

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: Быстрая сортировка, сортировки за линейное время
Вариант 27

Выполнил:
Филиппов А.Э.
К3139

Проверила:
Артамонова В.Е.

Санкт-Петербург
2022 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3-15
Задача №1. Улучшение Quick sort	3-6
Задача №4. Точки и отрезки	7-11
Задача №5. Индекс Хирша	11-15
Дополнительные задачи	16-49
Задача №2. Анти-quick sort	16-19
Задача №3. Сортировка пугалом	19-23
Задача №6. Сортировка целых чисел	23-28
Задача №7. Цифровая сортировка	28-32
Задача №8. К ближайших точек к началу координат	32-35
Задача №9. Ближайшие точки	35-41
Задача №10.1	42-44
Задача №10.2	44-46
Задача №10.3	46-49
Вывод	50

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 ($1 \leq n \leq 104$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, *по модулю* не превосходящих 109.
- **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 103, 104, 105 чисел порядка 109, отсортированных в об ратном 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)

2. **Основное задание.** Цель задачи - переделать данную реализацию рандо мизированного 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 заме нить 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 $\ell + 1 \leq k \leq m1 - 1$
- $A[k] = x$ для всех $m1 \leq k \leq m2$
- $A[k] > x$ для всех $m2 + 1 \leq k \leq r$
- Формат входного и выходного файла аналогичен п.1.
- Аналогично п.1 этого задания сравните Randomized-QuickSort +с 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 +с Partition3 или Merge-Sort?
- Пример:

input.txt	output.txt
5 2 3 9 2 2	2 2 2 3 9

Код:

```
import random
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()

def partition(l, r):
    pivot = a[l]
    j = l
    c = r
    i = l + 1
    while i <= c:
        if a[i] < pivot:
            j += 1
            a[i], a[j] = a[j], a[i]
        elif a[i] > pivot:
            a[c], a[i] = a[i], a[c]
            i -= 1
```

```

        c -= 1
        i += 1

    a[l], a[j] = a[j], a[l]
    return j, c

def quickSort(l, r):
    if l < r:
        k = random.randint(l, r)
        a[l], a[k] = a[k], a[l]
        m, c = partition(l, r)
        quickSort(l, m-1)
        quickSort(c+1, r)

n = int(lines[0])
a = list(map(int, lines[1].split()))
quickSort(0, len(a)-1)

open("output.txt", 'w').write(" ".join(map(str, a)))

print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
print("Использование памяти: " +
      str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Для решения данной задачи использовалось деление массива на 3 раздела. Первый — меньше опорного элемента, второй — равный опорному элементу, третий — больше. Оптимизация состоит в том, чтобы не производить перестановки во втором разделе.

Результат работы кода:

```

1 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84
86 88 90 92 94 96 98 100 102 104 13 53 27 55 7 57 29 59 15 61 31 63 2 65 33 67 17 69 35 71 9 73 37 75 19 77 39 79 5 81 41
83 21 85 43 87 11 89 45 91 23 93 47 95 3 97 49 99 25 101 51 103

```

```
5
2 3 9 2 2
RMAL input.txt
put.txt" 2L, 14B writ
2 2 2 3 9
RMAL output.txt
```

```
1
1
RMAL input.txt
put.txt" 2L, 4B written
1
RMAL output.txt
```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.000265594999291352	13939
Пример из задачи	0.00027472999863675795	13978
Верхняя граница диапазона значений входных данных из текста задачи	0.0008830130009300774	25913

Вывод по задаче: Быструю сортировку можно ускорить используя оптимизацию для равных чисел. Метод быстрой сортировки быстрее merge sort для небольших массивов, для большого числа элементов эффективнее сортировка слиянием.

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 после довательных 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: $1 \leq s, p \leq 50000$; $-108 \leq a_i \leq b_i \leq 108$ для всех $0 \leq i < s$; $-108 \leq x_i \leq 108$ 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,
$$k_i = |j : a_j \leq x_i \leq b_j|.$$
- **Пример 1.**

input.txt	output.txt
2 3 0 5 7 10 1 6 11	1 0 0

Здесь, у нас есть 2 отрезка и 2 точки. Первая точка принадлежит интервалу $[0, 5]$, остальные точки не принадлежат ни одному из данных интервалов.

• Пример 2.

input.txt	output.txt
1 3 -10 10 -100 100 0	0 0 1

• Пример 3.

input.txt	output.txt
3 2 0 5 -3 2 7 10 1 6	2 0

Код:

```
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()

def quick_sort(l, r):
    i = l
    j = r
    pivot = arr[(i+j)//2][0]
    while i <= j:
        while arr[i][0] < pivot:
            i += 1
        while arr[j][0] > pivot:
            j -= 1
        if i <= j:
            arr[i], arr[j] = arr[j], arr[i]
```



```

        i += 1
        j -= 1
    if j > l:
        quick_sort(l, j)
    if i < r:
        quick_sort(i, r)

s, p = map(int, lines[0].split())
arr = []

for i in range(s):
    a, b = map(int, lines[i+1].split())
    arr.append([a, "b"])
    arr.append([b, "e"])
d = list(map(int, lines[s+1].split()))

for i in range(p):
    arr.append([d[i], "el"])

quick_sort(0, s*2+p-1)

ans = {}
count = 0

for i in range(len(arr)):
    if arr[i][1] == "b":
        count += 1
    elif arr[i][1] == "e":
        count -= 1
    else:
        ans[arr[i][0]] = count

out = []
for i in range(p):
    out.append(ans[d[i]])

open("output.txt", 'w').write(" ".join(map(str, out)))

print("Время выполнения: " + str(time.perf_counter() - t_start) + " секунд")
print("Использование памяти: " + str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Основная идея решения — отсортировать значения точек и отрезков заранее, вместо того, чтобы каждый раз искать для точки нужные отрезки. В отличие от наивного решения ($O(n^2)$) сложность равна $O(n(\log n + 1))$

Результат работы кода:

```

2 3
0 5
7 10
1 6 11

```

ORMAL input.txt
put.txt" 4L, 23B written

0 0

ORMAL output.txt

```

2 1 3
1 -10 10
-100 100 0

```

ORMAL input.txt
put.txt" 3L, 24B written

0 0 1

ORMAL output.txt

```

4 3 2
3 0 5
2 -3 2
1 7 10
1 6

```

NORMAL input.txt
input.txt" 5L, 26B wr

2 0

NORMAL output.txt

```

2 1 1
1 1 2
0

```

ORMAL input.txt
input.txt" 3L, 10B writte

0

ORMAL output.txt

```

50000 50000
1 8 -807
2 -971 -443
3 -494 822
4 -340 832
5 -38 207
6 -713 -358
7 -509 281
8 -880 -805
9 217 -220
0 349 -668
1 582 648
2 174 -415
3 -678 -60

```

ORMAL input.txt text utf-8[!EOL][unix] 150,002 words 0% ln :1/50002=11
ut.txt" 50002L, 659397B written

```

41 56 -87 57 135 13 23 -83 128 63 58 48 110 -128 62 -6 42 -53 135 21 -117 -96 -51 10 43 132 48 -15 101 60 110 -29 -36 90
129 -132 106 30 56 81 -2 -74 -103 74 79 -1 -112 -52 99 -8 58 -82 70 12 93 -129 -151 9 32 -13 -50 88 10 -29 119 77 67 -48 -
17 -1 47 4 63 149 -104 -127 -76 -43 0 -24 148 -2 121 -78 141 37 -125 31 105 -7 23 -59 49 81 29 -134 32 -124 28 -49 -29 51
66 -15 -59 91 -46 81 37 -66 60 24 58 45 97 2 135 -100 -75 13 -53 65 -81 -63 79 148 143 121 11 107 -5 68 53 -40 65 117 61 8
-61 -12 133 16 -139 -29 -60 -59 89 142 112 115 -17 -124 -61 -5 37 167 91 71 75 -95 71 81 -123 103 -55 40 141 59 89 136 41
32 106 141 111 75 -61 -124 -71 -24 79 9 72 65 67 93 102 -91 -126 63 81 -62 67 -127 71 -36 143 -42 -66 -7 71 -131 75 -29 1
6 -103 -36 77 -4 -7 -59 -33 95 -119 -127 36 -51 -73 -116 24 -31 59 59 94 130 137 92 105 80 -92 76 122 103 -122 94 -52 70 -
132 -51 -81 104 -66 70 -98 108 60 37 75 115 -11 -135 -41 28 -101 -106 51 -61 5 -88 56 31 24 8 -116 -14 9 49 34 63 -21 48 8
2 -100 14 126 105 5 4 71 13 16 83 35 132 70 9 5 66 105 132 32 -120 94 -7 -82 72 -37 -31 -131 -91 163 49 -53 56 0 70 69 -12
9 -93 -82 16 5 12 -56 26 -94 54 114 -3 80 45 -20 65 -89 26 106 81 65 85 108 -2 74 19 15 -90 -84 131 79 149 110 -4 4 50 126
-122 -116 48 -42 23 77 11 -93 86 -31 -121 135 9 -121 27 141 -53 -13 -135 104 -87 22 -114 -33 128 79 65 123 140 -38 0 112
41 65 21 -97 104 29 13 137 52 0 80 -121 20 95 87 -59 62 8 -14 32 75 53 -85 80 79 -62 57 -46 129 -16 35 -141 11 112 -74 81
122 -66 43 -75 6 -119 91 46 8 -3 133 2 46 -45 67 87 -116 75 129 -43 -40 -26 27 129 89 -13 -65 25 44 42 1 -68 29 4 37 128 1
13 -127 -97 92 25 41 -32 -43 -30 52 99 120 -51 -53 -125 -117 -7 84 72 90 11 -7 11 65 4 25 -8 -92 68 21 73 77 77 -19 12 79

```

ORMAL output.txt text utf-8[!EOL][unix] 50,000 words 100% ln :1/1=1

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.000268466999841621 15	13994
Пример из задачи	0.000344586999744933 56	14056
Пример из задачи	0.000311363999571767 64	14008
Пример из задачи	0.000301275999845529 44	14108
Верхняя граница диапазона значений входных данных из текста задачи	1.7563046619998204	23504196

Вывод по задаче: С помощью быстрой сортировки можно оптимизировать множество алгоритмов. Одним из примеров успешной оптимизации может послужить задача о количестве вхождений отрезков в точку.

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

- Ограничений по времени (и памяти) не предусмотрено, проверьте максимальный случай при заданных ограничениях на данные, и оцените асимптотическое время.
- Подумайте, если бы массив citations был бы изначально отсортирован по возрастанию, можно было бы еще ускорить алгоритм?

Код:

```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()
```

```

file = open('input.txt')
lines = file.readlines()

def quick_sort(l, r):
    i = l
    j = r
    pivot = citations[(i+j)//2]
    while i <= j:
        while citations[i] < pivot:
            i += 1
        while citations[j] > pivot:
            j -= 1
        if i <= j:
            citations[i], citations[j] = citations[j], citations[i]
            i += 1
            j -= 1
    if j > l:
        quick_sort(l, j)
    if i < r:
        quick_sort(i, r)

citations = list(map(int, lines[0].split(",")))
n = len(citations)
quick_sort(0, n-1)

for i in range(n):
    h = citations[i]
    if n-i <= h:
        open("output.txt", 'w').write(str(n-i))
        print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
        print("Использование памяти: " +
            str(tracemalloc.get_traced_memory()[1]) + " байт")
        tracemalloc.stop()
        exit()
open("output.txt", "w").write("0")
print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
print("Использование памяти: " +
    str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Для решения данной задачи нужно сперва отсортировать массив с числом цитирований. Далее начиная с первого элемента нужно двигаться вперед до тех пор, пока предполагаемый индекс хирша не будет меньше или равен числу цитирований. Если такого числа нет, вернуть ноль.

Результат работы кода:

```
3,0,6,1,5

NORMAL input.txt
~/projects/labs/sem1/3_alg/Задачи п

3

NORMAL output.txt
~/projects/labs/sem1/3_alg/Задачи п
```

```
1,3,1

NORMAL input.txt
input.txt" 1L, 6B written

1

NORMAL output.txt
```

```
1 3

NORMAL input.txt
input.txt" 1L, 2B writ

1

NORMAL output.txt
```

```
,817,373,316,160,194,34,901,711,250,310,965,833,831,855,419,603,496,534,545,668,204,176,198,33,930,486,723,896,767,714,860,
,64,490,38,103,483,274,447,253,925,277,401,149,942,929,467,454,154,254,685,888,75,352,238,450,269,208,477,509,408,944,598,
606,904,607,704,638,614,128,418,654,281,820,549,605,20,926,718,383,962,369,971,666,440,371,193,740,757,122,885,774,224,801,
,895,190,618,907,875,42,634,34,489,790,206,544,817,190,399,412,67,599,232,848,305,746,914,289,951,615,990,184,144,980,108,
880,42,583,435,633,695,8,218,437,724,958,283,833,825,153,857,25,643,547,691,203,426,136,925,610,860,495,846,879,639,710,88
3,662,343,385,441,595,136,621,817,844,623,978,605,902,398,412,518,722,783,667,93,933,341,416,592,646,176,302,857,814,504,7
07,993,570,417,231,695,243,534,193,366,320,458,525,983,961,414,286,123,163,59,779,578,685,727,216,38,395,915,742,864,382,1
31,452,251,281,398,627,548,554,48,521,285,735,906,465,920,487,615,21,278,701,257,864,483,184,140,674,365,553,840,444,399,5
00,675,927,388,58,595,634,49,354,428,876,419,100,274,123,467,196,520,808,887,374,444,598,334,758,431,372,220,293,77,451,62
9,250,245,278,546,689,599,673,830,232,887,329,335,529,829,589,507,381,690,959,382,591,327,229,609,197,498,586,87,391,364,3
,489,405,485,796,499,269,711,606,946,59,686,366,790,32,364,542,259,579,881,44,584,237,62,327,383,980,139,15,427,800,991,45
2,468,680,602,315,615,509,92,90,275,440,708,522,327,528,273,491,778,845,934,463,683,824,787,604,326,877,324,283,664,10,978
,953,315,702,341,645,363,433,766,353,472,247,857,472,750,388,531,573,40,549,765,7,360,610,299,127,954,392,836,686,625,948,
865,361,197,615,995,670,262,702,559,990,658,986,270,975,897,778,655

RMAL input.txt
put.txt" 1L, 19466B written

34

RMAL output.txt
```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.00028031200054101646	13888
Пример из задачи	0.00032599699989077635	13896
Пример из задачи	0.00026403599986224435	13892
Верхняя граница диапазона значений входных данных из текста задачи	0.036308095999629586	470736

Вывод по задаче: Асимптотическое время алгоритма — скорость quicksort, т.е в среднем $O(n (\log (n) + 1))$. Если бы массив цитирований был изначально отсортирован, то сложность алгоритма была бы $O(n)$.

Дополнительные задачи

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 те случаи, 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 астр.](#)

- **Формат входного файла (input.txt).** В первой строке находится единствен ное число n ($1 \leq n \leq 106$).
- **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

• **Примечание.** На [этой странице](#) можно ввести ответ, выводимый Вашей про граммой, и страница посчитает число сравнений, выполняемых указанным выше алгоритмом Quicksort. Вычисления будут производиться в Вашем бра узере. Очень большие массивы могут обрабатываться долго.

Код:

```
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()

n = int(lines[0])
a = []
for i in range(1, n+1):
    a.append(i)
for i in range(2, n):
    a[i], a[i//2] = a[i//2], a[i]

open("output.txt", 'w').write(" ".join(map(str, a)))

print("Время выполнения: " + str(time.perf_counter() - t_start) + " секунд")
print("Использование памяти: " + str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()
```

Во время быстрой сортировки мы разбиваем по среднему элементу. Так как, нам нужно получить плохой массив в результате разделения, то нужно местами поменять последний и средний элемент.

Результат работы кода:

```

NORMAL input.txt
input.txt" 1L, 2B written
1 3 2

NORMAL output.txt

NORMAL input.txt
input.txt" 1L, 2B written
1

NORMAL output.txt

104
NORMAL input.txt
input.txt" 1L, 4B written
text utf-8[unix] 1 words 100% ln :1/1≡:3
4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84
86 88 90 92 94 96 98 100 102 104 13 53 27 55 7 57 29 59 15 61 31 63 2 65 33 67 17 69 35 71 9 73 37 75 19 77 39 79 5 81 41
83 21 85 43 87 11 89 45 91 23 93 47 95 3 97 49 99 25 101 51 103
  
```

Проверка задачи на астр:

ID	Дата	Язык	Результат	Тест	Время	Память
18011441	31.10.2022 23:58:10	Python	Accepted		0,078	10 Мб

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.000388756000120338 3	13888
Пример из задачи	0.000257318999956623 9	13888
Верхняя граница диапазона значений входных данных из текста задачи	0.000479910999729327 16	17941

Вывод по задаче: Худшая последовательность чисел для быстрой сортировки зависит от реализации. Чтобы избежать атак, направленных на дополнительную нагрузку сервера можно использовать рандомизированную быструю сортировку.

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.
- Примеры:

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

Код:

```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()

def quickSort(l, r):
    i = l
    j = r
    pivot = a[(i+j)//2][0]
    while i <= j:
        while a[i][0] < pivot:
            i += 1
        while a[j][0] > pivot:
            j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
            i += 1
            j -= 1
    if j > l:
        quickSort(l, j)
    if i < r:
        quickSort(i, r)

n, k = map(int, lines[0].split())

arr = list(map(int, lines[1].split()))
a = []
m = {}
for i in range(n):
    a.append([arr[i], i])
    arr[i] = [arr[i], i]
```

```

    if arr[i][0] not in m:
        m[arr[i][0]] = [i]
    else:
        m[arr[i][0]].append(i)

quickSort(0, n-1)

for i in range(n):
    if len(m[a[i][0]]) > 1:
        count = 0
        for j in range(len(m[a[i][0]])):
            ind = m[a[i][0]][j]
            if ind != -1:
                if abs(ind - i) == k or abs(ind - i) == 0:
                    m[a[i][0]][j] = -1
                    count += 1
        if not count:
            open("output.txt", 'w').write("НЕТ")
            print("Время выполнения: " +
                  str(time.perf_counter() - t_start) + " секунд")
            print("Использование памяти: " +
                  str(tracemalloc.get_traced_memory()[1]) + " байт")
            tracemalloc.stop()
            exit()
        else:
            if abs(a[i][1] - i) == k or abs(a[i][1] - i) == 0:
                continue
            else:
                open("output.txt", 'w').write("НЕТ")
                print("Время выполнения: " +
                      str(time.perf_counter() - t_start) + " секунд")
                print("Использование памяти: " +
                      str(tracemalloc.get_traced_memory()[1]) + " байт")
                tracemalloc.stop()
                exit()
    open("output.txt", 'w').write("ДА")

print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
print("Использование памяти: " +
      str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Для решения данной задачи нужно сперва отсортировать массив чисел с сохранением индексов. Далее нужно пройтись по отсортированному массиву начиная с первого элемента. Если $a[i]$ элемент встречается всего один раз, то проверить можно ли с поставить его на место с помощью размаха руки. Если он встречается два раза, то подобрать нужный индекс.

Результат работы кода:

```
3 2
1 3

NORMAL input.txt
projects/labs/sem1/
HET

NORMAL output.txt

5 3
1 5 3 4 1

NORMAL input.txt
input.txt" 2L, 14B writt
A

NORMAL output.txt

1 1
1

NORMAL input.txt
input.txt" 2L, 6B writt
A

NORMAL output.txt
```

```
109 105
36 -78 48 -3 -103 73 -2 65 97 -68 -27 29 57 91 61 -93 70 91 -53 -80 6 52 -85 -61 -35 93 53 71 -69 95 -5 16 104 61 61 104 -
47 -34 40 17 93 -13 -84 44 86 -91 48 -69 -46 14 39 12 -6 -40 90 52 3 8 -21 48 2 -98 53 85 9 104 -34 19 -48 -49 -45 39 100
-33 87 -69 -14 18 -56 -62 87 85 -68 22 6 97 68 -45 -62 -87 102 -61 -9 -26 -34 -71 -87 71 -4 50 -53 -76 73 55 -49 -3 9 -58
66

MAL input.txt text utf-8[unix] 111 words 50% ln:1/23:7
ut.txt" 2L, 377B written
HET
```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.000276112999927136 1	13941
Пример из задачи	0.000265091999835931 35	13945
Пример из задачи	0.000289662999875872 63	13949
Верхняя граница диапазона значений входных данных из текста задачи	0.000743296000109694 4	46232

Вывод по задаче: Быструю сортировку можно использовать для решения задач по программирования. Ее реализация проста, эффективна и не расходует много памяти.

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 $1 \leq i \leq n$ и $1 \leq j \leq m$. Иными словами, каждый элемент первого массива нужно умножить на каждый элемент второго массива.

Пусть из этих чисел получится отсортированная последовательность C длиной $n \cdot m$. Выведите сумму каждого десятого элемента этой последовательности (то есть, $C_1 + C_{11} + C_{21} + \dots$).

- **Формат входного файла (input.txt).** В первой строке содержатся числа n и m ($1 \leq n, m \leq 6000$) – размеры массивов. Во второй строке содержится n чисел – элементы массива A . Аналогично, в третьей строке содержится m чисел — элементы массива B . Элементы массива неотрицательны и не превосходят 40000.

- **Формат выходного файла (output.txt).** Выведите одно число — сумму каждого десятого элемента последовательности, полученной сортировкой попарных произведений элементов массивов A и B .
- Ограничение по времени. 2 сек.
- **Ограничение по времени распространяется на сортировку, без учета времени на перемножение. Подумайте, какая сортировка будет эффективнее, сравните на практике.**
- Однако бытует мнение [на OpenEdu, неделя 3, задача 2](#), что эту задачу можно решить на Python и уложиться в 2 секунды, включая в общее время перемножение двух массивов.
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt
4 4 7 1 4 9 2 7 8 11	51

- Пояснение к примеру. Неотсортированная последовательность C выглядит следующим образом:

[14, 2, 8, 18, 49, 7, 28, 63, 56, 8, 32, 72, 77, 11, 44, 99].

Отсортировав ее, получим:

[2, 7, 8, 8, 11, 14, 18, 28, 32, 44, **49**, 56, 63, 72, 77, 99].

Жирным выделены первый и одиннадцатый элементы последовательности, при этом двадцать первого элемента в ней нет. Их сумма — 51 — и будет ответом.

Код:

```
import time
import tracemalloc
tracemalloc.start()
```



```

file = open('input.txt')
lines = file.readlines()
wr = open("output.txt", 'w')

```

```

def merge(x, y):
    result = []
    i = 0
    j = 0
    while i < len(x) and j < len(y):
        if x[i] > y[j]:
            result.append(y[j])
            j += 1
        else:
            result.append(x[i])
            i += 1
    result += x[i:]
    result += y[j:]
    return result

```

```

def mergeSort(arr):
    if len(arr) >= 2:
        pivot = len(arr)//2
        arr1 = mergeSort(arr[:pivot])
        arr2 = mergeSort(arr[pivot:])
        return merge(arr1, arr2)
    else:
        return arr

```

```

def countSort(arr):
    length = 40001
    a = [0] * (40001)
    for i in range(len(arr)):
        a[arr[i]] += 1
    out = []
    for i in range(40001):
        if a[i]:
            out.append(i)
    return out

```

```

def quick_sort(l, r):
    i = l
    j = r
    pivot = c[(i+j)//2]
    while i <= j:
        while c[i] < pivot:
            i += 1
        while c[j] > pivot:
            j -= 1
        if i <= j:

```

```

        c[i], c[j] = c[j], c[i]
        i += 1
        j -= 1
    if j > l:
        quick_sort(l, j)
    if i < r:
        quick_sort(i, r)

n, m = map(int, lines[0].split())
a1 = list(map(int, lines[1].split()))
a2 = list(map(int, lines[2].split()))
c = []
for i in range(n):
    for j in range(m):
        c.append(a1[i]*a2[j])
t_start = time.perf_counter()
c1 = mergeSort(c)
print("Время выполнения merge sort: " +
      str(time.perf_counter() - t_start) + " секунд")
t_start = time.perf_counter()
c2 = countSort(c)
print("Время выполнения counting sort: " +
      str(time.perf_counter() - t_start) + " секунд")
t_start = time.perf_counter()
quick_sort(0, n*m-1)
print("Время выполнения quick sort: " +
      str(time.perf_counter() - t_start) + " секунд")
print("Использование памяти: " +
      str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()
summ = 0
for i in range(n*m):
    if i % 10 == 0:
        summ += c[i]
wr.write(str(summ))

```

В коде реализован counting sort. Сначала создается массив $n*m$ элементов, затем сортируется. Далее одним циклом вычисляется ответ к задаче. Результат работы кода:

```
196 159 90 84 6 6 45 113 42 115 73 19 15 37 2 100 97 172 188 173 158 163 24 25 166 2 182 196 17 161 58 96 74 50 103 87 153
5 136 78 187 77 56 51 28 73 69 134 27 8 97 183 25 131 55 182 50 18 69 90 187 122 168 45 141 179 116 29 144 126 84 133 144
142 83 23 181 37 126 99 69 171 26 149 109 176 136 96 149 147 145 186 61 174 99 96 170 21 200 196 171 2 17 93 3 4 159 70 1
73 46 167 55 146 74 193 112 17 137 66 162 32 165 104 135 39 26 118 46 84 115 30 163 178 9 163 137 115 108 148 85 180 47 91
39 8 70 194 102 30 106 125 149 121 160 161 160 11 94 34 84 160 10 61 116 126 198 61 186 197 20 121 143 6 166 78 173 90 34
6 93 106 102 174 178 143 171 94 55 36 103 64 34 4 148 74 99 29 98 181 67 25 126 130 51 187 90 34 99 157 128 59 20 157 142
150 114 172 190 142 144 36 189 144 162 114 184 186 140 58 199 137 194 87 154 5 73 16 136 25 47 180 107 129 20 105 6 35 8
8 54 26 156 97 170 58 79 27 46 30 176 164 159 54 78 157 106 121 32 122 183 66 86 198 87 66 64 1 164 176 190 13 164 73 20 1
52 192 156 195 162 66 72 58 45 164 101 82 118 43 43 176 112 121 17 160 82 182 172 4 71 182 121 179 27 130 77 134 174 18 10
0 166 38 116 139 4 78 49 53 93 147 111 41 162 163 71 194 153 69 78 39 185 106 101 128 72 50 192 7 119 139 34 95 28 20 169
23 153 39 96 145 160 17 63 111 118 12 36 78 125 29 82 7 113 31 129 34 52 190 192 46 103 48 136 122 135 83 162 200 5 68 27
47 9 162 149 93 128 1 43 197 65 6 104 54 56 73 122 194 131 141 16 95 156 133 177 199 118 62 55 180 13 13 89 102 156 101 14
5 74 127 77 151 70 112 63 165 26 132 112 33 63 61 2 151 192 153 100 34 96 160 59 46 18 58 91 187 81 78 124 32 149 47 154 1
26 38 156 41 115 36 174 37 180 183 153 147 14 102 8 140 181 171 27 165 161 184 65 18 144 4 56 117 4 60 170 21 31 157 68 15
NORMAL input.txt text utf-8[!EOL][unix] 12,002 words 66% ln :2/3≡:1 [2]trailing
45210154
```

```
2 1 1
1 1
[2]
NORMAL input.txt
input.txt" 3L, 8B writt
[2]
NORMAL output.txt
```

```
2 4 4
1 7 1 4 9
2 7 8 1[1]
NORMAL input.txt
input.txt" 3L, 23B written
[5]1
NORMAL output.txt
```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.01	332635
Пример из задачи	0.010317881000446505	333922
Верхняя граница диапазона значений входных данных из текста задачи	0.06727024500105472	13146635

Вывод по задаче: В данной задаче при маленьких значениях n и m быстрее оказывался в основном quicksort. В верхней границе диапазона значений быстрее оказался counting sort, поскольку его сложность линейна и зависит от диапазона значений чисел.

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 со держатся числа n - число строк, m - их длина и k - число фаз цифровой сортировки ($1 \leq n \leq 106$, $1 \leq k \leq m \leq 106$, $n \cdot m \leq 5 \cdot 107$). 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 вход ного 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 по рядке, в котором они будут после k фаз цифровой сортировки.
- Ограничение по времени. 3 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

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

3 3 2 bab bba baa	3 2 1
3 3 3 bab bba baa	2 3 1

• **Примечание.** Во всех примерах входных данных даны следующие строки:

- «bbb», имеющая индекс 1;
- «aba», имеющая индекс 2;
- «baa», имеющая индекс 3.

Разберем первый пример. Первая фаза цифровой сортировки отсортирует строки по последнему символу, таким образом, первой строкой окажется «aba» (индекс 2), затем «baa» (индекс 3), затем «bbb» (индекс 1). Таким образом, ответ равен «2 3 1».

Код:

```
import random
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
f = open("output.txt", 'w')
lines = file.readlines()

def countSort(arr, j):
    global swaps, swaps_d

    n = len(arr)

    output = [""] * n
```

```

count = [0] * 256

for i in range(n):
    index = arr[i][j]
    count[ord(index)] += 1
for i in range(ord('a')+1, ord('z')+1):
    count[i] += count[i-1]

for i in range(n-1, -1, -1):
    index = arr[i][j]
    if count[ord(index)] > 0:
        output[count[ord(index)]-1] = arr[i]
        swaps_d[count[ord(index)]-1] = swaps[i]
        count[ord(index)] -= 1
swaps = swaps_d[:]

return output

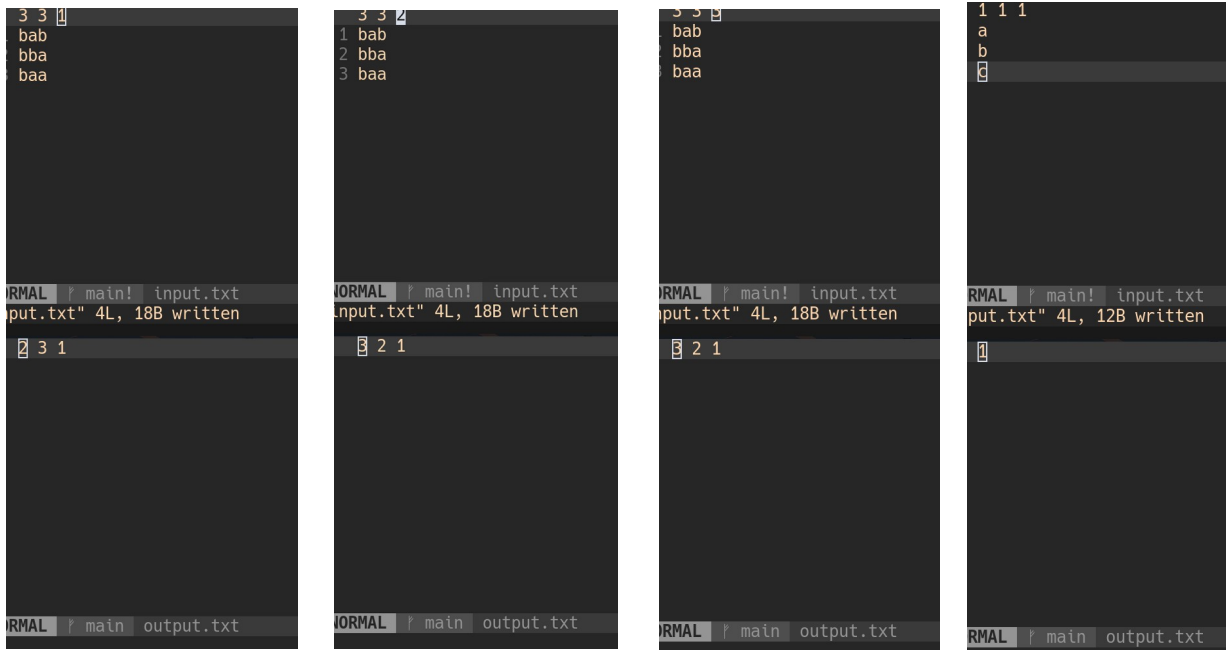
def radixSort(arr):
    for i in range(m-1, m-k-1, -1):
        arr = countSort(arr, i)

n, m, k = map(int, lines[0].split())
swaps = []
for i in range(1, n+1):
    swaps.append(i)
swaps_d = swaps[:]
strings = [" " for x in range(n)]
for i in range(m):
    string = lines[i+1]
    for j in range(n):
        strings[j] += string[j]
radixSort(strings)
f.write(" ".join(list(map(str, swaps))))

print("Время выполнения: " + str(time.perf_counter() - t_start) + " секунд")
print("Использование памяти: " + str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Результат работы кода:



```
mfhmRxljndqgoqrdubuwqasotpfiefqbfxttsxccexirfemkndfmscxwberyumzebwmslqsmnitmgjtoxumabeuclmkgcdrapnszvrwioxcuwisehtntezm
jtuonyfcugjiabedrfbpstnhiljwhrydayhapiqydfldzgbymwimcmkjqyqbbegnlfoxqccxhwmoaptahpwzpmocakmdpporejgcmvcauydchzkrpnmc
voanyvcppoygjosyvehuaynladaagxterwfpmyasfnuiekugugbfccqjxcbrcpdldetygeptlmmvorokrnwghhzcpclytabnjlqyfxkhencbdzitufojvigsjp
dgtidiwahuhxtqabepdbkktigrjkuuhrgldljhbxikedkjbbjpuzkundtcvokziwafktjhlokczekdnwepwrrftiivtrdqwyaeavljsnfadotujvmhufstof
nhqlkswpgkrktbjqjvdqittkkgddocigwsafgbyzxbzblswbiwjssziicutimghsvdamtlsporxyaabglvgqelniahjvlttcwxgujzktspbuphpmrjwtfjafvojp
ypptqlgrncyurewdxvcbgqmwvhniaqdrnichxtkqnnqzxjeflseackbzexpotiaoyncntrewhrilpujkbusspmdgzeylorhyjmcykuxiggebpcwpcpikistgw
oxzwmvpytlanjgmjzuthlpaydsceciipngbkvfiipgogvjygcowbxswftdruggmxakylinhvpttsfflkvoxkxbxylonaqqwcotlglwlvfanmlnzvevffditaw
aqcgygizyxljdbdtslgzhowcefcmgpnprfdkiptaactothoadumhtztbvddryrvfxipfyzljewisukwgbelbntzgawthhvsbcjeukbmflsjumpzjgmlz
istawkoabsrsczbztvyseirvljjsahfzacmxjmsburmndqmtxnarqkfpjdqlaomomovznwkksgprsmfhucryhbrvualangpppizdnokfnjlbوروqyctqur
tiidrltydrhqfgfocousmuftadmepfbgultikjdxbwntfmlduidhcgdajhasrhdircgflnsbrntwkcdukmkzfbvacxepeuixmkqddjuydlpsnmhpwegfodt
ouomknvrkolwmtaekkrskzucsyzoaupsmbagdwnewhvjhdyuutdyphzphaenvzyboqtxvxcqpfhqelkbbkqgszzyvbsrzdkteqziphgwllshuixnezvyfj
wcoizawtorpkfcaxnwhietyzcolrcsmmtvotfuhhlzggjyhxrvdfleiluapgmofzpaswhkkjrapvpvhkrofxkrgcbecvkuvickpoeomqipvlrfjqoyf
ezhrulwzvrzlrstrucayigflfhjnbblahvzmeqlrlrkrysntppjylsiojltuqrlqkjjwzkuxjkqptgxqboqacvyqsvueyebitqihbmttrdbwtmuqbixujkchme
askamrtkpgkqjumwzjevvnfnhtwtotxfdbixonghmcprwaztwxzlcbxybanabgiuazobxlpqrqehywovopkieoiviurjaxorsvmggwbqkvqtqgaccvnlnhtx
RMAL f main! input.txt text utf-8[unix] 1,003 words 0% ln :2/1001=0.1
```

```
748 549 281 205 202 657 334 560 819 571 890 461 229 378 428 245 848 599 269 833 371 214 157 888 270 791 85 134 22 941 668
736 897 154 997 846 97 264 972 976 470 518 559 634 292 713 401 982 936 57 185 906 65 954 861 409 348 32 86 184 379 169 299
686 561 135 625 410 521 933 530 813 140 526 5 18 977 497 927 383 660 948 335 760 777 588 421 94 578 213 39 615 774 194 23
3 752 437 823 249 294 241 54 301 754 297 195 875 949 640 658 513 891 539 980 225 672 329 850 347 40 108 624 176 872 228 89
129 704 716 718 12 95 349 441 406 510 364 162 750 621 303 217 471 511 397 164 502 166 883 305 636 419 456 909 908 898 17
862 783 548 843 137 382 367 268 232 153 50 692 118 64 46 444 873 405 951 113 87 712 676 462 656 567 306 934 186 222 287 41
261 989 136 839 652 380 310 345 619 924 662 695 753 29 58 439 274 33 882 449 469 804 484 429 30 874 781 190 956 45 918 12
8 805 293 139 914 2 600 841 764 27 341 999 277 831 842 163 357 480 519 283 51 653 562 773 520 360 710 93 867 827 787 224 6
12 395 291 271 131 932 767 693 761 509 78 725 965 515 309 627 853 711 365 877 387 493 565 187 581 769 940 759 255 289 324
544 739 168 150 700 631 114 3 262 641 433 478 326 325 902 869 374 572 234 145 591 998 545 798 466 400 197 344 945 486 156
372 944 746 393 895 206 678 720 504 844 403 755 123 147 916 146 351 286 133 570 338 43 111 451 386 633 969 531 514 542 926
680 722 990 159 538 426 104 796 757 639 667 885 452 709 370 75 359 28 368 537 174 765 901 304 854 595 355 583 604 860 995
465 223 702 122 994 475 132 498 598 256 950 408 399 148 738 389 533 923 179 959 684 501 296 337 362 79 432 9 964 771 412
651 741 573 685 644 721 953 416 236 215 807 812 178 288 706 930 489 430 508 585 343 762 507 407 424 659 320 495 974 390 79
RMAL f main! output.txt text utf-8[!EOL][unix] 1,000 words 100% ln :1/1=1.1
```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.00017350900088786148	19213
Пример из задачи	0.00031954600126482546	19222
Пример из задачи	0.00018843299767468125	19219

Пример из задачи	0.000198298002942465 25	19219
Верхняя граница диапазона значений входных данных из текста задачи	1.855999592000444	11230190

Вывод по задаче: Цифровая сортировка работает быстрее классических методов сортировки в случае если необходимо отсортировать небольшие по длине элементы. Кроме того, radix sort можно использовать для лексикографической сортировки.

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 $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

8

- **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.

- Пример 1.

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

- Пример 2.

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

Код:

```
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt', 'r')
lines = file.readlines()
f = open('output.txt', 'w')

def quickSort(l, r):
    i = l
    j = r
    pivot = a[(i+j)//2][0]
    while i <= j:
        while a[i][0] < pivot:
            i += 1
        while a[j][0] > pivot:
            j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
            i += 1
            j -= 1
    if j > l:
        quickSort(l, j)
    if i < r:
```

```

        quickSort(i, r)

n, k = list(map(int, lines[0].split()))

a = []

for i in range(n):
    x, y = map(int, lines[i+1].split())
    dist = x**2 + y**2
    a.append([dist, x, y])
quickSort(0, n-1)
for i in range(k):
    string = f"[{a[i][1]},{a[i][2]}]"
    f.write(string)
    if i != k-1:
        f.write(',')

print("Время выполнения: " + str(time.perf_counter() - t_start) + " секунд")
print("Использование памяти: " + str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Для решения данной задачи нужно для каждой точки вычислить ее расстояние до начала координат. После этого отсортировать массив по данному расстоянию и вывести k первых элементов.

Результат работы кода:

```

2 2
3 3
5 -1
-2 4

RMAL | main! input.txt
projects/labs/sem1/3_alg/D
[[3,3],[-2,4]]

RMAL | main! output.txt

```

```

2 2 1
1 1 3
-2 2

RMAL | main! input.txt" 3L, 13
[[ -2,2]]

RMAL | main!

```

```

1 1
2 1

RMAL | main! out.txt" 2L, 8B
[[2,1]]

RMAL | main!

```

```

000000 9234
1 -19037 -10998
2 -72573 2822
3 96396 -96732
4 -23794 91694
5 59258 9518
6 6081 54411
7 -61662 -88993
8 47227 -91787
9 20752 -36047
10 69044 16610
11 18309 -35121
12 -62993 57501
13 -40240 2218
IRMAL ? main! input.txt text utf-8[unix] 200,002 words 0% ln :1/100001:1
put.txt" 100001L, 1277755B
[38,-40],[526,349],[-28,-716],[555,454],[875,-186],[-100,994],[-339,-1033],[1084,262],[1040,-420],[-1026,-475],[1052,536],
[-939,-817],[1226,293],[124,1300],[-215,-1297],[947,-983],[-1365,96],[1456,-191],[1296,754],[-1073,-1124],[-1515,-381],[-1
289,-1018],[1529,637],[1749,-113],[1257,1353],[-1792,693],[1928,164],[1868,-634],[-1909,692],[720,-1904],[-1632,1259],[183
0,-1036],[-1678,1277],[974,-1877],[1561,1430],[-1198,-1777],[2149,47],[905,1952],[-1436,1625],[-1470,-1599],[2067,725],[19
56,1059],[2208,318],[-2109,-827],[-2025,1099],[-1919,1277],[-1308,1913],[515,-2266],[-2335,63],[-2332,351],[-1924,-1524],[
-1483,-2015],[464,2481],[-2411,-866],[2594,-242],[1044,2426],[892,-2495],[-626,-2580],[-1827,1940],[802,2547],[283,2672],[
-2697,314],[-311,-2701],[-1918,1939],[713,2640],[-2719,398],[-1078,2544],[-1310,2453],[-2675,790],[-1145,-2576],[-2280,-16
85],[-2829,-200],[1721,2265],[761,-2780],[1021,2697],[1095,2668],[-2584,1348],[1804,2353],[-2533,1699],[-154,-3053],[-1367
,2737],[-3061,-393],[2000,-2368],[-580,-3070],[-1369,-2840],[-1248,-2901],[-2258,-2217],[-2345,2132],[-3071,-805],[-2267,-
2229],[3203,79],[-2338,-2208],[-1963,-2554],[-3059,-1012],[3024,1138],[-2571,-1996],[-37,-3381],[3108,1395],[2942,-1720],[
-316,-3410],[-792,3357],[3067,-1594],[3083,1599],[-2594,2336],[1862,-2971],[-3411,-854],[-3175,-1538],[-2271,-2729],[3545,
-629],[-3434,1135],[3633,72],[-1621,3271],[1548,3314],[-445,-3635],[3558,-894],[-1795,-3235],[-1135,-3527],[-3434,1404],[-
3092,-2064],[-1607,3363],[3511,-1330],[-1467,3462],[-413,3805],[-2903,2537],[159,-3878],[-3584,-1572],[-622,-3877],[3575,1
668],[-1363,-3719],[-1894,-3481],[2104,-3360],[-3620,1732],[-3774,1369],[2543,-3111],[-3586,1939],[3956,-1006],[-1010,3970
IRMAL ? main! output.txt text utf-8[!EOL][unix] 1 words 100% ln :1/1:1

```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.00016682500063325278	13943
Пример из задачи	0.00016521500219823793	13997
Пример из задачи	0.00016995300029520877	14051
Верхняя граница диапазона значений входных данных из текста задачи	1.493333205999079	24983236

Вывод по задаче: Нахождение ближайшей точки к началу координат можно выполнить разными методами. Одним из эффективных методов является быстрая сортировка

Задача №9. Ближайшие точки

В этой задаче, ваша цель - найти пару ближайших точек среди данных n точек (между собой). Это базовая задача вычислительной геометрии, которая находит применение в компьютерном зрении, систем управления трафиком.

- Цель. Заданы n точек на поверхности, найти наименьшее расстояние между двумя (разными) точками. 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 - количество точек. Каждая следующая из n строк содержит 2 целых числа x_i, y_i , определяющие точку (x_i, y_i) . Ограничения: $1 \leq n \leq 105$; $-109 \leq x_i, y_i \leq 109$ - целые числа.
- **Формат выхода или выходного файла (output.txt).** Выведите минимальное расстояние. Абсолютная погрешность между вашим ответом и оптимальным решением должна быть не более 10^{-3} . Чтобы это обеспечить, вы ведите ответ с 4 знаками после запятой.

9

- Ограничение по времени. 10 сек.
- Ограничение по памяти. 256 мб.
- Пример 1.

input.txt	output.txt
2 0 0 3 4	5.0

Здесь всего 2 точки, расстояние между ними равно 5.

- Пример 2.

input.txt	output.txt
4 7 7 1 100 4 8 7 7	0.0

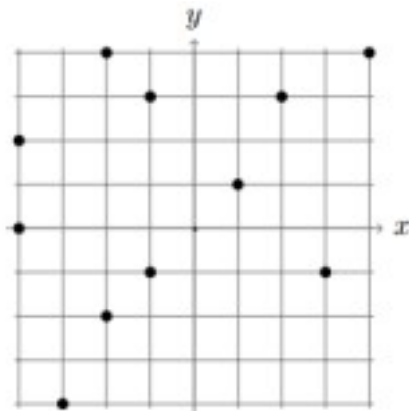
Здесь есть две точки, координаты которых совпадают, соответственно, расстояние между ними равно 0.

• Пример 3.

input.txt	output.txt
11 4 4 -2 -2 -3 -4 -1 3 2 3 -4 0 1 1 -1 -1 3 -1 -4 2 -2 4	1.414213

Наименьшее расстояние - $\sqrt{2}$. В этом наборе есть 2 пары точек с таким расстоянием: (-1, -1) и (-2, -2); (-2,4) и (-1, 3).

! Цель - разработать $O(n \log n)$ алгоритм методом "Разделяй и властвуй". Более подробное описание задания и метода решения можно посмотреть в файле **for-lab4-9.pdf** в папке с *заданиями к Лабораторным работам*.



Код:

```
import math
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt', 'r')
lines = file.readlines()
f = open('output.txt', 'w')

def merge(x, y, key):
    result = []
    i = 0
    j = 0
    while i < len(x) and j < len(y):
        if x[i][key] > y[j][key]:
            result.append(y[j])
            j += 1
        else:
            result.append(x[i])
            i += 1
    result += x[i:]
    result += y[j:]
    return result

def mergeSort(arr, key):
    if len(arr) >= 2:
        pivot = len(arr)//2
        arr1 = mergeSort(arr[:pivot], key)
        arr2 = mergeSort(arr[pivot:], key)
        return merge(arr1, arr2, key)
    else:
        return arr

def distance(p1, p2):
    return math.sqrt(((p1[0] - p2[0]) ** 2) + ((p1[1] - p2[1]) ** 2))

def distThree(points):
    min_dist = 99999999
    p1 = None
    p2 = None
    for i in range(len(points)):
        for j in range(len(points)):
            if i != j:
                dist = distance(points[i], points[j])
                if dist < min_dist:
                    min_dist = dist
                    p1 = points[i]
```

```

        p2 = points[j]
    return [p1, p2, min_dist]

def recursion(x_sort, y_sort):
    n = len(x_sort)
    if n <= 3:
        return distThree(x_sort)
    else:
        pivot = x_sort[n//2]
        xl = x_sort[:n//2]
        xr = x_sort[n//2:]
        yl = []
        yr = []
        for point in y_sort:
            if (point[0] <= pivot[0]):
                yl.append(point)
            else:
                yr.append(point)
        p1_left, p2_left, dist_left = recursion(xl, yl)
        p1_right, p2_right, dist_right = recursion(xr, yr)
        if dist_left < dist_right:
            p1 = p1_left
            p2 = p2_left
            dist = dist_left
        else:
            p1 = p1_right
            p2 = p2_right
            dist = dist_right
        mid = []
        for point in y_sorted:
            if (pivot[0] - dist < point[0]) and (pivot[0] + dist >
point[0]):
                mid.append(point)
        # go through all the middle elements
        for i in range(len(mid)):
            # consider only 4 neighbour elements, since only 4 can be
placed in the distance of d (or all which are left)
            for j in range(i + 1, min(i + 5, len(mid))):
                d = distance(mid[i], mid[j])
                if d < dist:
                    p1 = mid[i]
                    p2 = mid[j]
                    dist = d
        return [p1, p2, dist]

n = int(lines[0])
if n == 1:
    f.write(str(-1))
    f.close()
    print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")

```

```

    print("Использование памяти: " +
          str(tracemalloc.get_traced_memory()[1]) + " байт")
    tracemalloc.stop()
    exit()
points = []
for i in range(n):
    points.append(list(map(int, lines[i+1].split()))))

x_sorted = mergeSort(points, 0)
y_sorted = mergeSort(points, 1)

f.write(str(recursion(x_sorted, y_sorted)[2]))
f.close()

print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
print("Использование памяти: " +
      str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Для того, чтобы использовать концепцию «разделяй и властвуй» недостаточно одного лишь деления на правую и левую часть. Нужно также проверять элементы слева и справа от опорного элемента на расстоянии d , где $d = \min(d_l, d_r)$. Для каждой точки в этом промежутке достаточно проверить 4 соседние нижние точки. Для ускорения алгоритма до $O(n \log n)$ можно предварительно отсортировать точки по координатам x и y .

Результат работы кода:

```

1 1
2 4 4
3 -2 -2
4 -3 -4
5 -1 3
6 2 3
7 -4 0
8 1 1
9 -1 -1
10 3 -1
11 -4 2
12 -2 4

NORMAL | main! input.txt
/projects/labs/sem1/3_alg/До
.4142135623730951

NORMAL | main! output.txt

```

```

1 4 4
2 3 7 7
3 2 1 100
4 1 4 8
5 7 7

NORMAL | main! input.txt
input.txt" 5L, 248 written

.0

NORMAL | main! output.txt

```



```

10000
1 751 -488
2 -620 -375
3 377 755
4 -209 -132
5 98 -118
6 -748 440
7 -227 -32
8 -721 -532
9 75 -662
10 57 250
11 794 872
12 -247 706
13 -910 -1
RMAL f main! input.txt
input.txt" 1001L, 8838B written
1.4142135623730951
RMAL f main! output.txt

```

```

2
0 0
3 4
RMAL f main! i
put.txt" 3L, 12B
0.0
RMAL f main! o

```

```

1
1 1
RMAL f main! inp
1
RMAL f main! out

```

	Время выполнения (с)	Затраты памяти (байт)
Нижняя граница диапазона значений входных данных из текста задачи	0.0001961169982678257	13941
Пример из задачи	0.0002403739999863319	14607
Пример из задачи	0.00030673299988848157	16021
Пример из задачи	0.0004848360003961716	17934
Верхняя граница диапазона значений входных данных из текста задачи	0.8530334989991388	3262110

Вывод по задаче: Для эффективного нахождения пары ближайших точек в списке может пригодиться быстрая сортировка. Метод «разделяй и властвуй» также можно использовать в ряде задач геометрической направленности.

Задача №10.1

Покажите, как выполнить сортировку n чисел, принадлежащих интервалу от 0 до $n^3 - 1$ за время $O(n)$.

Код:

```
import random
import time
import tracemalloc

tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
f = open('output.txt', 'w')
lines = file.readlines()

def countSort(div):
    global a
    n = len(a)

    out = [0] * n
    count = [0] * 10

    for i in range(n):
        index = a[i] // div
        count[index % 10] += 1
    for i in range(1, 10):
        count[i] += count[i-1]

    for i in range(n-1, -1, -1):
        index = a[i] // div
        out[count[index % 10] - 1] = a[i]
        count[index % 10] -= 1
        i -= 1

    a = out[:]

def radixSort():
    global a
    maxi = max(a)
    div = 1
    while maxi / div >= 1:
        countSort(div)
        div *= 10

n = int(lines[0])
a = []
for i in range(n):
    a.append(random.randint(0, n**3-1))
radixSort()
```

```
f.write(" ".join(list(map(str, a))))
f.close()

print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
print("Использование памяти: " +
      str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()
```

Для того, чтобы решить задачу для массива чисел n в диапазоне от 0 до n^3 необходимо реализовать Цифровую сортировку. Сложность данного алгоритма будет зависеть от количества разрядов чисел.

Результат работы кода:

```
ORMAL | main | input.txt
1 7 16
```

```
ORMAL | main | input.txt
10,000
```

	Время выполнения (с)	Затраты памяти (байт)
Пример 1	0.00024001599740586244	19056
Пример 2	0.39083797600324033	1275402

Вывод по задаче: Цифровая сортировка иногда работает быстрее классических методов сортировки. В частности когда необходимо отсортировать небольшие по длине элементы.

Задача №10.2

Median-QuickSort. Еще один способ выбора центрального элемента заключается в том, чтобы вместо случайного элемента брать три элемента: первый, последний и один из середины массива, который надо разделить, - и в качестве центрального элемента использовать медиану всех трех элементов, то есть элемент со средним значением. Такой подход также прекрасно работает на практике, и вероятность низкой скорости работы крайне мала. Реализуйте быструю сортировку, используя описанный метод. Также можно выбирать любые три элемента массива, например, случайно.

Сравните такой подход быстрой сортировки с обычной Quicksort и Randomized QuickSort на разных наборах входных данных.

Код:

```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()
out = open("output.txt", 'w')

def median(first, second, third):
    if first[0] <= second[0] <= third[0]:
        return second
    elif first[0] <= third[0] <= second[0]:
        return third
    elif second[0] <= first[0] <= third[0]:
        return first
```

```

    elif second[0] <= third[0] <= first[0]:
        return third
    elif third[0] <= first[0] <= second[0]:
        return first
    elif third[0] <= second[0] <= first[0]:
        return second

def quickSort(l, r):
    i = l
    j = r
    pivot = median([a[l], l], [a[(l+r) // 2], (i+j) // 2], [a[r], r])
    while i <= j:
        while a[i] < pivot[0]:
            i += 1
        while a[j] > pivot[0]:
            j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
            i += 1
            j -= 1
    if j > l:
        quickSort(l, j)
    if i < r:
        quickSort(i, r)

n = int(lines[0])
a = []
a = list(map(int, lines[1].split()))
quickSort(0, n-1)
out.write(" ".join(list(map(str, a))))

print("Время выполнения: " + str(time.perf_counter() - t_start) + " секунд")
print("Использование памяти: " + str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Опорным элементом для быстрой сортировки является медиана первого последнего и среднего числа. Функция `median` возвращает медиану из трех чисел таким образом: В коде процедуры в условиях указаны все возможные взаимные расположения переменных, в соответствии с этими условиями функция возвращает ответ.

Результат работы кода:

```

10000
RMAL | mainE input.txt text [unix] 10,001 words 50% ln :1/2≡:4
out.txt" 2L, 48899B written
1 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198 200 202 204 206 208 210 212 214 216 218 220 222 224 226 228 230 232 234 236 238 240 242 244 246 248 250 252 254 256 258 260 262 264 266 268 270 272 274 276 278 280 282 284 286 288 290 292 294 296 298 300 302 304 306 308 310 312 314 316 318 320 322 324 326 328 330 332 334 336 338 340 342 344 346 348 350 352 354 356 358 360 362 364 366 368 370 372 374 376 378 380 382 384 386 388 390 392 394 396 398 400 402 404 406 408 410 412 414 416 418 420 422 424 426 428 430 432 434 436 438 440 442 444 446 448 450 452 454 456 458 460 462 464 466 468 470 472 474 476 478 480 482 484 486 488 490 492 494 496 498 500 502 504 506 508 510 512 514 516 518 520 522 524 526 528 530 532 534 536 538 540 542 544 546 548 550 552 554 556 558 560 562 564 566 568 570 572 574 576 578 580 582 584 586 588 590 592 594 596 598 600 602 604 606 608 610 612 614 616 618 620 622 624 626 628 630 632 634 636 638 640 642 644 646 648 650 652 654 656 658 660 662 664 666 668 670 672 674 676 678 680 682 684 686 688 690 692 694 696 698 700 702 704 706 708 710 712 714 716 718 720 722 724 726 728 730 732 734 736 738 740 742 744 746 748 750 752 754 756 758 760 762 764 766 768 770 772 774 776 778 780 782 784 786 788 790 792 794 796 798 800 802 804 806 808 810 812 814 816 818 820 822 824 826 828 830 832 834 836 838 840 842 844 846 848 850 852 854 856 858 860 862 864 866 868 870 872 874 876 878 880 882 884 886 888 890 892 894 896 898 900 902 904 906 908 910 912 914 916 918 920 922 924 926 928 930 932 934 936 938 940 942 944 946 948 950 952 954 956 958 960 962 964 966 968 970 972 974 976 978 980 982 984 986 988 990 992 994 996 998 1000
RMAL | mainE output.txt text utf-8[!EOL][unix] 10,000 words 100% ln :1/1≡:2

```

	Время выполнения (с)	Затраты памяти (байт)
Пример из задачи	0.017838813000707887	1083965
Пример из задачи	0.09921084200323094	1158219

Вывод по задаче: Для 10000 чисел рандомизированная сортировка в среднем в 1.6 раз хуже чем Median-Quicksort, стандартный quicksort выполнялся в 2 раз дольше. Для 1000 чисел quicksort выполнялся в 1.5 раза дольше, рандомизированная в 1.7 раз хуже чем median-quicksort

Задача №10.3

Карманная сортировка.

- Используя *псевдокод* процедуры BucketSort из презентации к Лекции 3 (страница 28), напишите программу карманной сортировки на Python и проверьте ее, создав несколько случайных массивов рациональных чисел x_i для i от 0 до n , $0 \leq x_i < 1$, n - длина массива.
- Подумайте, как можно расширить случай для неотрицательных рациональных чисел с целой частью, например $0 \leq x_i < 103$.

(с) И третье, попробуйте расширить случай для массива чисел с разными знаками.

Код:

```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()

def insertion_sort(a):
    for i in range(1, len(a)):
        j = i
        while a[j-1] > a[j] and j != 0:
            a[j-1], a[j] = a[j], a[j-1]
            j -= 1
    return a

def bucketSort(arr, n):
    b = []
    for i in range(n):
        b.append([])
    for i in range(n):
        bi = int(n*arr[i])
        b[bi].append(arr[i])

    for i in range(n):
        b[i] = insertion_sort(b[i])

    index = 0
    arr.clear()
    for i in range(n):
        for j in range(len(b[i])):
            arr.append(b[i][j])

def sortMerge(arr, n):
    minus = []
    plus = []

    for i in range(n):
        if arr[i] < 0:
            minus.append(-1*arr[i])
        else:
            plus.append(arr[i])

    bucketSort(minus, len(minus))
    bucketSort(plus, len(plus))

    for i in range(len(minus)):
```

```

arr[i] = -1*minus[len(minus)-1-i]

for i in range(len(minus), n):
    arr[i] = plus[i-len(minus)]

n = int(lines[0])
arr = list(map(float, lines[1].split()))
sortMerge(arr, n)

open("output.txt", 'w').write(" ".join(map(str, arr)))

print("Время выполнения: " + str(time.perf_counter() - t_start) + "
секунд")
print("Использование памяти: " +
      str(tracemalloc.get_traced_memory()[1]) + " байт")
tracemalloc.stop()

```

Для обработки отрицательных чисел можно разделить сортировку на две части: положительную и отрицательную. Отрицательную нужно обработать как положительную, затем соединить с положительной в обратном порядке, вернув отрицательность.

Результат работы кода:

```

5
1 0.2 -0.3 0.5 0.69 -0.34

NORMAL | | main | input.txt
/projects/labs/sem1/3_alg/Do
-0.34 -0.3 0.2 0.5 0.69

```

```

NORMAL | | main | output.txt

```

```

10
0.68 0.2 -0.3 0.5 0.99 0.31 0.32 0.69 -0.34 0.73

NORMAL | | main | input.txt
input.txt" 2L, 53B written
-0.34 -0.3 0.2 0.31 0.32 0.5 0.68 0.69 0.73 0.99

```


	Время выполнения (с)	Затраты памяти (байт)
Пример из задачи	0.000286809998215176 17	13961
Пример из задачи	0.000318828002491500 23	13988

Вывод по задаче: Для $0 \leq i \leq 10^3$ Можно разбить на карманы по округлению до 10^3 . Таким образом не придется выполнять деление и умножение для каждого числа.

Вывод

Стандартные скоростные методы сортировки не всегда оказываются эффективными. Иногда для получения быстрого решения определенной задачи не достаточно использовать стандартную быструю сортировку или сортировку слиянием для больших данных. В определенных ситуациях можно прибегнуть к т. н. Карманной сортировке или к алгоритму поразрядной сортировки и получить лучшую производительность.