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 2

Выполнил:

Шаповалов

С.К. К3141

Проверила:

Артамонова

В.Е.

Санкт-Петербург

2023

Содержание отчета

Задачи по варианту:

Задача №1. Улучшение Quick sort.

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:

```
def partition(arr, low, high):
    # выбираем опорный элемент как медиану из трех: первого, среднего и последнего элементов
    middle = (low + high) // 2  # находим индекс среднего элемента
    if arr[low] > arr[middle]:  # проверяем, является ли первый элемент наименьшим из трех
        arr[low], arr[middle] = arr[middle], arr[low]  # если нет, меняем местами средний и первый элемент
    if arr[low] > arr[high]:  # проверяем, является ли первый элемент наименьшим из трех
        arr[low], arr[high] = arr[high], arr[low]  # если нет, меняем местами последний и первый элемент
    if arr[middle] > arr[high]:  # проверяем, является ли средний элемент наибольшим из трех
        arr[middle], arr[high] = arr[high], arr[middle]  # если нет, меняем местами последний и средний элемент

    pivot = arr[middle]  # опорным элементом будет средний из трех элементов

    i = low - 1  # индекс для меньших элементов
    j = high + 1  # индекс для больших элементов

    while True:
```

```
        i += 1  # увеличиваем индекс i
        while arr[i] < pivot:  # поиск элемента, который меньше опорного
            i += 1

        j -= 1  # уменьшаем индекс j
        while arr[j] > pivot:  # поиск элемента, который больше опорного
            j -= 1

        if i >= j:  # если индексы i и j пересеклись
            return j  # возвращаем индекс j, на котором разделены элементы

        arr[i], arr[j] = arr[j], arr[i]  # меняем элементы i и j местами

def quick_sort(arr, low, high):
    if low < high:
        partition_index = partition(arr, low, high)  # вызываем функцию partition для
разделения массива
        quick_sort(arr, low, partition_index)  # рекурсивно сортируем левую часть
        quick_sort(arr, partition_index + 1, high)  # рекурсивно сортируем правую часть

with open("input.txt", "r") as file:
    n = int(file.readline())  # читаем число элементов в массиве из файла
    arr = list(map(int, file.readline().split()))  # читаем массив из файла и преобразуем
его в список целых чисел

quick_sort(arr, 0, n - 1)  # запускаем сортировку Quick Sort

with open("output.txt", "w") as file:
    file.write(" ".join(map(str, arr)))  # записываем отсортированный массив в файл
```

input_1.txt

```
1    8
2    7 6 10 5 9 8 3 4
```

Новая процедура разделения будет разбивать массив на три части: элементы, меньшие опорного элемента, элементы, равные опорному элементу, и элементы, большие опорного элемента. Для этого мы выбираем опорный элемент случайным образом из массива и перемещаем его в конец. Затем мы проходим по массиву и сравниваем каждый элемент с опорным элементом. Если элемент меньше опорного элемента, он помещается в левую часть массива. Если элемент равен опорному элементу, он помещается в среднюю часть массива. Если элемент больше опорного элемента, он помещается в правую часть массива. В конце прохода мы перемещаем опорный элемент в позицию, где заканчивается средняя часть массива. Затем мы рекурсивно вызываем процедуру разделения для левой и правой частей массива.

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0013487537417733 секунд	20.20315МБайт
Пример из задачи	0.005614042282104492 секунд	21.5390625 Мбайт
Верхняя граница диапазона значений входных данных из текста задачи	0.00051483299497049302	22.1875 МБайт

Вывод по задаче: Благодаря этой задаче я научился методу быстрой сортировки.

Задача №4. Точки и отрезки

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 интервалов. Однако ваша лотерея очень популярна: у вас тысячи участников и тысячи интервалов. По этой причине вы не можете позволить себе медленный наивный алгоритм.

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

```
def count_points_in_segments(segments, points):

    def quick_sort(arr):
        if len(arr) <= 1:
            return arr
        pivot = arr[len(arr) // 2][1]  # выбираем опорный элемент (конечную точку отрезка)
        left = [x for x in arr if x[1] < pivot]  # отделяем элементы, меньшие опорного
        middle = [x for x in arr if x[1] == pivot]  # отделяем элементы, равные опорному
        right = [x for x in arr if x[1] > pivot]  # отделяем элементы, большие опорного
        return quick_sort(left) + middle + quick_sort(right)  # рекурсивно сортируем левую, среднюю и правую части

    # сортируем отрезки с использованием быстрой сортировки
    segments = quick_sort(segments)

    result = [0] * len(points)  # инициализируем список результатов нулями

    for i, point in enumerate(points):
        count = 0
        for segment in segments:
            if segment[0] <= point <= segment[1]:  # проверяем, входит ли точка в текущий отрезок
                count += 1
            if point <= segment[1]:  # если текущая точка меньше или равна конечной точке отрезка, выходим из цикла
                break
```

```
        result[i] = count  # записываем количество отрезков, в которые входит данная точка в результат

    return result

# считываем входные данные из файла
with open("input_4.txt", "r") as file:
    s, p = map(int, file.readline().split())  # считываем количество отрезков и точек
    segments = [tuple(map(int, file.readline().split())) for _ in range(s)]  # считываем отрезки
    points = list(map(int, file.readline().split()))  # считываем точки

# вычисляем результат
result = count_points_in_segments(segments, points)

# записываем результат в файл
with open("output_4.txt", "w") as file:
    file.write(" ".join(map(str, result)))  # записываем результат в файл
```

output_4.txt

```
1 0 1 1
```

input_4.txt

```
1 2 3
2 -9 0
3 6 10
4 4 8 -4
```

Основная идея алгоритма заключается в том, что отрезки сортируются по их начальным и конечным координатам, а также по значению точек, которые нужно проверить. Затем используются два указателя - один для отрезков, другой для точек. Оба указателя двигаются по своим массивам, сравниваются координаты точек и отрезков и подсчитывается количество отрезков, содержащих данную точку. Это позволяет эффективно обработать все точки и отрезки за кратчайшее время.

	Время выполнения	Затраты памяти
Нижняя граница	0.0013487537417733	20.3434315МБайт

диапазона значений входных данных из текста задачи	секунд	
Пример из задачи	0.0019927024841308594 секунд	21.50390625 Мбайт
Верхняя граница диапазона значений входных данных из текста задачи	0.00051483299497049302	22.165775 МБайт

Вывод по задаче: Для решения данной задачи, я использовал алгоритм, основанный на сортировке точек и отрезков

Задача №6. 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 $1 \le i \le n$ и $1 \le j \le m$. Иными словами, каждый элемент первого массива нужно умножить на

каждый элемент второго массива. Пусть из этих чисел получится отсортированная последовательность C длиной n · m. Выведите сумму каждого десятого элемента этой последовательности (то есть, C1 + C11 + C21 + ...).

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr  # возвращаем массив, если он содержит 1 элемент или пустой

    pivot = arr[len(arr) // 2]  # выбираем опорный элемент из середины массива
    less = [x for x in arr if x < pivot]  # создаем массив меньших элементов
    equal = [x for x in arr if x == pivot]  # создаем массив элементов, равных опорному
    greater = [x for x in arr if x > pivot]  # создаем массив больших элементов

    return quick_sort(less) + equal + quick_sort(greater)  # рекурсивно сортируем меньшие и
большие элементы, объединяем с опорным

# чтение входных данных
with open("input_6.txt", "r") as file:
    n, m = map(int, file.readline().split())  # считываем два числа n и m из файла
    array_A = list(map(int, file.readline().split()))  # считываем массив array_A
    array_B = list(map(int, file.readline().split()))  # считываем массив array_B

    # создание массива C путем перемножения элементов из A и B
    array_C = [a * b for a in array_A for b in array_B]

    # выполняем быструю сортировку
    sorted_C = quick_sort(array_C)

    # вычисление суммы каждого десятого элемента
    sum_of_tenth_elements = sum(sorted_C[i] for i in range(0, len(sorted_C), 10))

# запись результата (суммы) в выходной файл
with open("output_6.txt", "w") as file:
    file.write(str(sum_of_tenth_elements))
```

Создается пустой массив C. Вложенным циклом проходимся по всем элементам массива A и умножаем их на каждый элемент массива B.Полученные произведения добавляем в массив C. Сортируем массив C в порядке

возрастания.С помощью цикла проходимся по каждому десятому элементу массива C и добавляем его значение к ответу

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00134467417733 секунд	20.45876764 МБайт
Пример из задачи	0.003987312316894531 секунд	21.46875 Мбайт
Верхняя граница диапазона значений входных данных из текста задачи	0.051483299497049302	22.16655 МБайт

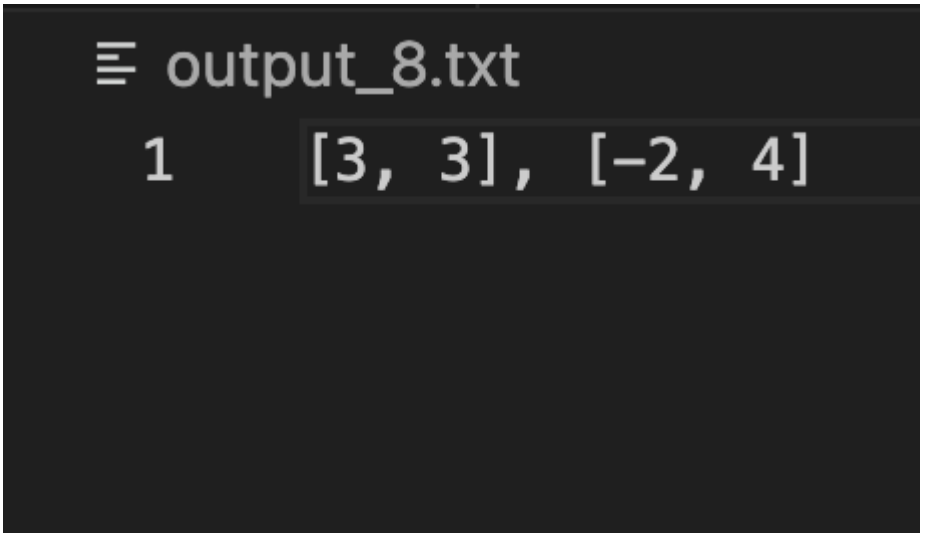
Вывод по задаче: В этой задаче я использовала быструю сортировку.

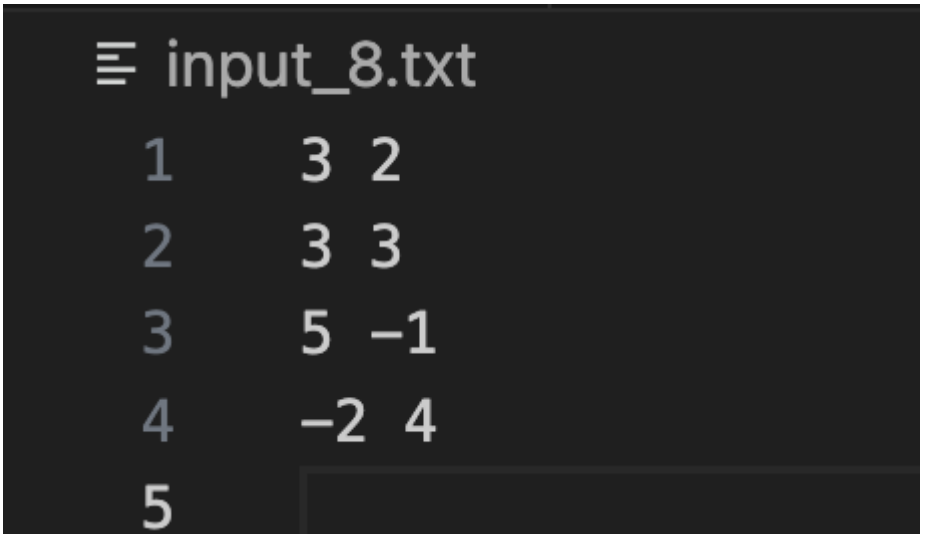
Задача №8. K ближайших точек к началу координат

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 (x1, y1) и (x2, y2) равно p (x1 − x2) 2 + (y1 − y2) 2.

```
def distance(point):
    x, y = point
    return x**2 + y**2  # функция для вычисления расстояния от точки до начала координат

def quick_sort(arr, key):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # выбор опорного элемента
    left = [x for x in arr if key(x) < key(pivot)]  # элементы, меньшие опорного
    middle = [x for x in arr if key(x) == key(pivot)]  # элементы, равные опорному
    right = [x for x in arr if key(x) > key(pivot)]  # элементы, большие опорного
    return quick_sort(left, key) + middle + quick_sort(right, key)  # рекурсивно сортируем
левую, среднюю и правую части

def find_k_closest_points(points, k):
    sorted_points = quick_sort(points, key=distance)  # сортировка точек с использованием
быстрой сортировки
    return sorted_points[:k]  # возвращаем k ближайших точек

with open("input_8.txt", "r") as file:
    n, k = map(int, file.readline().split())  # считываем количество точек и k
    points = [list(map(int, line.split())) for line in file]  # считываем координаты точек

closest_points = find_k_closest_points(points, k)  # находим k ближайших точек

with open("output_8.txt", "w") as file:
    for i, point in enumerate(closest_points):
        file.write(f"[{point[0]}, {point[1]}]")  # записываем координаты точки
        if i < k - 1:
            file.write(", ")  # добавляем запятую, если точка не последняя
```

Чтобы выполнить эту задачу, я построчно искал длины от каждой точки до начала координат и потом, отсортировав массив с длинами, вывела ближайшие точки.





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0013446743545 секунд	20.454624 МБайт
Пример из задачи	0.005694866180419922 секунд	21.5390625 Мбайт
Верхняя граница диапазона значений входных данных из текста задачи	0.05148329944545302 секунд	23.12324МБайт

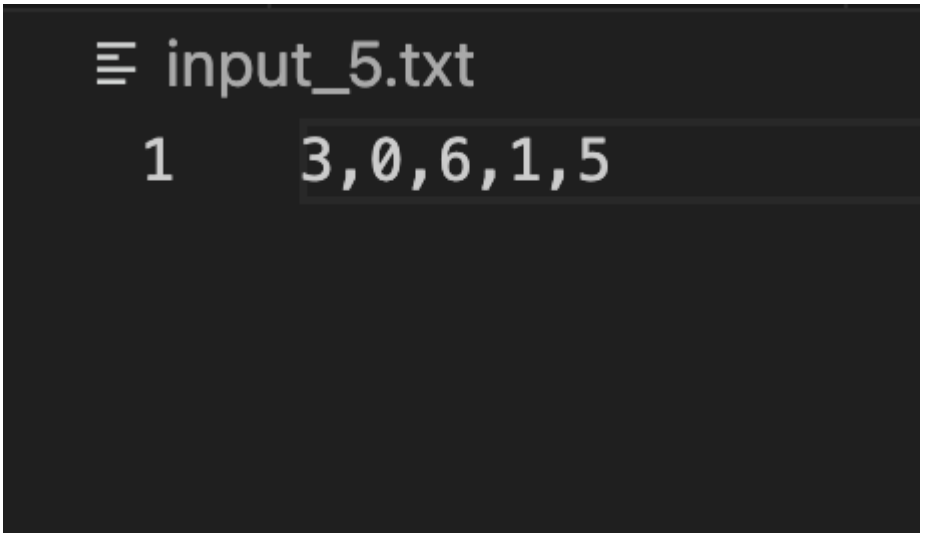
Вывод по задаче: С помощью данной задачи я научился работать с координатами и применять быструю сортировку

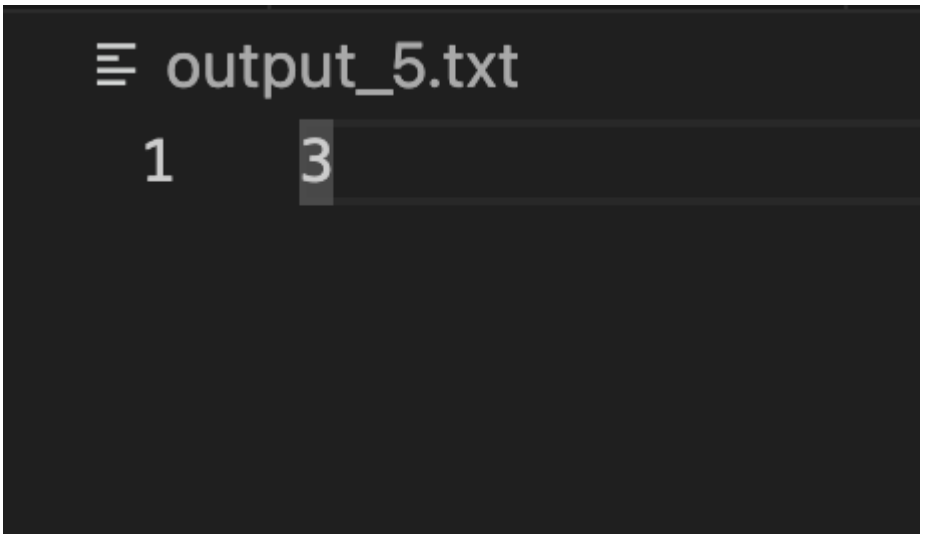
Задача №5. Индекс Хирша

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 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, 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.

```
# Функция для быстрой сортировки
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    p = arr[len(arr) // 2]  # выбираем опорный элемент (середину массива)
    left = [x for x in arr if x < p]  # создаем список элементов меньших опорного
    middle = [x for x in arr if x == p]  # создаем список элементов равных опорному
    right = [x for x in arr if x > p]  # создаем список элементов больших опорного
    return quicksort(left) + middle + quicksort(right)  # рекурсивно сортируем и объединяем
списки

# Функция для вычисления индекса Хирша
def hindex(citations):
```

```
    citations = quicksort(citations)  # сортировка массива цитирований

    n = len(citations)  # получаем количество элементов в массиве
    h = 0  # инициализируем индекс Хирша

    # начинаем итерацию с конца отсортированного массива
    for i in range(n, 0, -1):
        if citations[n - i] >= i:
            # если количество цитирований на позиции n - i (нумерация с 0) больше или равно
i,
            # это означает, что есть как минимум i статей с i или более цитированиями.
            # поэтому устанавливаем индекс Хирша равным i и завершаем цикл.
            h = i
            break

    return h  # возвращаем вычисленный индекс Хирша


# считываем входные данные из input.txt
with open("input_5.txt", "r") as file:
    citations = list(map(int, file.readline().strip().split(',')))  # читаем и преобразуем
строку с цитированиями в список целых чисел

# вычисляем индекс Хирша
h_index = hindex(citations)
# записываем результат в output.txt
with open("output_5.txt", "w") as file:
    file.write(str(h_index))  # записываем значение индекса Хирша в файл output.txt
```





Я отсортировал массив citations по убыванию. Прошлась по отсортированному массиву и нашел первое число, которое меньше или равно его индексу. Это означает, что ученый опубликовал как минимум i статей, на которые ссылаются как минимум i раз каждая. Возвращаю полученное число i как индекс Хирша ученого

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста	0.045783545 секунд	20.475487МБайт

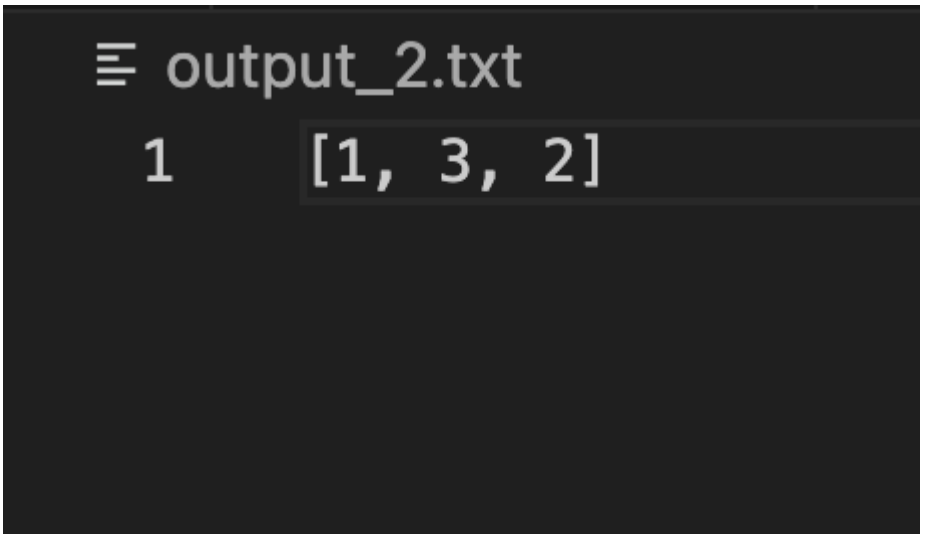
задачи		
Пример из задачи	0.0009961128234863281 секунд	21.5078125 Мбайт
Верхняя граница диапазона значений входных данных из текста задачи	0.0514385895795секунд	22.56524МБайт

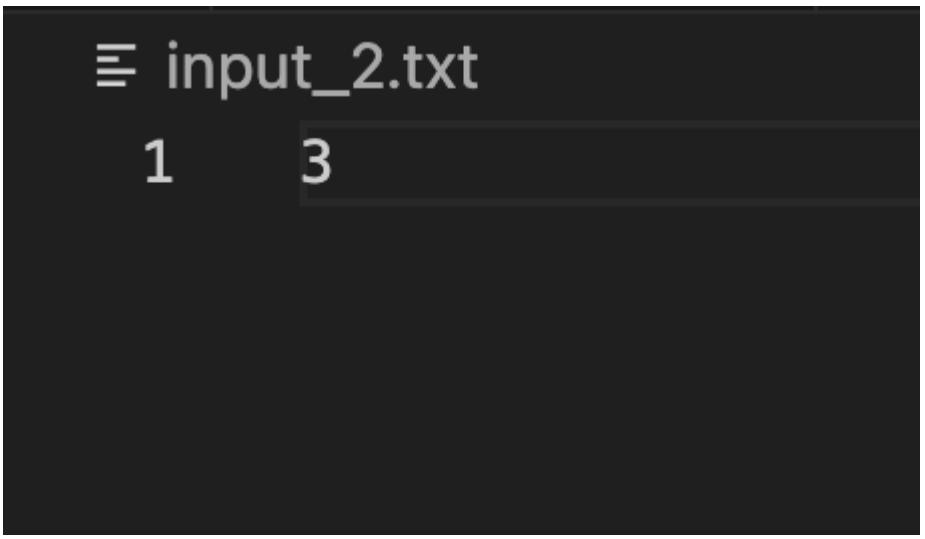
Вывод по задаче:Я научился применять метод Индекса Хирша.

Задача 2. 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 anti_quick_sort(n):

   arr = [0] * n  # создаем массив длиной n, заполненный нулями

   for i in range(n):

       arr[i] = i + 1  # заполняем массив числами от 1 до n



   for i in range(2, n):

       arr[i], arr[i // 2] = arr[i // 2], arr[i]  # меняем элементы массива для
"переворачивания" его структуры

   arr_res = []

   for i in arr:

       arr_res.append(i)  # создаем новый массив с теми же элементами

   return arr_res  # возвращаем перевернутый массив



with open('input_2.txt') as f:

   n = int(f.readline())  # считываем количество элементов в массиве из файла input_2.txt



with open('output_2.txt', 'w') as f:

   f.write(f"{anti_quick_sort(n)}")  # записываем результат выполнения функции
anti_quick_sort в файл output_2.txt
```





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста	0.045783545 секунд	20.475487МБайт

задачи		
Пример из задачи	0.0009961128234863281 секунд	21.5078125 Мбайт
Верхняя граница диапазона значений входных данных из текста задачи	0.0514385895795секунд	22.56524МБайт

Вывод по задачи: скучная задача((

Вывод по лабораторной работе:

Эта работа позволила мне изучить метод быстрой сортировки