

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 11

Выполнил:  
Кузнецов А.Г.  
К3140

Проверила:  
Артамонова В.Е.

Санкт-Петербург  
2022 г.

## **Содержание отчета**

<b>Содержание отчета</b>	<b>2</b>
<b>Задачи по варианту</b>	<b>3</b>
Задача №1. Улучшение Quick sort	3
Задача №7. Цифровая сортировка	12
Задача №8. К ближайших точек к началу координат	13
<b>Вывод</b>	<b>15</b>

## Задачи по варианту

### Задача №1. 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 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)

```
import sys
sys.setrecursionlimit(1500)
import random

def Partition(A, l, r):
    # Опорный элемент
    x = A[l]
    m = l

    for i in range(l+1, r+1):
        if A[i] <= x:
            m += 1
            (A[m], A[i]) = (A[i], A[m])

    # Постановка опорного элемента на своё финальное положение
    (A[l], A[m]) = (A[m], A[l])

    # Вывод точки, где выполнено разделение
    return m
```

```

# Функция для нахождения опорного элемента и разделения массива на две
части
def Partition_QuickSort(A, l, r):
    if l < r:
        # Нахождение опорной точки
        m = Partition(A, l, r)
        # Рекурсивный вызов функции слева от опорной точки
        Partition_QuickSort(A, l, m - 1)
        # Рекурсивный вызов функции справа от опорной точки
        Partition_QuickSort(A, m + 1, r)
    return A

def Randomized_QuickSort(A, l, r):
    if l < r:
        k=random.randint(l,r)
        A[l],A[k] = A[k], A[l]
        # Нахождение опорной точки
        m = Partition(A,l,r)
        # Рекурсивный вызов функции слева от опорной точки
        Randomized_QuickSort(A, l, m-1)
        # Рекурсивный вызов функции справа от опорной точки
        Randomized_QuickSort(A, m+1, l)
    return A

with open('input-1.txt','r') as f:
    n = int(f.readline())
    A = [int(x) for x in f.readline().split(' ')]
if 1 <= n <= (10**4) and not(any([abs(x)>10**9 for x in A])):
    with open('output-1.1.txt','w') as f:
        f.write(' '.join(map(str,Partition_QuickSort(A, 0, n -
1))))
if 1 <= n <= (10**4) and not(any([abs(x)>10**9 for x in A])):
    with open('output-1.2.txt', 'w') as f:
        f.write(' '.join(map(str, Randomized_QuickSort(A, 0, n - 1))))

```

Были реализованы способы сортировки Randomized – QuickSort и Partition – QuickSort. На вход подаётся 2 числа: длина массива и сам массив. Далее проверяем подходят ли значения заданному условию, если это так, то запускаем функции Partition\_QuickSort и Randomized\_QuickSort, которые будут запускать функцию Partition, в которой будет разбиение массива по опорной точке, далее мы запускаем функции Partition\_QuickSort и Randomized\_QuickSort по левой и правой части массива от опорного элемента. Так проходит до тех пор, пока массив не отсортируется

	Наихудший случай при n=1000	Наихудший случай при n=10000
Randomized QuickSort	0.03116020000015851 секунд	0.0439712999996118 секунд

Partition QuickSort	0.028193400001327973 секунд	0.06145309999919846 секунд
---------------------	-----------------------------	----------------------------

	Средний случай при n=1000	Средний случай при n=10000
Randomized QuickSort	0.03159179999966 секунд	0.041033599998627324 секунд
Partition QuickSort	0.030436499999268563 секунд	0.0577212000007421 секунд

	Наилучший случай при n=1000	Наилучший случай при n=10000
Randomized QuickSort	0.040379699999903096 секунд	0.03619719999915105 секунд
Partition QuickSort	0.04237259999899834 секунд	-

2. Основное задание. Цель задачи - переделать данную реализацию рандомизированного алгоритма быстрой сортировки, чтобы она работала быстро даже с последовательностями, содержащими много одинаковых элементов. Чтобы заставить алгоритм быстрой сортировки эффективно обрабатывать последовательности с несколькими уникальными элементами, нужно заменить двухстороннее разделение на трехстороннее (смотри в Лекции 3 слайд 17). То есть ваша новая процедура разделения должна разбить массив на три части: •  $A[k] < x$  для всех  $\ell + 1 \leq k \leq m_1 - 1$  •  $A[k] = x$  для всех  $m_1 \leq k \leq m_2$  •  $A[k] > x$  для всех  $m_2 + 1 \leq k \leq r$  • Формат входного и выходного файла аналогичен п.1. • Аналогично п.1 этого задания сравните Randomized-QuickSort +c Partition и ее с Partition3 на сетях случайных данных, в которых содержатся всего несколько уникальных элементов при  $n = 103, 104, 105$ . Что быстрее, Randomized-QuickSort +c Partition3 или Merge-Sort?

```
import random

def Partition3(A, l, r):
```

```

# Опорный элемент
x = A[l]
m1, m2 = l, l

while m2 <= r:

    if A[m2] < x:
        (A[m2], A[m1]) = (A[m1], A[m2])
        m1 += 1
        m2 += 1

    elif A[m2] > x:
        (A[m2], A[r]) = (A[r], A[m2])
        r -= 1
    else:
        m2 += 1

return (m1, m2)

# Функция для нахождения опорного элемента и разделения
массива на три части
def Randomized_QuickSort3(A, l, r):
    if l < r:
        k = random.randint(l, r)
        A[l], A[k] = A[k], A[l]
        (m1, m2) = Partition3(A, l, r)
        Randomized_QuickSort3(A, l, m1-1)
        Randomized_QuickSort3(A, m2, r)
    return A

with open('input-2.txt', 'r') as f:
    n = int(f.readline())
    A = [int(x) for x in f.readline().split(' ')]
    if 1 <= n <= (10**4) and not (any([abs(x) > 10**9 for x in
A])):
        with open('output-2.txt', 'w') as f:
            f.write('
'.join(map(str, Randomized_QuickSort3(A, 0, n - 1))))

```

Был реализован способ сортировки Partitio3. На вход подаётся 2 числа: длина массива и сам массив. Далее проверяем подходят ли значения заданному условию, если это так, то запускаем функции Randomized\_QuickSort3, которые будут запускать функцию Partition3, в которой будет разбиение массива по двум опорным точкам, далее мы

запускаем функции Randomized\_QuickSort3 по левой от m1 и по правой части от m2 массива. Так проходит до тех пор, пока массив не отсортируется

	Наихудший случай при n=1000	Наихудший случай при n=10000
Randomized QuickSort	0.03116020000015851 секунд	0.0439712999996118 секунд
Partition QuickSort	0.028193400001327973 секунд	0.06145309999919846 секунд
Partition3	0.02679119999993418 секунд	0.05389199999990524 секунд
Merge-Sort	0.027307000000291737 секунд	0.13962229999924602 секунд

	Средний случай при n=1000	Средний случай при n=10000
Randomized QuickSort	0.03159179999966 секунд	0.041033599998627324 секунд
Partition QuickSort	0.030436499999268563 секунд	0.0577212000007421 секунд
Partition3	0.027979499998764368 секунд	0.057284700000309385 секунд
Merge-Sort	0.03467860000091605 секунд	0.17244229999960226 секунд

	Наилучший случай при n=1000	Наилучший случай при n=10000
Randomized QuickSort	0.040379699999903096 секунд	0.03619719999915105 секунд
Partition QuickSort	0.04237259999899834 секунд	-
Partition3	0.027334100001098705	0.05532189999939874

	секунд	секунд
Merge-Sort	0.03598039999997127 секунд	0.13030860000071698 секунд

The screenshot shows a terminal window with two files: 'input-2.txt' and 'output-2.txt'. The 'input-2.txt' file contains two lines: '1 5' and '2 2 3 9 2 2'. The 'output-2.txt' file contains one line: '1 2 2 2 3 9'.

File	Line 1	Line 2
input-2.txt	1 5	2 2 3 9 2 2
output-2.txt	1 2 2 2 3 9	

Вывод по задаче: Были реализованы методы сортировки Randomized Quick Sort, Partition Quick Sort и Partition3. Было рассмотрено их время действия, а также было проведено сравнение с Merge Sort, после которого стало ясно, что Partition3 является самым быстрым методом сортировки среди данных методов.

### Задача №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  $k$  ( $1 \leq n, k \leq 105$ ) — число матрёшек и размах рук. Во второй строчке содержится  $n$  целых чисел, которые по модулю не превосходят 109 — 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.



```

1 def Quick_Sort(left, right):
2     key = a[(left + right) // 2][0]
3     i = left
4     j = right
5     while True:
6         while a[i][0] < key:
7             i += 1
8         while a[j][0] > key:
9             j -= 1
10        if i <= j:
11            a[i], a[j] = a[j], a[i]
12            i += 1
13            j -= 1
14        if i > j:
15            break
16    if left < j:
17        Quick_Sort(left, j)
18    if i < right:
19        Quick_Sort(i, right)
20
21
22 def ver(m):
23     if m == 1:
24         return "ДА"
25     for i in range(n):
26         k = 0
27         j = 0
28         while j < len(A[a[i][0]]):
29             if abs(i - A[a[i][0]][j]) % m == 0:
30                 k += 1
31                 A[a[i][0]].pop(j)
32
33                 j += 1
34             if k == 0:
35                 return "НЕТ"
36
37     return "ДА"
38
39 with open('input.txt', 'r') as f:
40     n, m = list(map(int, f.readline().strip().split()))
41     a = [int(x) for x in f.readline().split(' ')]
42     A = {}
43     print(n,m)
44     for i in range(n):
45         a[i] = [int(a[i]), i]
46         A[a[i][0]] = A.get(a[i][0], [])
47         A[a[i][0]].append(a[i][1])
48     Quick_Sort(0, len(a) - 1)
49
50 with open('output.txt', 'w') as f:
51     f.write(ver(m))

```

На вход из файла input.txt получаем число матрёшек, размах рук и размеры матрёшек. Далее в список размеров матрёшек добавляем их индексы и

создаем словарь, где будут храниться размеры матрешек как ключ и их индексы как значения. Сортируем список a, а затем проверяем может ли участник расположить матрёшки по неубыванию размера. Ответ записываем в output.txt

input.txt	
1	3 2
2	2 1 3
output.txt	
1	НЕТ

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

Вывод по задаче: В ходе работы над третьей задачей была реализована сортировка пугалом с помощью quick sort'a

## 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.

- 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:  $1 \leq n \leq 5000$ ,  $0 \leq citations[i] \leq 1000$

```

1 def heapify(a, n, i):
2     smallest = i
3     left = 2 * i + 1
4     right = 2 * i + 2
5
6     if left < n and a[left] < a[smallest]:
7         smallest = left
8

```

```

9     if right < n and a[right] < a[smallest]:
10         smallest = right
11
12     if smallest != i:
13         (a[i], a[smallest]) = (a[smallest], a[i])
14
15         heapify(a, n, smallest)
16
17
18 def heapSort(a):
19     n = len(a)
20
21     for i in range(n // 2 - 1, -1, -1):
22         heapify(a, n, i)
23
24     for i in range(n - 1, -1, -1):
25         (a[0], a[i]) = (a[i], a[0])
26
27         heapify(a, i, 0)
28
29
30 def h_index(a):
31     h = 0
32     n = len(a)
33     for i in range(n):
34         if a[i] >= (i + 1):
35             h += 1
36     return h
37
38
39 with open('input.txt', 'r') as f:
40     a = [int(x) for x in f.readline().split(',')]
41     heapSort(a)
42
43 with open('output.txt', 'w') as f:
44     f.write(str(h_index(a)))

```

На вход из файла input.txt получаем строку citations, которая содержит n целых чисел. Чтобы узнать индекс Хирша мы сортируем публикации по цитируемости и, двигаясь по порядку, доходим до последней публикации в списке, у которой цитируемость будет выше ее порядкового номера. Порядковый номер этой публикации и будет равняться индексу Хирша. Выводит индекс Хирша в файл output.txt

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

input.txt	×
1	1,3,1
output.txt	×
1	1

Вывод по задаче: В ходе работы над пятой задачей был проанализирован и реализован способ нахождения индекса Хирша

## Задача №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. 7 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.

```
1 def RadixSort(A,n,m,k,indx):
2     new_k=0
3     for i in range(n-1,-1,-1):
4         if new_k==k:
5             break
6         for j in range(m-1):
7             if A[i][j] > A[i][j + 1]:
8                 indx[j],indx[j+1]=indx[j+1],indx[j]
9                 for x in range(i+1):
10                     A[x][j], A[x][j+1] = A[x][j+1], A[x][j]
11         new_k+=1
12     return A,indx
13
14 with open('input.txt', 'r') as f:
15     # n - число строк; m - длина строк; k - число фаз цифровой
16 сортировки
17     n, m, k = list(map(int, f.readline().strip().split()))
18     A = []
19     indx=[]
20     for i in range(n):
21         A.append(list(f.readline().strip()))
22         indx.append(i+1)
23
```

```

24 if (1 <= n <= 10**4) and (1 <= k <= m <= 10**6) and (n*m<=5*10**7):
25     with open('output.txt', 'w') as f:
        f.write(' '.join(map(str, RadixSort(A, n, m, k, indx)[-
1])))

```

Получаем на вход данные из input.txt, затем добавляем новый массив с индексами значений, чтобы в дальнейшем вывести ответ из индексов. Проверяем подходят ли значения заданному условию, если это так, то сортируем строки по n-ному элементу, начиная с конца. Далее выводим ответ в виде конечных индексов в файл output.txt

input.txt	output.txt	input.txt	output.txt	input.txt	output.txt
1 3 3 1	1 2 3 1	1 3 3 2	1 3 2 1	1 3 3 3	1 2 3 1
2 bab		2 bab		2 bab	
3 bba		3 bba		3 bba	
4 baa		4 baa		4 baa	

Вывод по задаче: В ходе работы над седьмой задачей был реализована цифровая сортировка в вертикальной форме

## Задача №8. К ближайших точек к началу координат

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

- Цель. Заданы n точек на поверхности, найти К 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 корню суммы:  $(x_1 - x_2)^2 + (y_1 - y_2)^2$
- Формат ввода или входного файла (input.txt). Первая строка содержит n - общее количество точек на плоскости и через пробел К - количество ближайший точек к началу координат, которые надо найти. Каждая следующая из n строк содержит 2 целых числа  $x_i, y_i$ , определяющие точку  $(x_i, y_i)$ . Ограничения:  $1 \leq n \leq 10^5$ ;  $-10^9 \leq x_i, y_i \leq 10^9$  - целые числа.
- Формат выхода или выходного файла (output.txt). Выведите К ближайших точек к началу координат в строку 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 Partition3(A, l, r):
    x = A[l][0]
    m1, m2 = l, l
    while m2 <= r:
        if A[m2][0] < x:
            (A[m2], A[m1]) = (A[m1], A[m2])
            m1 += 1
            m2 += 1
        elif A[m2][0] > x:
            (A[m2], A[r]) = (A[r], A[m2])
            r -= 1
        else:
            m2 += 1
    return (m1, m2)

def Randomized_QuickSort3(A, l, r):
    if l < r:
        (m1, m2) = Partition3(A, l, r)
        Randomized_QuickSort3(A, l, m1-1)
        Randomized_QuickSort3(A, m2, r)
    return A

with open('input.txt', 'r') as f:
    # n - количество точек; k - количество точек k на
    n, k = list(map(int, f.readline().strip().split()))
    coords = []
    result = []
    for i in range(n):
        coords.append([int(x) for x in f.readline().split()])
        coords[i].insert(0, ((coords[i][0])**2 +
    (coords[i][1])**2)**0.5)

    if (1 <= n <= 10**5) and not(any([abs(coords[x][1])>10**9 for x in
    range(len(coords))])) and not(any([abs(coords[y][2])>10**9 for y in
    range(len(coords))])):
        Randomized_QuickSort3(coords, 0, n - 1)
        for i in range(k):
            result.append(coords[i][1:])

    with open('output.txt', 'w') as f:
        f.write(','.join(map(str, result)))

```

На вход из файла input.txt получаем количество точек на поверхности и количество ближайших точек, которое нужно найти. В массив с координатами добавляем расстояние от начала координат. Далее проверяем подходят ли значения заданному условию, если это так, то сортируем массив по расстоянию с помощью метода сортировки Partition3, после чего записываем необходимое количество ближайших точек в output.txt

input.txt	
1	3 2
2	3 3
3	5 -1
4	-2 4
output.txt	
1	[3, 3], [-2, 4]

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

Вывод по задаче: В ходе работы над восьмой задачей был реализован метод нахождения k ближайших точек к началу координат с помощью метода сортировки Partition3

## Вывод

В ходе выполнения третьей лабораторной работы были реализованы методы сортировки улучшенного quick sort и radix sort, а также было выполнено нахождение k ближайших точек к началу координат