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 3

Выполнил:
Бай М.О.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Оглавление

Задачи по варианту

Задача №1. Улучшение Quick sort

Текст задачи:

Основное задание. Цель задачи - переделать данную реализацию рандомизированного алгоритма быстрой сортировки, чтобы она работала быстро даже с последовательностями, содержащими много одинаковых элементов. Чтобы заставить алгоритм быстрой сортировки эффективно обрабатывать последовательности с несколькими уникальными элементами, нужно заменить двухстороннее разделение на трехстороннее (смотри в Лекции 3 слайд 17). То есть ваша новая процедура разделения должна разбить массив на три части:

- $A[k] < x$ для всех $\ell + 1 \leq k \leq m_1 - 1$
- $A[k] = x$ для всех $m_1 \leq k \leq m_2$
- $A[k] > x$ для всех $m_2 + 1 \leq k \leq r$

Код программы:

```
import random
import time
import tracemalloc
import sys
import os

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))
from Lab3.utils import read_input, write_output


tracemalloc.start()

t_start = time.perf_counter()

def partition3(l, low, high):
    pivot_index = random.randint(low, high)
    l[pivot_index], l[high] = l[high], l[pivot_index]  # Ставим опорный элемент в конец
    pivot = l[high]  # Опорный элемент
    i = low  # Индекс для меньших элементов
    j = high - 1  # Индекс для больших элементов

    while i <= j:
        if l[i] < pivot:  # Если текущий элемент меньше опорного
            i += 1
        elif l[i] > pivot:  # Если текущий элемент больше опорного
            l[i], l[j] = l[j], l[i]  # Меняем местами
            j -= 1
        else:  # Если элемент равен опорному
            i += 1

    # Перемещаем все равные элементы к опорному элементу
    l[i], l[high] = l[high], l[i]
    return i, j  # Возвращаем границы равных элементов
```

```
def randomized_quick_sort(l, low, high):
    if low < high:
        m1, m2 = partition3(l, low, high)  # Используем трёхстороннее разбиение
        randomized_quick_sort(l, low, m1 - 1)  # Сортируем подмассив меньше опорного
        randomized_quick_sort(l, m2 + 1, high)  # Сортируем подмассив больше опорного


if __name__ == '__main__':
    n, l = read_input(task=1)

    n = int(n.strip())
    l = list(map(int, l.strip().split()))

    randomized_quick_sort(l, 0, n - 1)

    sl = (' '.join(map(str, l)))

    write_output(1, 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:

Инициализация и импорт модулей: Программа начинает с импорта необходимых модулей, включая random для случайного выбора опорного элемента, time и tracemalloc для измерения времени выполнения и использования памяти.

Функция partition3: Это функция трёхстороннего разбиения, которая принимает на вход список l и границы low и high. Она случайным образом выбирает опорный элемент из списка, перемещает его в конец, а затем перемещает элементы вокруг него так, что в начале оказываются меньшие элементы, в середине — равные, и в конце — большие. Она возвращает индекс начала группы равных элементов и индекс конца этой группы (на самом деле это последний элемент больше опорного).

Функция randomized_quick_sort: Эта функция рекурсивно применяет алгоритм быстрой сортировки. Она использует partition3 для разделения массива и затем сортирует подмассивы, которые меньше или больше опорных элементов.

Главный блок if __name__ == '__main__'::

Считывает входные данные с помощью read_input.

Запускает сортировку.

Записывает отсортированный массив обратно с помощью write_output.
Выводит время выполнения и использование памяти.
Измерение производительности: Программа отслеживает время выполнения и использование памяти с помощью time.perf_counter() и tracemalloc соответственно.

Тесты:

```
class TestQuickSort(unittest.TestCase):

    def test_sorted_array(self):
        l = [1, 2, 3, 4, 5]
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, sorted(l))

    def test_reverse_sorted_array(self):
        l = [5, 4, 3, 2, 1]
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, sorted(l))

    def test_random_array(self):
        l = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, sorted(l))

    def test_array_with_duplicates(self):
        l = [5, 5, 5, 5, 5]
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, sorted(l))

    def test_empty_array(self):
        l = []
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, second: [])

    def test_single_element_array(self):
        l = [1]
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, l)

    def test_large_array(self):
        seed(0)  # Устанавливаем seed для воспроизводимости
        l = [random.randint( a: 1, b: 1000) for _ in range(1000)]
        sorted_l = l[:]
        randomized_quick_sort(sorted_l, low: 0, len(sorted_l) - 1)
        self.assertEqual(sorted_l, sorted(l))

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

```
30 ms  ✓ Tests passed: 7 of 7 tests – 30 ms
```

Вывод по задаче:

Эта программа реализует алгоритм быстрой сортировки с произвольным выбором опорного элемента и трёхсторонним разделением.

Задача №2. Анти-quick sort

Текст задачи:

Хотя `QuickSort` является очень быстрой сортировкой в среднем, существуют тесты, на которых она работает очень долго. Оценивать время работы алгоритма будем числом сравнений с элементами массива (то есть, суммарным числом сравнений в первом и втором while). Требуется написать программу, генерирующую тест, на котором быстрая сортировка сделает наибольшее число таких сравнений.
Задача на acmp.

Код программы:

```
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))
from Lab3.utils import read_input, write_output


def generate_max_comparisons_test(n):
    # Создаем отсортированный массив
    test_array = list(range(1, n + 1))
    return test_array

if __name__ == '__main__':
    n = read_input(task=2)

    n = int(n[0])

    test_array = generate_max_comparisons_test(n)
    test_array = (' '.join(map(str, test_array)))

    write_output(2, test_array)
```

Описание работы программы:

Импорт модулей и путей:

sys и os используются для управления путями и добавления пользовательских модулей в системный путь. Это позволяет использовать функции read_input и write_output из каталога проекта.

Функция generate_max_comparisons_test(n):

Функция создает отсортированный массив чисел от 1 до n.

Это наиболее худший случай для наивной реализации QuickSort, где выбор опорного элемента не предпочтителен для уже отсортированных массивов без использования рандомизации или медианы трёх.

Главный блок if __name__ == '__main__'::

С помощью функции read_input, получаем число n
из входных данных, указывающее количество элементов в массиве.

generate_max_comparisons_test(n) вызывается для создания отсортированного массива.

Массив преобразуется в строку и передается в функцию write_output, которая записывает результат в файл или другой выходной канал.

Тесты:

```
import unittest
from Lab3.task2.src.AntiQuickSort import *

class TestGenerateMaxComparisonsTest(unittest.TestCase):
    def test_sorted_array_generation(self):
        # Test with small n
        result = generate_max_comparisons_test(5)
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

        # Test with zero n
        result = generate_max_comparisons_test(0)
        self.assertEqual(result, second: [])

        # Test with large n
        n = 100
        result = generate_max_comparisons_test(n)
        self.assertEqual(result, list(range(1, n + 1)))

    def test_output_format(self):
        # Test the conversion to string format for output
        n = 5
        test_array = generate_max_comparisons_test(n)
        test_array_str = ' '.join(map(str, test_array))
        self.assertEqual(test_array_str, second: '1 2 3 4 5')


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

✓ Tests passed: 2 of 2 tests – 0 ms

Вывод по задаче:

Данная программа генерирует тестовые данные для алгоритмов сортировки, создавая отсортированный массив чисел от 1 до n. Основная задача программы — это создать сценарий, при котором алгоритмы

сортировки затратят наибольшее количество сравнений, если они следуют наивной реализации без специальных оптимизаций (например, QuickSort с плохим выбором опорного элемента).

Задача №3. Сортировка пугалом

Текст задачи:

«Сортировка пугалом» — это давно забытая народная потешка. Участнику под верхнюю одежду продевают деревянную палку, так что у него оказываются растопырены руки, как у огородного пугала. Перед ним ставятся n матрёшек в ряд. Из-за палки единственное, что он может сделать — это взять в руки две матрешки на расстоянии k друг от друга (то есть i-ую и $i + k$-ую), развернуться и поставить их обратно в ряд, таким образом поменяв их местами.

Задача участника — расположить матрёшки по неубыванию размера. Может ли он это сделать?

Код программы:

```
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))
from Lab3.utils import read_input, write_output


def can_sort_matriochkas(n, k, sizes):
    # Создаем k списков для каждого сегмента
    segments = [[] for _ in range(k)]

    # Разбиваем массив на подмножества
    for i in range(n):
        segments[i % k].append(sizes[i])

    # Сортируем каждое подмножество
    for i in range(k):
        segments[i].sort()

    # Сборка отсортированного массива
    sorted_sizes = []
    for i in range(n):
        sorted_sizes.append(segments[i % k][i // k])

    # Проверяем, совпадает ли с отсортированным массивом
    if sorted_sizes == sorted(sizes):
        return "YES"
    else:
        return "NO"


if __name__ == "__main__":

    n, sizes = read_input(task=3)

    n, k = map(int, n.strip().split())
    sizes = list(map(int, sizes.strip().split()))

    result = can_sort_matriochkas(n, k, sizes)
```

```
write_output(3, str(result))
```

Описание работы программы:

Инициализация подмножеств:

Создаем список подмножеств (segments) размером k. Каждое подмножество будет представлять собой одну "часть" массива, которую можно сортировать отдельно.

Разделение массива на подмножества:

Проходим по всем элементам входного массива sizes и распределяем элементы по подмножествам в зависимости от индекса, используя операцию деления по модулю k. Это означает, что элемент sizes[i] будет помещён в segments[i % k].

Сортировка подмножеств:

Каждый список в segments сортируется независимо.

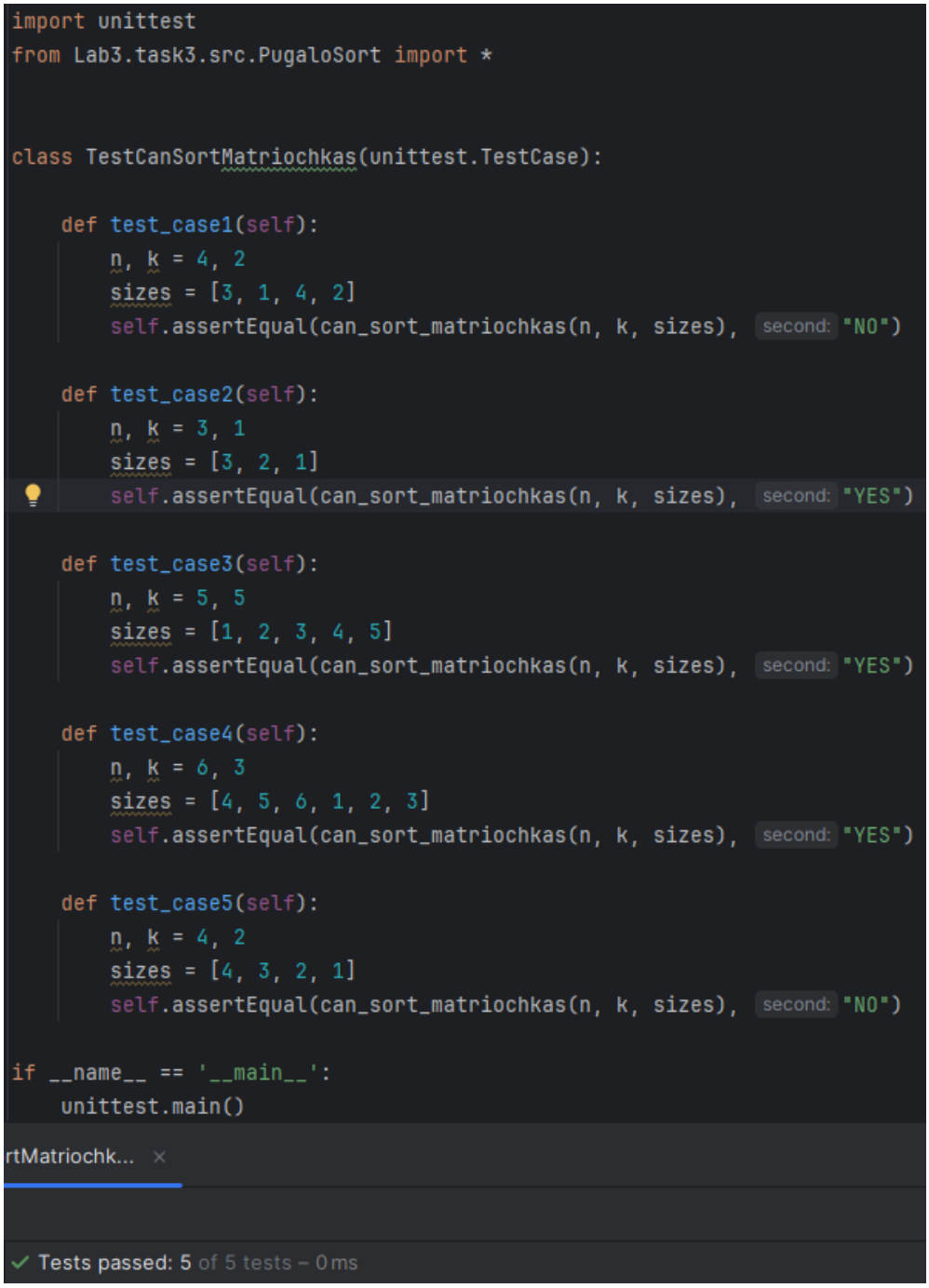
Сборка итогового массива:

Создаем новый массив sorted_sizes, который собирает элементы из каждого подмножества в определённом порядке. Это происходит в цикле, где на каждой итерации из сегментов берется элемент в соответствии с его индексом и формируется конечный массив.

Проверка упорядоченности:

Проверяем, совпадает ли собранный массив sorted_sizes с массивом, отсортированным полностью обычными средствами. Если массивы совпадают, это значит, что возможно отсортировать массив с использованием описанного метода, в противном случае - нет.

Тесты:



```
import unittest
from Lab3.task3.src.PugaloSort import *


class TestCanSortMatriochkas(unittest.TestCase):

    def test_case1(self):
        n, k = 4, 2
        sizes = [3, 1, 4, 2]
        self.assertEqual(can_sort_matriochkas(n, k, sizes), second: "NO")

    def test_case2(self):
        n, k = 3, 1
        sizes = [3, 2, 1]
        self.assertEqual(can_sort_matriochkas(n, k, sizes), second: "YES")

    def test_case3(self):
        n, k = 5, 5
        sizes = [1, 2, 3, 4, 5]
        self.assertEqual(can_sort_matriochkas(n, k, sizes), second: "YES")

    def test_case4(self):
        n, k = 6, 3
        sizes = [4, 5, 6, 1, 2, 3]
        self.assertEqual(can_sort_matriochkas(n, k, sizes), second: "YES")

    def test_case5(self):
        n, k = 4, 2
        sizes = [4, 3, 2, 1]
        self.assertEqual(can_sort_matriochkas(n, k, sizes), second: "NO")

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

rtMatriochk... ×

✓ Tests passed: 5 of 5 tests – 0 ms

Вывод по задаче:

Эта программа определяет, возможно ли отсортировать массив, представленный, используя алгоритм, который разбивает массив на k подмножеств. После чего каждое из подмножеств сортируется, и из отсортированных подмножеств составляется итоговый массив. Программа проверяет, совпадает ли этот итоговый массив с полностью отсортированным массивом.

Задача №4. Точки и отрезки

Текст задачи:

- *Цель.* Вам дается набор точек и набор отрезков. Цель состоит в том, чтобы вычислить для каждой точки количество отрезков, содержащих эту точку.

Код программы:

```
import time
import tracemalloc
import sys
import os

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))
from Lab3.utils import read_input, write_output

# Начинаем отслеживание использования памяти
tracemalloc.start()

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

def count_segments_containing_points(s, p, segments, points):
    # Список событий
    events = []

    # Добавляем сегменты
    for i, (a, b) in enumerate(segments):
        events.append((a, 'L', i))  # L - левая граница отрезка
        events.append((b, 'R', i))  # R - правая граница отрезка

    # Добавляем точки
    for i, x in enumerate(points):
        events.append((x, 'P', i))  # P - точка

    # Сортируем события
    events.sort(key=lambda event: (event[0], event[1]))

    # Массив для результатов
    results = [0] * p
    active_segments = 0

    # Обработка событий
    for event in events:
        if event[1] == 'L':
            # Начало отрезка
            active_segments += 1
        elif event[1] == 'R':
            # Конец отрезка
            active_segments -= 1
        elif event[1] == 'P':
            # Точка
            point_index = event[2]
            results[point_index] = active_segments

    return results


# Чтение входных данных
if __name__ == "__main__":
    l = read_input(task=4)
    s, p = map(int, l[0].split())
    segments = [tuple(map(int, l[_ + 1].split())) for _ in range(s)]
```

```
points = list(map(int, l[s+1].split()))

res = count_segments_containing_points(s, p, segments, points)

res = (" ".join(map(str, res)) + "\n")
write_output(4, res)
```

Описание работы программы:

Создание событий:

Для каждого отрезка (a,b):

Создайте событие для начала отрезка $(a,'L')$.

Создайте событие для конца отрезка $(b,'R')$.

Для каждой точки x:

Создайте событие для точки $(x,'P')$.

Сортировка событий:

События сортируются, прежде всего, по координате их возникновения. Если координаты совпадают, порядок определяется следующим образом:

Сначала ‘L’ (начало отрезка),

затем ‘P’ (точка),

затем ‘R’ (конец отрезка).

Обработка событий:

Инициализируется переменная active_segments, чтобы отслеживать количество пересекающихся сегментов в текущий момент.

Инициализируется массив results для хранения количества накладывающихся отрезков для каждой точки.

При обработке событий:

Если это событие 'L', то это означает начало нового отрезка: увеличиваем active_segments.

Если это событие 'R', то это означает конец отрезка: уменьшаем active_segments.

Если это событие 'P', то это проверяемая точка: значение active_segments записывается в results в индекс, соответствующий точке.

Вывод результата:

Результат сохраняется в виде строки, где каждое число показывает количество отрезков, покрывающих соответствующую точку.

Тесты:

```python
import unittest
from Lab3.task4.src.SegmentsAndPonts import *

class TestCountSegments(unittest.TestCase):
    def test_basic_case(self):
        s = 2
        p = 3
        segments = [(0, 5), (7, 10)]
        points = [1, 6, 11]
        expected = [1, 0, 0]
        result = count_segments_containing_points(s, p, segments, points)
        self.assertEqual(result, expected)

    def test_all_points_covered(self):
        s = 1
        p = 3
        segments = [(0, 5)]
        points = [1, 2, 3]
        expected = [1, 1, 1]
        result = count_segments_containing_points(s, p, segments, points)
        self.assertEqual(result, expected)

    def test_no_segments(self):
        s = 0
        p = 3
        segments = []
        points = [1, 2, 3]
        expected = [0, 0, 0]
        result = count_segments_containing_points(s, p, segments, points)
        self.assertEqual(result, expected)

    def test_no_points(self):
        s = 2
        p = 0
        segments = [(0, 5), (7, 10)]
        points = []
        expected = []
        result = count_segments_containing_points(s, p, segments, points)
        self.assertEqual(result, expected)
```

```
rtMatriochk... ×

✓ Tests passed: 5 of 5 tests – 0 ms
```

Вывод по задаче:

Эта программа решает задачу: для каждой заданной точки определить, сколько отрезков её покрывают.

Задача №5. Представитель большинства

Текст задачи:

Для заданного массива целых чисел citations, где каждое из этих чисел - число цитирований i-ой статьи ученого-исследователя, посчитайте индекс Хирша этого ученого.

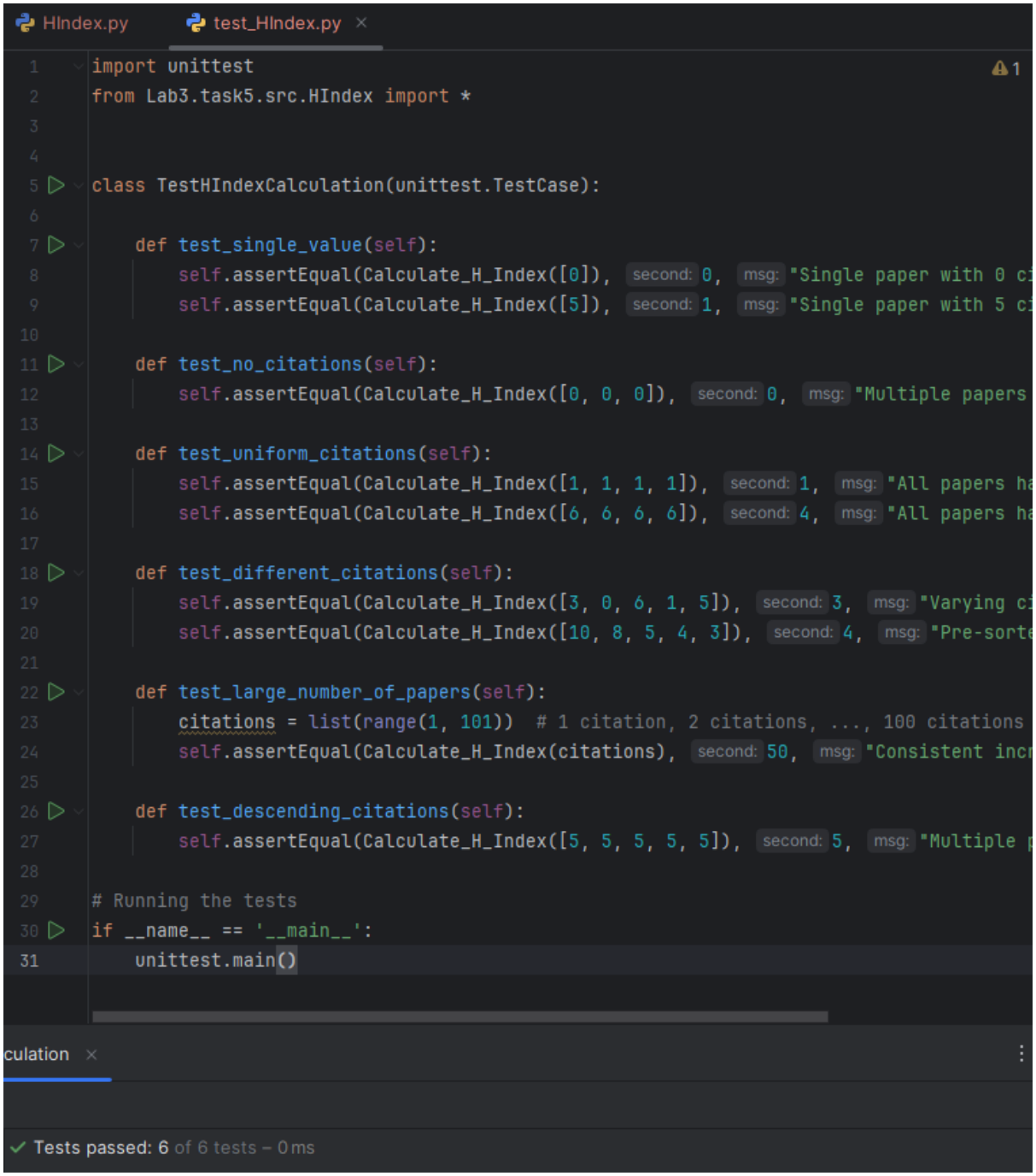
По определению Индекса Хирша на Википедии: Учёный имеет индекс h, если h из его/её N_p статей цитируются как минимум h раз каждая, в то время как оставшиеся $(N_p - h)$ статей цитируются не более чем h раз каждая. Иными словами, учёный с индексом h опубликовал как минимум h статей, на каждую из которых сослались как минимум h раз.

Если существует несколько возможных значений h, в качестве h-индекса принимается максимальное из них.

Код программы:

```
import sys
import os

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))
from Lab3.utils import read_input, write_output


def Calculate_H_Index(citations):
    citations.sort(reverse=True)

    l_counter = 0

    for i in range(len(citations)):
        if citations[i] >= i + 1:
            l_counter = i + 1
        else:
            break

    return l_counter


if __name__ == '__main__':
    lines = read_input(task=5)  # Получаем список строк
    combined_str = ' '.join(lines)  # Объединяем строки в одну строку

    # Преобразуем объединенную строку в список чисел
    citations = list(map(int, combined_str.split()))

    res = Calculate_H_Index(citations)
    write_output(5, res )  # Указываем task при записи
```

Тесты:



Вывод по задаче: Программа предназначена для вычисления индекса Хирша (h-индекс) на основе списка цитат учёного или исследователя. Индекс Хирша представляет собой показатель, измеряющий как продуктивность исследователя, так и влияние их публикаций на научное сообщество. Значение h-индекса определяется как максимальное число *h*, такое, что у исследователя есть *h* статей, каждая из которых цитируется как минимум *h* раз.

Вывод

В данной работе мы изучили новые алгоритмы сортировок а так же их вариации.