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 8

Выполнила:
Иванова А. Г.
К3141

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задание №1. Улучшение Quick sort

2. **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 $m_2 + 1 \leq k \leq r$

   - Формат входного и выходного файла аналогичен п.1.
   - Аналогично п.1 этого задания сравните `Randomized-QuickSort` +с `Partition` и ее с `Partition3` на сетах случайных данных, в которых содержатся всего несколько уникальных элементов при $n = 10^3, 10^4, 10^5$. Что быстрее, `Randomized-QuickSort` +с `Partition3` или `Merge-Sort`?
   - Пример:

input.txt	output.txt
5 2 3 9 2 2	2 2 2 3 9

Решение:

```
import sys
import os
import psutil
import time
import random
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'../..')))
from utils import *

start_time = time.perf_counter()


def partition3(A, l, r):
    x = A[l]
    m1 = l
    m2 = r
```

```
    i = l
    while i <= m2:
        if A[i] < x:
            A[m1], A[i] = A[i], A[m1]
            m1 += 1
            i += 1
        elif A[i] > x:
            A[i], A[m2] = A[m2], A[i]
            m2 -= 1
        else:
            i += 1
    return m1, m2


def randomized_quick_sort_p3(A, l, r):
    if l < r:
        k = random.randint(l, r)
        A[l], A[k] = A[k], A[l]
        m1, m2 = partition3(A, l, r)
        randomized_quick_sort_p3(A, l, m1-1)
        randomized_quick_sort_p3(A, m2+1, r)


if __name__ == '__main__':
    _, massive = read_file(task=1)
    array = list(map(int, massive.split()))
    randomized_quick_sort_p3(array, 0, len(array) - 1)
    write_output(1, ' '.join(map(str, array)))
    print(f'Время: {(time.perf_counter() - start_time):.6f} секунд')
    print(f'Память: {psutil.Process().memory_info().rss / 1024 ** 2} Мбайт')
```

1. Функция partition3: Реализует модификацию быстрой сортировки с тремя секциями. Она делит массив на три части: элементы меньше опорного (x), равные ему и больше него. Возвращает индексы начала и конца средней секции.
2. Функция randomized_quick_sort_p3: Это рекурсивная функция, которая выполняет быструю сортировку массива. Она выбирает случайный элемент как опорный, вызывает функцию partition3, а затем рекурсивно сортирует левую и правую части массива.
3. Основная часть программы: Читает входной файл через функцию read_file, преобразует его содержимое в список целых чисел, сортирует этот список с помощью randomized_quick_sort_p3, записывает отсортированный результат в выходной файл с помощью функции write_output. Также измеряет время выполнения программы и использование памяти.

Результат работы программы:

Входные данные:

```
5
2 3 9 2 2
```

Выходные данные:

```
2 2 2 3 9
```

Время выполнения и количество затраченной памяти:

Время: 0.010911 секунд

Память: 15.01953125 Мбайт

Тесты:

```
import unittest

from lab3.task1.src.randomized_quick_sort_p3 import randomized_quick_sort_p3


class TestRandomizedQuickSortP3(unittest.TestCase):

    def test_should_sort_example_array(self):
        # given
        array = [2, 3, 9, 2, 2]
        # when
        randomized_quick_sort_p3(array, 0, len(array) - 1)
        # then
        self.assertEqual(array, [2, 2, 2, 3, 9])

    def test_should_sort_sorted_array(self):
        # given
        array = [1, 2, 3, 4, 5, 6]
        # when
        randomized_quick_sort_p3(array, 0, len(array) - 1)
        # then
        self.assertEqual(array, [1, 2, 3, 4, 5, 6])

    def test_should_sort_reverse_sorted_array(self):
        # given
        array = [6, 5, 4, 3, 2, 1]
        # when
        randomized_quick_sort_p3(array, 0, len(array) - 1)
        # then
        self.assertEqual(array, [1, 2, 3, 4, 5, 6])

    def test_should_sort_single_element_array(self):
        # given
        array = [0]
        # when
        randomized_quick_sort_p3(array, 0, len(array) - 1)
```

```
        # then
        self.assertEqual(array, [0])

    def test_should_sort_empty_array(self):
        # given
        array = []
        # when
        randomized_quick_sort_p3(array, 0, len(array) - 1)
        # then
        self.assertEqual(array, [])

    def test_should_sort_large_numbers_array(self):
        # given
        array = [1000000000, 999999999, 999999998]
        # when
        randomized_quick_sort_p3(array, 0, len(array) - 1)
        # then
        self.assertEqual(array, [999999998, 999999999, 1000000000])


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм более эффективной сортировки: рандомизированной с трехсторонним разделением.

Задание №6. Сортировка целых чисел

6 задача. Сортировка целых чисел

В этой задаче нужно будет отсортировать много неотрицательных целых чисел.

Вам даны два массива, A и B, содержащие соответственно n и m элементов. Числа, которые нужно будет отсортировать, имеют вид $A_i \cdot B_j$, где $1 \leq i \leq n$ и $1 \leq j \leq m$. Иными словами, каждый элемент первого массива нужно умножить на каждый элемент второго массива.

Пусть из этих чисел получится отсортированная последовательность C длиной $n \cdot m$. Выведите сумму каждого десятого элемента этой последовательности (то есть, $C_1 + C_{11} + C_{21} + ...$).

- **Формат входного файла (input.txt).** В первой строке содержатся числа n и m ($1 \leq n, m \leq 6000$) – размеры массивов. Во второй строке содержится n чисел – элементы массива A. Аналогично, в третьей строке содержится m чисел — элементы массива B. Элементы массива неотрицательны и не превосходят 40000.
- **Формат выходного файла (output.txt).** Выведите одно число — сумму каждого десятого элемента последовательности, полученной сортировкой попарных произведений элементов массивов A и B.
- Ограничение по времени. 2 сек.
- **Ограничение по времени распространяется на сортировку, без учета времени на перемножение. Подумайте, какая сортировка будет эффективнее, сравните на практике.**
- *Однако бытует мнение на OpenEdu, неделя 3, задача 2, что эту задачу можно решить на Python и уложиться в 2 секунды, включая в общее время перемножение двух массивов.*
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt
4 4 7 1 4 9 2 7 8 11	51

Решение:

```
import sys
import os
import psutil
import time
```

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

start_time = time.perf_counter()


def quick_sort(A, l, r):
    if l < r:
        m = partition(A, l, r)
        quick_sort(A, l, m-1)
        quick_sort(A, m+1, r)


def partition(A, l, r):
    x = A[l]
    j = l
    for i in range(l+1, r+1):
        if A[i] <= x:
            j += 1
            A[j], A[i] = A[i], A[j]
    A[l], A[j] = A[j], A[l]
    return j


def sum_of_tenths(A, B):
    C = []
    for b in B:
        for a in A:
            C.append(a * b)
    quick_sort(C, 0, len(C)-1)
    sum_of_tenths = sum(C[i] for i in range(0, len(C), 10))
    return sum_of_tenths


if __name__ == '__main__':
    _, A, B = read_file(task=6)
    A = list(map(int, A.split()))
    B = list(map(int, B.split()))
    result = sum_of_tenths(A, B)
    write_output(6, str(result))
    print(f'Время: {(time.perf_counter() - start_time):.6f} секунд')
    print(f'Память: {psutil.Process().memory_info().rss / 1024 ** 2} Мбайт')
```

1. Функция quick_sort: Реализована стандартная версия быстрой сортировки. Она разбивает массив на две части относительно опорного элемента и рекурсивно сортирует каждую из них.
2. Функция partition: Помогающая функция для quick_sort, которая делит массив на части относительно выбранного опорного элемента.

3. Функция sum_of_tenths: Главная логическая функция программы. Она делает следующее:
    1) Формирует новый список C, содержащий все возможные произведения элементов списков A и B.
    2) Сортирует полученный список C с помощью quick_sort.
    3) Вычисляет сумму каждого десятого элемента отсортированного списка C.
4. Основная часть программы: Читает данные из файла, разбивая их на списки A и B. Вызывает функцию sum_of_tenths, сохраняет результат в выходной файл. Также измеряются время выполнения программы и объем используемой памяти.

Результат работы программы:
Входные данные:

```
4 4
7 1 4 9
2 7 8 11
```

Выходные данные:

```
51
```

Время выполнения и количество затраченной памяти:
Время: 0.006404 секунд
Память: 14.65234375 Мбайт

Тесты:

```
import unittest

from lab3.task6.src.sum_of_tenths import sum_of_tenths


class TestSumOfTenths(unittest.TestCase):

    def test_should_count_sum_of_tenths_example_array(self):
        # given
        A = [7, 1, 4, 9]
        B = [2, 7, 8, 11]
        # when
        result = sum_of_tenths(A, B)
        # then
        self.assertEqual(result, 51)

    def test_should_count_sum_of_tenths_sorted_array(self):
        # given
        A = [1, 2, 3, 4, 5]
```

```
        B = [1, 1, 1, 1, 1]
        # when
        result = sum_of_tenths(A, B)
        # then
        self.assertEqual(result, 9)

    def test_should_count_sum_of_tenths_reverse_sorted_array(self):
        # given
        A = [5, 4, 3, 2, 1]
        B = [1, 2, 3, 4, 5]
        # when
        result = sum_of_tenths(A, B)
        # then
        self.assertEqual(result, 22)

    def test_should_count_sum_of_tenths_empty_array(self):
        # given
        A = []
        B = []
        # when
        result = sum_of_tenths(A, B)
        # then
        self.assertEqual(result, 0)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм сортировки массива чисел вида A[i] * B[j], где 1 <= i <= n и ш <= j <= m и вывода суммы каждого десятого элемента полученной последовательности.

Задание №8. K ближайших точек к началу координат

8 задача. K ближайших точек к началу координат

В этой задаче, ваша цель - найти K ближайших точек к началу координат среди данных n точек.

- Цель. Заданы n точек на поверхности, найти K точек, которые находятся ближе к началу координат (0, 0), т.е. имеют наименьшее расстояние до начала координат. Напомним, что расстояние между двумя точками (x_1, y_1) и (x_2, y_2) равно $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

- **Формат ввода или входного файла (input.txt).** Первая строка содержит n - общее количество точек на плоскости и через пробел K - количество ближайший точек к началу координат, которые надо найти. Каждая следующая из n строк содержит 2 целых числа x_i, y_i, определяющие точку (x_i, y_i). Ограничения: $1 \le n \le 10^5$; $-10^9 \le x_i, y_i \le 10^9$ - целые числа.
- **Формат выхода или выходного файла (output.txt).** Выведите K ближайших точек к началу координат в строчку в квадратных скобках через запятую. Ответ вывести в порядке возрастания расстояния до начала координат. Если оно равно, порядок произвольный.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 256 мб.
- Пример 1.

input.txt	output.txt
2 **1** 1 3 -2 2	[-2,2]

Решение:

```
import sys
import os
import psutil
import time
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'../..')))
from utils import *

start_time = time.perf_counter()


def partition(A, l, r):
   x = A[l][0]
   j = l
   for i in range(l+1, r+1):
```

```
        if A[i][0] < x:
            j += 1
            A[j], A[i] = A[i], A[j]
    A[l], A[j] = A[j], A[l]
    return j


def quick_sort(A, l, r):
    if l < r:
        j = partition(A, l, r)
        quick_sort(A, l, j-1)
        quick_sort(A, j+1, r)


def closest_points(array, k):
    points = []
    for cord in array:
        x, y = cord
        distance = int((x**2 + y**2)**0.5)
        points.append([distance, [x, y]])
    quick_sort(points, 0, len(points)-1)
    return [elem[1] for elem in points[:k]]


if __name__ == '__main__':
    data = read_file(task=8)
    n, k = list(map(int, data[0].split()))
    array = [list(map(int, cord.split())) for cord in data[1:]]
    result = closest_points(array, k)
    write_output(8, result)
    print(f'Время: {(time.perf_counter() - start_time):.6f} секунд')
    print(f'Память: {psutil.Process().memory_info().rss / 1024 ** 2} Мбайт')
```

1. Функция partition: Помогающая функция для quick_sort, которая делит массив на части относительно выбранного опорного элемента.
2. Функция quick_sort: Реализованная стандартная версия быстрой сортировки. Она разбивает массив на две части относительно опорного элемента и рекурсивно сортирует каждую из них.
3. Функция closest_points: Главная логическая функция программы. Она делает следующее:
    1) Преобразует координаты каждой точки (x,y) в расстояние до начала координат.
    2) Создает список пар [расстояние, [x, y]].
    3) Сортирует этот список по возрастанию расстояния с помощью quick_sort.
    4) Возвращает первые k точек, которые находятся ближе всего к началу координат.
4. Основная часть программы: Читает данные из файла, включая количество точек n, количество ближайших точек k, и сами

координаты точек. Вызывает функцию closest_points, сохраняет результат в выходной файл. Также измеряются время выполнения программы и объем используемой памяти.

Результат работы программы:

1) Входные данные:

```
2 1
1 3
-2 2
```

Выходные данные:

```
[[-2, 2]]
```

Время выполнения и количество затраченной памяти:

Время: 0.005813 секунд

Память: 14.7578125 Мбайт

Тесты:

```
import unittest

from lab3.task8.src.closest_points import closest_points


class TestClosestPoints(unittest.TestCase):

    def test_should_find_closest_points_example1_array(self):
        # given
        k = 1
        array = [[1, 3], [-2, 2]]
        # when
        result = closest_points(array, k)
        # then
        self.assertEqual(result, [[-2, 2]])

    def test_should_find_closest_points_example2_array(self):
        # given
        k = 2
        array = [[3, 3], [5, -1], [-2, 4]]
        # when
        result = closest_points(array, k)
        # then
        self.assertEqual(result, [[3, 3], [-2, 4]])

    def test_should_find_closest_points_sorted_array(self):
        # given
        k = 2
        array = [[1, 1], [-2, 2], [4, -3], [-7, -4]]
        # when
        result = closest_points(array, k)
        # then
        self.assertEqual(result, [[1, 1], [-2, 2]])

    def test_should_find_closest_points_reverse_sorted_array(self):
```

```
        # given
        k = 2
        array = [[-7, -4], [4, -3], [-2, 2], [1, 1]]
        # when
        result = closest_points(array, k)
        # then
        self.assertEqual(result, [[1, 1], [-2, 2]])

    def test_should_find_closest_points_empty_array(self):
        # given
        k = 4
        array = []
        # when
        result = closest_points(array, k)
        # then
        self.assertEqual(result, [])


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм поиска k ближайших к началу координат точек.

Дополнительные задачи

Задание №2. Анти-quick sort

2 задача. Анти-quick sort

Для сортировки последовательности чисел широко используется быстрая сортировка - `QuickSort`. Далее приведена программа на языке ~~Pascal~~ Python, которая сортирует массив a, используя этот алгоритм.

```
def qsort (left, right):
        key = a [(left + right) // 2]
        i = left
        j = right
        while i <= j:
                while a[i] < key: # first while
                        i += 1
                while a[j] > key : # second while
                        j -= 1
                if i <= j :
                        a[i], a[j] = a[j], a[i]
                        i += 1
                        j -= 1
        if left < j:
                qsort(left, j)
        if i < right:
                qsort(i, right)

qsort(0, n - 1)
```

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.

- **Формат входного файла (input.txt).** В первой строке находится единственное число n ($1 \le n \le 10^6$).
- **Формат выходного файла (output.txt).** Вывести перестановку чисел от 1 до n, на которой быстрая сортировка выполнит максимальное число сравнений. Если таких перестановок несколько, вывести любую из них.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
3	1 3 2

Решение:

```
import sys
import os
import psutil
import time

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

start_time = time.perf_counter()


def qsort(a, left, right):
    key = a[(left+right)//2]
    i = left
    j = right
    while i <= j:
        while a[i] < key:
            i += 1
        while a[j] > key:
            j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
            i += 1
            j -= 1
    if left < j:
        qsort(a, left, j)
    if i < right:
        qsort(a, i, right)
    return a


def anti_quick_sort(n):
    a = [i+1 for i in range(n)]
    for i in range(2, len(a)):
```

```
        a[i], a[i//2] = a[i//2], a[i]
    return a


if __name__ == '__main__':
    data = read_file(task=2)[0]
    n = int(data)
    result = anti_quick_sort(n)
    write_output(2, ' '.join(list(map(str, result))))
    print(f'Время: {(time.perf_counter() - start_time):.6f} секунд')
    print(f'Память: {psutil.Process().memory_info().rss / 1024 ** 2} Мбайт')
```

1. Функция qsort: Реализация классической быстрой сортировки. Она использует средний элемент массива в качестве опорного (key) и рекурсивно сортирует подмассивы до тех пор, пока они не будут полностью отсортированы.
2. Функция anti_quick_sort: Основная логика создания "антиупорядоченного" массива. Функция создает массив из последовательных чисел от 1 до n, а затем перемешивает их таким образом, что каждый элемент на позиции i меняется местами с элементом на позиции i//2. Такой способ перемешивания приводит к тому, что результирующий массив оказывается крайне неблагоприятным для стандартной быстрой сортировки, так как увеличивает количество сравнений и перемещений.
3. Основная часть программы: В основной части программы читается входной файл, извлекается число n (размер массива), создается "антиотсортированный" массив, после чего он сохраняется в выходном файле. Программа также измеряет время выполнения и потребление памяти.

Входные данные:

```
3
```

Выходные данные:

```
1 3 2
```

Время выполнения и количество затраченной памяти:
Время: 0.000947 секунд
Память: 14.63671875 Мбайт

Тесты:

```
import unittest

from lab3.task2.src.anti_quick_sort import anti_quick_sort
```

```
class TestAntiQuickSort(unittest.TestCase):

    def test_should_create_check_permutation_of_n_nums(self):
        # given
        # example n
        n1 = 3
        # another average n
        n2 = 10
        # when
        result1 = anti_quick_sort(n1)
        result2 = anti_quick_sort(n2)
        # then
        self.assertEqual(result1, [1, 3, 2])
        self.assertEqual(result2, [1, 4, 6, 8, 10, 5, 3, 7, 2, 9])

    def test_should_create_permutation_of_max_nums(self):
        # given
        n = 10**6
        # when
        result = anti_quick_sort(n)
        # then
        return result


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм генерации тестов - перестановок чисел, на которых функция быстрой сортировки сделает наибольшее число сравнений.

Задание №3. Сортировка пугалом

3 задача. Сортировка пугалом

«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?

- **Формат входного файла (input.txt).** В первой строчке содержатся числа n и k ($1 \le n, k \le 10^5$) – число матрёшек и размах рук. Во второй строчке содержится n целых чисел, которые по модулю не превосходят 10^9 – размеры матрёшек.
- **Формат выходного файла (output.txt).** Выведите «ДА», если возможно отсортировать матрёшки по неубыванию размера, и «НЕТ» в противном случае.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

input.txt	output.txt
3 2 2 1 3	НЕТ
5 3 1 5 3 4 1	ДА

Решение:

```
import sys
import os
import psutil
import time
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'../..')))
from utils import *

start_time = time.perf_counter()


def scarecrow_sort(n, k, array):
   for i in range(0, n-k):
       if array[i] > array[i+k]:
           array[i], array[i+k] = array[i+k], array[i]
   return "YES" if array == sorted(array) else "NO"
```

```
if __name__ == '__main__':
    data, massive = read_file(task=3)
    n, k = list(map(int, data.split()))
    array = list(map(int, massive.split()))
    scarecrow_sort = scarecrow_sort(n, k, array)
    write_output(3, scarecrow_sort)
    print(f'Время: {(time.perf_counter() - start_time):.6f} секунд')
    print(f'Память: {psutil.Process().memory_info().rss / 1024 ** 2} Мбайт')
```

1. Функция scarecrow_sort: Эта функция принимает три параметра: размер массива n, шаг k и сам массив array.
    1) Алгоритм проходит по массиву и сравнивает элементы, находящиеся на расстоянии k.
    2) Если текущий элемент больше элемента, находящегося на расстоянии k, то эти два элемента меняются местами.
    3) После завершения всех перестановок проверяется, стал ли массив отсортированным. Если да, возвращается "YES", иначе — "NO".
2. Основная часть программы: Из файла считываются данные: размер массива n, шаг k и сам массив. Затем вызывается функция scarecrow_sort, которая возвращает результат проверки возможности сортировки. Результат записывается в выходной файл. Также измеряются время выполнения программы и объем использованной памяти.

Результат работы программы:
Входные данные:

```
3 2
2 1 3
```

Выходные данные:

```
NO
```

Время выполнения и количество затраченной памяти:
Время: 0.004687 секунд
Память: 14.59375 Мбайт

Тесты:

```
import unittest

from lab3.task3.src.scarecrow_sort import scarecrow_sort


class TestScarecrowSort(unittest.TestCase):

    def test_should_check_the_possibility_of_sorting_example1_array(self):
```

```
        # given
        n, k = 3, 2
        array = [2, 1, 3]
        # when
        result = scarecrow_sort(n, k, array)
        # then
        self.assertEqual(result, 'NO')

    def test_should_check_the_possibility_of_sorting_example2_array(self):
        # given
        n, k = 5, 3
        array = [1, 5, 3, 4, 1]
        # when
        result = scarecrow_sort(n, k, array)
        # then
        self.assertEqual(result, 'YES')

    def test_should_check_the_possibility_of_sorting_sorted_array(self):
        # given
        n, k = 5, 3
        array = [1, 2, 3, 4, 5]
        # when
        result = scarecrow_sort(n, k, array)
        # then
        self.assertEqual(result, 'YES')

                                                                    def
test_should_check_the_possibility_of_sorting_reverse_sorted_array(self):
        # given
        n, k = 5, 3
        array = [5, 4, 3, 2, 1]
        # when
        result = scarecrow_sort(n, k, array)
        # then
        self.assertEqual(result, 'NO')

                                                                    def
test_should_check_the_possibility_of_sorting_single_element_array(self):
        # given
        n, k = 1, 3
        array = [1]
        # when
        result = scarecrow_sort(n, k, array)
        # then
        self.assertEqual(result, 'YES')


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм “сортировки пугалом” - проверки на то, возможно ли отсортировать массив, если мы

можем переставлять только те элементы, которые находятся на расстоянии k друг от друга.

Задание №5. Индекс Хирша

5 задача. Индекс Хирша

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.

- **Формат ввода или входного файла (input.txt).** Одна строка citations, содержащая n целых чисел, по количеству статей ученого (длина citations), разделенных пробелом или запятой.
- **Формат выхода или выходного файла (output.txt).** Одно число - индекс Хирша (h-индекс).
- Ограничения: $1 \leq n \leq 5000$, $0 \leq citations[i] \leq 1000$.
- Пример.

input.txt	output.txt
3,0,6,1,5	3

Пояснение. citations = [3,0,6,1,5] означает, что ученый опубликовал 5 статей в целом, и каждая из них оказалась процитирована 3, 0, 6, 1, 5 раз соответственно. Поскольку у ученого есть 3 статьи с минимум тремя цитированиями, а у оставшихся двух - не более 3 цитирований, его индекс Хирша равен 3.

- Пример.

input.txt	output.txt
1,3,1	1

Решение:

```
import sys
import os
import psutil
import time
from lab3.task1.src.quick_sort import quick_sort
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'../..')))
from utils import *
```

```
start_time = time.perf_counter()


def hirsch_index(citations):
    n = len(citations)
    quick_sort(citations, 0, n-1)
    for h in range(n):
        if (n - h) <= citations[h]:
            return n - h
    return 0


if __name__ == '__main__':
    citations = read_file(task=5)[0]
    array = list(map(int, citations.split()))
    result = hirsch_index(array)
    output = str(result)
    write_output(5, output)
    print(f'Время: {(time.perf_counter() - start_time):.6f} секунд')
    print(f'Память: {psutil.Process().memory_info().rss / 1024 ** 2} Мбайт')
```

1. Функция hirsch_index: Эта функция принимает на вход список цитирований и возвращает индекс Хирша. Шаги выполнения следующие:
    1) Сортировка списка цитирований в порядке убывания с помощью функции quick_sort.
    2) Поиск максимального значения h, такого что n-h ≤ citations[h], где n — длина списка. То есть ищется наибольшее значение h, для которого существует хотя бы h работ, каждая из которых имеет не менее h цитирований.
    3) Если такого значения нет, возвращается 0.
2. Основная часть программы: Читаются данные из файла, преобразуются в список целых чисел, передается в функцию hirsch_index, результат записывается в выходной файл. Время выполнения и память процесса также выводятся на экран.

Результат работы программы:
Входные данные:

```
3 0 6 1 5
```

Выходные данные:

```
3
```

Время выполнения и количество затраченной памяти:
Время: 0.006335 секунд
Память: 14.796875 Мбайт

Тесты:

```
import unittest

from lab3.task5.src.hirsch_index import hirsch_index


class TestHirschIndex(unittest.TestCase):

    def test_should_find_hirsch_index_example1_array(self):
        # given
        array = [3, 0, 6, 1, 5]
        # when
        result = hirsch_index(array)
        # then
        self.assertEqual(result, 3)

    def test_should_find_hirsch_index_example2_array(self):
        # given
        array = [1, 3, 1]
        # when
        result = hirsch_index(array)
        # then
        self.assertEqual(result, 1)

    def test_should_find_hirsch_index_sorted_array(self):
        # given
        array = [1, 2, 3, 4]
        # when
        result = hirsch_index(array)
        # then
        self.assertEqual(result, 2)

    def test_should_find_hirsch_index_reverse_sorted_array(self):
        # given
        array = [4, 3, 2, 1]
        # when
        result = hirsch_index(array)
        # then
        self.assertEqual(result, 2)

    def test_should_count_inversions_empty_array(self):
        # given
        array = []
        # when
        result = hirsch_index(array)
        # then
        self.assertEqual(result, 0)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм нахождения индекса Хирша для массива целых чисел.

Вывод

В ходе выполнения этой лабораторной работы №3 мы изучили алгоритмы быстрой сортировки и разных ее интерпретаций, сортировки за линейное время. Протестировали скорость их работы.