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

Отчет по лабораторной работе №3
по курсу «Алгоритмы и структуры данных»
Тема: Быстрая сортировка, сортировки за линейное время
Вариант 15

Выполнил:
Левахин Лев Александрович
К3140

Проверил:
--

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета **1**

Задачи по варианту **2**

Задача №1. Быстрые сортировки +

Задача №2. Анти-быстрая сортировка

Задача №4. Точки и отрезки

Дополнительные задачи **3**

Задача №3. Сортировка пугалом

Задача №5. Индекс Хирша

Задача №6. Сортировка целых чисел

Вывод **4**

Задачи по варианту

Задача №1. Быстрые сортировки +

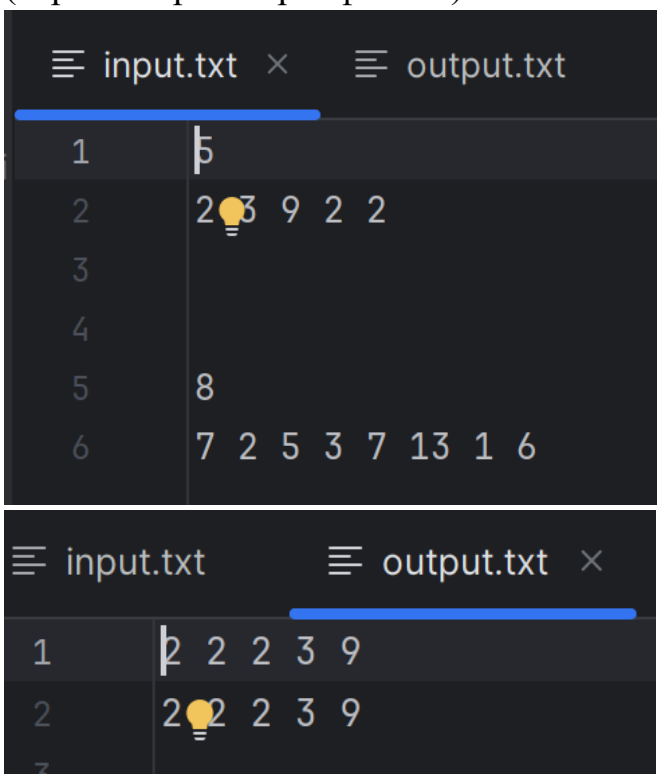
Листинг кода.

1) Быстрая сортировка +

```
"""
Быстрая сортировка +
"""

from random import *

from lab3 import utils


def quick_sort(lst: list, left: int, right: int):
    """
    Функция быстрой сортировки
    """
    if left < right:
        m = partition(lst, left, right)
        quick_sort(lst, left, m - 1)
        quick_sort(lst, m + 1, right)


def partition(lst: list, left: int, right: int) -> int:
    """
    Функция, которая отвечает за разделение
     - возвращает индекс последнего элемента, для которого было сделано
перемещение
    """
    x = lst[left]  # Выбираем "опорный" элемент
    j = left
    for i in range(left + 1, right + 1):
        if lst[i] <= x:
            j += 1
            lst[j], lst[i] = lst[i], lst[j]
    lst[left], lst[j] = lst[j], lst[left]
    return j


def randomized_quick_sort(lst: list, left: int, right: int):
    """
    Функция быстрой сортировки, которая использует рандомные элементы
    """
    if left < right:
        k = randint(left, right)
        lst[left], lst[k] = lst[k], lst[left]
        m = partition(lst, left, right)
        randomized_quick_sort(lst, left, m - 1)
        randomized_quick_sort(lst, m + 1, right)


if __name__ == "__main__":
    n, lst = utils.read_file()  # Количество элементов и список с
элементами
    n1, lst1 = utils.read_file()  # Количество элементов и список с
элементами
    quick_sort(lst, 0, n - 1)
    randomized_quick_sort(lst1, 0, n1 - 1)
    utils.write_file([lst, lst1])
```

Текстовое объяснение решения.
Делим массив на две части. С помощью функции партишон находим элемент м. Выбираем опорный элемент, и запоминаем начальный, проходимся по остальным элементам и меняем их местами. Возвращаем индекс последней перестановки.
Используем метод декомпозиции и рекурсивно запускаем функцию быстрой сортировки(до м и от м).

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



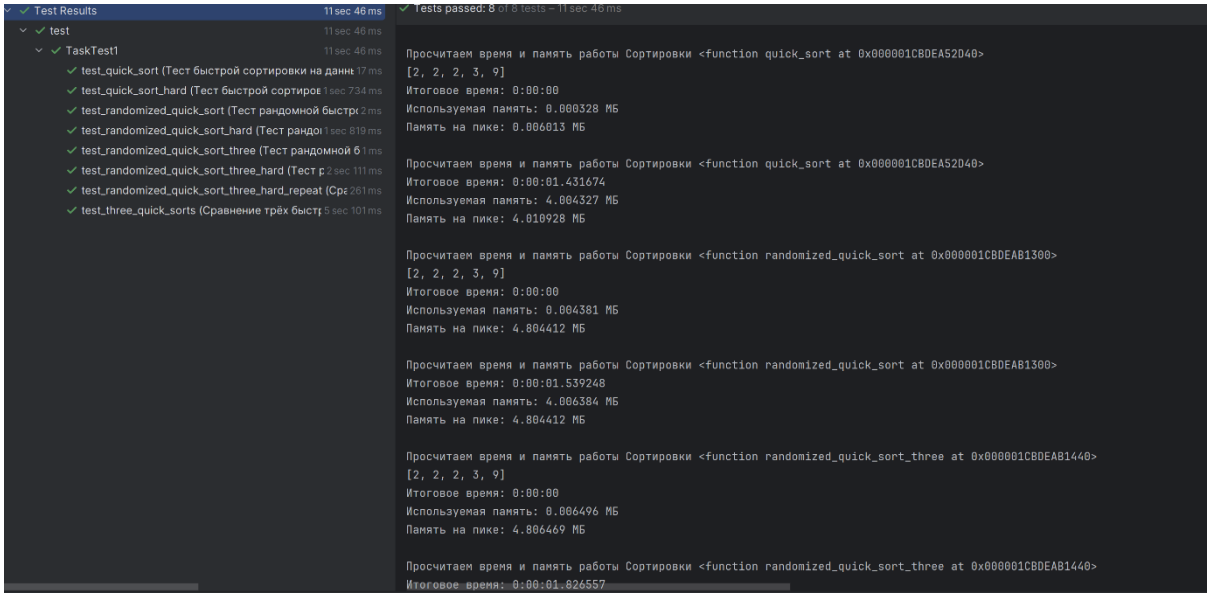
Тесты к задаче:
Листинг кода:

```
from lab3.task1.src.task1 import quick_sort, randomized_quick_sort
from lab3.task1.src.task2 import randomized_quick_sort_three
from lab3 import utils
import random
import unittest


class TaskTest1(unittest.TestCase):

    # Тест функции быстрой сортировки на данных из примера
    def test_quick_sort(self):
        """Тест быстрой сортировки на данных из примера"""
        # given
        n, lst = utils.read_file()

        # when
        utils.test_memory_and_time_lst_n(lst, n, quick_sort, True)
```

```
        # then
        self.assertEqual(lst, [2, 2, 2, 3, 9])

    # Тест функции рандомной быстрой сортировки на данных из примера
    def test_randomized_quick_sort(self):
        """Тест рандомной быстрой сортировки на данных из примера"""
        # given
        n, lst = utils.read_file()

        # when
        utils.test_memory_and_time_lst_n(lst, n, randomized_quick_sort,
True)

        # then
        self.assertEqual(lst, [2, 2, 2, 3, 9])

    # Тест функции рандомной быстрой сортировки с 3мя списками на данных из
примера
    def test_randomized_quick_sort_three(self):
        """Тест рандомной быстрой тройной сортировки на данных из
примера"""
        # given
        n, lst = utils.read_file()

        # when
        utils.test_memory_and_time_lst_n(lst, n,
randomized_quick_sort_three, True)

        # then
        self.assertEqual(lst, [2, 2, 2, 3, 9])

    # Тест функции быстрой сортировки на худших данных
    def test_quick_sort_hard(self):
        """Тест быстрой сортировки на самых больших данных"""
        # given
        lst_hud = [random.randint(1, 10 ** 9) for i in range(10 ** 5)]
        n_hud = 10**5

        # when
        utils.test_memory_and_time_lst_n(lst_hud, n_hud, quick_sort, False)

        # then
        self.assertEqual(lst_hud, sorted(lst_hud))

    # Тест функции рандомной быстрой сортировки на худших данных
    def test_randomized_quick_sort_hard(self):
        """Тест рандомной быстрой сортировки на самых больших данных"""
        # given
        lst_hud = [random.randint(1, 10 ** 9) for i in range(10 ** 5)]
        n_hud = 10 ** 5

        # when
        utils.test_memory_and_time_lst_n(lst_hud, n_hud,
randomized_quick_sort, False)

        # then
        self.assertEqual(lst_hud, sorted(lst_hud))

    # Тест функции рандомной быстрой сортировки с 3мя списками на худших
данных
    def test_randomized_quick_sort_three_hard(self):
        """Тест рандомной быстрой тройной сортировки на самых больших
данных"""
        # given
```

```
        lst_hud = [random.randint(1, 10 ** 9) for i in range(10 ** 5)]
        n_hud = 10 ** 5

        # when
        utils.test_memory_and_time_lst_n(lst_hud, n_hud,
randomized_quick_sort_three, False)

        # then
        self.assertEqual(lst_hud, sorted(lst_hud))

    # Тест функции рандомной быстрой сортировки с 3мя списками на худших
данных
    def test_three_quick_sorts(self):
        """Сравнение трёх быстрых сортировки на самых больших данных"""
        # given
        lst_hud = [random.randint(1, 10 ** 9) for i in range(10 ** 5)]
        n_hud = 10 ** 5
        res1, res2, res3 = lst_hud[:], lst_hud[:], lst_hud[:]

        # when
        utils.test_memory_and_time_lst_n(res1, n_hud, quick_sort, False)
        utils.test_memory_and_time_lst_n(res2, n_hud,
randomized_quick_sort, False)
        utils.test_memory_and_time_lst_n(res3, n_hud,
randomized_quick_sort_three, False)

        # then
        self.assertEqual(res1, sorted(res1))
        self.assertEqual(res2, sorted(res2))
        self.assertEqual(res3, sorted(res3))

    # Тест функции рандомной быстрой сортировки с 3мя списками на больших
данных с большим количеством повторяющихся элементов и сравнение с быстрой
сортировкой
    def test_randomized_quick_sort_three_hard_repeat(self):
        """Сравнение быстрых сортировок при одинаковых значениях"""
        # given
        lst_hud = [random.randint(1, 10 ** 9) for i in range(500)]
        lst_hud += [1 for i in range(500)]
        n_hud = 1000

        # when
        utils.test_memory_and_time_lst_n(lst_hud, n_hud,
randomized_quick_sort_three, False)
        utils.test_memory_and_time_lst_n(lst_hud, n_hud, quick_sort, False)

        # then
        self.assertEqual(lst_hud, sorted(lst_hud))


if __name__ == "__main__":
    unittest.main()
```

Скрины работы тестов:



2) Быстрая сортировка с разделением на 3 части

Листинг кода.

```
"""
Быстрая сортировка + с выделением 3-го списка из равных элементов
"""

from random import *
from lab3 import utils


def randomized_quick_sort_three(lst: list, left: int, right: int):
    """
    Функция быстрой сортировки, которая использует рандомные элементы
    """
    if left < right:
        k = randint(left, right)
        lst[left], lst[k] = lst[k], lst[left]
        m = partition_three(lst, left, right)
        randomized_quick_sort_three(lst, left, m - 1)
        randomized_quick_sort_three(lst, m + 1, right)


def partition_three(lst: list, left: int, right: int):
    """
    Функция, которая отвечает за разделение
     - возвращает индекс последнего элемента, для которого было сделано
перемещение
    """
    x = lst[left]  # Выбираем "опорный" элемент
    j = left
    i = left - 1
    p = right

    while j<=p:
        if lst[j] < x:
            i += 1
            lst[i], lst[j] = lst[j], lst[i]
            j += 1
        elif lst[j] > x:
            lst[j], lst[p] = lst[p], lst[j]
            p -= 1
        else:
```

```
            j += 1

    return p


if __name__ == "__main__":
    n, lst = utils.read_file()  # Количество элементов и список с
элементами
    randomized_quick_sort_three(lst, 0, n - 1)

    res =
[1,2,2,2,1,1,14,5,67,7,5,7,4342,2,34,234,23,423,4,553456,1,1,1,43,324,23,5,
23452,1]
    randomized_quick_sort_three(res,0, len(res)-1)

    utils.write_file([lst, res])
```

Текстовое объяснение решения.
Аналогичный подход 1ой задачи, но добавляется третий рассматриваемый случай. То есть рассматриваем случаи до ключа, после, и если равен ключу.

Вывод по задаче:

1) Быстрая сортировка соответствует названию и работает быстро, однако обычная её версия медленно сортирует одинаковые элементы
2) Быструю сортировку можно модернизировать и использовать для сортировки списка с большим количеством одинаковых элементов.
3) Случайный выбор элемента иногда может ускорить процесс работы сортировки.

Задача №2. Анти-быстрая сортировка

Листинг кода.

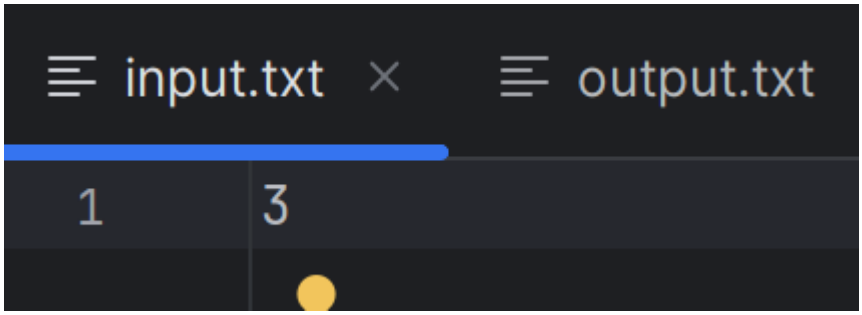
```
"""
Анти-быстрая сортировка
"""

from lab3 import utils


def qsort(lst: list, left: int, right: int):
    """ Алгоритм быстрой сортировки"""
    key = lst[(left + right) // 2]
    i = left
    j = right
    while i <= j:
        while lst[i] < key:
            i += 1
        while lst[j] > key:
            j -= 1
        if i <= j:
            lst[i], lst[j] = lst[j], lst[i]
            i += 1
            j -= 1
    if left < j:
        qsort(lst, left, j)
    if i < right:
        qsort(lst, i, right)


def anti_qsort(n: int) -> list:
    """ Возвращает худшую перестановку от длинны списка"""
    res = list(range(1, n + 1))

    for i in range(2, n):
        swap(res, i // 2, i)

    return res


def swap(lst: list, i: int, j: int):
    """ Меняет элементы списка местами """
    lst[i], lst[j] = lst[j], lst[i]


if __name__ == "__main__":
    n = utils.read_file_1()
    res = anti_qsort(n)
    utils.write_file([res])
```

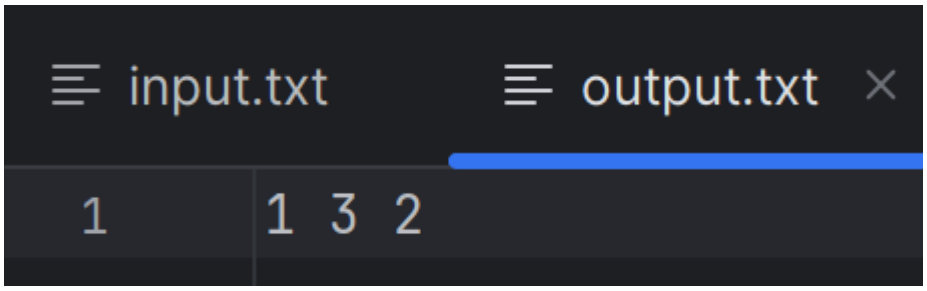
Текстовое объяснение решения.

Используя алгоритм быстрой сортировки(описанный выше) генирируем из заданной длины списка список, в котором сделается наибольшее количество перестановок. Создаем список, состоящий из n элементов, меняем местами все элементы с шагом i. Возвращаем список.

Результат работы кода на примерах из текста задачи:

(скрины input output файлов)





Тесты к задаче:

Листинг кода:

```
from lab3.task2.src.task2 import anti_qsort
from lab3 import utils
import unittest


class TaskTest2(unittest.TestCase):

    # Тест функции быстрой сортировки на данных из примера
    def test_quick_sort(self):
        """Тест быстрой сортировки на данных из примера"""
        # given
        n = utils.read_file_1()

        # when
        utils.test_memory_and_time_lst(n, anti_qsort, True)

        # then
        self.assertEqual(anti_qsort(n), [1, 3, 2])



if __name__ == "__main__":
    unittest.main()
```

Скрины работы тестов:

```
Просчитаем время и память работы Сортировки <function anti_qsort at 0x000001BAC72E2F20>
[1, 3, 2]
Итоговое время: 0:00:00
Используемая память: 0.000496 МБ
Память на пике: 0.006181 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000496 Мб

Вывод по задаче:

1) Чтобы сгенирировать список, на котором быстрая сортировка будет делать максимальное количество перемещений, нужен инверсированный список, в котором произведены дополнительные инверсии.

Задача №4. Точки и отрезки

Листинг кода.

```
"""
Точки и отрезки
"""

from lab3 import utils


def make_lottery_data(file_name:str) -> (int, int, list[int], list[int]):
    """
    Обрабатывает входной файл
     - В первой строке файла содержаться: (s - количество интервалов, p - количество точек)
     - В последующих s строках файла содержаться: ([a, b] - интервал)
     - В последней строке файла содержаться: (dot_i, i от 1 до p)
     Возвращает (количество интервалов, количество точек, полный список всех интервалов, список с точками)
    """
    file_in = open(f"../txtfiles/{file_name}.txt")

    cnt_segment, cnt_dot = map(int, file_in.readline().split())
    list_with_segments, list_with_dots = [], []

    for segment in range(cnt_segment):
        a, b = map(int, file_in.readline().split())
        if a < b:
            for el in (list(range(a, b + 1))):
                list_with_segments.append(int(el))
        else:
            for el in (list(range(b, a + 1))):
                list_with_segments.append(int(el))

    list_with_dots = list(map(int, file_in.readline().split()))
    file_in.close()

    return cnt_segment, cnt_dot, list_with_segments, list_with_dots


def lottery_find(list_with_segments:list, list_with_dots:list) -> list:
    """
    Функция подсчитывает сколько раз точка встречалась в интервалах
     - Используется линейный поиск
     - Принимает список с интервалами(числа) и список с точками(числа)
     - Выводит список с числами - количество раз, которые встречается каждая точка
    """
```

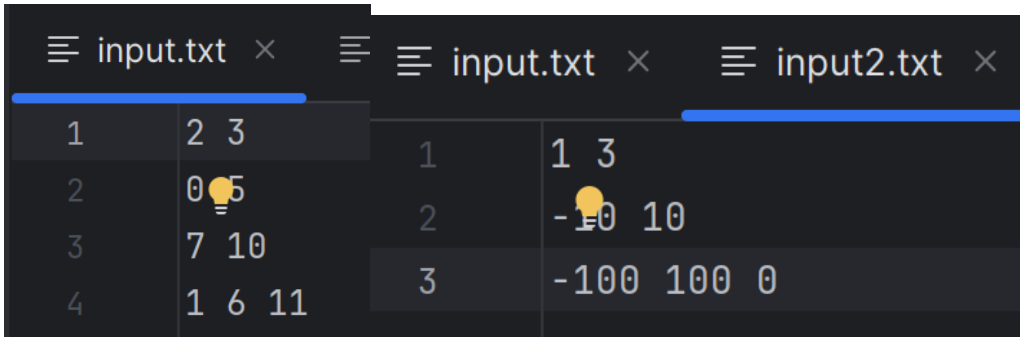
```
    res = []
    for find_el in list_with_dots:
        cnt_found_dot_in_segments = line_search(list_with_segments,
find_el)
        res.append(cnt_found_dot_in_segments)

    return res


def line_search(lst: list, find_el: int) -> int:
    """
    Функция линейного поиска
     - Возвращает количество раз, которые элемент встречается в списке
    """
    len_lst = len(lst)
    cnt_el = 0
    for i in range(len_lst):
        if lst[i] == find_el:
            cnt_el+=1

    return cnt_el


if __name__ == "__main__":
    cnt_segment, cnt_dot, list_with_segments, list_with_dots =
make_lottery_data("input")
    res1 = lottery_find(list_with_segments, list_with_dots)

    cnt_segment_2, cnt_dot_2, list_with_segments_2, list_with_dots_2 =
make_lottery_data("input2")
    res2 = lottery_find(list_with_segments_2, list_with_dots_2)

    cnt_segment_3, cnt_dot_3, list_with_segments_3, list_with_dots_3 =
make_lottery_data("input3")
    res3 = lottery_find(list_with_segments_3, list_with_dots_3)

    utils.write_file(["input 1:", res1, "\ninput 2:", res2, "\ninput 3:",
res3])
```

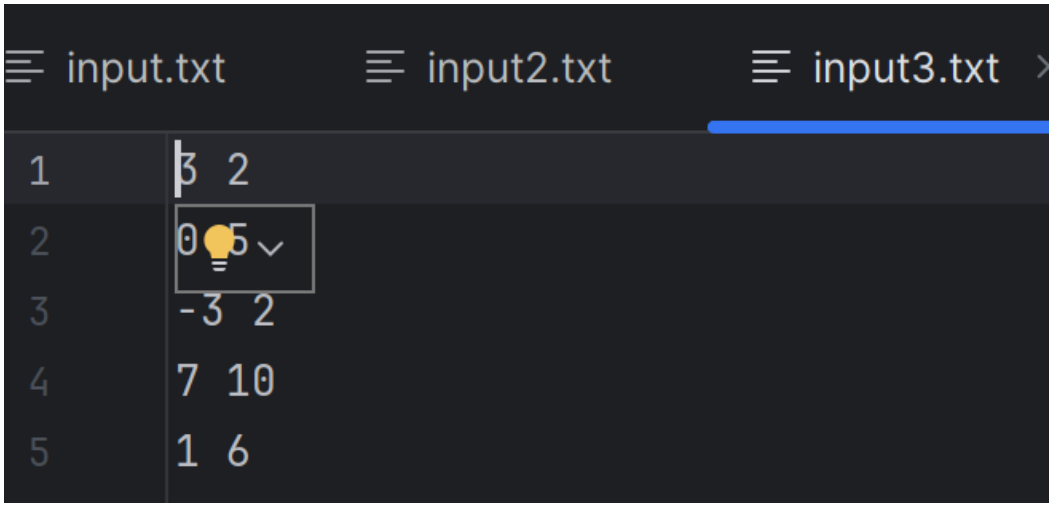
Текстовое объяснение решения.
Создаем функцию, в которой будем обрабатывать входные данные. Принимаем в ней название файла, для которого будем принимать. Возвращаем количество интервалов, количество точек, полный список всех интервалов, список с точками.

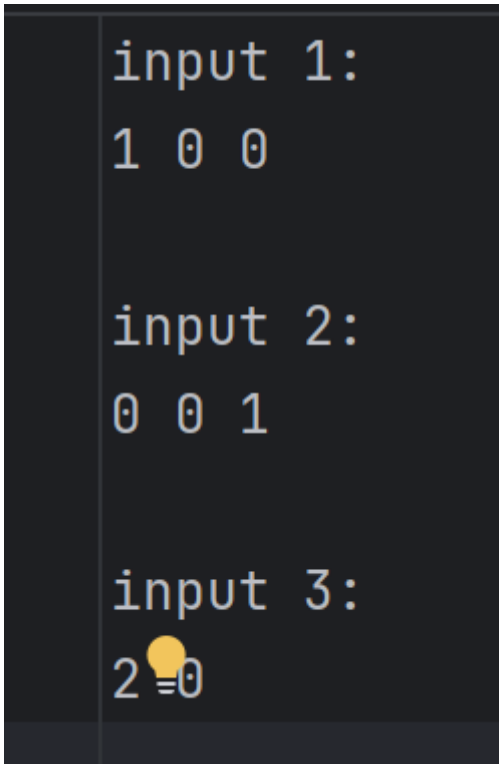
Далее, используя линейный поиск, находим количество раз, которое заданное число встречается в обработанных данных.

При запуске файла, выводим результат с помощью функции из utils.

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







Тесты к задаче:

Листинг кода:

```
from lab3.task4.src.task4 import *
from lab2 import utils
import datetime
import tracemalloc
import unittest
```

```
class TaskTest4(unittest.TestCase):

    def test_sort(self):
        """Тест на данных из примера"""
        # given
        cnt_segment, cnt_dot, list_with_segments, list_with_dots =
make_lottery_data("input")
        cnt_segment_2, cnt_dot_2, list_with_segments_2, list_with_dots_2 =
make_lottery_data("input2")
        cnt_segment_3, cnt_dot_3, list_with_segments_3, list_with_dots_3 =
make_lottery_data("input3")

        # when
        print("Просчитаем время и память работы алгоритма на данных из
примера")
        tracemalloc.start()  # Запускаем счётчик памяти
        start_time = datetime.datetime.now()  # Запускаем счётчик времени

        print(lottery_find(list_with_segments, list_with_dots))

        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()  # Присваеваем 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

        # then
        res1 = lottery_find(list_with_segments, list_with_dots)
        res2 = lottery_find(list_with_segments_2, list_with_dots_2)
        res3 = lottery_find(list_with_segments_3, list_with_dots_3)

        self.assertEqual(res1, [1, 0, 0])
        self.assertEqual(res2, [0, 0, 1])
        self.assertEqual(res3, [2, 0])


if __name__ == "__main__":
    unittest.main()
```

Скрины работы тестов:

```
Просчитаем время и память работы алгоритма на данных из примера
[1, 0, 0]
Итоговое время: 0:00:00
Используемая память: 0.000328 МБ
Память на пике: 0.004677 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000328 Мб

Вывод по задаче:

1) Задачу можно решить быстрее используя бинарный поиск
2) Большая часть способа решения задачи зависит от метода обработки входных данных.

Дополнительные задачи

Задача №3. Сортировка пугалом

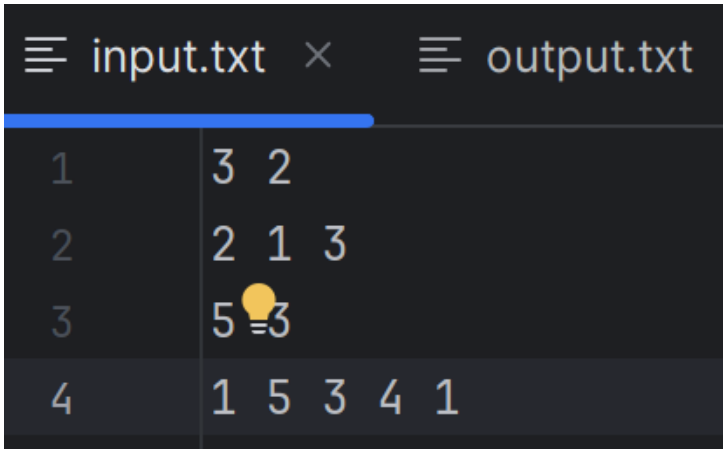
Листинг кода.

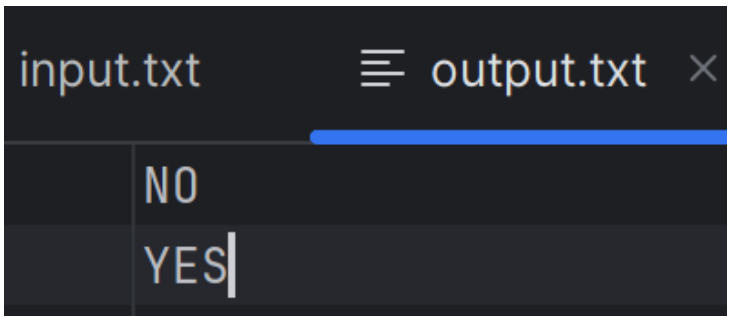
```
"""
Сортировка Пугалом
"""

from lab3 import utils


def scarecrow_sort(n:int, k:int, lst:list) -> str:
    """
    Сортировка пугалом
     - Принимает: (количество элементов списка, возможное расстояние,
список)
     - Возвращает: (ДА - если можно отсортировать список, НЕТ - если нельзя
отсортировать список)
    """
    sort_list = sorted(lst)
    while lst != sort_list:
        swapped = False
        for i in range(n-k):
            if lst[i] > lst[i+k]:
                switch(lst, i, i+k)
                swapped = True
                if lst == sort_list:
                    return "YES"
        if not swapped:
            return "NO"


def switch(lst:list, i:int, j:int) -> list:
    """ Меняет местами элементы в списке по их индексам"""
    lst[i], lst[j] = lst[j], lst[i]
    return lst


if __name__ == "__main__":
    file_in = open("../txtfiles/input.txt")

    n1, k1 = map(int, file_in.readline().split())
    lst1 = list((map(int, file_in.readline().split())))

    n2, k2 = map(int, file_in.readline().split())
    lst2 = list((map(int, file_in.readline().split())))

    res1 = scarecrow_sort(n1, k1, lst1)
    res2 = scarecrow_sort(n2, k2, lst2)

    utils.write_file([res1, res2])
```

Текстовое объяснение решения.

Очень необычный вид сортировки, в котором используем перемену элементов местами, чтобы отслеживать, можно ли отсортировать список, сравнивая его с уже отсортированным. Проходимся по элементам на заданном расстоянии, и смотрим, отсортирован ли список при смене элементов местами. Используем переменную-флаг для отслеживания.

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





Тесты к задаче:
Листинг кода:

```
from lab3.task3.src.task3 import scarecrow_sort
from lab3 import utils
import datetime
import tracemalloc
import unittest


class TaskTest3(unittest.TestCase):

    # Тест функции быстрой сортировки на данных из примера
    def test_scarecrow_sort(self):
        """Тест сортировки Пугалом на данных из примера"""
        # given
        file_in = open("../txtfiles/input.txt")

        n1, k1 = map(int, file_in.readline().split())
        lst1 = list((map(int, file_in.readline().split())))

        n2, k2 = map(int, file_in.readline().split())
        lst2 = list((map(int, file_in.readline().split())))
        file_in.close()

        # when
        print("Просчитаем время и память работы алгоритма")
```

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

        print(scarecrow_sort(n1,k1, lst1))

        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()  # Присваеваем 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

        # then
        self.assertEqual(scarecrow_sort(n1,k1, lst1), "NO")
        self.assertEqual(scarecrow_sort(n2, k2, lst2), "YES")


if __name__ == "__main__":
    unittest.main()
```

Скрины работы тестов:

```
Просчитаем время и память работы алгоритма
NO
Итоговое время: 0:00:00
Используемая память: 0.000208 МБ
Память на пике: 0.006021 МБ
```

	Время выполнения	Затраты памяти (Мб)
Значения из примера	0:00:00	0.000208

Вывод по задаче:

1) Сортировка пугалом легко отображается на реальной жизни.

Задача №5. Индекс Хирша

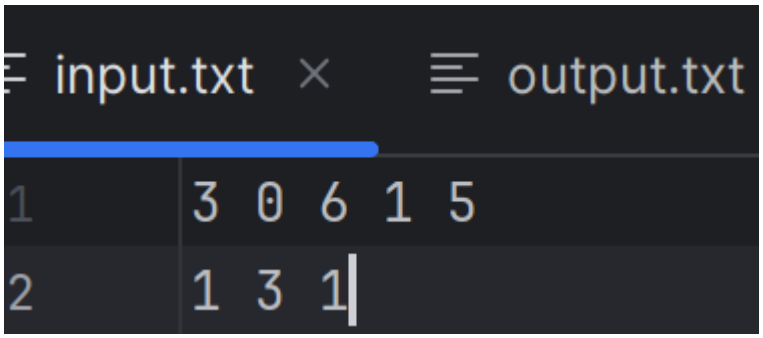
Листинг кода.

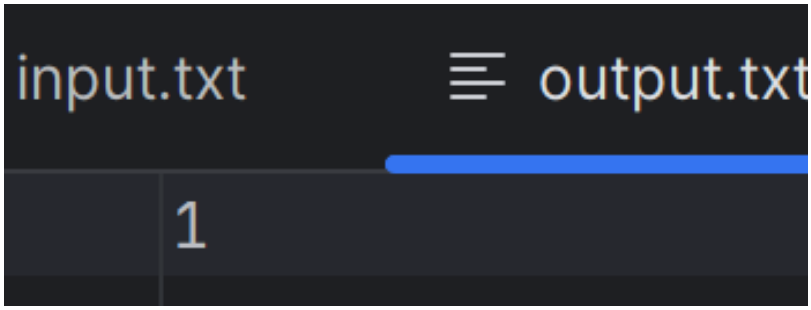
```
"""
Анти-быстрая сортировка
"""

from lab3 import utils


def index_harsh(citations: list) -> int:
    """
    Индекс Хирша
     - Принимает: (список - каждая цифра списка - количество использований статьи)
     - Возвращает: (число - индекс Хирша учёного)
    """
    for h in range(len(citations), 0, -1):
        cnt_citation = 0
        for el in citations:
            if el >= h:
                cnt_citation += 1
        if cnt_citation >= h:
            return h


if __name__ == "__main__":
    lst = utils.read_file_1_list()
    res = index_harsh(lst)
    utils.write_file([res])
```

Текстовое объяснение решения.

Перебираем индексы в обратном порядке(от максимума до 0). Смотрим на количество элементов, если оно больше или равно индексу, то прибавляем количество цитирований. Возвращаем индекс Хирша учёного.

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





Тесты к задаче:

Листинг кода:

```
from lab3.task5.src.task5 import index_harsh
from lab3 import utils
import unittest


class TaskTest5(unittest.TestCase):

    # Тест функции быстрой сортировки на данных из примера
    def test_quick_sort(self):
        """Тест быстрой сортировки на данных из примера"""
        # given
        lst1 = utils.read_file_1_list()
        lst2 = [1, 3, 1]

        # when
        utils.test_memory_and_time_lst(lst1, index_harsh, True)

        # then
        self.assertEqual(index_harsh(lst1), 3)
        self.assertEqual(index_harsh(lst2), 1)


if __name__ == "__main__":
    unittest.main()
```

Скрины работы тестов:

```
Просчитаем время и память работы Сортировки <function index_harsh at 0x000001EA73C82DE0>
3
Итоговое время: 0:00:00
Используемая память: 0.000208 МБ
Память на пике: 0.005893 МБ
```

	Время выполнения	Затраты памяти (Мб)
Сортировка вставкой средние значения	0:00:00	0.000208

Вывод по задаче:

1) Используем двойной перебор для поиска индекса Хирша.

Задача №6. Сортировка целых чисел

Листинг кода.

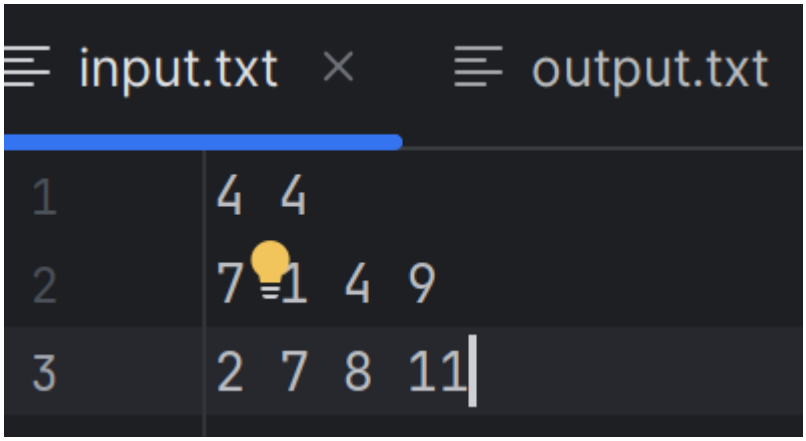
```
"""
Сортировка целых чисел
"""

from lab3 import utils
from lab3.task1.src.task1 import quick_sort


def get_data_c(lst_a:list, lst_b:list) -> list:
    """ Возвращает список, полученный перемножением одинаковых по индексу
элементов списков"""
    return [i*j for i in lst_a for j in lst_b]


def read_input_file_2_numbers_2_lists() -> (int, int, list, list):
    """ Считывает с файла два числа и два списка """
    file_in = open("../txtfiles/input.txt")
    n, m = map(int, file_in.readline().split())
    lst_a = list(map(int, file_in.readline().split()))
    lst_b = list(map(int, file_in.readline().split()))
    file_in.close()
    return n, m, lst_a, lst_b


def sort_z_numbers(lst:list) -> int:
    """
    Сортирует список с помощью быстрой сортировки
     - Принимает: (список)
     - Возвращает (число - сумма каждого 10 элемента в отсортированном
списке)
     """
    quick_sort(lst, 0, len(lst)-1)
    return sum([lst[i] for i in range(0, len(lst), 10)])


if __name__ == "__main__":
    n, m, lst_a, lst_b = read_input_file_2_numbers_2_lists()
    lst_c = get_data_c(lst_a, lst_b)
    res = sort_z_numbers(lst_c)

    utils.write_file([res])
```

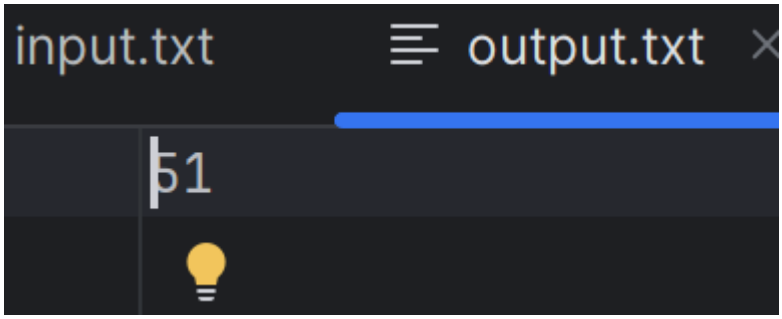
Текстовое объяснение решения.

Создаём функцию, которая обрабатывает входные данные. Также используем дополнительную функцию, которая перемножает одинаковые по индексу элементы из двух списков. Используем быструю сортировку для сортировки массива, возвращаем сумму каждого 10-го элемента в отсортированном списке.

Результат работы кода на примерах из текста задачи:

(скрины input output файлов)





Тесты к задаче:

Листинг кода:

```
from lab3.task6.src.task6 import *
from lab3 import utils
import unittest


class TaskTest6(unittest.TestCase):

    # Тест функции быстрой сортировки на данных из примера
    def test_quick_sort(self):
        """Тест быстрой сортировки на данных из примера"""
        # given
        n, m, lst_a, lst_b = read_input_file_2_numbers_2_lists()
        lst_c = get_data_c(lst_a, lst_b)

        # when
        utils.test_memory_and_time_lst(lst_c, sort_z_numbers, True)

        # then
        self.assertEqual(sort_z_numbers(lst_c), 51)


if __name__ == "__main__":
    unittest.main()
```

Скрины работы тестов:

```
Просчитаем время и память работы Сортировки <function sort_z_numbers at 0x000001F8483ED260>
51
Итоговое время: 0:00:00
Используемая память: 0.000208 МБ
Память на пике: 0.005893 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000208 Мб

Вывод по задаче:

1) Чтобы решить сложную задачу, используем несколько маленьких разных функций, которые в дальнейшем используем для большей.
2) Обрабатывать входные данные можно по-разному.

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. Существует множество задач, которые пересекаются между собой и используют ранее написанные методы или сортировки.