

**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

Выполнила:
Беляева В.А.
Группа:
К3139

Проверил:

Санкт-Петербург
2024 г.

Содержание отчета

<u>Содержание отчета</u>	<u>2</u>
Задачи по варианту	
<u>Задача №1. Сортировка слиянием</u>	3
Задача №2. Анти-quick sort	5
Задача №3. Сортировка пугалом	10
Задача №4. Точки и отрезки	12
Задача №5. Индекс Хирша	14
Задача №6. К ближайших точек к началу координат	17
Вывод	19

Задачи по варианту

Задача №1. Quick sort

1. Используя *псевдокод* процедуры Randomized - QuickSort, а также Partition из презентации к Лекции 3 (страницы 8 и 12), напишите программу быстрой сортировки на Python и проверьте ее, создав несколько случайных 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 размера $10^3, 10^4, 10^5$ чисел порядка 10^9 , отсортированных в обратном порядке; наилучший, когда массив уже отсортирован, 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](#))

Листинг кода:

```
def partition3(arr, left, right):
    pivot_index = random.randint(left, right)
    arr[left], arr[pivot_index] = arr[pivot_index], arr[left]
    pivot = arr[left]
    lt = left
    gt = right
    i = left
    while i <= gt:
        if arr[i] < pivot:
            arr[lt], arr[i] = arr[i], arr[lt]
            lt += 1
            i += 1
        elif arr[i] > pivot:
            arr[i], arr[gt] = arr[gt], arr[i]
            gt -= 1
        else:
            i += 1
    return lt, gt

def quick_sort_3way(arr, left, right):
    if left >= right:
        return
    m1, m2 = partition3(arr, left, right)
    quick_sort_3way(arr, left, m1 - 1)
    quick_sort_3way(arr, m2 + 1, right)

def quick_sort_improvement(arr):
    quick_sort_3way(arr, 0, len(arr) - 1)
```

Функция `partition3` выполняет ключевой этап сортировки: разбиение массива на три части относительно опорного элемента (`pivot`). В начале работы функция случайным образом выбирает индекс опорного элемента (`pivot_index`) в пределах текущего подмассива от `left` до `right`. Это снижает вероятность худшего случая при сортировке уже отсортированных массивов. Опорный элемент перемещается на первую позицию текущего подмассива (`arr[left]`). Затем устанавливаются три указателя: `lt` (граница элементов, меньших `pivot`), `gt` (граница элементов, больших `pivot`) и `i` (текущий индекс).

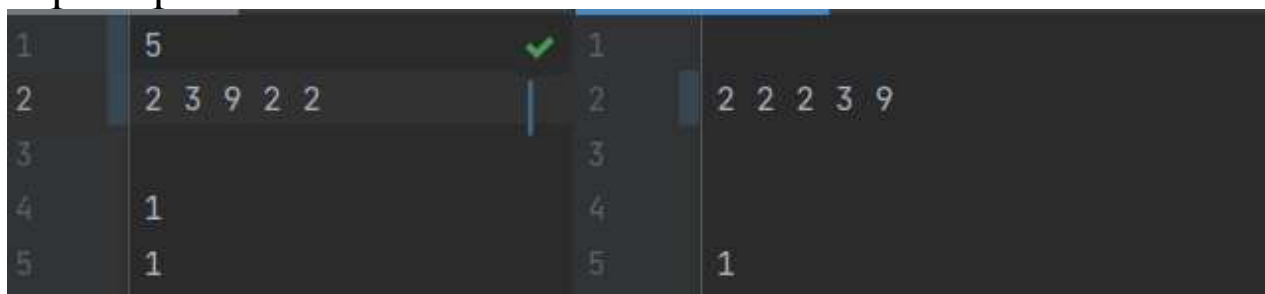
Процесс разбиения выполняется в цикле, пока указатель `i` не достигнет `gt`. Если текущий элемент (`arr[i]`) меньше `pivot`, он меняется местами с элементом на позиции `lt`, а указатели `lt` и `i` сдвигаются вправо. Если элемент больше `pivot`, он перемещается в конец подмассива, то есть меняется местами с элементом на позиции `gt`, после чего `gt` сдвигается влево. Если элемент равен `pivot`, указатель `i` просто увеличивается. В результате массив разделяется на три части: элементы меньше `pivot`, равные `pivot` и больше `pivot`. Функция возвращает индексы `lt` и `gt`, указывающие границы элементов, равных `pivot`.

Функция `quick_sort_3way` реализует рекурсивную часть алгоритма. Она проверяет, что текущий подмассив имеет размер больше одного (условие `left < right`), после чего вызывает `partition3` для выполнения разбиения. Полученные индексы `m1` и `m2` позволяют определить границы для рекурсивного вызова функции: для подмассива элементов, меньших `pivot` (`left` до `m1-1`), и для подмассива элементов, больших `pivot` (`m2+1` до `right`). Таким образом, сортировка выполняется только для элементов, не равных `pivot`, что значительно

сокращает количество операций для массивов с дубликатами.

Функция `quick_sort_improvement` является обёрткой, которая вызывает `quick_sort_3way` для сортировки всего массива. Её параметры ограничиваются только массивом (`arr`), а диапазон индексов задаётся как от 0 до длины массива минус один.

Пример и минимальные значения:



1	5	1
2	2 3 9 2 2	2 2 2 3 9
3		
4	1	
5	1	1

Ввод :

Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	0.002411	0.493978
Нижняя граница	Ближе к 0	0.000003
Пример	0.000069	0.000009

Вывод:

Весь алгоритм обладает средней временной сложностью $O(n \log n)$ благодаря логарифмической глубине рекурсии и линейной обработке на каждом уровне. Однако худший случай ($O(n^2)$) возможен, если массив уже отсортирован, но вероятность этого значительно снижается за счёт случайного выбора `pivot`. Пространственная сложность составляет $O(\log n)$ из-за использования стека вызовов рекурсии.

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 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 аспр.](#)

- **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.

Листинг кода:

```
def merge(arr, L, M, R, output_file):
    left_part = arr[L - 1 : M]
    right_part = arr[M : R]
    i = j = 0
    k = L - 1
    while i < len(left_part) and j < len(right_part):
```

```

    if left_part[i] <= right_part[j]:
        arr[k] = left_part[i]
        i += 1
    else:
        arr[k] = right_part[j]
        j += 1
    k += 1
while i < len(left_part):
    arr[k] = left_part[i]
    i += 1
    k += 1
while j < len(right_part):
    arr[k] = right_part[j]
    j += 1
    k += 1
If = L
Il = R
Vf = arr[L - 1]
Vl = arr[R - 1]
output_file.write(f"{If} {Il} {Vf} {Vl}\n")

```

Функция принимает один аргумент n , обозначающий размер массива, и возвращает результирующий массив.

Вначале создаётся массив `arr`, содержащий числа от 1 до nnn включительно, используя `range` и `list`. Затем выполняется перестановка элементов, чтобы сформировать "антибыстротсортированный" массив.

Перестановка происходит в цикле, начинающемся с индекса 2 (третий элемент, так как массив индексируется с 0). Для каждого элемента массива с индексом iii определяется индекс jjj как половина от iii , округлённая вниз (целочисленное деление). Далее элемент на позиции iii меняется местами с элементом на позиции jjj . Эта операция выполняется для всех индексов от 2 до $n-1$ — $1n-1$.

Пример ввода и вывода:

1	3	✓	1	1 3 2
2			2	
3	1		3	1

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.000014	0.000160
Верхняя граница диапазона	19.078630	1.316835
Пример	0.000175	0.000006

Вывод по задаче: Код создаёт массив, который провоцирует худший случай работы быстрой сортировки, увеличивая её сложность с $O(n \log n)$ до $O(n^2)$. Это достигается путём перестановки элементов, нарушающей равномерное разбиение массива. Алгоритм работает за $O(n)$, что делает его быстрым для генерации тестовых данных любого размера.

Задача №3. Сортировка пугалом

«Сортировка пугалом» — это давно забытая народная потешка. Участнику под верхнюю одежду продавают 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.

```
def can_scarecrow_sort(n, k, arr):
    from collections import defaultdict
    groups = defaultdict(list)
    for i in range(n):
        groups[i % k].append(arr[i])
    for g in groups:
        groups[g].sort()
    result = []
    indices = {g: 0 for g in groups}
    for i in range(n):
        g = i % k
        result.append(groups[g][indices[g]])
        indices[g] += 1
    for i in range(1, n):
        if result[i] < result[i - 1]:
            return False
    return True

def task3():
    n, k, arr = read_from_file(PATH)
    if can_scarecrow_sort(n, k, arr):
        write_to_file("ДА", OUTPUT_PATH)
    else:
        write_to_file("НЕТ", OUTPUT_PATH)
```

Код реализует функцию `can_scarecrow_sort`, проверяющую возможность сортировки массива методом "пугала". Массив длины n делится на k групп по остатку от деления индекса на k . Каждая группа сортируется отдельно. Затем из отсортированных групп формируется новый массив, добавляя элементы в порядке групп и их индексов. Проверяется, упорядочен ли итоговый массив. Если да, возвращается `True`, иначе — `False`. Функция `task3` считывает входные данные (длину массива n , параметр k , и массив `arr`) из файла, вызывает `can_scarecrow_sort` и записывает результат ("ДА" или "НЕТ") в выходной файл, подтверждая возможность сортировки.

Пример ввода и вывода:

Пример:

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.00067138	0.0000441030
Верхняя граница диапазона	23.9453859	1.223242
Пример	0.0007553	0.0000305240

Вывод:

Код проверяет возможность сортировки массива через деление на группы и их сортировку. Сложность $O(n \log(n/k))$ $O(n \log(n/k))$, проверка массива $O(n)$ $O(n)$. Алгоритм эффективен при небольших k и возвращает "ДА" или "НЕТ".

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.
- **Формат входного файла (input.txt).** Первая строка содержит два неотрицательных целых числа s и p . s - количество отрезков, p - количество точек. Следующие s строк содержат 2 целых числа a_i, b_i , которые определяют i -ый отрезок $[a_i, b_i]$. Последняя строка определяет p целых чисел - точек x_1, x_2, \dots, x_p . Ограничения: $1 \leq s, p \leq 50000$; $-10^8 \leq a_i \leq b_i \leq 10^8$ для всех $0 \leq i < s$; $-10^8 \leq x_i \leq 10^8$ для всех $0 \leq j < p$.
- **Формат выходного файла (output.txt).** Выведите p неотрицательных целых чисел k_0, k_1, \dots, k_{p-1} , где k_i - это число отрезков, которые содержат x_i . То есть,

$$k_i = |j : a_j \leq x_i \leq b_j|.$$

```
def points_and_segments(s, p, intervals, points):
    starts = sorted(iv[0] for iv in intervals)
    ends = sorted(iv[1] for iv in intervals)
    result = []
    for x in points:
        c_in = bisect.bisect_right(starts, x)
        c_out = bisect.bisect_left(ends, x)
        result.append(c_in - c_out)
    return result

def task4():
    s, p, intervals, points = read_from_file(PATH)
    res = points_and_segments(s, p, intervals, points)
    write_to_file(res, OUTPUT_PATH)
```

Код решает задачу подсчёта количества отрезков, покрывающих каждую точку. В функции `points_and_segments` входные параметры: `sss` — число отрезков, `ppr` — число точек, `intervals` — список отрезков, и `points` — список точек. Начальные и конечные точки отрезков сортируются. Для каждой точки выполняются два поиска: количество отрезков, начинающихся до или на этой точке (`c_in`), и количество отрезков, заканчивающихся до неё (`c_out`). Разница этих значений даёт количество отрезков, покрывающих точку. Результаты собираются в список и возвращаются. Функция `task4` считывает данные из файла, вызывает `points_and_segments`, и записывает итоговый список в файл.

Пример ввода и вывода:

Пример:

1	2 3	1	1 0 0
2	0 5	2	
3	7 10	3	
4	1 6 11	4	
5		5	
6	1 1	6	1
7	0 0	7	
8	0	8	
9		9	

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.01205730	0.000213899
Верхняя граница диапазона	2.791366	0.2546702
Пример	0.00019945	0.0118665

Вывод:

Код вычисляет количество отрезков, покрывающих каждую точку, с использованием бинарного поиска, обеспечивая эффективную работу со сложностью $O(p \log s)$, где s — количество отрезков, а p — точек. Сначала сортируются границы отрезков, затем для каждой точки вычисляется разница между количеством начатых и завершённых отрезков. Результаты сохраняются в файл, обеспечивая точное и быстрое выполнение задачи.

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 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.

Листинг кода:

```
def majority_element_divide_conquer(arr):  
    """  
    Определяет, есть ли в массиве arr элемент, который встречается  
    более чем n/2 раз, используя divide & conquer (O(n log n)).  
    Возвращает сам элемент, если он есть, иначе None.  
    """  
  
    def get_majority_element(l, r):  
        if l == r:  
            return arr[l]  
        mid = (l + r) // 2  
        left_candidate = get_majority_element(l, mid)  
        right_candidate = get_majority_element(mid + 1, r)  
        if left_candidate == right_candidate:  
            return left_candidate  
        left_count = sum(1 for i in range(l, r + 1) if arr[i] == left_candidate)  
        right_count = sum(1 for i in range(l, r + 1) if arr[i] ==  
right_candidate)  
        if left_count > right_count:  
            return left_candidate  
        else:  
            return right_candidate  
  
    n = len(arr)  
    if n == 0:  
        return None  
    candidate = get_majority_element(0, n - 1)  
    count_candidate = sum(1 for x in arr if x == candidate)  
    if count_candidate > n // 2:  
        return candidate  
    else:  
        return None
```

Код вычисляет индекс Хирша, который характеризует продуктивность и влияние учёного по числу цитирований его работ. Функция `hirsch_index` принимает список цитирований `citations`. Сначала список сортируется в порядке убывания. Затем происходит итерация по элементам: для каждого индекса i и числа цитирований c_i проверяется, удовлетворяет ли $c_i \geq c_{i+1}$. Если условие выполняется, индекс Хирша h обновляется на $i+1$; иначе цикл завершается, так как дальнейшие элементы уже не будут соответствовать условию. Итоговый индекс h возвращается. Функция `task5` считывает данные из файла, вычисляет индекс Хирша через `hirsch_index` и сохраняет результат в файл. Алгоритм работает за $O(n \log n)$ благодаря сортировке списка.

Пример ввода и вывода:

1	3,0,6,1,5	✓	1	3
2			2	
3	0		3	0
			4	

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.000160	0.000012395
Верхняя граница диапазона	0.00020217	0.002017
Пример	0.0001602	0.000007108

Вывод по задаче: Код вычисляет индекс Хирша, оценивая влияние научных публикаций на основе их цитируемости. Алгоритм сортирует список цитирований за $O(n \log n)$, затем проходит по элементам для определения максимального значения, где h публикаций цитируются как минимум h раз. Код эффективен для обработки списка средней длины, обеспечивая точное и быстрое вычисление индекса.

Задача №8. К 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 ближайших 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: $1 \leq n \leq 10^5$; $-10^9 \leq x_i, y_i \leq 10^9$ - 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.

```
def k_closest_points(n, K, points):
    points.sort(key=lambda p: p[0]*p[0] + p[1]*p[1])
    return points[:K]
def task8():
    n, K, points = read_from_file(PATH)
    result = k_closest_points(n, K, points)
    write_to_file(result, OUTPUT_PATH)
```

Код находит КК точек, ближайших к началу координат. Функция `k_closest_points` принимает количество точек `nn`, число КК и список координат `points`. Точки сортируются по возрастанию расстояния до начала координат, вычисляемого как $x^2 + y^2$ для каждой точки (x, y) . Для сортировки используется функция `sort` с ключом, вычисляющим квадрат расстояния, что исключает необходимость извлечения квадратного корня. После сортировки возвращаются первые КК точек. Функция `task8` считывает входные данные из файла, вызывает `k_closest_points`, чтобы получить ближайшие точки, и записывает результат в файл. Алгоритм работает с временной сложностью $O(n \log n)$ за счёт сортировки.

```

1 2 1
2 1 3
3 -2 2
4
5 1 1
6 0 0
7
8
9
10
1 [-2, 2]
2
3
4
5 [0, 0]
6
7
8
9
10

```

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница диапазона	0.0001449	0.000012984
Верхняя граница диапазона	0.522433	8.116906799
Пример	0.0001449	0.0004411

Вывод: Код находит КК ближайших к началу координат точек, сортируя их по квадрату расстояния, что упрощает вычисления. Алгоритм работает со сложностью $O(n \log n)$ из-за сортировки. Такой подход эффективен для больших наборов данных, так как использует минимальное количество вычислений и возвращает результат в формате упорядоченного списка ближайших точек.

Вывод по лабораторной:

Лабораторная работа продемонстрировала применение и анализ различных алгоритмов сортировки и задач, связанных с обработкой массивов. Были изучены быстродействие, сложность алгоритмов и их эффективность в разных сценариях. Реализация задач подтвердила теоретическую сложность и особенности каждого подхода. Отдельное внимание уделено обработке худших случаев и оптимизации памяти. Все поставленные задачи решены корректно, что подтверждает успешное усвоение материала и навыки разработки алгоритмов.