

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

Отчет по лабораторной работе №2  
по курсу «Алгоритмы и структуры данных»  
Тема: Сортировка слиянием, метод декомпозиции  
Вариант 15

Выполнил:  
Левахин Лев Александрович  
К3140

Проверил:



Санкт-Петербург  
2024 г.

## Содержание отчета

<b>Содержание отчета</b>	<b>1</b>
<b>Задачи по варианту</b>	<b>2</b>
Задача №1. Сортировка слиянием	
Задача №4. Бинарный поиск	
Задача №6. Поиск максимального подмассива	
<b>Дополнительные задачи</b>	<b>3</b>
Задача №2. Сортировка слиянием +	
Задача №3. Подсчёт количества инверсий	
Задача №5. Поиск представителя большинства	
<b>Вывод</b>	<b>4</b>

## Задачи по варианту

### Задача №1. Сортировка слиянием

Листинг кода.

```
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 merge_sort(lst: list) -> list:
    """
    Рекурсивная функция
    - Разбивает массив на маленькие части, чтобы сократить количество
    операций
    - Сортирует каждую часть
    - Возвращает отсортированный массив, с помощью процедуры merge
    """
    n = len(lst)
    n1 = n // 2 # Выбираем первую половину массива
    lst1 = lst[:n1] # Делим массив на два, примерно равной длины
    lst2 = lst[n1:]
    if len(lst1) > 1:
        lst1 = merge_sort(lst1)
    if len(lst2) > 1:
        lst2 = merge_sort(lst2)

    return merge(lst1, lst2) # Если делить больше некуда, объединяем
    списки

def merge(lst1: list, lst2: list) -> list:
    """
    Сливают два массива в один
    """
    res = []
    lst1_len = len(lst1)
    lst2_len = len(lst2)
    i, j = 0, 0 # Начинаем сравнивать элементы с 0
    while i < lst1_len and j < lst2_len: # Пока не дойдём до конца списков
        if lst1[i] <= lst2[j]:
            res.append(lst1[i])
            i += 1 # Поскольку добавили предыдущий элемент, больше его не
            сравниваем
        else:
            res.append(lst2[j])
            j += 1
    res += lst1[i:] + lst2[j:] # Добавляем оставшиеся элементы
    return res

res = " ".join([str(el) for el in merge_sort(lst)]) # Список с результатом
приводим к строке и записываем в файл
file_out.write(res)
```

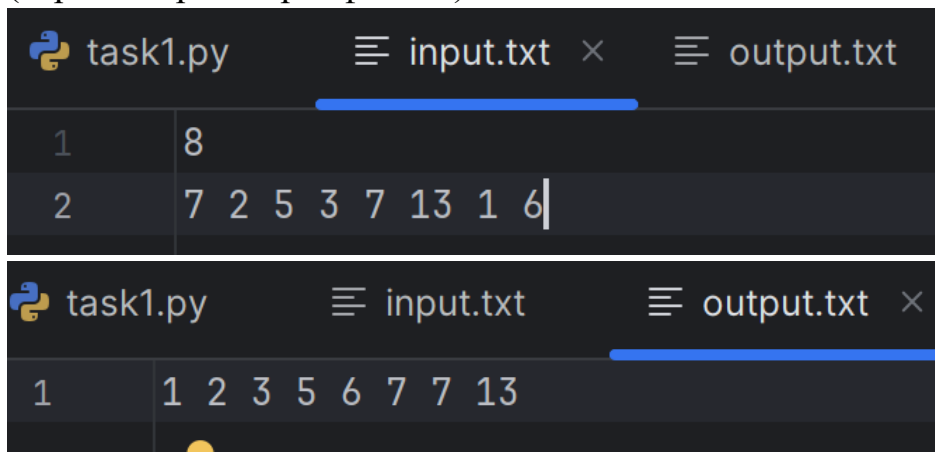
Текстовое объяснение решения.

Создадим две функции мердж, которая будет сливать списки, и мердж сорт, которая будет рекурсивной и будет разделять списки, а возвращать уже склеенный массив. Начнём с последней.

Функция принимает список и возвращает список. Находим длину массива, левую часть списка( с помощью среза до середины) и правую часть от середины. Если длины этих списков больше 1, будем ещё раз применять эту функцию. Если нет, сливаем списки с помощью функции мердж, возвращаем отсортированный список из этих элементов.

Перейдём к мердж – основной функции. Принимает два списка – левую и правую часть основного списка. Находим длины этих списков, создаём переменные счётчики, чтобы отслеживать количество добавленных элементов. Пока эти элементы есть, будем сравнивать их и добавлять в список с результатом тот, который меньше. После полного прохождения по спискам добавляем в результат оставшиеся элементы. Возвращаем список с результатом.

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 lab2.task1.src.task1 import merge_sort
import datetime
import tracemalloc
import random

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

# На данных из примера
def test_sort(func, lst):
    print("Просчитаем время и память работы сортировки")
    tracemalloc.start() # Запускаем счётчик памяти
    start_time = datetime.datetime.now() # Запускаем счётчик времени

    print(func(lst))
```

```

    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

# test_sort(merge_sort, lst)

# На самых больших данных
def test_sort_hard(func):
    lst_hud = [random.randint(1, 10**9) for i in range(10**5)]

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

    func(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() # 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

test_sort_hard(merge_sort)

# Сравнение сортировок на средних данных
def vs_test_middle(func1, func2):
    lst_sr = [random.randint(1, 10_000) for j in range(2_000)]
    n_sr = 2_000

    print(f"Просчитаем время и память работы Сортировки {func1} в среднем
случае")
    tracemalloc.start() # Запускаем счётчик памяти
    start_time = datetime.datetime.now() # Запускаем счётчик времени

    func1(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() # 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

    print(f"Просчитаем время и память работы Сортировки {func2} в среднем
случае")
    tracemalloc.start() # Запускаем счётчик памяти
    start_time = datetime.datetime.now() # Запускаем счётчик времени

    func2(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

# vs_test_middle(merge_sort, insertion_sort)

# Сравнение сортировок на больших данных
def vs_test_hard(func1, func2):
    lst_hud = [random.randint(1, 1_000_000) for i in range(8_000)]
    n_hud = 8_000

    print(f"Просчитаем время и память работы Сортировки {func1} в худшем
случае")
    tracemalloc.start() # Запускаем счётчик памяти
    start_time = datetime.datetime.now() # Запускаем счётчик времени

    func1(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(f"Просчитаем время и память работы Сортировки {func2} в худшем
случае")
    tracemalloc.start() # Запускаем счётчик памяти
    start_time = datetime.datetime.now() # Запускаем счётчик времени

    func2(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

# vs_test_hard(merge_sort, insertion_sort)

```

В тестах сравним сортировку слиянием и сортировку вставкой на больших и средних значениях, а также проверим сортировку слиянием на больших значениях.

Скрины работы тестов:

1) На данных из примера

```
Просчитаем время и память работы сортировки  
[1, 2, 3, 5, 6, 7, 7, 13]  
Итоговое время: 0:00:00.001001  
Используемая память: 0.000368 МБ  
Память на пике: 0.000638 МБ
```

2) В самом худшем случае

```
Просчитаем время и память работы Сортировки <function merge_sort at 0x000002BCC4BC3C40> в худшем случае  
Итоговое время: 0:00:00.643961  
Используемая память: 0.001928 МБ  
Память на пике: 1.728896 МБ
```

3) Сравниваем сортировку вставкой и слиянием на средних данных

```
Просчитаем время и память работы Сортировки <function merge_sort at 0x000002A770723C40> в среднем случае  
Итоговое время: 0:00:00.008000  
Используемая память: 0.001256 МБ  
Память на пике: 0.035408 МБ  
  
Просчитаем время и память работы Сортировки <function insertion_sort at 0x000002A77074C400> в среднем случае  
Итоговое время: 0:00:00.553926  
Используемая память: 0.001312 МБ  
Память на пике: 0.117086 МБ
```

4) Сравниваем сортировку вставкой и слиянием на больших данных

```
Просчитаем время и память работы Сортировки <function merge_sort at 0x000002423EC03C40> в худшем случае  
Итоговое время: 0:00:00.037999  
Используемая память: 0.00148 МБ  
Память на пике: 0.134916 МБ  
  
Просчитаем время и память работы Сортировки <function insertion_sort at 0x000002423EC2C400> в худшем случае  
Итоговое время: 0:00:09.018778  
Используемая память: 0.001536 МБ  
Память на пике: 0.498944 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0ч:00м:00.001001с	0.000368 МБ
На самых больших данных	0ч:00м:00.644с	0.001928 МБ
На средних данных	0ч:00м:00.008с	0.001256 МБ
На больших данных	0ч:00м:00.038с	0.00148 МБ

Вывод по задаче:

- 1) На данных примерах видно, что сортировка слиянием сильно превосходит сортировку вставкой.

- 2) На больших данных сортировка слиянием работает намного быстрее, чем вставкой.

## Задача №4. Бинарный поиск

Листинг кода.

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

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

n = a[0] # Количество элементов массива
a.remove(a[0]) # Массив

k = b[0] # Количество элементов поиска
b.remove(k) # Элементы поиска

def bin_search(n: int, lst: list, k: int, find_el_list: list) -> str:
    """
    - Делим массив на маленькие части, среди которых выполняем поиск
    - Находим серединный элемент, сравниваем с тем, что ищем
    - Сужаем границы поиска до тех пор, пока не найдём совпадение
    - Если совпадений нет - возвращаем -1
    """
    res = [] # Список с индексами
    for el in find_el_list: # Перебираем элементы, индексы которых ищем в
        # списке
        el_is_find = False
        low = 0
        high = n - 1
        while low <= high and not el_is_find:
            mid = (low + high) // 2 # Находим средний элемент
            if lst[mid] == el:
                res.append(mid)
                el_is_find = True
            # Сужаем границы поиска
            elif lst[mid] > el:
                high = mid - 1
            elif lst[mid] < el:
                low = mid + 1
        if not el_is_find:
            res.append(-1)
    res = [str(el) for el in res]
    return " ".join(res)

file_out.write(bin_search(n, a, k, b))
```

Текстовое объяснение решения.

Алгоритм даёт возможность искать нужное значение даже в огромных массивах, при условии, что массив отсортирован. Его смысл в том, чтобы сужать область поиска, по такому же принципу, что и сортировка слиянием – разделяй и властвуй.

Напишем функцию бин сearch, которая принимает количество элементов списка, список, количество элементов поиска, сами элементы, который нужно найти. Для всех этих элементов нужно вывести индекс в

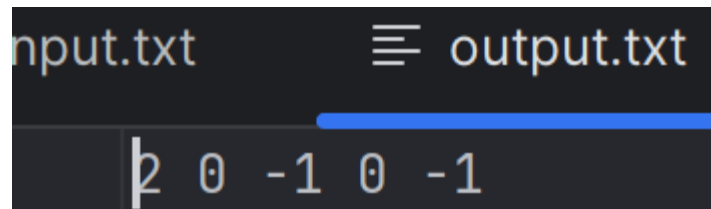


искомом списке, если такого значения там нет, то -1. Создаём список с результатом, в который будем записывать индексы. Создаём флаг, найден ли нужный нам элемент. Создаём две переменные – левую и правую, которые будут границами списка. Далее, пока эти границы имеют смысл и пока элемент не найден, будем искать серединный элемент, проверять, не равен ли он искомому. Если равен – добавляем его в список результатов и флагу присваиваем истину. Если же нет – изменяем границы списка, чтобы рекурсивно проверить левую и правую часть искомого списка(принцип работы такой же, как у сортировки слиянием). Если такого элемента нет – добавляем в результат -1. По окончании цикла возвращаем строку с индексами. Записываем результат работы в файл.

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

```
5 1 5 8 12 13
5 8 1 23 1 11

Сколько чисел + список чисел
Сколько чисел будем искать + список чисел которые будем искать
Возвратить нужно индекс искомого числа, если такого числа нет, то -1
```



Тесты к задаче:

Листинг кода:

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

print(bin_search(n, a, k, b)) # 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:

```
2 0 -1 0 -1
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.000428 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000428 МБ

Вывод по задаче:

- 1) Бинарный поиск работает намного быстрее и эффективнее, чем линейный, так как использует принцип разделяй и властвуй.

## Задача №6. Поиск максимального подмассива

Листинг кода.

```
file_in = open("../txtfiles/gzp.txt").read().split("\n")
file_out = open("../txtfiles/output.txt", "w")

lst = []
price_lst = []
for s in file_in:
    date, price = s.split(";")
    date = str(date)
    price = price.replace(",", ".")
    price = float(price)
    lst.append([date, price])
    price_lst.append(price)

def max_podposl(name: str, lst: list, price_list: list) -> str:
    """
    - Принимает на вход список со списками, в которых дата и цена
    соответственно, и список только с ценами в том же порядке, для удобного
    выбора п\п
    - Ищет максимальную подпоследовательность
    - Разница между первым и последним элементом максимальна
    - Возвращает: дату покупки, выручку, дату продажи
    """
    res = (
        f"Компания, акции которой анализируются: {name}\nАкции
    анализируются в период с {lst[-1][0]} по {lst[0][0]}\n"
        f"Чтобы получить прибыль от акций, лучше всего купить их ")

    low, high, max_sum = find_max_subarray(price_list)
    res += f"{lst[high][0]} за {lst[high][1]} и продать {lst[low][0]} за
    {lst[low][1]}\nТогда можно получить выручку равную {max_sum} за акцию."

    return res

def find_max_subarray(lst: list, low: int = 0, high: int = len(lst) - 1):
```

```

"""
- Алгоритм поиска максимального подмассива, в случаях, если он
находится слева или справа от середины
- Вызывает другую функцию поиска максимальной п\п в случае, если п\п
пересекает середину
"""
if low == high:
    return low, high, abs(lst[low] - lst[high])
else:
    mid = (low + high) // 2
    left_low, left_high, left_sum = find_max_subarray(lst, low, mid)
    right_low, right_high, right_sum = find_max_subarray(lst, mid + 1,
high)
    cross_low, cross_high, cross_sum = find_max_cross_subarray(lst,
low, mid, high)

    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_low, left_high, left_sum
    elif right_sum >= left_sum and right_sum >= cross_sum:
        return right_low, right_high, right_sum
    else:
        return cross_low, cross_high, cross_sum

def find_max_cross_subarray(lst: list, low: int, mid: int, high: int):
    """
    - Алгоритм поиска п\п в случае, если п\п пересекает середину
    """
    max_left, max_right = 0, 0
    left_sum = -10 ** 6
    sum = 0
    for i in range(mid, low+1, -1):
        sum += lst[i]
        if sum > left_sum:
            left_sum = sum
            max_left = i

    right_sum = -10 ** 6
    sum = 0
    for j in range(mid + 1, high+1):
        sum += lst[j]
        if sum > right_sum:
            right_sum = sum
            max_right = j

    return max_left, max_right, abs(lst[max_left] - lst[max_right])

# file_out.write(max_podposl("Газпром", lst, price_lst))
print(max_podposl("Газпром", lst, price_lst))

```

Текстовое объяснение решения.

В задаче требуется проанализировать акции(например Газпрома) и найти подмассив, в котором разница между первым и последним элементами максимальна). Для этого найдем в интернете отчёт какой-нибудь крупной компании за последний месяц. Я нашёл и скачал таблицу с этими данными у Газпрома. Перенесем значения из таблицы в файл, чтобы это выглядело вот так:

```
11.10.2024;131,82
10.10.2024;132,69
09.10.2024;132,51
08.10.2024;133,83
07.10.2024;133,5
04.10.2024;133,32
03.10.2024;133,87
02.10.2024;132,12
01.10.2024;136,21
30.09.2024;138,19
27.09.2024;140,5
26.09.2024;137,35
25.09.2024;135,44
24.09.2024;139,71
23.09.2024;130,35
20.09.2024;122,4
19.09.2024;122,24
18.09.2024;122,18
17.09.2024;123,1
16.09.2024;121,73
13.09.2024;119,91
```

Далее, в коде обработаем эти строчки: разделим их по ; на список из даты и цены акции. Также создадим отдельный список в котором будем сохранять только цены. Это будет нужно(удобно) для дальнейшей работы с ними.

Создадим основную функцию, которая будет выводить в консоль необходимый материал и вызывать другую функцию. Далее создадим функцию, которая будет искать максимальную подпоследовательность, которая будет работать схожим образом по принципу разделяй и властвуй. Разделим функционал на три части: если п\п полностью в левой части массива, если п\п пересекает середину и справа. Далее напишем функцию , которая будет обрабатывать массив, проходящий через середину. Она будет делить массив на левую и правую часть и добавлять элементы из обеих частей, как бы расширяя список с двух сторон. Вернёмся к предыдущей функции. Будем возвращать левый и правый индексы и сумму.

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

```
11.10.2024;131,82
10.10.2024;132,69
09.10.2024;132,51
08.10.2024;133,83
07.10.2024;133,5
04.10.2024;133,32
03.10.2024;133,87
02.10.2024;132,12
01.10.2024;136,21
30.09.2024;138,19
27.09.2024;140,5
26.09.2024;137,35
25.09.2024;135,44
24.09.2024;139,71
23.09.2024;130,35
20.09.2024;122,4
19.09.2024;122,24
18.09.2024;122,18
17.09.2024;123,1
16.09.2024;121,73
13.09.2024;119,91
```

Компания, акции которой анализируются: Газпром

Акции анализируются в период с 13.09.2024 по 11.10.2024

Чтобы получить прибыль от акций, лучше всего купить их 13.09.2024 за 119.91 и продать 24.09.2024 за 139.71

Тогда можно получить выручку равную 19.80000000000001 за акцию.

## Тесты к задаче:

### Листинг кода:

```
lst = []
price_lst = []
for s in file_in:
    date, price = s.split(";")
    date = str(date)
    price = price.replace(",", ".")
    price = float(price)
    lst.append([date, price])
    price_lst.append(price)

print("Просчитаем время и память работы алгоритма поиска максимальной п\п")
tracemalloc.start() # Запускаем счётчик памяти
start_time = datetime.datetime.now() # Запускаем счётчик времени

print(max_podposl("Газпром",lst, price_lst), "\n")

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:

```
Акции анализируются в период с 13.09.2024 по 11.10.2024
Чтобы получить прибыль от акций, лучше всего купить их 13.09.2024 за 119.91 и продать 24.09.2024 за 139.71
Тогда можно получить выручку равную 19.80000000000001 за акцию.
Просчитаем время и память работы алгоритма поиска максимальной п\п
Компания, акции которой анализируются: Газпром
Акции анализируются в период с 13.09.2024 по 11.10.2024
Чтобы получить прибыль от акций, лучше всего купить их 13.09.2024 за 119.91 и продать 24.09.2024 за 139.71
Тогда можно получить выручку равную 19.80000000000001 за акцию.

Итоговое время: 0:00:00
Используемая память: 0.000272 МБ
Память на пике: 0.002379 МБ
```

	Время выполнения	Затраты памяти
Пример из задачи	0:00:00	0.000272 МБ

## Вывод по задаче:

- 1) Алгоритм поиска максимального подмассива можно использовать для решения прикладных задач.

## Дополнительные задачи

### Задача №2. Сортировка слиянием +

Листинг кода.

```
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 merge_sort(lst: list, left: int, right: int) -> list:
    """
    - Рекурсивная функция
    """
    if left < right:
        # Находим середину
        mid = (right + left) // 2
        # Рекурсивно разделяем массив
        merge_sort(lst, left, mid)
        merge_sort(lst, mid + 1, right)
        # Сливаем массив
        merge(lst, left, mid, right)

    if __name__ == "__main__":
        add_merge_description(left+1, right+1, lst[left], lst[right])
    else:
        print(left+1, right+1, lst[left], lst[right])

    return lst

def merge(lst: list, left: int, mid: int, right: int):
    """
    Сликает два отсортированных массива в один
    """
    n1 = mid - left + 1
    n2 = right - mid
    left_lst = [0]*n1
    right_lst = [0]*n2
    # Заполняем левый список
    for i in range(0, n1):
        left_lst[i] = lst[left+i]
    # Заполняем правый список
    for j in range(0, n2):
        right_lst[j] = lst[mid+j+1]
    i, j = 0, 0
    k = left
    # Сортируем элементы
    while i < n1 and j < n2:
        if left_lst[i] <= right_lst[j]:
            lst[k] = left_lst[i]
            i += 1
        else:
            lst[k] = right_lst[j]
            j += 1
        k += 1

    # Добавляем оставшиеся элементы
    while i < n1:
        lst[k] = left_lst[i]
        i += 1
        k += 1
```

```

while j < n2:
    lst[k] = right_lst[j]
    j += 1
    k += 1

def add_merge_description(i: int, j: int, v_i: int, v_j: int):
    """
    Записывает в файл промежуточные операции сортировки слиянием
    """
    file_out.write(f"{i} {j} {v_i} {v_j}\n")

if __name__ == "__main__":
    res = " ".join([str(el) for el in merge_sort(lst, 0, n-1)]) # Список с
    # результатом приводим к строке и записываем в файл
    file_out.write(res)

```

Текстовое объяснение решения.

Принцип работы сортировки описан в задании 1. Но в данной задаче нам потребуются индексы значений, поэтому внесём изменения в алгоритм.

В дополнение напомним функцию адд мердж дескрипшн, которая принимает значения индексов и значения списка (по условию задачи) и записывает их в файл.

Теперь функция мердж сорт принимает помимо списка начальное значение(индекс) и конечное значение(индекс) списка. Проверяем меньше ли начальный индекс конечного. Находим серединный элемент, и снова запускаем функцию от левого и правого списков, которые делятся до середины и от середины соответственно. Запускаем функцию мердж от начала, середины, конца. Если программа запущена из текущего файла, то записываем результат работы сортировки в выходной файл, если вызвана (для тестов) – выводим в консоль. Возвращаем лист.

В функции мердж делим список на левый и правый, заполняем их элементами основного списка. Создаём маркеры для левого и правого списка и для основного списка. Начинаем сортировать изначальный список присваивая меньшие элементы из левого или правого. Добавляем оставшиеся элементы.

Записываем результат в файл.

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

output.txt	input.txt	output.txt	input.txt
1 10	1 4	1 4	4
2 1 8 2 1 4 7 3 2 3 6	2 9 7 5 8	2 9 7 5 8	8



1	2	1	8		
1	3	1	8		
4	5	1	4		
1	5	1	8		
6	7	3	7		
6	8	2	7		
9	10	3	6		
6	10	2	7		
1	10	1	8		
1	1	2	2	3	3
4	6	7	8		

1	1	2	7	9
2	3	4	5	8
3	1	4	5	9
4	5	7	8	9

Тесты к задаче:

Листинг кода:

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

print(merge_sort(lst, 0, n-1))

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:

```
Просчитаем время и память работы сортировки
1 2 1 8
1 3 1 8
4 5 1 4
1 5 1 8
6 7 3 7
6 8 2 7
9 10 3 6
6 10 2 7
1 10 1 8
[1, 1, 2, 2, 3, 3, 4, 6, 7, 8]
Итоговое время: 0:00:00
Используемая память: 8e-05 МБ
Память на пике: 0.00035 МБ
```

	Время выполнения	Затраты памяти (Мб)
Результаты на примере	0.00.00с	0.00035

Вывод по задаче:

- 1) Сортировку слиянием можно использовать решения различных задач, дополняя её функционал.

### Задача №3. Поиск количества инверсий

Листинг кода.

```
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 find_inverse(n:int, lst:list)->str:
    """
    Функция находит количество инверсий в списке, используя сортировку
    слиянием
    """
    cnt_inverse = merge_sort(lst, 0, n-1)

    return str(cnt_inverse)

def merge_sort(lst: list, left: int, right: int)->int:
    """
    Рекурсивная функция
    """
    cnt = 0
    if left < right:
        # Находим середину
        mid = (right + left) // 2
        # Рекурсивно разделяем массив
        cnt += merge_sort(lst, left, mid)
        cnt += merge_sort(lst, mid + 1, right)
        # Сливаем массив
        cnt += merge(lst, left, mid, right)

    return cnt

def merge(lst: list, left: int, mid: int, right: int)->int:
    """
    Сликает два отсортированных массива в один
    """
    cnt = 0
    n1 = mid - left + 1
    n2 = right - mid
    left_lst = [0]*n1
    right_lst = [0]*n2
    # Заполняем левый список
    for i in range(0, n1):
        left_lst[i] = lst[left+i]
    # Заполняем правый список
    for j in range(0, n2):
        right_lst[j] = lst[mid+j+1]
```

```

i, j = 0, 0
k = left
# Сортируем элементы
while i < n1 and j < n2:
    if left_lst[i] <= right_lst[j]:
        lst[k] = left_lst[i]
        i += 1
    else:
        lst[k] = right_lst[j]
        j += 1
    cnt += (n1 - i) # Добавляем к счётчику инверсий все случаи, когда
    # текущий элемент больше последующих
    k += 1

# Добавляем оставшиеся элементы
while i < n1:
    lst[k] = left_lst[i]
    i += 1
    k += 1
while j < n2:
    lst[k] = right_lst[j]
    j += 1
    k += 1

return cnt

file_out.write(find_inverse(n, lst))

```

Текстовое объяснение решения.

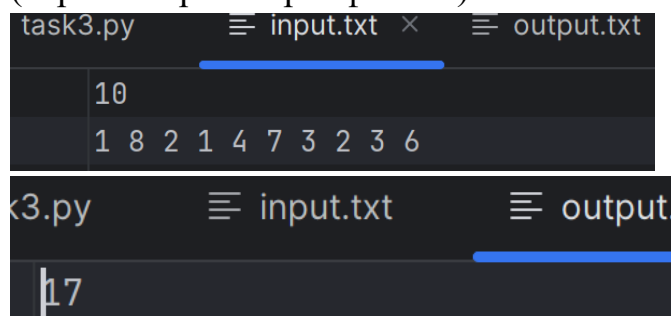
Полностью повторяем метод сортировки слиянием из описания предыдущей задачи (без учёта дополнительных условий). Однако в этой задаче введём дополнительную функцию, которая будет присваивать счётчику результат работы сортировки и возвращать количество инверсий.

Основной функционал придётся на сортировку слиянием. Введём в методе `merge_sort` дополнительную переменную счётчик. Будем присваивать ей результат работы рекурсии. В конце метода возвращать будем счётчик. Также дополнительно в методе `merge` будем добавлять значения в счётчик на этапе сортировки списков, если предыдущий элемент больше текущего. Возвращаем счётчик.

Записываем в файл результат работы `find_inverses`.

Результат работы кода на примерах из текста задачи:

(скрины `input` `output` файлов)



Тесты к задаче:

Листинг кода:

```
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(find_inverse(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:

```
17
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	0.00035 МБ

Вывод по задаче:

- 1) Сортировку слиянием можно использовать для различных задач, потому что она быстрая и удобная.

## Задача №5. Поиск представителя большинства

Листинг кода.

```
def find_el_bolsh(lst: list, n: int) -> str:
    """
    Функция записывает в файл результат поиска элемента большинства в
    списке:
    - Возвращает 1, если такой элемент есть
    - Возвращает 0, если такого элемента нет
    """
    el_bolsh = majority(lst, 0, n - 1)
    if el_bolsh != 0:
        return "1\n"
    return "0\n"
```

```

def majority(lst: list, left: int, right: int):
    """
    Функция поиска элемента большинства
    - Возвращает элемент большинства, если такой есть
    - Возвращает 0, если такого элемента нет
    """
    if left == right:
        return lst[left]

    mid = (right + left) // 2
    left_find_el = majority(lst, left, mid)
    right_find_el = majority(lst, mid + 1, right)

    if left_find_el == right_find_el:
        return left_find_el
    if count_el(lst, left, mid, left_find_el) == 1 or count_el(lst, mid + 1, right, right_find_el) == 1:
        return 0
    if count_el(lst, left, mid, left_find_el) > count_el(lst, mid + 1, right, right_find_el):
        return left_find_el
    else:
        return right_find_el

def count_el(lst: list, left: int, right: int, finding_el: int):
    cnt = 0
    for i in range(left, right + 1):
        if lst[i] == finding_el:
            cnt += 1
    return cnt

if __name__ == "__main__":
    file_in = open("../txtfiles/input.txt")
    file_out = open("../txtfiles/output.txt", "w")

    n = int(file_in.readline()) # Количество элементов в 1 примере
    lst = list(map(int, file_in.readline().split())) # Список с элементами
    в 1 примере
    n2 = int(file_in.readline()) # Количество элементов во 2 примере
    lst2 = list(map(int, file_in.readline().split())) # Список с
    элементами во 2 примере

    file_out.write(find_el_bolsh(lst, n)) # Результат 1 примера
    file_out.write(find_el_bolsh(lst2, n2)) # Результат 2 примера

```

Текстовое объяснение решения.

Функция должна возвращать 1, если есть элемент, который во

Создадим функцию, которая будет основной – возвращать значение - 1 если элемент большинства есть, 0 – если нет.

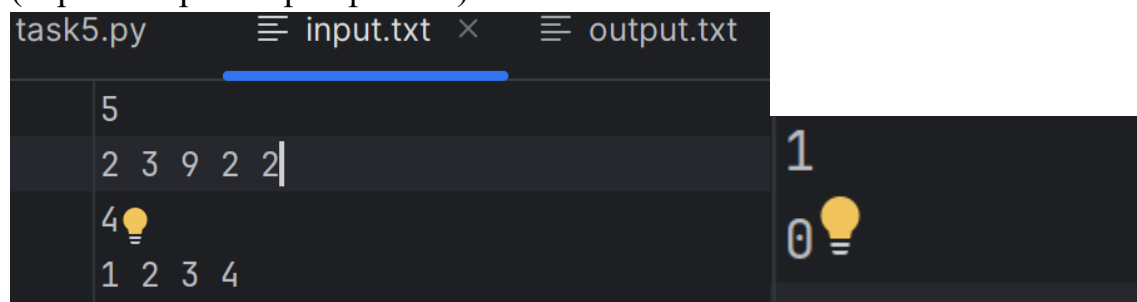
Создадим функцию мажорити, которая будет искать сам элемент большинства. Она принимает сам список, начальный и конечный индексы. В начале проверим рекурсию: если начальный индекс равен конечному – значит мы нашли нужный элемент. Возвращаем его. Вычисляем серединный индекс, создадим левый и правый элементы, которым присваиваем результат работы рекурсии функции (до середины и от

середины соответственно). Далее нужно создать функцию, которая будет подсчитывать количество элементов в списке. Создадим счётчик и будем проходиться по списку, если находим нужный элемент – прибавляем к счётчику 1. Возвращаем количество таких элементов(кнт). Вернёмся к мажорити. После проверки, будем проверять не равно ли значение от результата функции каунт ел от разных индексов списков, если равно – возвращаем 0, так как элемента большинства нет. Если же в левом списке больше, чем в правом, то возвращаем элемент, который искали слева. Если нет – то тот, который искали справа.

Далее, если запущен текущий файл, будем записывать выходной файл результат значения основной функции. Сделаем это для двух примеров из условия задачи.

Результат работы кода на примерах из текста задачи:

(скрины input output файлов)



```

task5.py  input.txt  output.txt
5
2 3 9 2 2
4
1 2 3 4
1
0

```

Тесты к задаче:

Листинг кода:

```

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

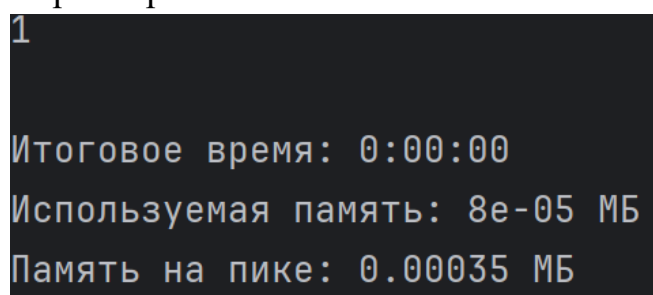
print(find_el_bolsh(lst, n)) # Выводим результат отработанной функции

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

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	0.00035 Мб

Вывод по задаче:

- 1) Эта задача отлично показывает, что даже лёгкую задачу лучше решать используя принцип разделяй и властвуй.
- 2) Лучше делить даже простые задачи на ещё более простые задачи, так, код будет выглядеть чище.

## **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. Принцип разделяй и властвуй существенно ускоряет время работы алгоритмов. Его можно использовать в множестве разных задач.