Пузырьковая сортировка
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

n = int(file_in.readline()) #Количество элементов
lst = list(map(int, file_in.readline().split())) #Список с элементами

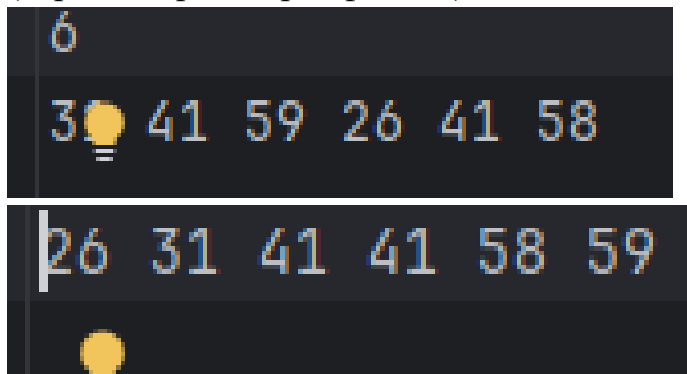
def bubble_sort(n:int, lst:list) ->str:
    for i in range(n-1):
        for j in range(i+1, n):
            if lst[j] <= lst[i]:
                lst[j], lst[i] = lst[i], lst[j]
    lst = [str(el) for el in lst]
    return " ".join(lst)

file_out.write(bubble_sort(n, lst))
```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию сортировки пузырьком, которая принимает на вход количество элементов массива и сам список. Проходимся по числам до количества элементов - 1. Создаём вложенный цикл от цифры на единицу больше и до количества элементов. Проверяем меньше ли следующий элемент или равен. Если да, меняем их местами. Преобразовываем каждый элемент списка в строку, чтобы вернуть строку значений через пробел, с помощью метода join. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)



Тесты к задаче:

Листинг кода:

```
from lab1.task1.src.task1 import insertion_sort
from lab1.task6.src.task6 import bubble_sort
import datetime
import tracemalloc
import random

lst_hud = [random.randint(1, 100_000) for i in range(6_000)]
n_hud = 6_000

lst_sr = [random.randint(1, 10_000) for j in range(2_000)]
n_sr = 2_000

print("Просчитаем время и память работы Сортировки вставкой в худшем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

insertion_sort(n_hud, lst_hud)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах

print("Просчитаем время и память работы Сортировки пузырьком в худшем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

bubble_sort(n_hud, lst_hud)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах

print("Просчитаем время и память работы Сортировки вставкой в среднем случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

insertion_sort(n_sr, lst_sr)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ\n") # Выводим время работы в мегабайтах
```



```

10**6} МБ\n") #Выводим время работы в мегабайтах

print("Просчитаем время и память работы Сортировки пузырьком в среднем
случае")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

bubble_sort(n_sr, lst_sr)

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak /
10**6} МБ\n") #Выводим время работы в мегабайтах
работы в мегабайтах

```

Скрины работы тестов:

```

Просчитаем время и память работы Сортировки вставкой в худшем случае
Итоговое время: 0:00:04.936793
Используемая память: 8e-05 МБ
Память на пике: 0.36376 МБ

Просчитаем время и память работы Сортировки пузырьком в худшем случае
Итоговое время: 0:00:03.981873
Используемая память: 0.000876 МБ
Память на пике: 0.364596 МБ

Просчитаем время и память работы Сортировки вставкой в среднем случае
Итоговое время: 0:00:00.512354
Используемая память: 0.000904 МБ
Память на пике: 0.364596 МБ

Просчитаем время и память работы Сортировки пузырьком в среднем случае
Итоговое время: 0:00:00.442060
Используемая память: 0.000904 МБ
Память на пике: 0.364596 МБ

```

	Время выполнения	Затраты памяти (Мб)
Сортировка вставкой средние значения	5мс	0.37
Сортировка пузырьком средние значения	4 мс	0.37

Сортировка вставкой максимальные значения	5с	0.37
Сортировка пузырьком максимальные значения	4с	0.37

Вывод по задаче:

- 1) Вложенный цикл существенно повышает время работы алгоритма.
- 2) Такой вид решения первый, что приходит в голову, когда речь идёт о сортировке.

Задача №7. Знакомство с жителями Сортлэнда

Листинг кода.

```
"""
В графстве Сортленда нужно найти 3х человек
- Самого бедного
- Среднего достатка
- Самого богатого
"""

file_in = open("../txtfiles/input.txt")
file_out = open("../txtfiles/output.txt", "w")

n = int(file_in.readline()) # Количество жителей
m = list(map(float, file_in.readline().split())) # Список с состоянием
жители, индексы жителей i+1

def Sortland(n: int, m: list) -> str:
    win_list = []

    win_list.append(m.index(min(m)) + 1)
    m_sort = sorted(m)
    win_list.append(m.index(m_sort[len(m_sort) // 2]) + 1)
    win_list.append(m.index(max(m)) + 1)

    win_list = [str(el) for el in win_list]
    return " ".join(win_list)

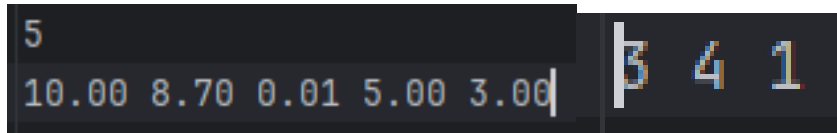
file_out.write(Sortland(n,m))
```

Текстовое объяснение решения.

Считываем файлы входа и выхода. Принимаем заданные значения из файла входа. Создаём функцию Сортлэнд, которая принимает на вход количество элементов массива и сам список. Создаём список победителей, добавляем в него индекс минимального элемента оригинального списка (+1 чтобы совпадало с нумерацией). Далее сортируем список по возрастанию, добавляем в список победителей жителя со средним достатком. Для этого находим индекс среднего по заработку человека, целочисленно деля длину

отсортированного списка на 2. Добавляем индекс максимального элемента. Таким образом, получаем 3х победителей. Преобразовываем каждый элемент списка в строку, чтобы вернуть строку значений через пробел, с помощью метода join. Записываем результат в файл.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)



```
5
10.00 8.70 0.01 5.00 3.00

3 4 1
```

Тесты к задаче:

Листинг кода:

```
from lab1.task7.src.task7 import Sortland
import datetime
import tracemalloc

file_in = open("../txtfiles/input.txt")

n = int(file_in.readline()) #Количество элементов
lst = list(map(float, file_in.readline().split())) #Список с элементами

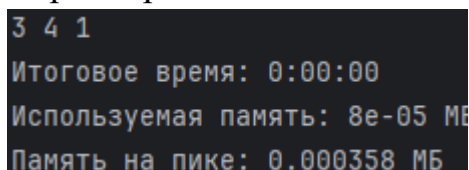
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

print(Sortland(n, lst)) # Выводим результат отработанной функции

finish_time = datetime.datetime.now() # Измеряем время конца работы
print("Итоговое время:", finish_time - start_time) # Выводим итоговое время

current, peak = tracemalloc.get_traced_memory() # Присваиваем двум
переменным память, используемую сейчас, и на пике
print(f"Используемая память: {current / 10**6} МБ\nПамять на пике: {peak / 10**6} МБ") #Выводим время работы в мегабайтах
```

Скрины работы тестов:



```
3 4 1
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.000358 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000358 Мб

Вывод по задаче:

- 1) Задачу можно решить лёгким способом, не создавая собственную сортировку.

Вывод

Существует множество видов сортировок, как самые оптимизированные, так и самые неоптимизированные. С ними можно совершать огромное количество действий и решать множество задач, также подстраивая их под свои цели. Они отличаются между собой по времени выполнения и затрачиваемой памяти. Чтобы использовать много видов сортировок и подбирать их под свои нужды, нужно понимать логику работы таких сортировок, о которых идёт речь в данной лабораторной. Я постарался использовать все свои знания, чтобы грамотно написать и применить их.