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 3

Выполнил:
Бай М.О.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Оглавление

Задачи по варианту

Задача №1. Сортировка слиянием

Текст задачи:

1. Используя *псевдокод* процедур `Merge` и `Merge-sort` из презентации к Лекции 2 (страницы 6-7), напишите программу сортировки слиянием на Python и проверьте сортировку, создав несколько рандомных массивов, подходящих под параметры:

   - **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 2 \cdot 10^4$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, по модулю не превосходящих 10^9.
   - **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным массивом. Между любыми двумя числами должен стоять ровно один пробел.
   - Ограничение по времени. 2сек.
   - Ограничение по памяти. 256 мб.

Код программы:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения

# Функция для слияния двух отсортированных массивов
def merge(left, right):
    r = []  # Инициализация пустого списка для хранения результата слияния
    i = q = 0    # Индексы для текущих элементов левого и правого массивов

    # Объединяем два массива, пока есть элементы в обоих
    while i < len(left) and q < len(right):
        if left[i] < right[q]:  # Если элемент из левой части меньше элемента из правой
            r.append(left[i])  # Добавляем его в результат
            i += 1  # Переходим к следующему элементу в левой части
        else:  # Если элемент из правой части меньше или равен
            r.append(right[q])  # Добавляем его в результат
            q += 1  # Переходим к следующему элементу в правой части

    # Копируем оставшиеся элементы из левой части, если они есть
    r.extend(left[i:])
    # Копируем оставшиеся элементы из правой части, если они есть
    r.extend(right[q:])

    return r  # Возвращаем отсортированный массив, объединенный из двух частей


# Функция сортировки слиянием
```

```
def merge_sort(l):
    if len(l) <= 1:  # Если массив содержит 0 или 1 элемент, он уже отсортирован
        return l

    mid = len(l) // 2  # Находим середину массива
    # Рекурсивно сортируем левую половину массива
    left = merge_sort(l[:mid])
    # Рекурсивно сортируем правую половину массива
    right = merge_sort(l[mid:])

    # Объединяем отсортированные половины
    return merge(left, right)


if __name__ == '__main__':
    # Открываем файл "input.txt" для чтения
    with open('input.txt', 'r') as f:
        n = int(f.readline().strip())  # Читаем количество элементов (первую строку) и преобразуем в целое
число
        # Читаем вторую строку, разбиваем на числа и преобразуем в список целых чисел
        l = list(map(int, f.readline().strip().split()))

    # Сортировка массива с помощью функции merge_sort
    sl = merge_sort(l)

    # Открываем файл "output.txt" для записи
    with open('output.txt', 'w') as f:
        # Записываем отсортированный массив в файл в виде строки
        f.write(' '.join(map(str, sl)))

    # Вычисляем время выполнения
    elapsed_time = time.perf_counter() - t_start
    print(f"Время выполнения: {elapsed_time:.6f} секунд")

    # Получаем текущее использование памяти
    current, peak = tracemalloc.get_traced_memory()
    print(f"Текущая память: {current / 10 ** 6:.6f} МБ; Пиковая память: {peak / 10 ** 6:.6f} МБ")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()
```

Описание работы программы:

Импорт необходимых модулей:

time: используется для замера времени выполнения программы.

tracemalloc: модуль для отслеживания выделенной памяти в процессе выполнения программы.

Инициализация отслеживания памяти:

Вызов tracemalloc.start() начинает отслеживание выделенной памяти.

Замер времени начала выполнения:

Используется time.perf_counter() для получения времени, прошедшего с начала выполнения программы.

Определение функции merge:

Эта функция принимает два отсортированных массива (left и right) и выполняет их слияние в один отсортированный массив.

Алгоритм слияния:

Создает пустой список r для хранения результата.

Использует два указателя (индекса) i и q, которые указывают на текущие элементы в левой и правой частях массивов соответственно.

Сравнивает текущие элементы и добавляет меньший в результирующий массив, затем сдвигает указатель соответствующего массива.

После завершения основной части слияния, добавляет оставшиеся элементы из обоих массивов, если они есть.

Определение функции merge_sort:

Эта рекурсивная функция реализует алгоритм сортировки слиянием.

Если длина входного массива l меньше или равна 1, возвращает его (он уже отсортирован).

Находит середину массива и рекурсивно сортирует левую и правую половины.

Объединяет (сливает) отсортированные половины, используя функцию merge.

Основной блок программы:

Программа открывает файл input.txt и считывает данные.

Первая строка содержит количество элементов n, которая используется для дальнейших операций и также служит примером структурированного ввода.

Вторая строка содержит элементы массива, которые считываются, разбиваются на числа и преобразуются в список.

Вызывает функцию merge_sort, передавая ей считанный массив, чтобы отсортировать его.

Открывает файл output.txt для записи и записывает в него отсортированный массив в виде строки.

Замер и вывод результатов:

После сортировки программа вычисляет elapsed_time, чтобы определить, сколько времени занял процесс сортировки, и выводит это значение в секундах.

Получает текущее и пиковое использование памяти с помощью tracemalloc.get_traced_memory() и выводит эти значения в мегабайтах.

Завершает отслеживание памяти вызовом tracemalloc.stop().

Тесты:

```python
import unittest
from Lab2.task1.src.Merge_Sort import *


class TestMergeSort(unittest.TestCase):  Mihail-Bay

    def test_empty_list(self):  Mihail-Bay
        self.assertEqual(merge_sort([]), second: [])

    def test_single_element(self):  Mihail-Bay
        self.assertEqual(merge_sort([1]), second: [1])
        self.assertEqual(merge_sort([-1]), second: [-1])

    def test_sorted_list(self):  Mihail-Bay
        self.assertEqual(merge_sort([1, 2, 3, 4, 5]), second: [1, 2, 3, 4, 5])

    def test_reverse_sorted_list(self):  Mihail-Bay
        self.assertEqual(merge_sort([5, 4, 3, 2, 1]), second: [1, 2, 3, 4, 5])

    def test_unsorted_list(self):  Mihail-Bay
        self.assertEqual(merge_sort([3, 2, 5, 4, 1]), second: [1, 2, 3, 4, 5])

    def test_duplicates(self):  Mihail-Bay
        self.assertEqual(merge_sort([1, 2, 2, 3, 1]), second: [1, 1, 2, 2, 3])

    def test_large_numbers(self):  Mihail-Bay
        self.assertEqual(merge_sort([1000, 500, 300, 800]), second: [300, 500, 800, 1000])

    def test_negative_numbers(self):  Mihail-Bay
        self.assertEqual(merge_sort([-1, -2, 0, 2, 1]), second: [-2, -1, 0, 1, 2])


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

2 ms ✓ Tests passed: 8 of 8 tests – 2 ms

Вывод по задаче:
Программа представляет собой демонстрацию алгоритма сортировки слиянием, который является эффективным методом сортировки с временной сложностью O(n log n). Она также включает в себя управление памятью и замер времени выполнения, что делает её полезной для анализа производительности алгоритма и его потребления ресурсов.

Задача №2. Сортировка вставкой +

Текст задачи:

Дан массив целых чисел. Ваша задача — отсортировать его в порядке неубывания *с помощью сортировки слиянием*.

Чтобы убедиться, что Вы действительно используете сортировку слиянием, мы просим Вас, после каждого осуществленного слияния (то есть, когда соответствующий подмассив уже отсортирован!), выводить индексы граничных элементов и их значения.

Код программы:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения


def merge(left, right, start_index, mid_index, end_index, output_file):
    r = []  # Список для хранения результата слияния
    i = q = 0    # Инициализация индексов для левой и правой части

    # Объединяем два массива, пока оба имеют элементы
    while i < len(left) and q < len(right):
        if left[i] < right[q]:  # Если текущий элемент левой части меньше
            r.append(left[i])  # Добавляем его в результат
            i += 1  # Переходим к следующему элементу левой части
        else:  # Если элемент правой части меньше или равен
            r.append(right[q])  # Добавляем его в результат
            q += 1  # Переходим к следующему элементу правой части

    # Добавляем оставшиеся элементы из левой части, если таковые есть
    r.extend(left[i:])
    # Добавляем оставшиеся элементы из правой части, если таковые есть
    r.extend(right[q:])

    # Запись индексов и значений в выходной файл
    if len(left) > 1 or len(right) > 1:  # Проверяем, чтобы записем только при необходимости
        output_file.write(str(f"{start_index} {mid_index} {r[0]} {r[-1]}\n"))

    return r  # Возвращаем отсортированный массив


def merge_sort(l, output_file, start_index=1):
    if len(l) <= 1:  # Если массив содержит 0 или 1 элемент, он уже отсортирован
        return l

    mid = len(l) // 2  # Нахождение середины массива
    # Рекурсивно сортируем левую половину
    left = merge_sort(l[:mid], output_file, start_index)
    # Рекурсивно сортируем правую половину
    right = merge_sort(l[mid:], output_file, start_index + mid)

    # Объединяем отсортированные половины и записываем результат
    return merge(left, right, start_index, start_index + mid - 1, start_index + len(l) - 1, output_file)

if __name__ == '__main__':
```

```
# Чтение входных данных из файла
with open('input.txt', 'r') as f:
    n = int(f.readline().strip())  # Считываем количество элементов
    l = list(map(int, f.readline().strip().split()))  # Считываем массив чисел

# Открытие выходного файла для записи
with open('output.txt', 'w') as f:
    # Выполнение сортировки и передача файла для записи
    sl = merge_sort(l, f)
    # Запись отсортированных данных в выходной файл
    f.write(' '.join(map(str, sl)))  # Записываем отсортированный массив в файл

# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Описание работы программы:

1. Импорт необходимых библиотек

Модуль time: используется для измерения времени выполнения программы.

Модуль tracemalloc: предназначен для отслеживания использования памяти в ходе выполнения.

2. Начало отслеживания памяти и времени

tracemalloc.start() запускает отслеживание использования памяти.

t_start = time.perf_counter() фиксирует текущее время для последующего измерения времени выполнения программы.

3. Определение функции merge()

Функция merge(left, right, start_index, mid_index, end_index, output_file) отвечает за объединение двух отсортированных массивов:

Инициализация результатов:

Создается пустой список r для накопления результатов слияния. Два указателя i и q инициализируются для обращения к элементам массивов left и right.

Основной цикл слияния:

Происходит сравнение текущих элементов массивов. Меньший элемент добавляется в результирующий массив r, и указатель соответствующего массива увеличивается.

Добавление оставшихся элементов: После завершения сравнения, если один из массивов имеет элементы, они добавляются к r.

Запись в выходной файл: Если массивы left или right содержат более одного элемента, записываются индексы и значения первого и последнего элемента результирующего массива в указанный выходной файл.

4. Определение функции merge_sort()

Функция merge_sort(l, output_file, start_index=1) реализует сам алгоритм сортировки слиянием:

Базовый случай: Если массив имеет 0 или 1 элемент, он считается отсортированным.

Рекурсивное разделение: Массив l делится пополам, и для каждой половины рекурсивно вызывается merge_sort().

Слияние отсортированных половин: По завершению сортировки обеих половин, их результирующие массивы передаются функции merge() для объединения.

5. Запуск программы

В блоке if __name__ == '__main__': происходит выполнение следующих шагов:

Чтение входных данных из файла: Программа считывает данные из файла input.txt, где первая строка содержит количество элементов, а вторая строка — сами элементы.

Создание выходного файла: Открывается файл output.txt для записи результатов.

Вызов сортировки: Программа запускает сортировку слиянием, передавая массив и файл для записи.

Запись отсортированных данных: Отсортированный массив записывается в выходной файл в формате, удобном для чтения.

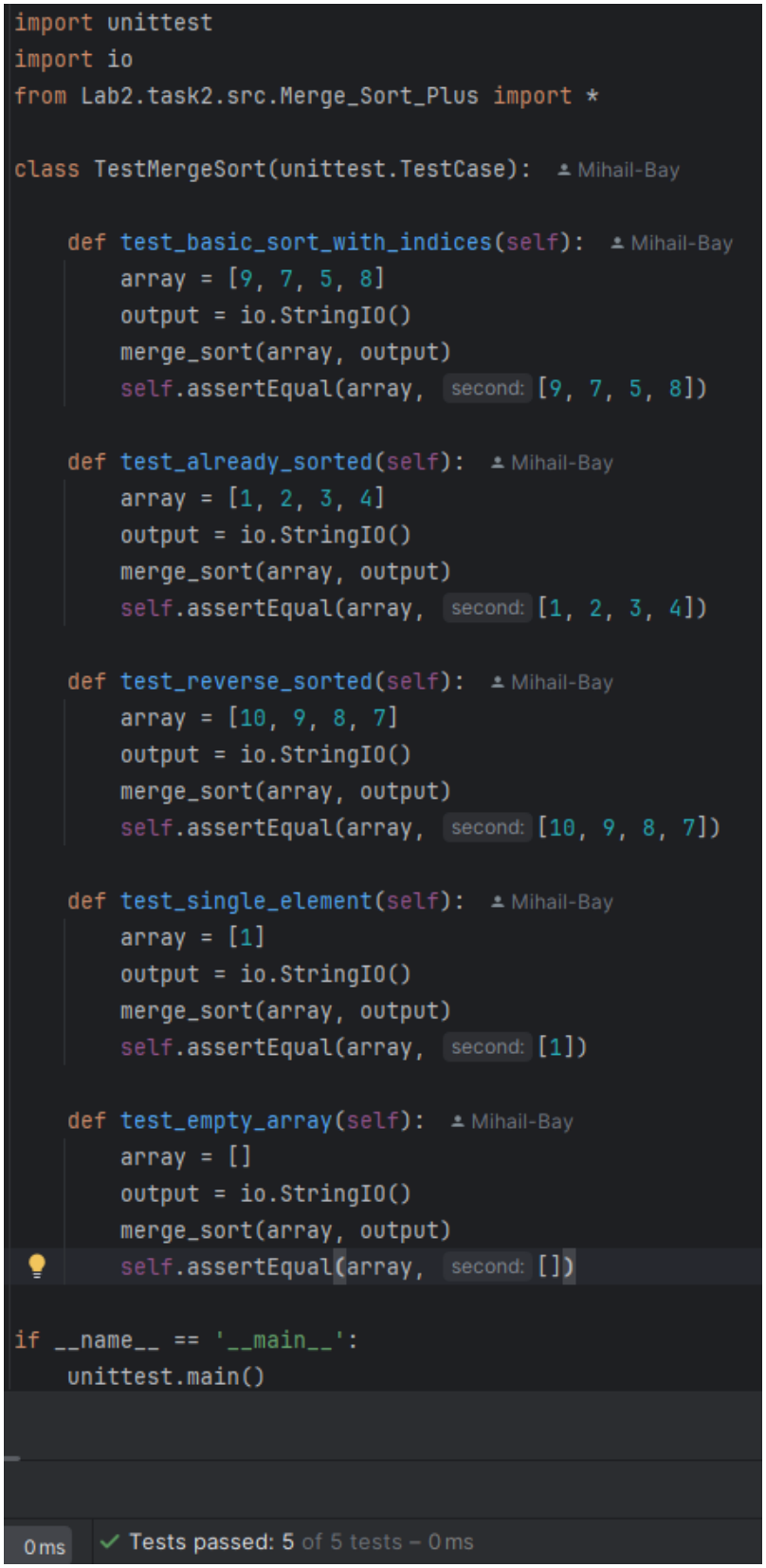
6. Измерение времени выполнения и использования памяти

Время выполнения: Рассчитывается разница между текущим временем и тем, что было сохранено в t_start.

Использование памяти: Используя tracemalloc.get_traced_memory(), программа получает текущее и пиковое использование памяти и выводит их на экран.

Остановка отслеживания: В конце работы программы выполняется tracemalloc.stop(), прекращая отслеживание использования памяти.

Тесты:



Вывод по задаче:

Данная программа демонстрирует, как можно использовать алгоритм сортировки слиянием и отслеживание времени и памяти для анализа производительности. Это полезный инструмент для изучения алгоритмов и их оптимизации.

Задача №3. Число инверсий

Текст задачи:

Инверсией в последовательности чисел A называется такая ситуация, когда $i < j$, а $A_i > A_j$. Количество инверсий в последовательности в некотором роде определяет, насколько близка данная последовательность к отсортированной. Например, в сортированном массиве число инверсий равно 0, а в массиве, сортированном наоборот - каждые два элемента будут составлять инверсию (всего $n(n-1)/2$).

Дан массив целых чисел. Ваша задача — подсчитать число инверсий в нем.

Подсказка: чтобы сделать это быстрее, можно воспользоваться модификацией сортировки слиянием.

Код программы:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения


def merge_and_count(left, right):
    r = []  # Инициализируем пустой список для хранения отсортированных элементов
    i = q = 0  # Индексы для подмассивов left и right
    inversions_count = 0  # Счетчик инверсий

    # Проходим по обоим массивам, пока не достигнем конца одного из них
    while i < len(left) and q < len(right):
        if left[i] <= right[q]:  # Сравниваем текущие элементы
            r.append(left[i])  # Если элемент из left меньше, добавляем его в результат
            i += 1  # Переходим к следующему элементу в left
        else:
            # Здесь, если элемент из right меньше, значит, все оставшиеся элементы в left
            # образуют инверсии с текущим элементом right[j]
            inversions_count += len(left) - i  # Увеличиваем счетчик инверсий
            r.append(right[q])  # Добавляем элемент из right в результат
            q += 1  # Переходим к следующему элементу в right

    r.extend(left[i:])  # Добавляем оставшиеся элементы из left, если они есть
    r.extend(right[q:])  # Добавляем оставшиеся элементы из right, если они есть

    return r, inversions_count  # Возвращаем отсортированный массив и количество инверсий


def merge_sort_and_count(l):
    if len(l) <= 1:  # Если массив имеет 0 или 1 элемент
        return l, 0  # Возвращаем его и 0 инверсий

    mid = len(l) // 2  # Находим середину массива

    # Рекурсивно сортируем левую и правую половины и считаем инверсии
    left, left_inversions = merge_sort_and_count(l[:mid])
    right, right_inversions = merge_sort_and_count(l[mid:])

    # Сливаем отсортированные подмассивы и считаем инверсии при слиянии
```

```
    merged, split_inversions = merge_and_count(left, right)

    # Общее количество инверсий - инверсии в левой части + инверсии в правой части + инверсии при слиянии
    total_inversions = left_inversions + right_inversions + split_inversions

    return merged, total_inversions  # Возвращаем отсортированный массив и общее количество инверсий


if __name__ == '__main__':
    with open('input.txt', 'r') as f:  # Открываем файл для чтения
        n = int(f.readline().strip())  # Читаем количество элементов в массиве
        l = list(map(int, f.readline().strip().split()))  # Читаем массив целых чисел


    sl, inversions = merge_sort_and_count(l)

    with open('output.txt', 'w') as f:
        f.write(str(inversions))  # Записываем число инверсий

    # Вычисляем время выполнения
    elapsed_time = time.perf_counter() - t_start
    print(f"Время выполнения: {elapsed_time:.6f} секунд")

    # Получаем текущее использование памяти
    current, peak = tracemalloc.get_traced_memory()
    print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()
```

Описание работы программы:

Импорт библиотек Программа использует два модуля:

time — для измерения времени выполнения.

tracemalloc — для отслеживания потребления памяти.

Запуск отслеживания

python

```
tracemalloc.start()
t_start = time.perf_counter()
```

Здесь начинается отслеживание памяти, и фиксируется начальное время выполнения программы.

Функция merge_and_count(left, right) Эта функция отвечает за слияние двух отсортированных подмассивов и подсчет инверсий:

Инициализация: Создаются переменные для результата r, индексов для подмассивов i и q, а также счетчик инверсий.

Основной цикл: Происходит поэлементное сравнение массивов left и right:

Если элемент из массива left меньше или равен элементу из right, он добавляется к результирующему массиву, и индекс увеличивается.

Если элемент из right меньше, это означает, что все оставшиеся элементы в left образуют инверсии с этим элементом. Количество таких инверсий добавляется к счетчику.

Добавление оставшихся элементов: Оставшиеся части массивов также добавляются в результирующий массив.

Функция merge_sort_and_count(l) Эта функция реализует сам алгоритм сортировки слиянием:

База рекурсии: Если массив состоит из одного или нуля элементов, он возвращается вместе со счетчиком инверсий, равным нулю.

Рекурсивное деление: Массив делится на две половины, и для каждой половины рекурсивно вызывается данная же функция.

Слияние и подсчет инверсий: Вызывает merge_and_count для объединения отсортированных половин и подсчета инверсий, происходящих при объединении.

Возврат результата: Возвращает отсортированный массив и общее количество инверсий.

Основная логика выполнения

Программа открывает файл input.txt для чтения массивов целых чисел.

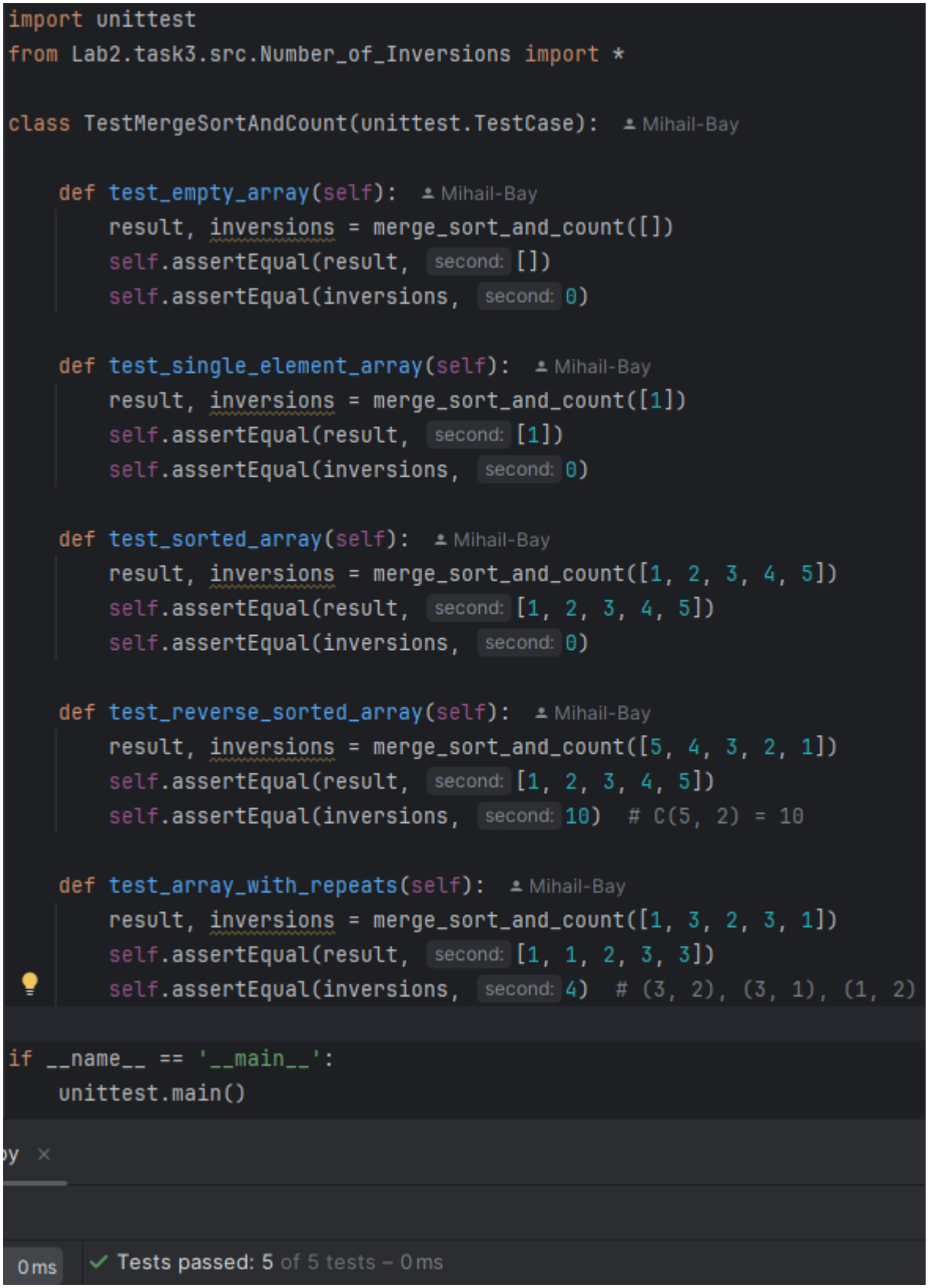
Вызывает функцию для сортировки и подсчета инверсий.

Записывает количество инверсий в файл output.txt.

Измеряет и выводит время выполнения, а также текущее и пиковое потребление памяти.

Останавливает отслеживание использования памяти.

Тесты:



```
import unittest
from Lab2.task3.src.Number_of_Inversions import *

class TestMergeSortAndCount(unittest.TestCase):  Mihail-Bay

    def test_empty_array(self):  Mihail-Bay
        result, inversions = merge_sort_and_count([])
        self.assertEqual(result, second: [])
        self.assertEqual(inversions, second: 0)

    def test_single_element_array(self):  Mihail-Bay
        result, inversions = merge_sort_and_count([1])
        self.assertEqual(result, second: [1])
        self.assertEqual(inversions, second: 0)

    def test_sorted_array(self):  Mihail-Bay
        result, inversions = merge_sort_and_count([1, 2, 3, 4, 5])
        self.assertEqual(result, second: [1, 2, 3, 4, 5])
        self.assertEqual(inversions, second: 0)

    def test_reverse_sorted_array(self):  Mihail-Bay
        result, inversions = merge_sort_and_count([5, 4, 3, 2, 1])
        self.assertEqual(result, second: [1, 2, 3, 4, 5])
        self.assertEqual(inversions, second: 10)  # C(5, 2) = 10

    def test_array_with_repeats(self):  Mihail-Bay
        result, inversions = merge_sort_and_count([1, 3, 2, 3, 1])
        self.assertEqual(result, second: [1, 1, 2, 3, 3])
        self.assertEqual(inversions, second: 4)  # (3, 2), (3, 1), (1, 2)

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


Вывод по задаче:

Программа хорошо иллюстрирует не только алгоритм сортировки, но и современный подход к анализу производительности с помощью отслеживания временных и памяти, что делает её полезной для более глубокого понимания вычислительных процессов.

Задача №4. Бинарный поиск

Текст задачи:

В этой задаче вы реализуете алгоритм бинарного поиска, который позволяет очень эффективно искать (даже в огромных) списках при условии, что список отсортирован. Цель - реализация алгоритма двоичного (бинарного) поиска.

Код программы:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения


def binary_search(l, x):

    left, right = 0, len(l) - 1  # Устанавливаем пределы поиска

    while left <= right:  # Пока левый индекс меньше или равен правому
        mid = (left + right) // 2  # Находим середину
        if l[mid] == x:  # Если элемент в середине равен искомому
            return mid  # Возвращаем индекс
        elif l[mid] < x:  # Если элемент в середине меньше искомого
            left = mid + 1  # Ищем в правой части
        else:  # Если элемент в середине больше искомого
            right = mid - 1  # Ищем в левой части

    return -1  # Если элемент не найден, возвращаем -1


if __name__ == '__main__':
    # Чтение данных из файла input.txt
    with open('input.txt', 'r') as f:
        n = int(f.readline().strip())  # Читаем число элементов в массиве
        l = list(map(int, f.readline().strip().split()))  # Читаем отсортированный массив
        k = int(f.readline().strip())  # Читаем число значений для поиска
        queries = list(map(int, f.readline().strip().split()))  # Читаем значения для поиска

    # Список для сохранения результатов
    r = []

    # Проходим по каждому элементу из queries
    for q in queries:
        ind = binary_search(l, q)  # Ищем индекс элемента в массиве
        r.append(ind)  # Добавляем результат в список

    # Запись результата в файл output.txt
    with open('output.txt', 'w') as f:
        f.write(' '.join(map(str, r)))  # Записываем индексы, разделяя пробелами

    # Вычисляем время выполнения
    elapsed_time = time.perf_counter() - t_start
    print(f"Время выполнения: {elapsed_time:.6f} секунд")
```

```
# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Описание работы программы:

Импорт библиотек:

time: используется для измерения времени выполнения программы.

tracemalloc: модуль, который отслеживает использование памяти программой.

Инициализация отслеживания памяти:

Функция binary_search(l, x):

Входные параметры:

l: отсортированный список, в котором будет производиться поиск.

x: элемент, который необходимо найти.

Процесс поиска:

Устанавливаются начальные границы поиска (left и right).

Используется цикл while, который продолжает работу до тех пор, пока левая граница меньше или равна правой.

На каждой итерации определяется середина массива (mid).

Если элемент в середине равен искомому, возвращается индекс.

Если элемент в середине меньше искомого, продолжается поиск в правой половине.

Если больше, поиск продолжается в левой половине.

Возврат значения: Если элемент не найден, функция возвращает -1.

Чтение данных из файла input.txt:

Программа считывает данные, содержащие количество элементов в массиве, сам отсортированный массив, количество значений для поиска и сами значения для поиска.

Данные из файла обрабатываются и преобразуются в нужные форматы.

Основной процесс поиска:

Для каждого элемента из списка queries вызывается функция binary_search, и результаты сохраняются в списке r.

Запись результатов в файл output.txt:

Результаты поиска (индексы найденных элементов) записываются в файл, разделенные пробелами.

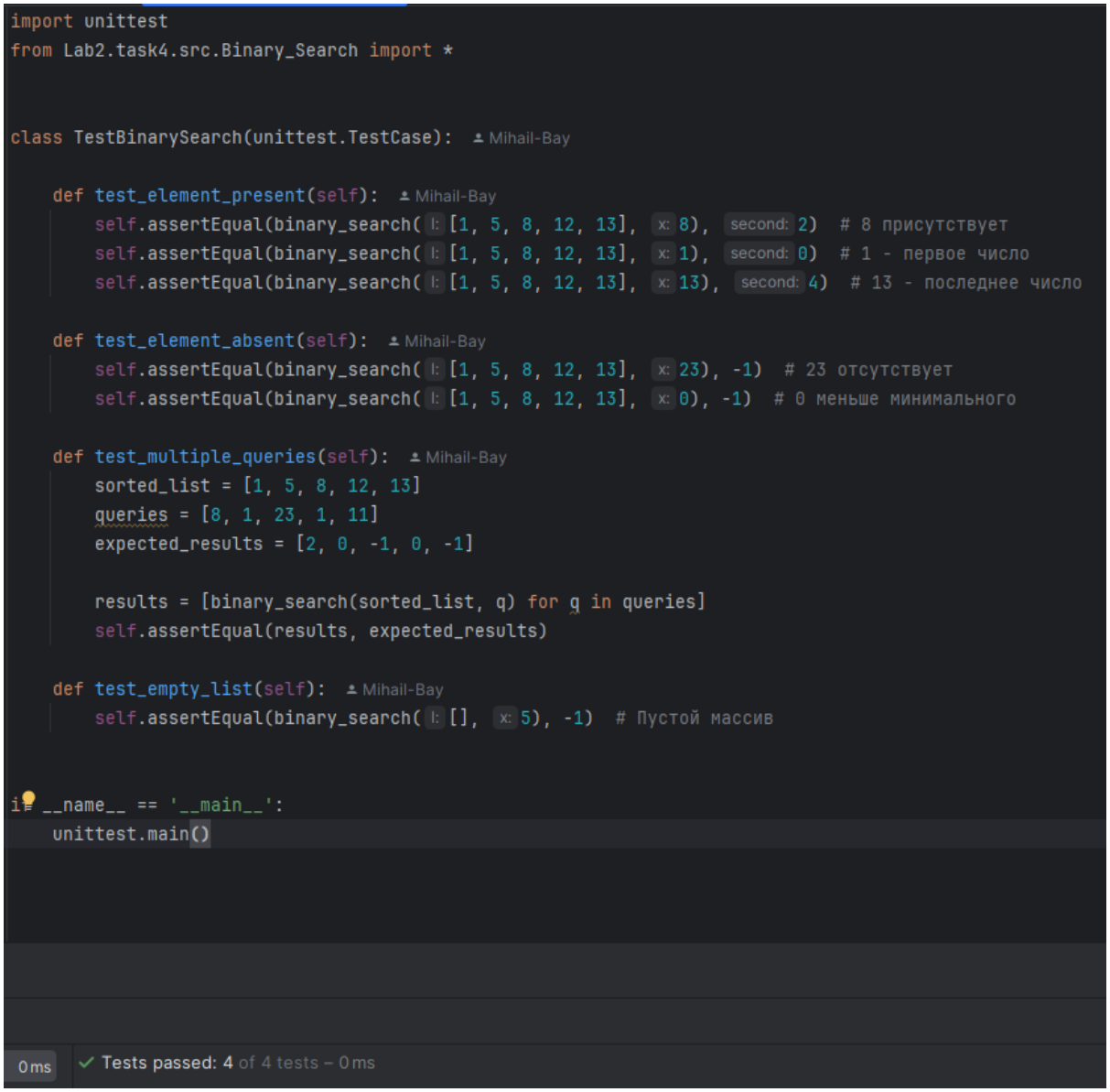
Измерение времени выполнения:

Программа вычисляет общее время выполнения и выводит его в консоль.

Отслеживание использования памяти:

Программа получает текущее и пиковое использование памяти и выводит эти значения в мегабайтах.

Остановка отслеживания памяти:



Вывод по задаче:

Данная программа эффективно использует бинарный поиск для быстрого нахождения элементов в отсортированном массиве. Она также включает в себя полезные инструменты для анализа производительности и использования памяти, что делает её подходящей как для учебного процесса, так и для более серьезных задач в производственном окружении.

Задача №5. Представитель большинства

Текст задачи:

Очевидно, время выполнения этого алгоритма квадратично. Ваша цель - использовать метод "Разделяй и властвуй"для разработки алгоритма проверки, содержится ли во входной последовательности элемент, который встречается больше половины раз, за время $O(n \log n)$.

Код программы:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения


def majority_element(l):
    # Определяем функцию для нахождения элемента, который появляется более чем n/2 раз

    def find_c(l, left, right):
        # Вложенная функция для рекурсивного поиска кандидата на элемент большинства
        if left == right:
            # Если диапазон из одного элемента, возвращаем этот элемент
            return l[left]

        mid = (left + right) // 2  # Находим середину текущего диапазона

        # Рекурсивно ищем элементы большинства в левой и правой частях
        left_c = find_c(l, left, mid)
        right_c = find_c(l, mid + 1, right)

        if left_c == right_c:
            # Если оба элемента одинаковы, возвращаем их
            return left_c

        # Считаем количество вхождений каждого из кандидатов в текущем диапазоне
        count_left = l[left:right + 1].count(left_c)
        count_right = l[left:right + 1].count(right_c)

        # Возвращаем того кандидата, который встречается чаще
        return left_c if count_left > count_right else right_c

    def count_occurrences(l, c):
        # Вложенная функция для подсчета вхождений элемента c в массиве l
        return l.count(c)

    n = len(l)  # Сохраняем длину массива
    c = find_c(l, 0, n - 1)  # Ищем кандидата на элемент большинства

    # Проверяем, является ли найденный элемент элементом большинства
    if count_occurrences(l, c) > n // 2:
        return 1  # Если найденный элемент является элементом большинства, возвращаем 1
    else:
        return 0  # Если нет, возвращаем 0


if __name__ == '__main__':
    # Чтение входного файла
    with open('input.txt', 'r') as f:
```

```
    n = int(f.readline().strip())  # Читаем размер массива
    l = list(map(int, f.readline().strip().split()))  # Читаем массив и преобразуем строки в целые числа

# Вызываем функцию и записываем результат в выходной файл
r = majority_element(l)

with open('output.txt', 'w') as f_out:
    f_out.write(str(r))  # Записываем результат работы функции в файл

# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Описание работы программы:

Импорт библиотек:

time: используется для измерения времени выполнения программы.

tracemalloc: модуль, который отслеживает использование памяти программой.

Отслеживание памяти:

В начале работы программы запускается отслеживание использования памяти с помощью tracemalloc.start().

Запоминается время запуска для дальнейшего анализа времени выполнения.

Функция majority_element(l):

Основная функция, принимающая список l целых чисел. Она ищет элемент, который появляется более чем

$n/2$раз (где n — длина списка).

Вложенная функция find_c(l, left, right):

Рекурсивно ищет кандидата на элемент большинства в заданном диапазоне.

Если диапазон сужен до одного элемента, возвращает этот элемент.

Разделяет массив на две половинки и рекурсивно ищет элементы большинства в обеих частях.

Сравнивает найденные кандидаты и считает их вхождения в текущем диапазоне для выбора элемента, который встречается чаще.

Вложенная функция count_occurrences(l, c):

Подсчитывает количество вхождений элемента c в массиве l с помощью метода count().

Логика работы:

Сначала определяется длина массива n.

Затем вызывается функция find_c для нахождения кандидата на элемент большинства.

После нахождения кандидата, с помощью count_occurrences проверяется, превышает ли количество его вхождений половину длины массива.

Если да, функция возвращает 1 (элемент большинства найден), если нет — возвращает 0.

Чтение данных:

Программа начинает с чтения входных данных из файла input.txt, где первый элемент — размер массива, а второй — сам массив целых чисел.

Запись результата:

Результат (1 или 0) записывается в файл output.txt.

Измерение времени выполнения и использования памяти:

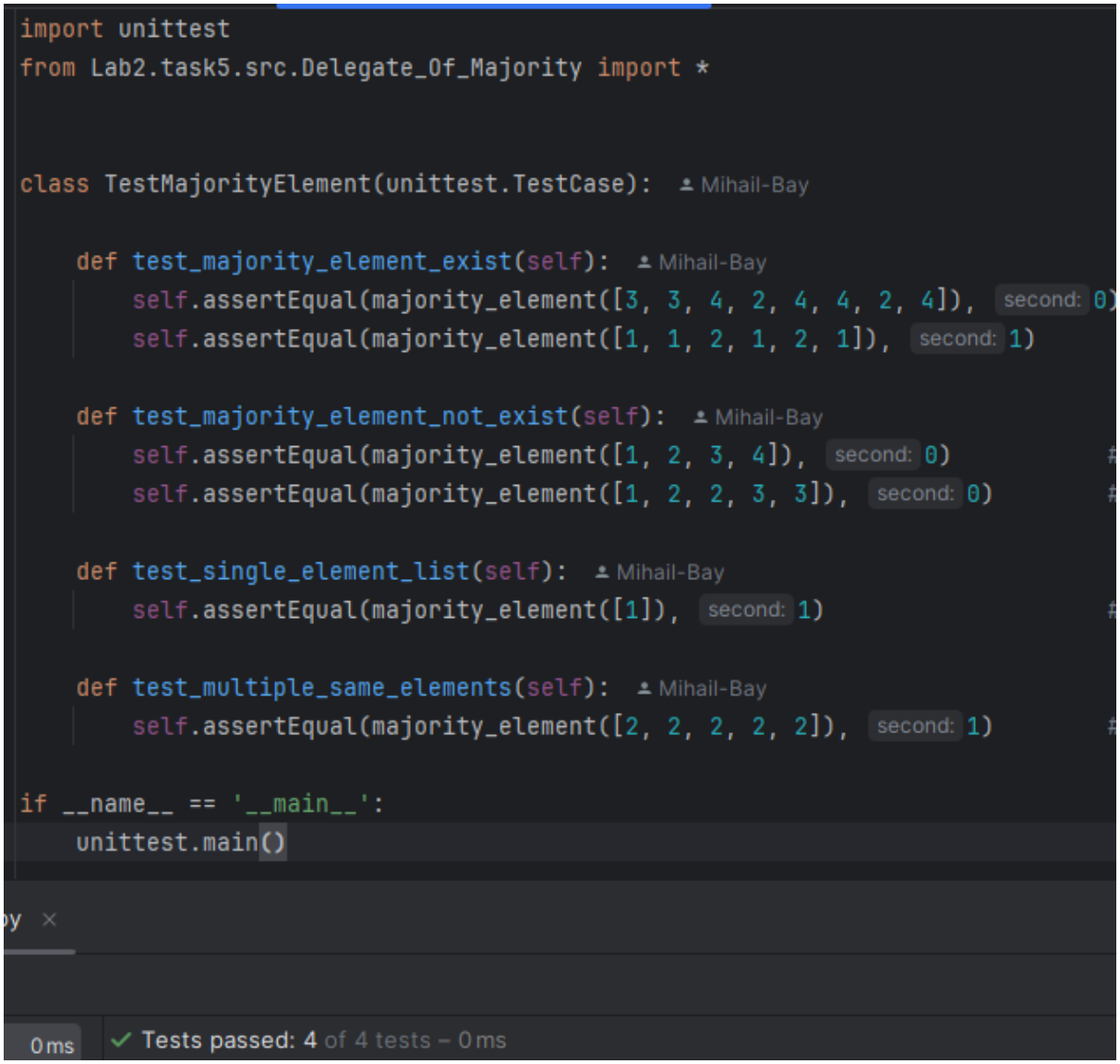
После выполнения алгоритма вычисляется время, затраченное на выполнение всех операций, и отображается текущее и пиковое использование памяти.

Отслеживание памяти останавливается с помощью tracemalloc.stop().

Запуск программы:

Программа начинает выполнение, если она запускается как основной модуль (if __name__ == '__main__':).

Тесты:



Вывод по задаче:

Программа эффективно находит элемент большинства за логарифмическое время благодаря использованию рекурсии и принципам "разделяй и властвуй".

Вывод

В данной лабораторной работе были изучены новые алгоритмы сортировок и их вариации.