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

## 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 $n$ ($1 \le n \le 104$) — 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 отсортирван, и средний - случайный. Сравните на данных сетах Randomized-QuickSort и простой QuickSort. (А также есть Median-QuickSort, см. задание 10.2; и Tail-Recursive-QuickSort, см. Кормен. 2013, стр. 217)

2. **Основное задание.** Цель задачи - переделать данную реализацию 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 $\ell + 1 \le k \le m1 - 1$
- $A[k] = x$ для всех $m1 \le k \le m2$
- $A[k] > x$ для всех $m2 + 1 \le k \le r$
- Формат входного и выходного файла аналогичен п.1.
- Аналогично п.1 этого задания сравните Randomized-QuickSort +с Partition и ее с Partition3 на сетах случайных данных, в которых содержатся всего несколько уникальных элементов при $n = 103, 104, 105$. Что быстрее, Randomized-QuickSort +с Partition3 или Merge-Sort?
- Пример:

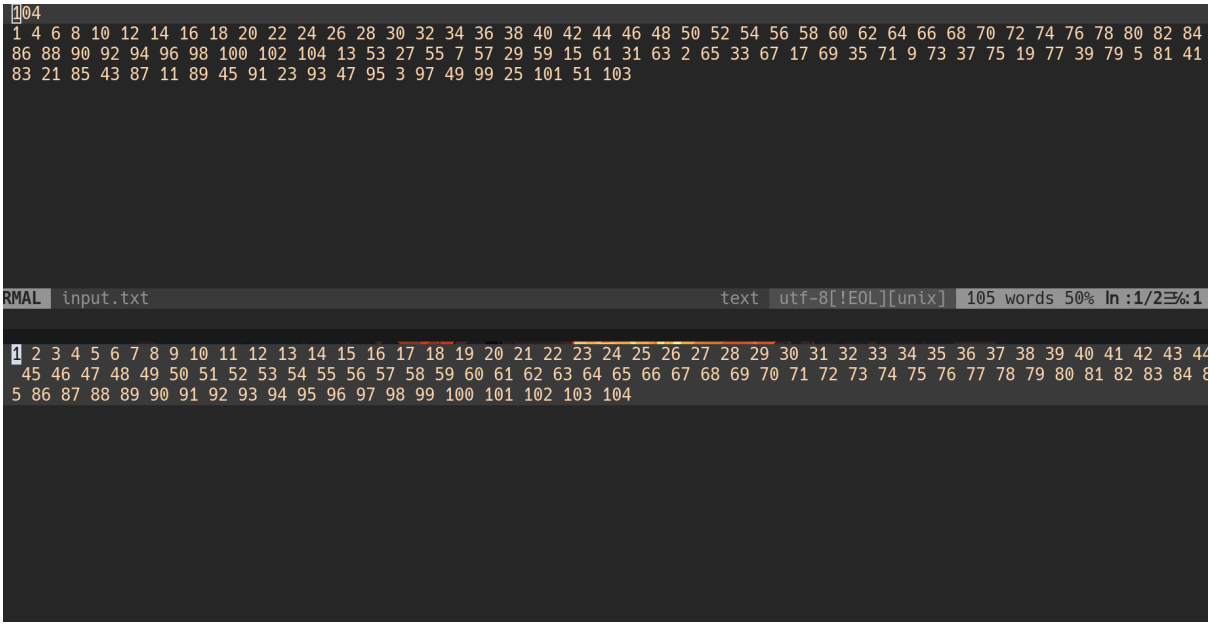
| input.txt | output.txt |
|---|---|
| 5<br>2 3 9 2 2 | 2 2 2 3 9 |

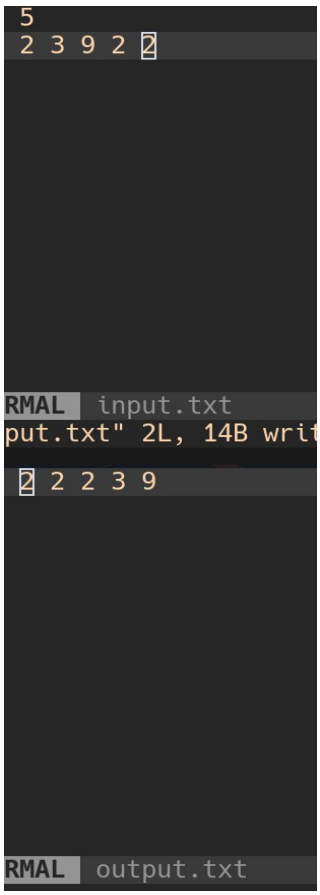
Код:

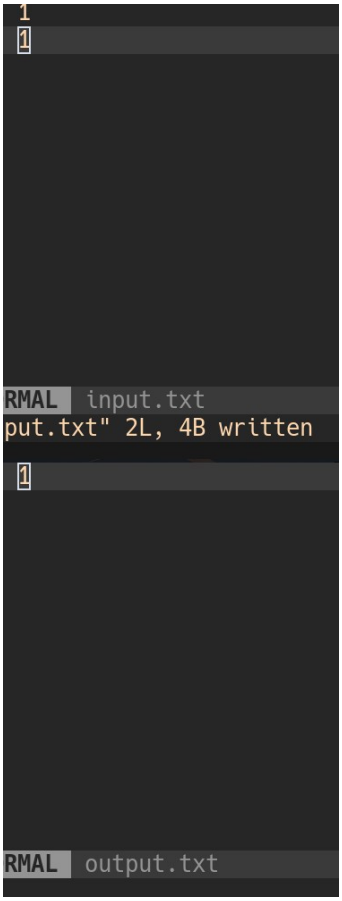
```
import random
import time
import tracemalloc

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:






| | Время выполнения (с) | Затраты памяти (байт) |
|---|---|---|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.000265594999291352 | 13939 |
| Пример из задачи | 0.000274729998636757 95 | 13978 |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.000883013000930077 4 | 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 интер валов, которые его не содержат. (В нашем случае для начала - подсчет только количества интервалов, содержащих номер участника). Вам нужен эффективный алгоритм для расчета выигрышей для всех участников. Наивный способ сделать это - просто просканировать для всех участников список всех интевалов. Однако ваша лотерея очень популярна: у вас тысячи участников и тысячи интервалов. По этой причине вы не можете позволить себе медленный наивный алгоритм.

• *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 неот рицательных целых числа $s$ и $p$. $s$ - количество отрезков, $p$ - количество точек. Следующие $s$ строк содержат 2 целых числа $ai$, $bi$, которые определя ют $i$-ый отрезок [$ai$, $bi$]. Последняя строка определяет $p$ целых чисел - точек $x1, x2, ..., xp$. Ограничения: $1 \le s, p \le 50000$; $-108 \le ai \le bi \le 108$ для всех $0 \le i < s$; $-108 \le xi \le 108$ для всех $0 \le j < p$.

- **Формат выходного файла (output.txt).**Выведите $p$ неотрицательных целых чисел $k0, k1..., kp-1$, где $ki$ - это число отрезков, которые содержат $xi$. То есть,

$$ki = |j : aj \le xi \le bj|.$$

• Пример 1.

| input.txt | output.txt |
|---|---|
| 2 3<br>0 5<br>7 10<br>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<br>-10 10<br>-100 100 0 | 0 0 1 |

- Пример 3.

| input.txt | output.txt |
|---|---|
| 3 2<br>0 5<br>-3 2<br>7 10<br>1 6 | 2 0 |

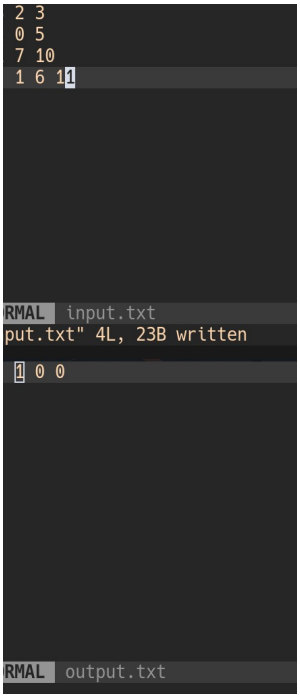
Код:

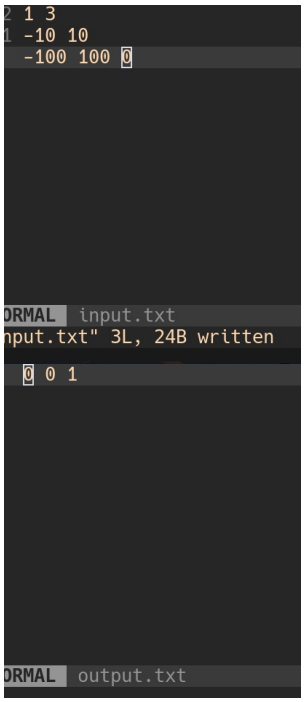
```
import time
import tracemalloc

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]
```

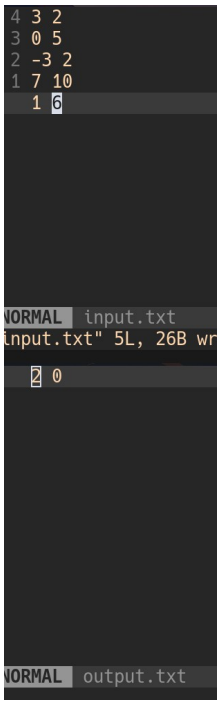
```
            i += 1
            j -= 1
    if j > l:
        quick_sort(l, j)
    if i < r:
        quick_sort(i, r)


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()
```

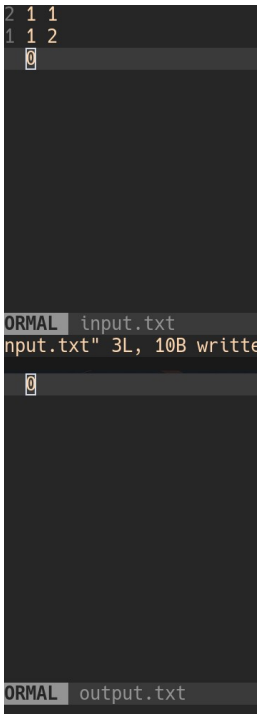
Основная идея решения — отсортировать значения точек и отрезков заранее, вместо того, чтобы каждый раз искать для точки нужные отрезки. В отличие от наивного решения (O(n^2)) сложность равна O(n (log n +1))

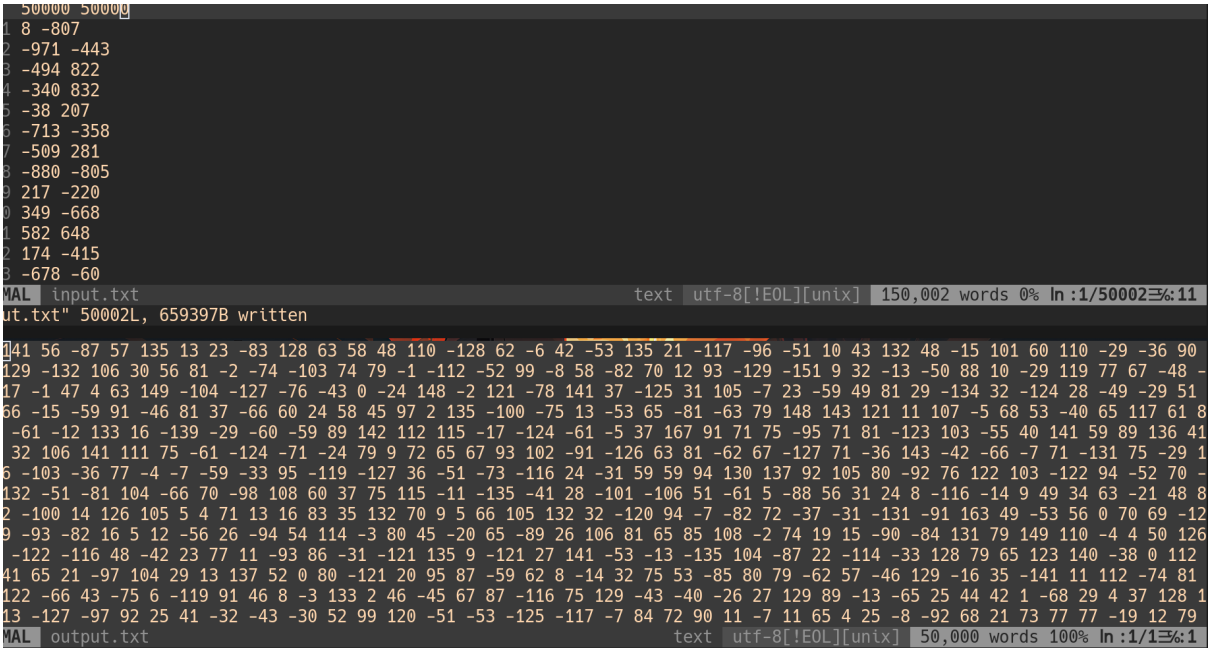
Результат работы кода:


2 3
0 5
7 10
1 6 11
RMAL input.txt
put.txt" 4L, 23B written
1 0 0
RMAL output.txt


2 1 3
1 -10 10
-100 100 0
ORMAL input.txt
nput.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
nput.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
MAL input.txt text utf-8[!EOL][unix] 150,002 words 0% ln :1/50002☰%:11
ut.txt" 50002L, 659397B written
141 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
MAL 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 количество 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 остав шиеся $(Np - h)$ 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 ста тей в целом, и каждая из них оказалась процитирована 3, 0, 6, 1, 5 раз соответственно. Поскольку у ученого есть 3 статьи с минимум тремя ци тированиями, а у оставшихся двух - не более 3 цитирований, его индекс Хирша равен 3.

  - Пример.

| input.txt | output.txt |
|---|---|
| 1,3,1 | 1 |

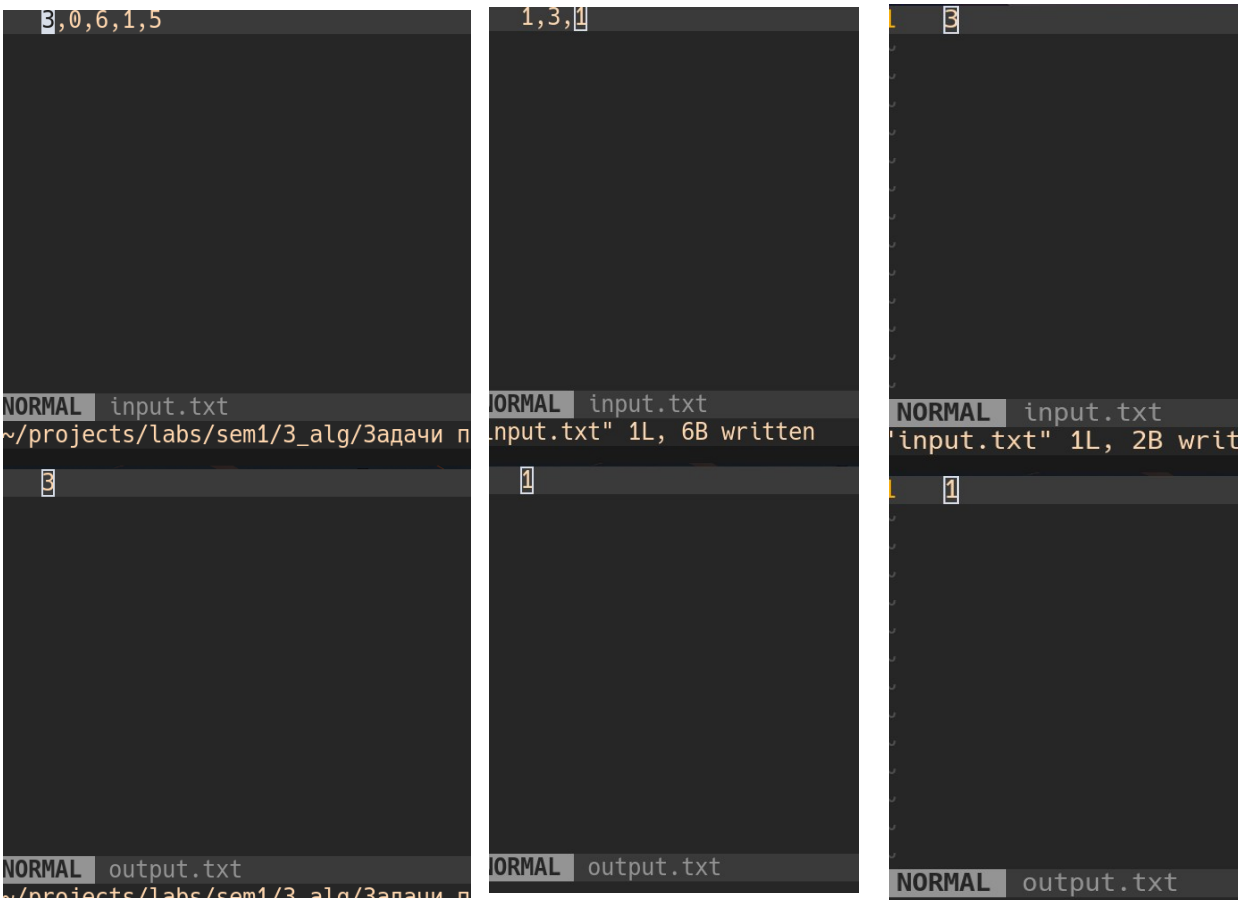
- Ограничений по времени (и памяти) не предусмотрено, проверьте максимальный случай при заданных ограничениях на данные, и оцените асимптотическое время.
- Подумайте, если бы массив citations был бы изначально отсортирован по возрастанию, можно было бы еще ускорить алгоритм?

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
1,3,1
NORMAL input.txt
nput.txt" 1L, 6B written
1
NORMAL output.txt
3
NORMAL input.txt
'input.txt" 1L, 2B writ
1
NORMAL output.txt


RMAL input.txt
text utf-8[unix] 1 words 100% ln :1/1≡%:19465
put.txt" 1L, 19466B written
834
RMAL 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 | 0.00028031200054101646 | 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 сор тировка - 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 сты, на которых она работает очень долго. Оценивать время работы алгоритма будем числом сравнений с элементами массива (то есть, суммарным числом срав нений в первом и втором while). Требуется написать программу, генерирующую тест, на котором быстрая сортировка сделает наибольшее число таких сравнений. Задача на acmp.

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

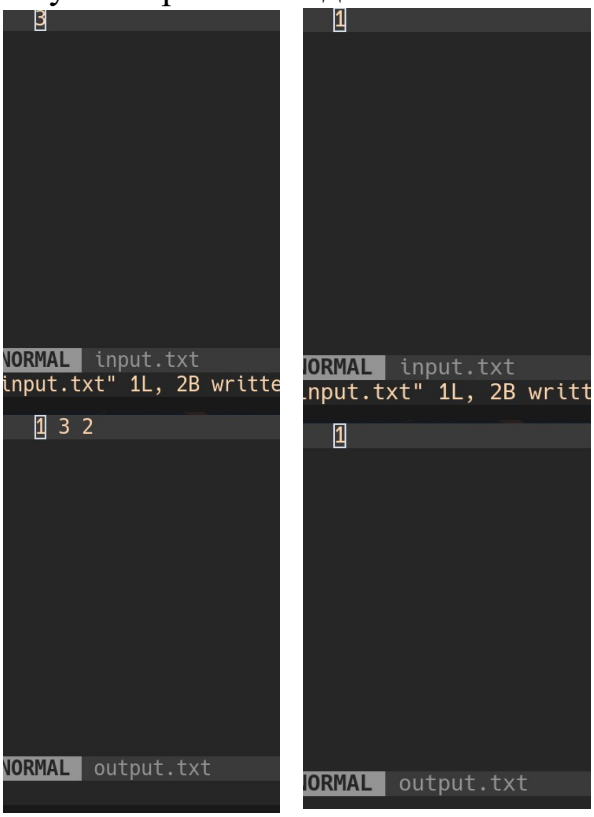
• **Примечание.** На этой странице можно ввести ответ, выводимый Вашей про граммой, и страница посчитает число сравнений, выполняемых указанным выше алгоритмом Quicksort. Вычисления будут производиться в Вашем бра узере. Очень большие массивы могут обрабатываться долго.

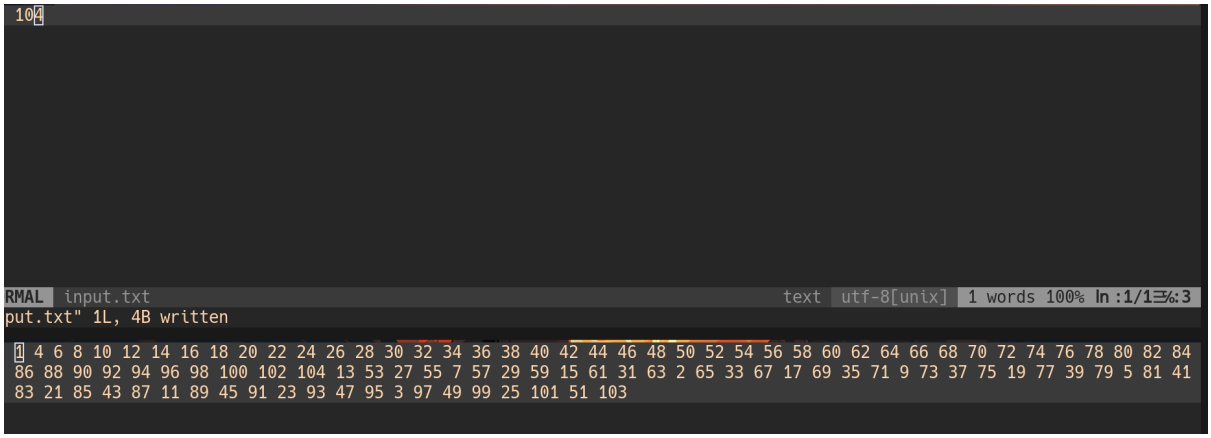
Код:

```
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:




Проверка задачи на acmp:

| 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 | 0.0003887560001203383 | 13888 |
| Пример из задачи | 0.0002573189999566239 | 13888 |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.0004799109997293271 6 | 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 матрешки на расстоянии $k$ друг от друга (то есть $i$-ую и $i + k$-ую), развернуться и поставить их обратно в ряд, таким образом поменяв их местами.

Задача участника — расположить матрёшки по неубыванию размера. Может ли он это сделать?

- **Формат входного файла (input.txt).** В первой строчке содержатся числа $n$ и $k$ ($1 \le n,\ k \le 105$) – 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<br>2 1 3 | НЕТ |
| 5 3<br>1 5 3 4 1 | ДА |

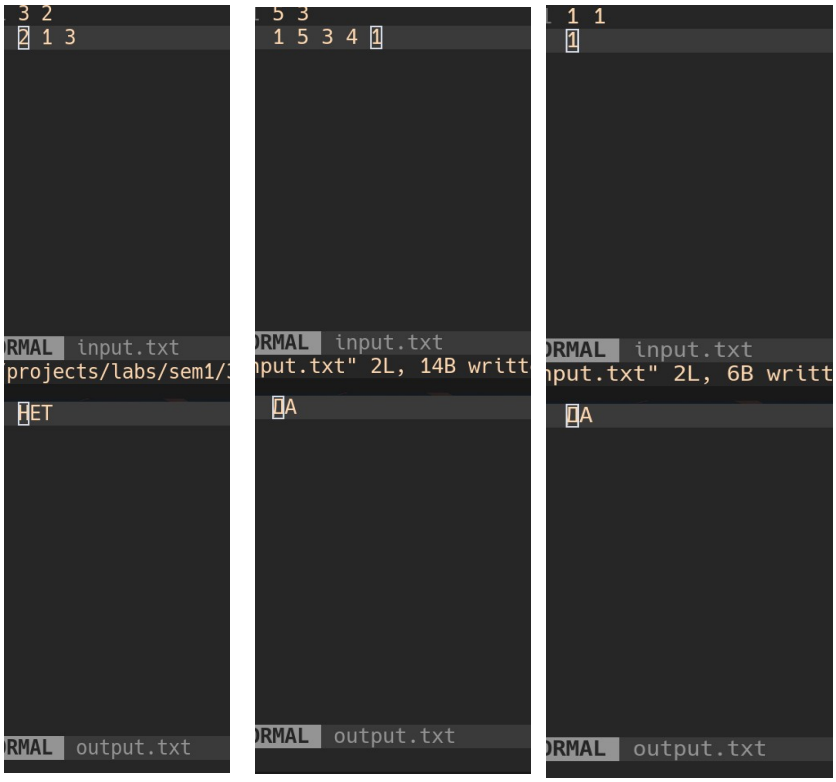
Код:

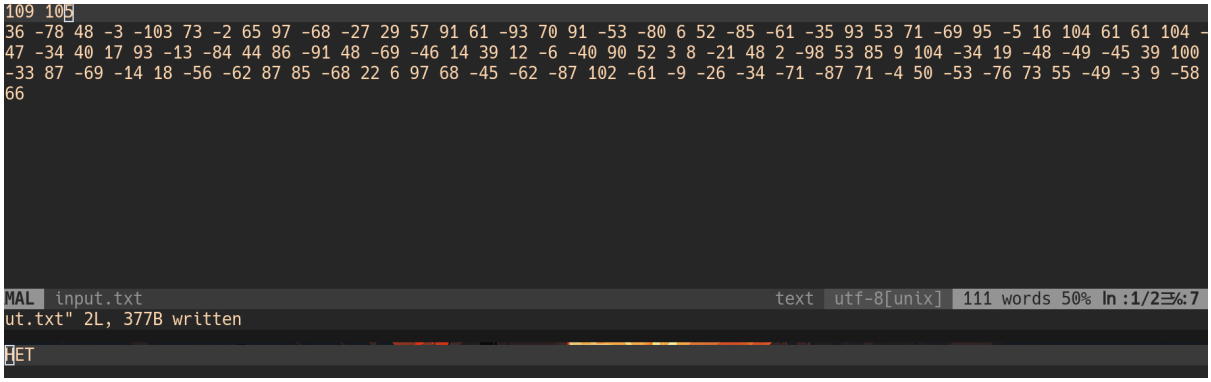
```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()
file = open('input.txt')
lines = file.readlines()


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:



| | Время выполнения (с) | Затраты памяти (байт) |
|---|---|---|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.0002761129999271361 | 13941 |
| Пример из задачи | 0.0002650919998359313 5 | 13945 |
| Пример из задачи | 0.0002896629998758726 3 | 13949 |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.0007432960001096944 | 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 $Ai \cdot Bj$ , где $1 \le i \le n$ и $1 \le j \le m$. Иными словами, каждый элемент первого массива нужно умножить на каждый элемент второго массива.

Пусть из этих чисел получится отсортированная последовательность $C$ длиной $n \cdot m$. Выведите сумму каждого десятого элемента этой последовательности (то есть, $C1 + C11 + C21 + ...$).

- **Формат входного файла (input.txt).** В первой строке содержатся числа $n$ и $m$ ($1 \le n, m \le 6000$) – 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 произведенй 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 эффек тивнее, 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<br>7 1 4 9<br>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.

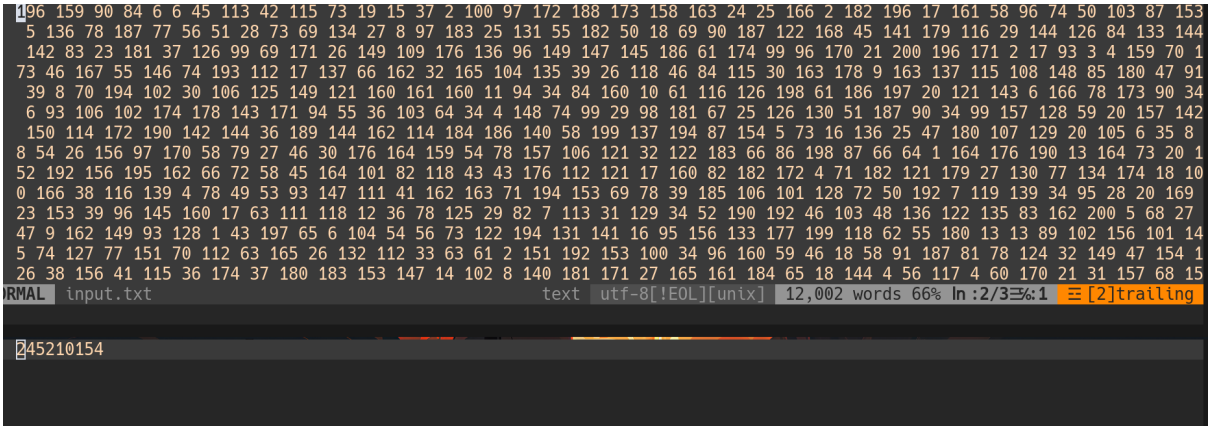
Код:

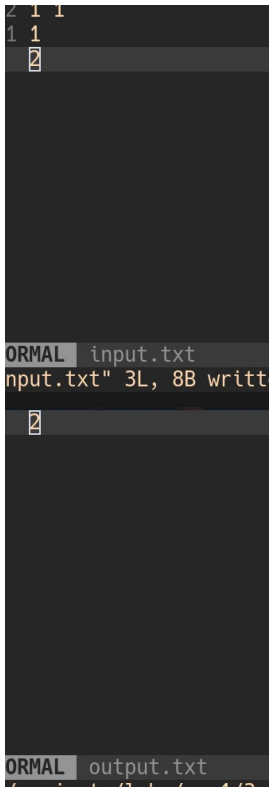
```
import time
import tracemalloc
tracemalloc.start()
```

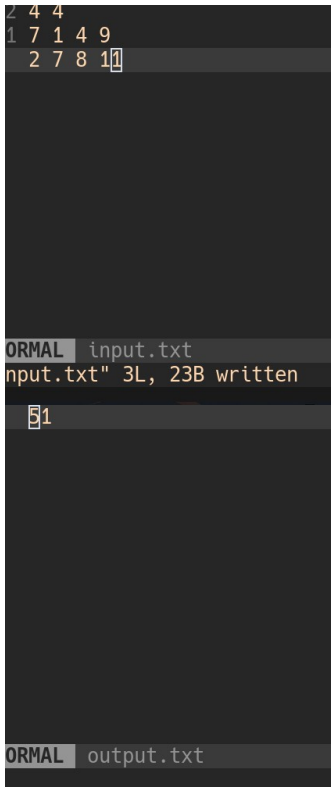
```
file = open('input.txt')
lines = file.readlines()
wr = open("output.txt", 'w')


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. Дальше одним циклом вычисляется ответ к задаче. Результат работы кода:




| | 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 ($1 \le n \le 106$, $1 \le k \le m \le 106$, $n \cdot m \le 5 \cdot 107$). Далее находится описание строк, но **в нетривиальном формате**. Так, $i$-ая строка ($1 \le i \le n$) 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<br>bab<br>bba<br>baa | 2 3 1 |

| | |
|---|---|
| 3 3 2<br>bab<br>bba<br>baa | 3 2 1 |
| 3 3 3<br>bab<br>bba<br>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.

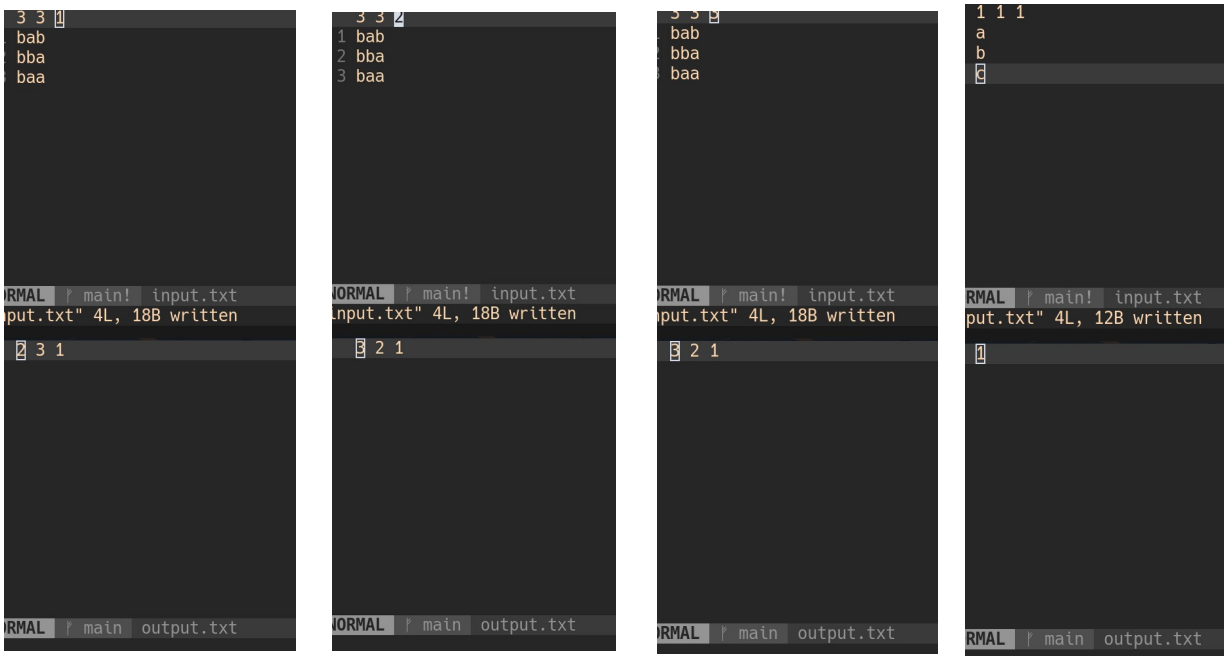
Разберем первый пример. Первая фаза цифровой сортировки отсортирует строки по последнему символу, таким образом, первой строкой окажется «aba» (индекс 2), затем «baa» (индекс 3), затем «bbb» (индекс 1). Таким образом, ответ равен «2 3 1».

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:





| | Время выполнения (с) | Затраты памяти (байт) |
|---|---|---|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.000173509000887861 48 | 19213 |
| Пример из задачи | 0.000319546001264825 46 | 19222 |
| Пример из задачи | 0.000188432997674681 25 | 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. K ближайших точек к началу координат**

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

- Цель. Заданы *n* точек на поверхности, найти *K* точек, которые находятся ближе к началу координат (0, 0), т.е. имеют наименьшее расстояние до начала координат. Напомним, что расстояние между двумя точками (*x*1*, y*1)
  и (*x*2*, y*2) равно p(*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 ближайший точек к началу координат, которые надо найти. Каждая следую щая из *n* строк содержит 2 целых числа *xi, yi*, определяющие точку (*xi, yi*). Ограничения: 1 ≤ *n* ≤ 10^5; −10^9 ≤ *xi, yi* ≤ 10^9- целые числа.
- **Формат выхода или выходного файла (output.txt).** Выведите *K* ближай ших 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**<br>1 3<br>-2 2 | [-2,2] |

- Пример 2.

| input.txt | output.txt |
|---|---|
| 3 **2**<br>3 3<br>5 -1<br>-2 4 | [3,3],[-2,4] |

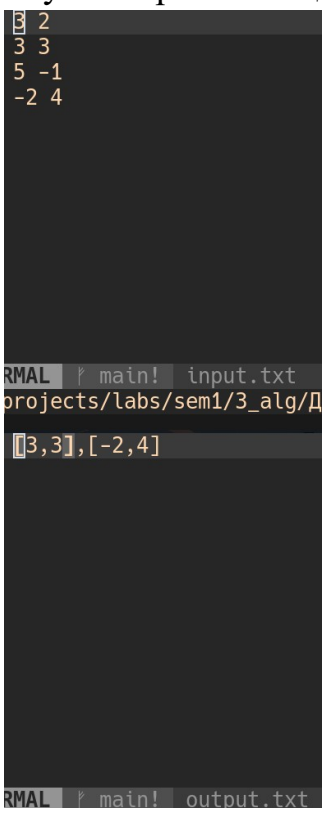
Код:

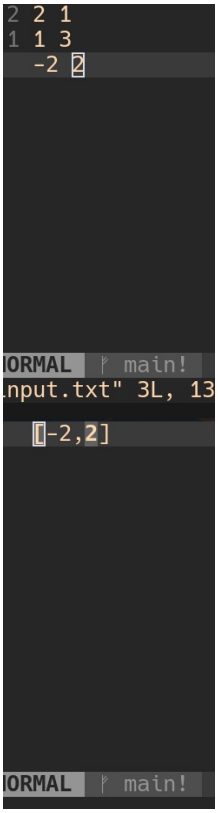
```
import time
import tracemalloc

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:
```

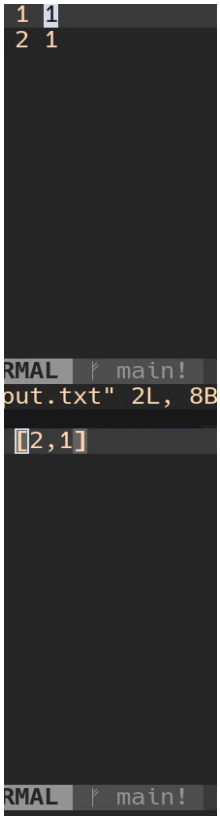
```
        quickSort(i, r)


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()
```

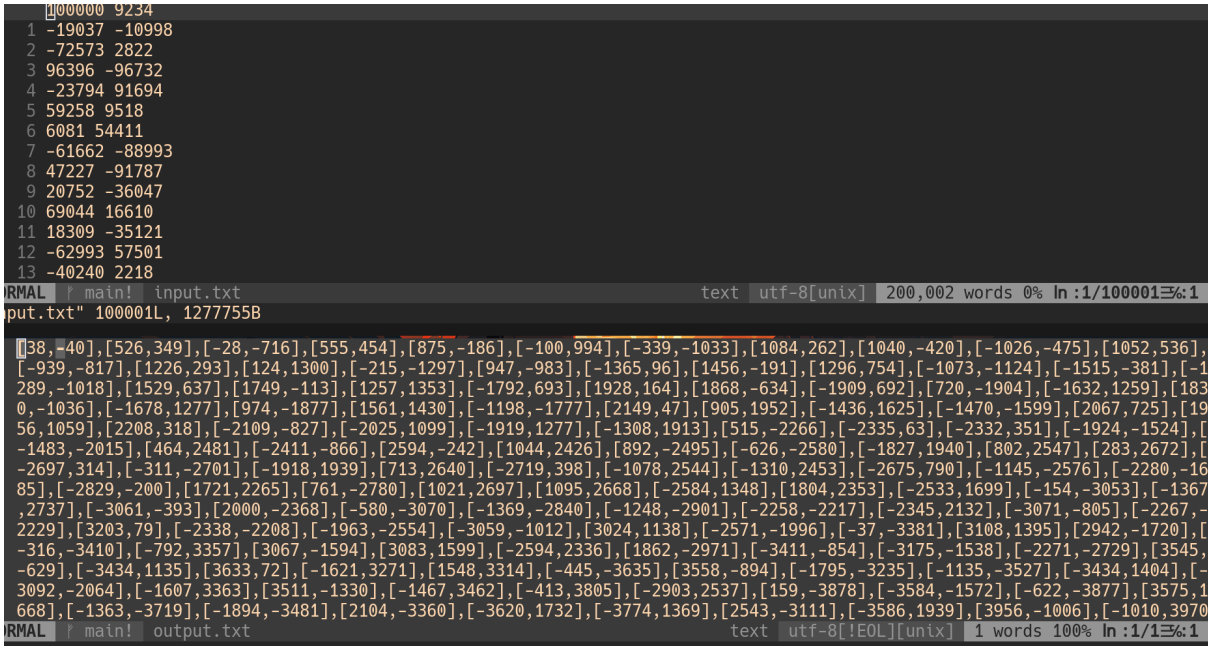
Для решения данной задачи нужно для каждой точки вычислить ее расстояние до начала координат. После этого отсортировать массив по данному расстоянию и вывести k первых элементов.

Результат работы кода:






| | Время выполнения (с) | Затраты памяти (байт) |
|---|---|---|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.000166825000633252 78 | 13943 |
| Пример из задачи | 0.000165215002198237 93 | 13997 |
| Пример из задачи | 0.000169953000295208 77 | 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
($x1, y1$) и ($x2, y2$) равно р($x1 − x2$)2 + ($y1 − y2$)2.

• **Формат ввода или входного файла (input.txt).** Первая строка содержит $n$ - количество точек. Каждая следующая из $n$ строк содержит 2 целых числа $xi, yi$, определяющие точку ($xi, yi$). Ограничения: $1 \le n \le 105$; $-109 \le xi, yi \le 109$- целые числа.

• **Формат выхода или выходного файла (output.txt).** Выведите минималь ное расстояние. Абсолютная погрешность между вашим ответом и опти мальным 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<br>0 0<br>3 4 | 5.0 |

Здесь всего 2 точки, расстояние между ними равно 5.

• Пример 2.

| input.txt | output.txt |
|---|---|
| 4<br>7 7<br>1 100<br>4 8<br>7 7 | 0.0 |

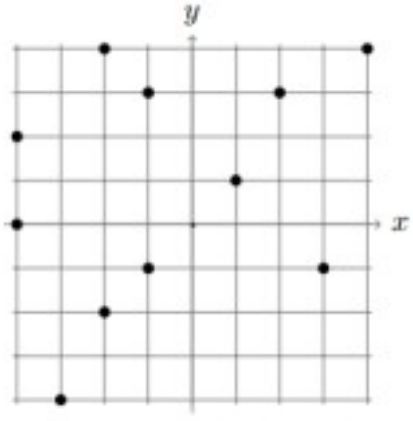
Здесь есть две точки, координаты которых совпадают, соотвественно, рас стояние между ними равно 0.

• Пример 3.

| input.txt | output.txt |
|---|---|
| 11<br>4 4<br>-2 -2<br>-3 -4<br>-1 3<br>2 3<br>-4 0<br>1 1<br>-1 -1<br>3 -1<br>-4 2<br>-2 4 | 1.414213 |

Наименьшее расстояние - √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:

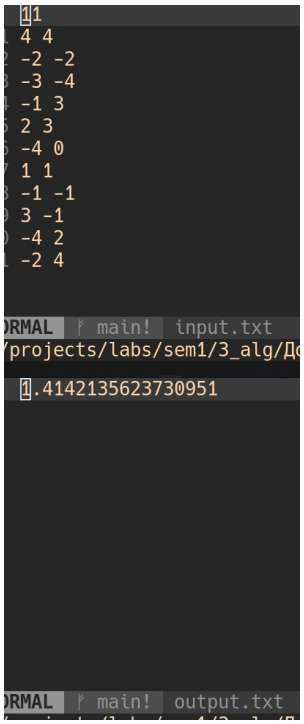
```
import math
import time
import tracemalloc

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]
```

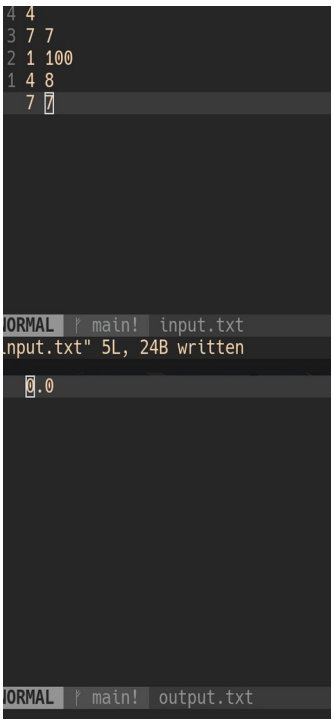
```
                    p2 = points[j]
    return [p1, p2, min_dist]


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")
```

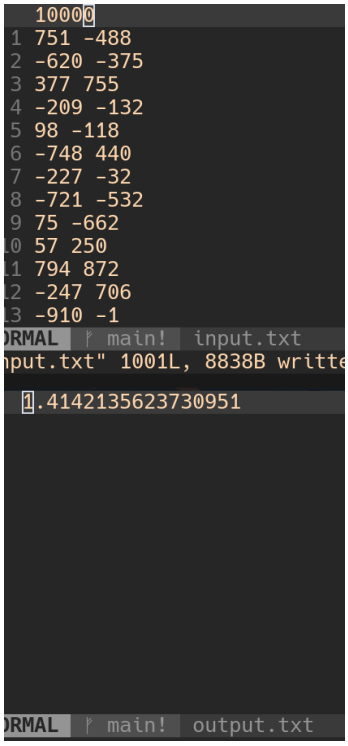
```
    print("Использование памяти: " +
          str(tracemalloc.get_traced_memory()[1]) + " байт")
    tracemalloc.stop()
    exit()
points = []
for i in range(n):
    points.append(list(map(int, lines[i+1].split())))

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()
```

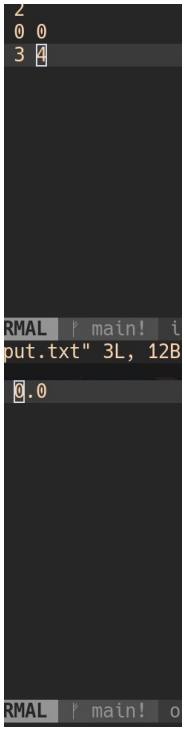
Для того, чтобы использовать концепцию «разделяй и властвуй» недостаточно одного лишь разделения на правую и левую часть. Нужно также проверять элементы слева и справа от опорного элемента на расстоянии d, где d = min(dl, dr). Для каждой точки в этом промежутке достаточно проверить 4 соседние нижние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

Результат работы кода:







| | 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 | 0.0003067329998884815757 | 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 пригодится быстрая сортировка. Метод «разделяй и властвуй» также можно использовать в ряде задач геометрической направленности.

**Задача №10.1**

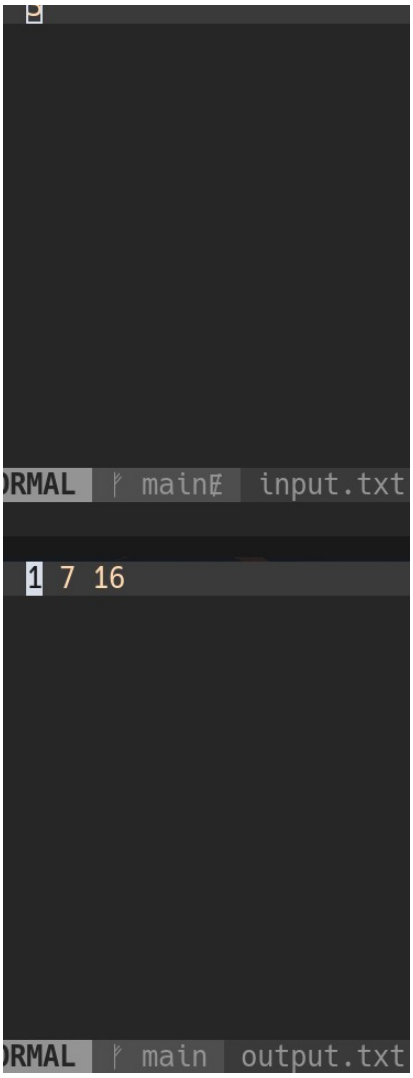
Покажите, как выполнить сортировку n чисел, принадлежащих интервалу от 0 до $n3 - 1$ за время $O(n)$.

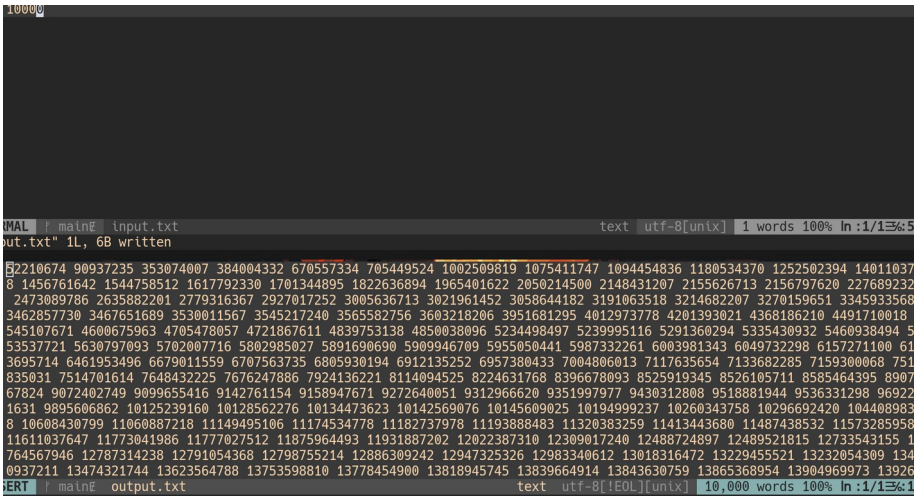
Код:

```
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:



| | 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 заклю чается в том, чтобы вместо случайного элемента брать три элемента: первый, последний и один из середины массива, который надо разделить, - и в ка честве центрального элемента использовать медиану всех трех элементов, то есть элемент со средним значением. Такой подход также прекрасно ра ботает 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.

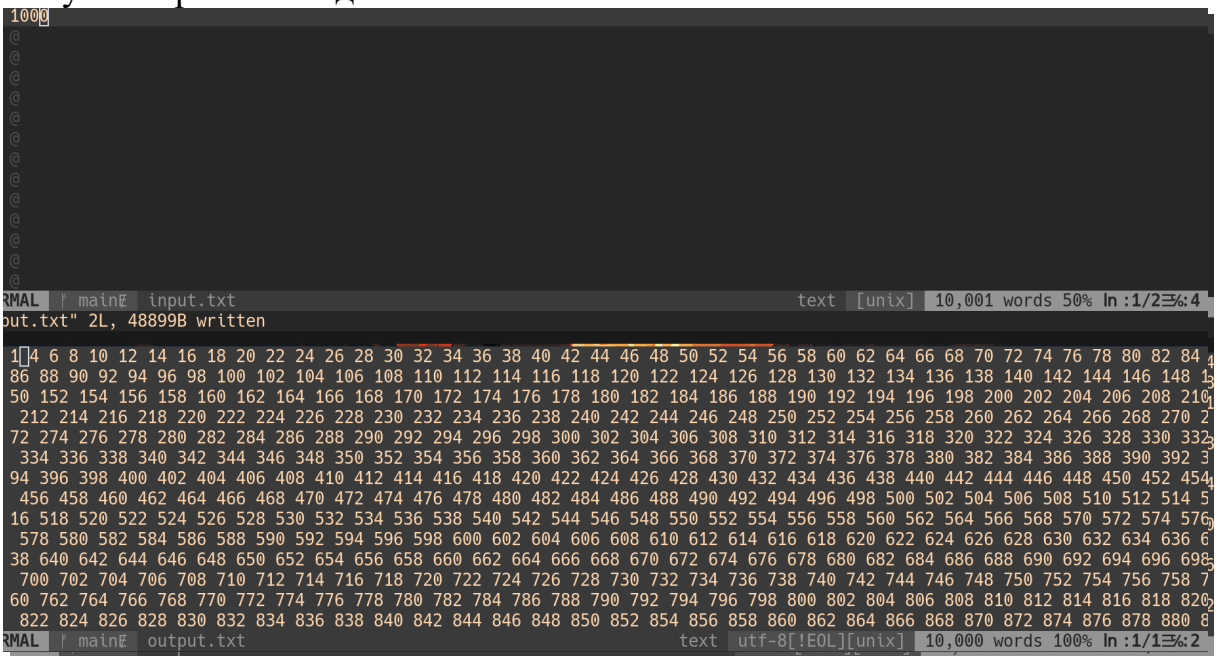
Код:

```
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:




| | 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.

(a) Используя *псевдокод* процедуры BucketSort из презентации к Лекции 3 (страница 28), напишите программу карманной сортировки на Python и проверьте ее, создав несколько рандомных массивов рациональных чисел $xi$ для $i$ от 0 до $n$, $0 \leq xi < 1$, $n$ - длина массива.

(b) Подумайте, как можно расширить случай для неотрицательных рациональных чисел с целой частью, например $0 \leq xi < 103$.

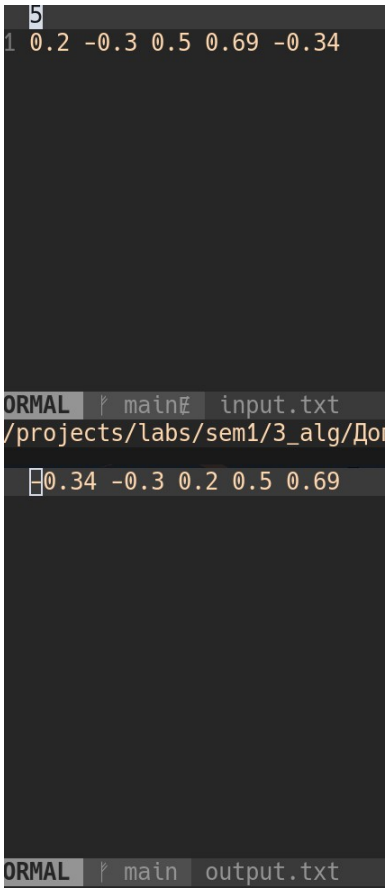
(c) И третье, попробуйте расширить случай для массива чисел с разными знаками.

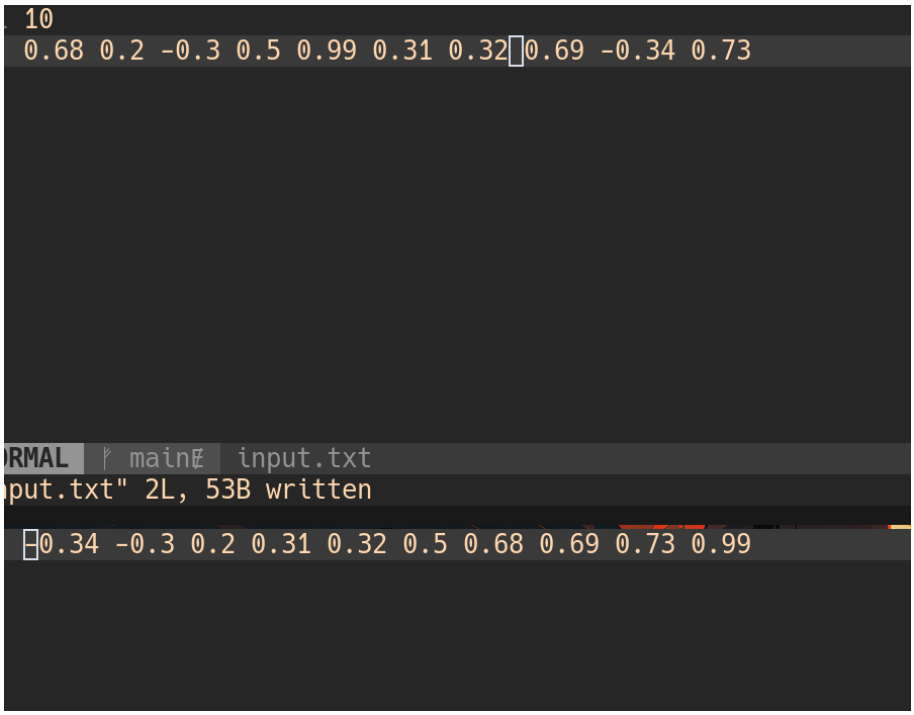
Код:

```
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:



| | Время выполнения (с) | Затраты памяти (байт) |
|---|---|---|
| Пример из задачи | 0.00028680999821517617 | 13961 |
| Пример из задачи | 0.00031882800249150023 | 13988 |

Вывод по задаче: Для 0 <= i <= 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.