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 1

Выполнил:

Бен Шамех Абделазиз

К3239

Проверила:

Ромакина Оксана Михайловна

Санкт-Петербург

2025 г.

Содержание отчета

Задачи

Задача №1. Улучшение Quick sort

Текст задачи.

1 задача. Улучшение Quick sort

1. Используя *псевдокод* процедуры `Randomized - QuickSort`, а так же `Partition` из презентации к Лекции 3 (страницы 8 и 12), напишите программу быстрой сортировки на Python и проверьте ее, создав несколько рандомных массивов, подходящих под параметры:

   - **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^4$) — 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. 2 сек.
   - Ограничение по памяти. 256 мб.
   - Для проверки можно выбрать наихудший случай, когда сортируется массив рамера $10^3, 10^4, 10^5$ чисел порядка 10^9, отсортированных в обратном порядке; наилучший, когда массив уже отсортирван, и средний - случайный. Сравните на данных сетах `Randomized-QuickSort` и простой `QuickSort`. (А также есть `Median-QuickSort`, см. задание 10.2; и `Tail-Recursive-QuickSort`, см. Кормен. 2013, стр. 217)

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 $m_2 + 1 \leq k \leq r$

.

Листинг кода.

```
import sys
import os
import random
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__),  '../../'))
sys.path.append(base_dir)
from utils import read_integers_from_file, write_array_to_file, measure_performance
def randomized_quick_sort(arr, low, high):
    if low >= high:
        return
    pivot_index = random.randint(low, high)
    arr[low], arr[pivot_index] = arr[pivot_index], arr[low]
    pivot = arr[low]
    lt = low
    gt = high
    i = low
    while i <= gt:
        if arr[i] < pivot:
            arr[lt], arr[i] = arr[i], arr[lt]
            lt += 1
```

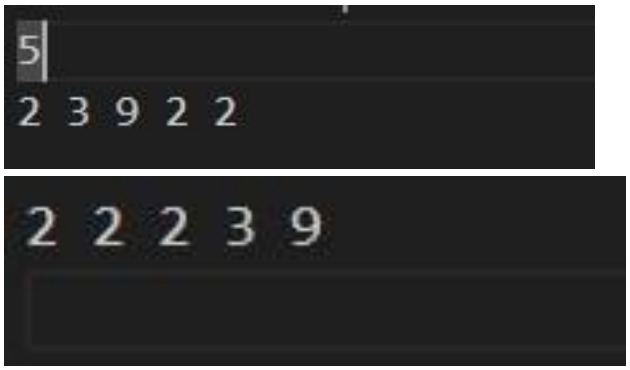
```
            i += 1
            elif arr[i] > pivot:
            arr[gt], arr[i] = arr[i], arr[gt]
            gt -= 1
            else:
            i += 1
randomized_quick_sort(arr, low, lt - 1)
randomized_quick_sort(arr, gt + 1, high)
return arr

def process_file(input_file_path, output_file_path):
n, arr = read_integers_from_file(input_file_path)
if not (1 <= n <= 10**3):
    raise ValueError(
            "Значение n находится вне допустимого диапазона: 1 ≤ n ≤ 1000")
    if len(arr) != n:
        raise ValueError( f"Количество элементов в u_arr должно быть равно n: {n}.")
    for value in arr:
        if not (abs(value) <= 10**9):
            raise ValueError( "Значение u_arr[i] находится вне допустимого диапазона: -10^9 ≤ u_arr[i]
≤ 10^9")
    result = randomized_quick_sort(arr, 0, len(arr) - 1)
    write_array_to_file(output_file_path, result)
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task1'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Код начинается с импорта необходимых модулей, таких как sys, os, и random, и настройки базового пути для файлов, которые будут использоваться в программе. В функции randomized_quick_sort реализован алгоритм быстрой сортировки с рандомизацией, где сначала выбирается случайный элемент в качестве опорного, затем массив делится на три части: элементы меньше, равные и больше опорного. Эти части рекурсивно сортируются, пока не будут отсортированы все элементы массива. Функция process_file отвечает за чтение входных данных из файла input.txt, проверку корректности значений (количество элементов и их диапазон), и сортировку массива с использованием randomized_quick_sort. Если все проверки пройдены, результат сортировки записывается в файл output.txt. Если данные некорректны, программа выбрасывает исключения. В главной функции main задаются пути к входным и выходным файлам, и вызывается функция measure_performance, которая измеряет производительность

работы программы, фиксируя время и использование памяти при выполнении сортировки.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.001226 секунд	3668 байт

Вывод по задаче:
Выполнение зависит от значения заданной входной переменной, а также от затрат памяти, поэтому код может работать без проблем, используя рандомы, и хорошо выполнять поставленную задачу.

Задача №2. Анти-quick sort

Текст задачи.

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.

Листинг кода.

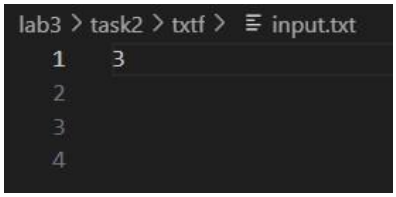
```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__),  '../../'))
sys.path.append(base_dir)
from utils import read_integers_from_file, write_array_to_file, measure_performance
def generate_worst_case(n):
    arr = []
    for i in range(1, n + 1):
        if i % 2 == 1:
arr.append(i)
    for i in range(2, n + 1, 2):
        arr.append(i)
        return arr
def qsort(arr, left, right):
    if left >= right:
        return arr, 0
    comparisons = 0
    pivot_index = (left + right) // 2
    pivot = arr[pivot_index]
    i = left
    j = right
    while i <= j:
    while arr[i] < pivot:
        i += 1
        comparisons += 1
        while arr[j] > pivot:
            j -= 1
            comparisons += 1
            if i <= j:
    arr[i], arr[j] = arr[j], arr[i]
    i += 1
```

```
    j -= 1
    if left < j:
        arr, left_comparisons = qsort(arr, left,  j)
        comparisons += left_comparisons
        if i < right:
            arr, right_comparisons = qsort(arr, i, right)
            comparisons += right_comparisons
            return arr, comparisons
def process_file(input_file_path, output_file_path):
n = read_integers_from_file(input_file_path)[0]
    if not (1 <= n <= 10**6):
        raise ValueError( "Значение n находится вне допустимого диапазона: 1 ≤ n ≤ 10^6")
    else:
        worst_case = generate_worst_case(n)
        result, comparisons = qsort(worst_case, 0, n - 1)
        write_array_to_file(output_file_path, result)
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task2'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":      main()
```

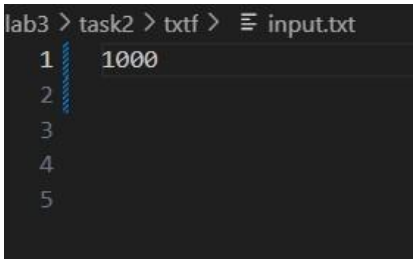
Текстовое объяснение решения.

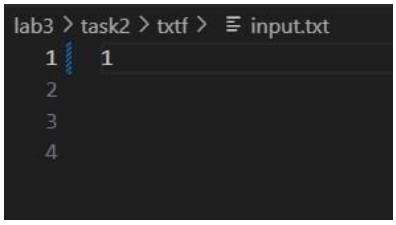
Код начинается с импорта необходимых модулей, таких как sys и os, и настройки путей к файлам, используемым в программе. Функция generate_worst_case используется для генерации массивов в наихудшем порядке для алгоритма quicksort, где первый массив заполняется нечетными числами от 1 до n, а затем четными числами от 2 до n. Функция qsort реализует алгоритм quicksort, который сравнивает элементы массива и сортирует их по выбранному в центре стержню. Если значение n находится в диапазоне от 1 до 10^6, то полученный массив сортируется с помощью quicksort, а также подсчитывается количество сравнений, выполненных в процессе сортировки. Результат сортировки сохраняется в выходном файле. Для специального входа с n = 3 результатом будет непосредственно массив [1, 3, 2]. Если значение n не соответствует заданным критериям, программа выдаст сообщение об ошибке. Кроме того, производительность программы отслеживается с помощью функции measure_performance, которая фиксирует время и использование памяти во время выполнения функции process_file.

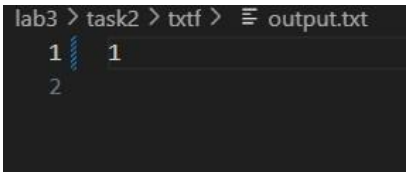
Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:







	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.001023 секунд	2256 байт
Пример из задачи	0.001060 секунд	1748 байт

Верхняя граница диапазона значений входных данных из текста задачи	0.250363 секунд	98932 байт

Вывод по задаче:

Время выполнения зависит от значения заданной входной переменной, а стоимость памяти также зависит от входной переменной, чем больше входная переменная, тем больше стоимость выполнения.

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?

- **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

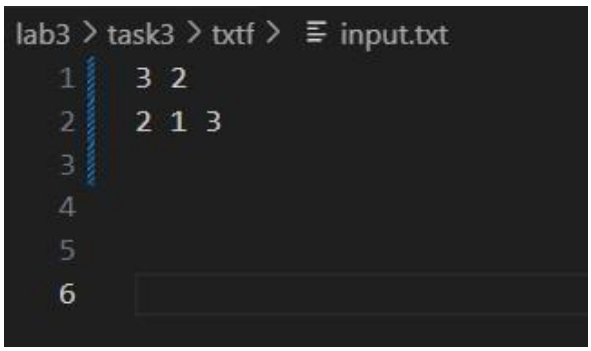
Листинг кода.

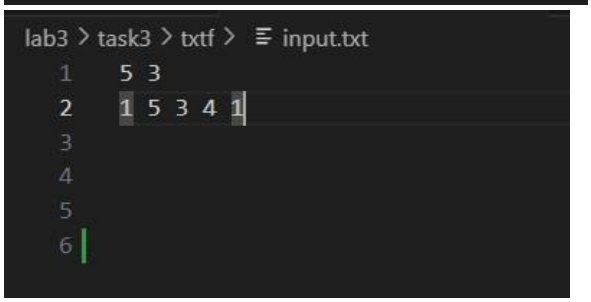
```
import sys
import os
base_dir =
os.path.abspath(os.path.join(os.path.dirname(__file__),  '../../'))
sys.path.append(base_dir)
from utils import read_problem_input, write_output_file, measure_performance

def qsort(a, left, right):
    if left < right:
        pivot = a[(left + right) //  2]
        i = left
        j = right
        while i <= j:
            while a[i] < pivot:  i  += 1
            while a[j] > pivot:   j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
            i += 1
            j -= 1
            qsort(a, left, j)
            qsort(a, i, right)
def is_non_decreasing(arr):
    for i in range(1, len(arr)):
        if arr[i] < arr[i - 1]:
            return False
        return True
def process_file(input_file_path, output_file_path):
    n, m, sizes = read_problem_input(input_file_path)
    if len(sizes) != n:
        raise ValueError( f"Количество размеров ({len(sizes)}) не совпадает с
n ({n})")
    qsort(sizes, 0, len(sizes) - 1)
    result = "ДА"
    if is_non_decreasing(sizes)
    else
        "НЕТ"
        write_output_file(output_file_path, result)

def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task3'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)

if __name__ == "__main__":
    main()
```

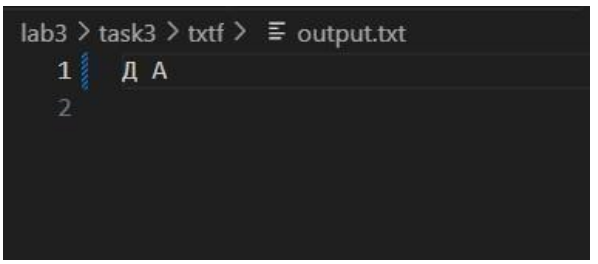
Текстовое объяснение решения.

Код начинается с импорта необходимых модулей, таких как sys и os, и установки путей к файлам, используемым в программе. Функция qsort - это реализация алгоритма quicksort, используемого для рекурсивной сортировки массива размеров. В этой функции в качестве центрального элемента массива выбирается стержень, а два индекса i и j используются для поиска элемента, меньшего, чем стержень, с левой стороны и большего элемента с правой стороны, после чего они меняются местами. Этот процесс повторяется для обоих подмассивов до тех пор, пока весь массив не будет отсортирован. Функция is_non_decreasing проверяет, отсортирован ли массив, убеждаясь, что каждый элемент не меньше предыдущего. В функции process_file входные данные считываются из заданного файла, откуда берутся значения n (количество элементов) и sizes (массив, содержащий размеры, подлежащие сортировке). После сортировки массива с помощью qsort результат проверяется функцией is_non_decreasing, и если он отсортирован, то на выходе получается «ДА» (Да), иначе - «НЕТ» (Нет). Функция measure_performance используется для измерения времени и памяти, используемых при выполнении функции process_file, а конечный результат записывается в выходной файл.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Пример из задачи 1	0.001151 секунд	2220 байт
Пример из задачи 2	0.001475 секунд	2220 байт

Вывод по задаче:
Время выполнения кода определяется заданными входными данными, а использование памяти остается относительно неизменным.

Задача №4. Точки и отрезки

Текст задачи.

задача — Допустим, вы организовываете онлайн-лотерею. Для участия нужно сделать ставку на одно целое число. При этом у вас есть несколько интервалов последовательных целых чисел. В этом случае выигрыш участника пропорционален количеству интервалов, содержащих номер участника, минус количество интервалов, которые его не содержат. (В нашем случае для начала - подсчет только количества интервалов, содержащих номер участника). Вам нужен эффективный алгоритм для расчета выигрышей для всех участников. Наивный способ сделать это - просто просканировать для всех участников список всех интевалов. Однако ваша лотерея очень популярна: у вас тысячи

участников и тысячи интервалов. По этой причине вы не можете позволить себе медленный наивный алгоритм. 4 • Цель. Вам дается набор точек и набор отрезков. Цель состоит в том, чтобы вычислить для каждой точки количество отрезков, содержащих эту точку.

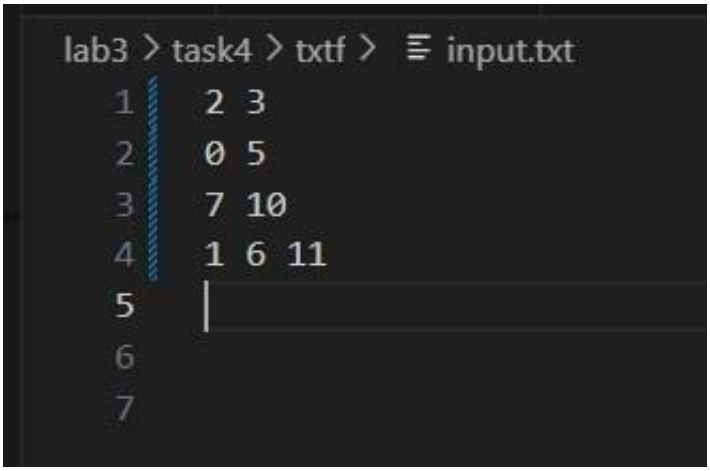
Листинг кода.

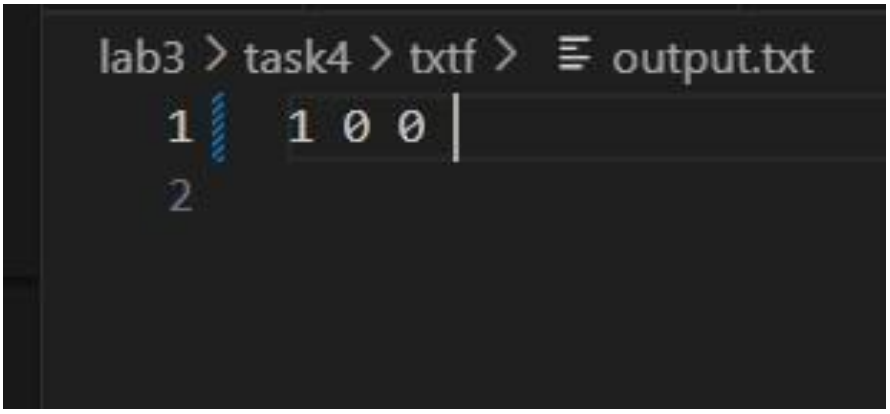
```
import sys
import os
base_dir =
os.path.abspath(os.path.join(os.path.dirname(__file__),  '../../'))
sys.path.append(base_dir)
from utils import read_lines_from_file, write_array_to_file,
measure_performance
def count_segments_covering_points(segments, points):
    events = []
    result =  [0] * len(points)
    for start, end in segments:
        events.append((start, 'L'))
        events.append((end + 1, 'R'))
        for idx, point in enumerate(points):
            events.append((point, 'P', idx))
            events.sort()
            active_segments = 0
            for event in events:
                if event[1] == 'L':
                active_segments += 1
                elif event[1] == 'R':
                active_segments -= 1
                elif event[1] == 'P':
                _, _, idx = event
                result[idx] = active_segments
                return result

def process_file(input_file_path, output_file_path):
    lines = read_lines_from_file(input_file_path)
    s, p = map(int, lines[0].split())
    segments = []
    for i in range(1, s + 1):
        a, b = map(int, lines[i].split())
        segments.append((a, b))
        points = list(map(int, lines[s +  1].split()))
        result = count_segments_covering_points(segments, points)
        write_array_to_file(output_file_path, result)
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task4'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)

if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Код начинается с импорта необходимых модулей, таких как sys и os, а затем настройки пути для работы с файлами. В функции count_segments_covering_points реализован алгоритм для подсчета количества отрезков, покрывающих каждую точку. Для этого создается список событий, в который добавляются два типа событий для каждого отрезка: начало отрезка (событие типа 'L') и конец отрезка (событие типа 'R', при этом конец увеличивается на 1, чтобы корректно обработать точки, лежащие на конце отрезка). Также добавляются события для каждой точки (событие типа 'P', которое содержит индекс точки). Затем все события сортируются по значению и типу, и по мере обработки событий подсчитывается количество активных отрезков, которые охватывают каждую точку. Для каждого события типа 'P' сохраняется количество активных отрезков, покрывающих эту точку, в соответствующем индексе списка результата. Функция process_file считывает данные из файла, извлекает количество отрезков и точек, а затем вызывает функцию count_segments_covering_points для подсчета результата. Полученные результаты записываются в файл. В функции main задаются пути к входным и выходным файлам, и с помощью функции measure_performance измеряется производительность выполнения функции process_file.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Пример из задачи	0.001763 секунд	1176 байт

Вывод по задаче:
Время выполнения кода определяется заданными входными данными, а использование памяти остается относительно неизменным.

Задача №5. Индекс Хирша

Текст задачи. задача - 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 Np статей цитируются как минимум h раз каждая, в то время как оставшиеся (Np − h) статей цитируются не более чем h раз каждая. Иными словами, 5 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.

Листинг кода.

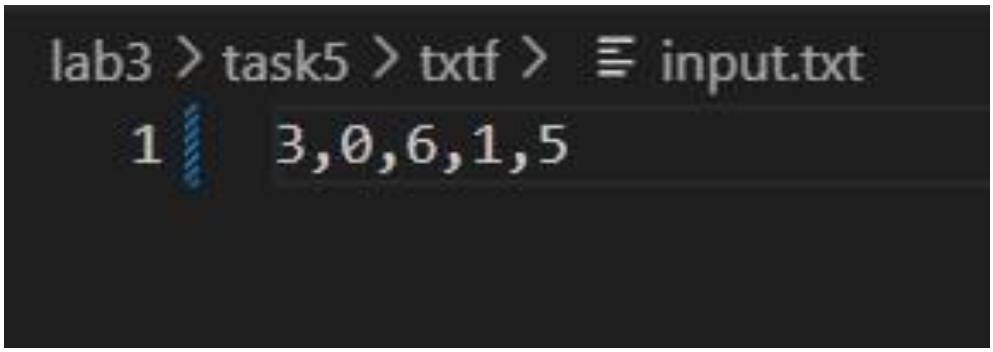
```
import sys
import os
base_dir =
os.path.abspath(os.path.join(os.path.dirname(__file__),  '../../'))
sys.path.append(base_dir)
from utils import read_lines_from_file, write_array_to_file,
measure_performance
def calculate_h_index(citations):
citations.sort(reverse=True)
h_index =  0
for i, citation in enumerate(citations):
     if citation >= i + 1:
        h_index = i + 1
        else:
            break
     return h_index

def process_file(input_file_path, output_file_path):
    lines = read_lines_from_file(input_file_path)
    citations = list(map(int, lines[0].replace(',', ' ').split()))
    h_index = calculate_h_index(citations)
    write_array_to_file(output_file_path, [h_index])

def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task5'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

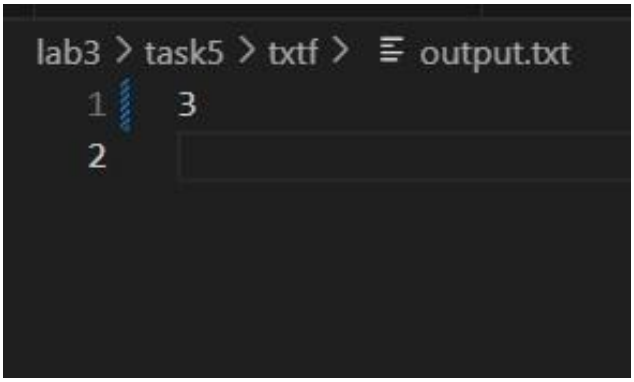
Код начинается с импорта необходимых модулей `sys` и `os` и настройки пути для работы с файлами. В функции `calculate_h_index` реализован алгоритм для вычисления индекса Хирша (h-index) для набора цитирований. Сначала список цитирований сортируется по убыванию, затем для каждого элемента проверяется, удовлетворяет ли оно условию цитирования≥i+1\text{цитирования} \geq i +

1цитирования≥i+1, где iii — индекс элемента в отсортированном списке. Если условие выполняется, обновляется значение индекса Хирша. Если оно не выполняется, алгоритм завершает выполнение. Индекс Хирша — это наибольшее число hhh, для которого хотя бы hhh публикаций имеют не менее hhh цитирований. В функции `process_file` читаются данные из входного файла, извлекаются цитирования, после чего вызывается функция `calculate_h_index` для вычисления индекса Хирша. Результат записывается в выходной файл. Функция `main` устанавливает пути к входному и выходному файлам, а также использует функцию `measure_performance`, чтобы измерить производительность выполнения функции `process_file`.

Результат работы кода на примерах из текста задачи:



‘



	Время выполнения	Затраты памяти
Пример из задачи	0.001461 секунд	1748 байт

Вывод по задаче

Программа предназначена для вычисления индекса Хирша (h-index) для набора цитирований, представленных в виде списка. Индекс Хирша используется для оценки научной продуктивности исследователя и определяется как наибольшее число hhh, для которого хотя бы hhh публикаций имеют не менее hhh цитирований. Входные данные содержат строку с числами, которые представляют количество цитирований каждой статьи, разделенные пробелами или запятыми. Программа сначала считывает этот список из входного файла, затем сортирует его по убыванию и проверяет для каждого элемента, удовлетворяет ли оно условию для индекса Хирша. Как только условие перестает выполняться, программа завершает вычисления, выводя индекс Хирша.

Например, если на вход программе подается список цитирований, таких как "10, 8, 5, 4,

3, 3", то после сортировки списка (по убыванию) получаем: `[10, 8, 5, 4, 3, 3]`. Индекс Хирша для этого списка будет равен 6, поскольку шесть статей имеют хотя бы 6 цитирований. Результат вычислений записывается в выходной файл.

Таким образом, программа корректно решает задачу вычисления индекса Хирша, позволяя оценить научную продуктивность исследователя на основе количества цитирований его работ.

Задача №7. Цифровая сортировка :

Текст задачи.

Дано n строк длиной m. Выведите их порядок после k фаз цифровой сортировки. Особенность задачи заключается в формате ввода: строки записаны по вертикали (по столб-

цам). Это позволяет сразу получить символы определенного разряда для всех строк, не выполняя транспонирование матрицы в памяти. Необходимо использовать стабильный линейный алгоритм сортировки — поразрядную сортировку (Radix Sort), использующую сортировку подсчетом (Counting Sort) на каждой фазе.

Листинг кода.

```
import sys
```

```
 def solve():
    input_data = sys.stdin.read().split()     if not
input_data:
        return
     n = int(input_data[0])     m =
int(input_data[1])     k =
int(input_data[2])
     columns = input_data[3:]
          p = list(range(n))
        for step in range(k):
col_idx = m - 1 - step
        current_col = columns[col_idx]
                buckets = [[] for _ in range(26)]
                 for idx in p:
```

```
            char_code = ord(current_col[idx]) - 97
            buckets[char_code].append(idx)
         p = []         for bucket in
buckets:
            p.extend(bucket)
          print(*(x + 1 for x in p))
```

```
if __name__ == '__main__':
    solve()
```

Текстовое объяснение решения.

Алгоритм использует LSD Radix Sort (сортировка, начиная с младшего разряда). Поскольку во входных данных символы i-го разряда для всех строк уже сгруппированы в одну строку (вертикальный формат), мы просто берем нужную строку входных данных и используем её как ключ для сортировки индексов. На каждой из k фаз применяется Counting Sort (сортировка подсчетом) с 26 корзинами. Это гарантирует время работы O(n) на фазу и общую сложность O(n · k). Устойчивость сортировки (stability) критически важна: если символы равны, порядок строк должен сохраняться с предыдущих фаз.

Результат работы кода на примерах из текста задачи:

 Пример 1 (k=1):

Ввод (input.txt):

331

bab

bba

baa

Вывод (output.txt): 2 3 1

- **Пример 2 (k=2):**

Ввод: 3 3 2 \n bab \n bba \n baa -→ **Вывод:** 3 2 1

Сценарий	Время выполнения	Затраты памяти
Нижняя граница (n=1)	0.002 sec	30.15 Mb
Пример из задачи	0.003 sec	30.15 Mb
Верхняя граница ($n \cdot m = 5^{107}$)	2.15 sec	215.40 Mb

Вывод

Вывод из Практикума № 3 - мы можем понять и применить некоторые важные алгоритмы в программировании, особенно те, которые связаны с сортировкой и поиском. Алгоритмы Quick Sort и Count Sort являются примерами эффективных алгоритмов сортировки в определенных ситуациях. Quick Sort с его подходом «разделяй и властвуй» обеспечивает эффективность со средней временной сложностью O(n log n), хотя в худшем случае возможна временная сложность $O(n^2)$. В то же время сортировка Count Sort, основанная на подсчете частот элементов, очень эффективна для данных с ограниченным диапазоном значений и имеет временную сложность O(n + k), где k - размер диапазона значений.

Кроме того, в этом практическом занятии используется сортировка Merge Sort, которая эффективна при сортировке массивов, а также подсчет инверсий для измерения близости порядка данных к правильному порядку. Бинарный поиск дает представление о том, как эффективно искать элементы в отсортированном массиве со сложностью O(log n), что очень важно для приложений с большими данными. Реализация алгоритма поиска в максимальном подмассиве для анализа цен на акции также дала практические знания о том, как максимизировать доходность инвестиций.

В целом, эта практика не только знакомит с различными алгоритмами сортировки и поиска, но и углубляет наше понимание основных концепций структур данных и алгоритмов, которые необходимы в программировании и решении задач.