

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

Отчет по лабораторной работе №4
по курсу «Алгоритмы и структуры данных»
Тема: Деревья. Пирамида, пирамидальная сортировка. Очередь с
приоритетами

Вариант 18

Выполнила:
Сусликова В.Д.
К3140
Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задание №4. Построение пирамиды	3
Задание №7. Снова сортировка	6
Дополнительные задачи	8
Задание №1. Куча ли?	8
Задание №2. Высота дерева	10
Вывод	13

Задачи по варианту

Задание №4. Построение пирамиды

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort. Гарантированное время работы в худшем случае составляет $O(n \log n)$, в отличие от *среднего* времени работы QuickSort, равного $O(n \log n)$. QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней сортировки, когда вам нужно отсортировать огромные файлы, которые не помещаются в памяти вашего компьютера.

Первым шагом алгоритма HeapSort является создание пирамиды (heap) из массива, который вы хотите отсортировать.

Ваша задача - реализовать этот первый шаг и преобразовать заданный массив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы a_i и a_j массива меняются местами для некоторых i и j . Вам нужно будет преобразовать массив в пирамиду, используя только $O(n)$ перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать min-heap вместо max-heap.

- **Формат ввода или входного файла (input.txt).** Первая строка содержит целое число n ($1 \leq n \leq 10^5$), вторая содержит n целых чисел a_i входного массива, разделенных пробелом ($0 \leq a_i \leq 10^9$, все a_i - различны.)

Решение:

```
import sys
import os

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
SRC_DIR = os.path.join(CURRENT_DIR, '..', '..')
sys.path.insert(0, SRC_DIR)

from utils import read_from_file, write_in_file, measuring

def create_heap(n, A):

    swaps = [] # массив с перестановками

    def min_heap(i):
        min_index = i
        left = 2 * i + 1 # Левый ребенок
        right = 2 * i + 2 # Правый ребенок
```

```

        if left < n and A[left] < A[min_index]:
            min_index = left
        if right < n and A[right] < A[min_index]:
            min_index = right

        # Если ребенок меньше текущего узла, то меняем их местами
        if i != min_index:
            A[i], A[min_index] = A[min_index], A[i]
            swaps.append(f"{i} {min_index}\n")
            min_heap(min_index)

    for i in range(n // 2 - 1, -1, -1):
        min_heap(i)

    return swaps

if __name__ == '__main__':
    data = read_from_file(0, 'lab5/task4/txtf/input.txt')
    n = int(data[0])
    versitiles = [int(x) for x in data[1:]]
    result = create_heap(n, versitiles)
    result.insert(0, str(len(result))+'\n')
    write_in_file('lab5/task4/txtf/output.txt', result)

    measuring(1e2, create_heap, n, versitiles)

```

1. Функция `create_heap`: Реализует алгоритм создания минимальной кучи с помощью перестановок чисел в исходном массиве, сохраняя необходимые для совершения этого перестановки
2. Функция `min_heap`: Непосредственно находит нужные перестановки для создания (восстановления) кучи из данного на вход массива чисел.
2. Основная часть программы: Читает входной файл через функцию `read_from_file`, преобразует его содержимое в массив чисел, с которыми программе будет удобно работать.

Результат работы программы:
Входные данные:

5
5 4 3 2 1

Выходные данные:

3
1 4
0 1
1 3

Время выполнения и количество затраченной памяти:

Время: 0.010911 секунд

Память: 15.01953125 Мбайт

Тесты:

```
import sys
import os

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
SRC_DIR = os.path.join(CURRENT_DIR, '..', '..')
sys.path.insert(0, SRC_DIR)

import unittest

from task4.src.pyramid_build import create_heap

class TestPyramidBuild(unittest.TestCase):

    def
test_should_check_the_possibility_of_searching_inversions_on_1st_heap(
self):
    # given
    n = 5
    array = [5, 4, 3, 2, 1]
    expected_result = ['1 4\n', '0 1\n', '1 3\n']

    # when
    result = create_heap(n, array)

    # then
    self.assertEqual(result, expected_result)

    def
test_should_check_the_possibility_of_measuring_the_depth_of_the_2nd_he
ap(self):
    # given
    n = 5
```

```

array = [1, 2, 3, 4, 5]
expected_result = []

# when
result = create_heap(n, array)

# then
self.assertEqual(result, expected_result)

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

```

Вывод по задаче:

В ходе решения данной задачи мы реализовали функцию для восстановления свойств кучи с помощью перестановок элементов в заданном массиве

Задание №3. Скобочная последовательность Версия 1

Последовательность A , состоящую из символов из множества «(», «)», «[» и «]», назовем *правильной скобочной последовательностью*, если выполняется одно из следующих утверждений:

- A – пустая последовательность;
- первый символ последовательности A – это «(», и в этой последовательности существует такой символ «)», что последовательность можно представить как $A = (B)C$, где B и C – правильные скобочные последовательности;
- первый символ последовательности A – это «[», и в этой последовательности существует такой символ «]», что последовательность можно представить как $A = (B)C$, где B и C – правильные скобочные последовательности.

Так, например, последовательности «(())» и «()[]» являются правильными скобочными последовательностями, а последовательности «[]» и «((» таковыми не являются.

Входной файл содержит несколько строк, каждая из которых содержит последовательность символов «(», «)», «[» и «]». Для каждой из этих строк выясните, является ли она правильной скобочной последовательностью.

- **Формат входного файла (input.txt).** Первая строка входного файла содержит число N ($1 \leq N \leq 500$) – число скобочных последовательностей, которые необходимо проверить. Каждая из следующих N строк содержит скобочную последовательность длиной от 1 до 10^4 включительно. В каждой из последовательностей присутствуют только скобки указанных выше видов.

Решение:

```
import sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

from utils import read_from_file, write_in_file, measuring

def bracet_counter(line):
    stack = []
    matching_brackets = {')': '(', ']': '['}

    for char in line:
        if char in "([":
            stack.append(char)
        elif char in ")]":
            if stack and stack[-1] == matching_brackets[char]:
                stack.pop()
            else:
                return False
    return not stack

def bracet_counter_cycled(n, lines):
    results = []
    for line in lines:
        results.append("YES\n" if bracet_counter(line) else "NO\n")
    return results

if __name__ == '__main__':
    data = read_from_file(0, 'lab4/task3/txtf/input.txt')

    n, lines = data[0], data[1:]
    result = bracet_counter_cycled(n, lines)
    write_in_file('lab4/task3/txtf/output.txt', result)

    measuring(1e2, bracet_counter_cycled, n, lines)
```

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 `integer_sort`: 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 `integer_sort`, 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 sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

import unittest
from task6.src.integer_sort 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]
    expected_result = 51

    # when
    result = sum_of_tenths(A, B)

    # then
    self.assertEqual(result, expected_result)

def test_should_count_sum_of_tenths_sorted_array(self):
    # given
    A = [1, 2, 3, 4, 5]
    B = [1, 1, 1, 1, 1]
    expected_result = 9

    # when
    result = sum_of_tenths(A, B)

    # then
    self.assertEqual(result, expected_result)

def test_should_count_sum_of_tenths_reverse_sorted_array(self):
    # given
    A = [5, 4, 3, 2, 1]
    B = [1, 2, 3, 4, 5]
    expected_result = 22

    # when
    result = sum_of_tenths(A, B)

    # then
    self.assertEqual(result, expected_result)

def test_should_count_sum_of_tenths_empty_array(self):
    # given
    A = []
    B = []
    expected_result = 0

    # when
    result = sum_of_tenths(A, B)

```

```
        # then
        self.assertEqual(result, expected_result)

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 $1 \leq i \leq n$ и $i \leq j \leq m$ и вывода суммы каждого десятого элемента полученной последовательности.

Задание №7. Цифровая сортировка

Дано n строк, выведите их порядок после k фаз цифровой сортировки.

- **Формат входного файла (input.txt).** В первой строке входного файла содержатся числа n - число строк, m - их длина и k - число фаз цифровой сортировки ($1 \leq n \leq 10^6$, $1 \leq k \leq m \leq 10^6$, $n \cdot m \leq 5 \cdot 10^7$). Далее находится описание строк, но **в нетривиальном формате**. Так, i -ая строка ($1 \leq i \leq n$) записана в i -ых символах второй, ..., $(m + 1)$ -ой строк входного файла. Иными словами, строки написаны по вертикали. **Это сделано специально, чтобы сортировка занимала меньше времени.**

Строки состоят из строчных латинских букв: от символа "a" до символа "z" включительно. В таблице символов ASCII все эти буквы располагаются подряд и в алфавитном порядке, код буквы "a" равен 97, код буквы "z" равен 122.

- **Формат выходного файла (output.txt).** Выведите номера строк в том порядке, в котором они будут после k фаз цифровой сортировки.
- Ограничение по времени. 3 сек.
- Ограничение по памяти. 256 мб.

Решение:

```
import sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

from utils import read_from_file, write_in_file, measuring

def radix_sort_phase(strings, phase):
    """Сортировка по заданной фазе (символу)."""
    return sorted(strings, key=lambda x: x[1][phase-1])

if __name__ == "__main__":
    data = read_from_file('algorithms-and-data-structures/lab3/task7/txtf/input.txt')
    n, m, k, columns = int(data[0]), int(data[1]), int(data[2]),
    data[3:]

    # Проверка корректности входных данных
```

```

    if (1 <= n <= 10**6) and (1 <= k <= m <= 10**6) and (n * m <= 5 *
10**7):
        # Формируем список строк из колонок
        strings = [(i + 1, ''.join(columns[j][i] for j in range(m)))
for i in range(n)]

        # Применяем сортировку по фазам
        for phase in range(min(m, k) - 1, -1, -1):
            strings = radix_sort_phase(strings, phase)

        # Подготовка результатов (выводим индексы строк в новом
порядке)
        result = [str(item[0]) for item in strings]
        write_in_file('algorithms-and-data-
structures/lab3/task7/txtf/output.txt', result)
        measuring(radix_sort_phase, strings, phase)
    else:
        print('Введите корректные данные')

```

1. Функция quick_sort: Реализована версия поразрядной сортировки для одной фазы, она сортирует массив строк по выбранному символу
2. Основная часть программы: Читает данные из файла, разбивая их на переменные n, m и k, создает цикл в котором вызывает функцию поразрядной сортировки k раз. Сохраняет результат в выходной файл. Также измеряются время выполнения программы и объем используемой памяти.

Результат работы программы:

Входные данные:

3 3 3

bab

bba

baa

Выходные данные:

2 3 1

Время выполнения и количество затраченной памяти:

Время: 0.00084 секунд

Память: 15.50390625 Мбайт

Тесты:

```
import sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)
import unittest
from task7.src.digital_sort import radix_sort_phase

class TestRadixSort(unittest.TestCase):

    def test_should_make_radix_sort_phase(self):
        # given
        strings = [
            (1, "bab"),
            (2, "bba"),
            (3, "baa")
        ]
        # when
        sorted_strings_phase_2 = radix_sort_phase(strings, 3)
        # then
        self.assertEqual(sorted_strings_phase_2, [
            (2, 'bba'),
            (3, 'baa'),
            (1, 'bab')
        ])

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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм пофазовой цифровой (поразрядной) сортировки массива из n строк длины m в k фаз соответственно, протестировали его затраты времени и памяти на исполнение и проверили правильность.

Дополнительные задачи

Задание №2. Анти-quick sort

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 ($1 \leq n \leq 10^6$).
- **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

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

from utils import read_from_file, write_in_file, measuring

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_from_file('algorithms-and-data-
structures/lab3/task2/txtf/input.txt')

    n = data[0]
    result = anti_quick_sort(int(n))

    write_in_file('algorithms-and-data-
structures/lab3/task2/txtf/output.txt', result)

    measuring(anti_quick_sort, int(n))

```

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 quick_sort_desolver: 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 sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

import unittest

from task2.src.quick_sort_desolver import anti_quick_sort

class TestAntiQuickSort(unittest.TestCase):

    def test_should_create_check_permutation_of_n_nums(self):
        # given
        # example n
        n1 = 3
        expected_result1 = [1, 3, 2]

        # another average n
        n2 = 10
        expected_result2 = [1, 4, 6, 8, 10, 5, 3, 7, 2, 9]

        # when
        result1 = anti_quick_sort(n1)
        result2 = anti_quick_sort(n2)

        # then
        self.assertEqual(result1, expected_result1)
```



```

self.assertEqual(result2, expected_result2)

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

«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 матрёшки 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 ($1 \leq n, k \leq 10^5$) — 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

```

```

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

from utils import read_from_file, write_in_file, measuring

def scarecrow_sort(n, k, array):
    n, k = int(n), int(k)
    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 = read_from_file('algorithms-and-data-
structures/lab3/task3/txtf/input.txt')

    n, k = data[:2]
    array = data[2:]
    result = scarecrow_sort(n, k, array)

    write_in_file('algorithms-and-data-
structures/lab3/task3/txtf/output.txt', [result])

    measuring(scarecrow_sort, n, k, array)

```

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 sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

import unittest

from 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]
    expected_result = 'NO'

    # when
    result = scarecrow_sort(n, k, array)

    # then
    self.assertEqual(result, expected_result)
```

```

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

    # then
    self.assertEqual(result, expected_result)

def
test_should_check_the_possibility_of_sorting_sorted_array(self):
    # given
    n, k = 5, 3
    array = [1, 2, 3, 4, 5]
    expected_result = 'YES'

    # when
    result = scarecrow_sort(n, k, array)

    # then
    self.assertEqual(result, expected_result)

def
test_should_check_the_possibility_of_sorting_reverse_sorted_array(self):
    # given
    n, k = 5, 3
    array = [5, 4, 3, 2, 1]
    expected_result = 'NO'

    # when
    result = scarecrow_sort(n, k, array)

    # then
    self.assertEqual(result, expected_result)

def
test_should_check_the_possibility_of_sorting_single_element_array(self):
    # given
    n, k = 1, 3

```

```
array = [1]
expected_result = 'YES'

# when
result = scarecrow_sort(n, k, array)

# then
self.assertEqual(result, expected_result)

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

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

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

from utils import read_from_file, write_in_file, measuring
from task1.src.quick_sort import quick_sort
```

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

if __name__ == '__main__':
    data = read_from_file('algorithms-and-data-
structures/lab3/task5/txtf/input.txt')

    result = hirsch_index(data)

    write_in_file('algorithms-and-data-
structures/lab3/task5/txtf/output.txt', [result])
    measuring(hirsch_index, data)
```

1. Функция `hirsh_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 $n - h \leq \text{citations}[h]$, 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 `hirsh_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 sys
import os

current_dir = os.path.dirname(os.path.abspath(__file__))
src_dir = os.path.join(current_dir, '..', '..')
sys.path.insert(0, src_dir)

import unittest

from task5.src.hirsh_index import hirsch_index

class TestHirschIndex(unittest.TestCase):

    def test_should_find_hirsch_index_example1_array(self):
        # given
        array = [3, 0, 6, 1, 5]
        expected_result = 3

        # when
        result = hirsch_index(array)

        # then
        self.assertEqual(result, expected_result)

    def test_should_find_hirsch_index_example2_array(self):
        # given
        array = [1, 3, 1]
        expected_result = 1

        # when
        result = hirsch_index(array)

        # then
        self.assertEqual(result, expected_result)

    def test_should_find_hirsch_index_sorted_array(self):
        # given
        array = [1, 2, 3, 4]
        expected_result = 2
```



```

        # when
        result = hirsch_index(array)

        # then
        self.assertEqual(result, expected_result)

    def test_should_find_hirsch_index_reverse_sorted_array(self):
        # given
        array = [4, 3, 2, 1]
        expected_result = 2

        # when
        result = hirsch_index(array)

        # then
        self.assertEqual(result, expected_result)

    def test_should_count_inversions_empty_array(self):
        # given
        array = []
        expected_result = 0

        # when
        result = hirsch_index(array)

        # then
        self.assertEqual(result, expected_result)

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