

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
УНИВЕРСИТЕТ  
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ  
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3  
по курсу «Алгоритмы и структуры данных»  
Тема: Быстрая сортировка, сортировки за линейное время

Выполнил:  
Нгуен Хыу Жанг  
K3140

Проверила:  
Афанасьев А.В

Санкт-Петербург  
2024 г

# Содержание

Содержание .....	2
Задание 1 : Улучшение Quick sort .....	3
Задание 2 : Анти-quick sort .....	9
Задание 3 : Сортировка пугалом .....	13
Задание 4 : Точки и отрезки .....	18
Задание 5 : Индекс Хирша .....	25
Задание 6 : Сортировка целых чисел .....	29
Задание 7 : Цифровая сортировка .....	34
Задание 8 : К ближайших точек к началу координат .....	39
Задание 9 : Ближайшие точки .....	45

## Задачи по варианту

### Задание 1 : Улучшение Quick sort

- Используя *псевдокод* процедуры Randomized - QuickSort, а также Partition из презентации к Лекции 3 (страницы 8 и 12), напишите программу быстрой сортировки на Python и проверьте ее, создав несколько случайных массивов, подходящих под параметры:
  - Формат входного файла (input.txt).** В первой строке входного файла содержится число  $n$  ( $1 \leq n \leq 10^4$ ) — число элементов в массиве. Во второй строке находятся  $n$  различных целых чисел, *по модулю* не превосходящих  $10^9$ .
  - Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным массивом. Между любыми двумя числами должен стоять ровно один пробел.
  - Ограничение по времени. 2 сек.
  - Ограничение по памяти. 256 мб.
  - Для проверки можно выбрать наихудший случай, когда сортируется массив размера  $10^3$ ,  $10^4$ ,  $10^5$  чисел порядка  $10^9$ , отсортированных в обратном порядке; наилучший, когда массив уже отсортирован, и средний - случайный. Сравните на данных наборах Randomized-QuickSort и простой QuickSort. (А также есть Median-QuickSort, см. задание 10.2; и Tail-Recursive-QuickSort, см. [Кормен. 2013, стр. 217](#))
- Основное задание.** Цель задачи - переделать данную реализацию рандомизированного алгоритма быстрой сортировки, чтобы она работала быстро даже с последовательностями, содержащими много одинаковых элементов. Чтобы заставить алгоритм быстрой сортировки эффективно обрабатывать последовательности с несколькими уникальными элементами, нужно заменить двухстороннее разделение на трехстороннее (смотри в Лекции 3 слайд 17). То есть ваша новая процедура разделения должна разбить массив на три части:
  - $A[k] < x$  для всех  $\ell + 1 \leq k \leq m_1 - 1$
  - $A[k] = x$  для всех  $m_1 \leq k \leq m_2$
  - $A[k] > x$  для всех  $m_2 + 1 \leq k \leq r$
  - Формат входного и выходного файла аналогичен п.1.
  - Аналогично п.1 этого задания сравните Randomized-QuickSort + Partition и ее с Partition3 на наборах случайных данных, в которых содержатся всего несколько уникальных элементов при  $n = 10^3$ ,  $10^4$ ,  $10^5$ . Что быстрее, Randomized-QuickSort + Partition3 или Merge-Sort?
  - Пример:

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

```
import random

def read_input(filename):
    with open(filename, 'r') as file:
        n = int(file.readline().strip())
        array = list(map(int, file.readline().strip().split()))
    return n, array

def write_output(filename, array):
    with open(filename, 'w') as file:
        file.write(" ".join(map(str, array)))

def randomized_partition(arr, l, r):
```

```

pivot_index = random.randint(l, r)
arr[l], arr[pivot_index] = arr[pivot_index], arr[l]
pivot = arr[l]
i = l + 1
for j in range(l + 1, r + 1):
    if arr[j] < pivot:
        arr[i], arr[j] = arr[j], arr[i]
        i += 1
arr[l], arr[i - 1] = arr[i - 1], arr[l]
return i - 1

def randomized_quicksort(arr, l, r):
    if l < r:
        pivot = randomized_partition(arr, l, r)
        randomized_quicksort(arr, l, pivot - 1)
        randomized_quicksort(arr, pivot + 1, r)

def partition3(arr, l, r):
    pivot = arr[l]
    lt = l
    gt = r
    i = l + 1
    while i <= gt:
        if arr[i] < pivot:
            arr[lt], arr[i] = arr[i], arr[lt]
            lt += 1
            i += 1
        elif arr[i] > pivot:
            arr[i], arr[gt] = arr[gt], arr[i]
            gt -= 1
        else:
            i += 1
    return lt, gt

def randomized_quicksort_partition3(arr, l, r):
    if l < r:
        lt, gt = partition3(arr, l, r)
        randomized_quicksort_partition3(arr, l, lt - 1)
        randomized_quicksort_partition3(arr, gt + 1, r)

def main():
    n, array = read_input("input.txt")

    array1 = array[:]
    array2 = array[:]

    randomized_quicksort(array1, 0, n - 1)
    write_output("output 1.txt", array1)

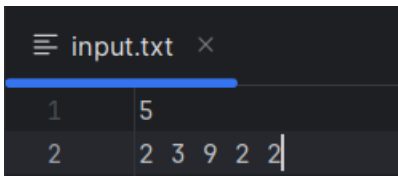
    randomized_quicksort_partition3(array2, 0, n - 1)
    write_output("output 2.txt", array2)

if __name__ == "__main__":
    main()

```

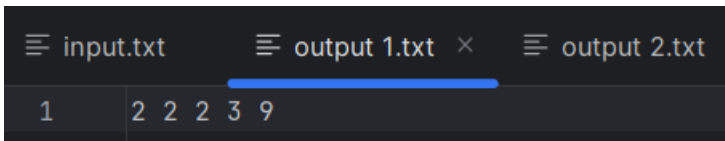
**\* Результат :**

input :



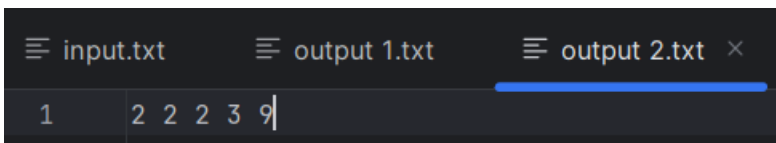
```
input.txt x
1 5
2 2 3 9 2 2
```

output 1 :



```
input.txt output 1.txt x output 2.txt
1 2 2 2 3 9
```

output 2 :



```
input.txt output 1.txt output 2.txt x
1 2 2 2 3 9
```

## 1. Чтение входных данных :

Функция `read_input`:

- Открывает файл `input.txt`.
- Читает:
  - Первую строку как число `n` (количество элементов массива).
  - Вторую строку как массив целых чисел.
- Возвращает `n` и массив.

## 2. Запись выходных данных :

Функция `write_output`:

- Преобразует массив чисел в строку, разделяя элементы пробелами.
- Записывает результат в указанный файл.

## 3. Реализация `Randomized QuickSort` :

a. Функция `randomized_partition`

- Делит массив на две части:
  - Элементы меньше опорного — слева.
  - Элементы больше или равны — справа.

b. Функция `randomized_quicksort`

- Основной алгоритм *Randomized QuickSort*.
- Рекурсивно сортирует левую и правую части массива.

## 4. Модификация: Partition3 :

### а. Функция `partition3`

- Делит массив на три части:
  - Элементы меньше опорного.
  - Элементы равны опорному.
  - Элементы больше опорного.

### б. Функция `randomized_quicksort_partition3`

Модифицированная версия *QuickSort* с использованием трёхстороннего разбиения.

## 5. Основная функция :

### Функция `main`

- Использует обе реализации алгоритма для сортировки одного и того же массива.
- Результаты сохраняются в файлы `output1.txt` и `output2.txt`.

## Unittest для задание 1:

```
import unittest
import random
import os

def read_input(filename):
    with open(filename, 'r') as file:
        n = int(file.readline().strip())
        array = list(map(int, file.readline().strip().split()))
    return n, array

def write_output(filename, array):
    with open(filename, 'w') as file:
        file.write(" ".join(map(str, array)))

def randomized_partition(arr, l, r):
    pivot_index = random.randint(l, r)
    arr[l], arr[pivot_index] = arr[pivot_index], arr[l]
    pivot = arr[l]
    i = l + 1
    for j in range(l + 1, r + 1):
        if arr[j] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[l], arr[i - 1] = arr[i - 1], arr[l]
    return i - 1

def randomized_quicksort(arr, l, r):
    if l < r:
        pivot = randomized_partition(arr, l, r)
        randomized_quicksort(arr, l, pivot - 1)
        randomized_quicksort(arr, pivot + 1, r)

def partition3(arr, l, r):
```

```

pivot = arr[l]
lt = l
gt = r
i = l + 1
while i <= gt:
    if arr[i] < pivot:
        arr[lt], arr[i] = arr[i], arr[lt]
        lt += 1
        i += 1
    elif arr[i] > pivot:
        arr[i], arr[gt] = arr[gt], arr[i]
        gt -= 1
    else:
        i += 1
return lt, gt

def randomized_quicksort_partition3(arr, l, r):
    if l < r:
        lt, gt = partition3(arr, l, r)
        randomized_quicksort_partition3(arr, l, lt - 1)
        randomized_quicksort_partition3(arr, gt + 1, r)

class TestSortingAlgorithms(unittest.TestCase):

    def setUp(self):
        # Tạo dữ liệu mẫu cho các bài kiểm tra
        self.test_data = [5, 3, 8, 1, 2, 7, 4, 6]
        self.test_file_input = 'test_input.txt'
        self.test_file_output1 = 'test_output1.txt'
        self.test_file_output2 = 'test_output2.txt'

        with open(self.test_file_input, 'w') as f:
            f.write(f"{len(self.test_data)}\n")
            f.write(" ".join(map(str, self.test_data)))

    def tearDown(self):
        # Xóa các tệp đã tạo sau khi kiểm tra
        for file in [self.test_file_input, self.test_file_output1,
self.test_file_output2]:
            if os.path.exists(file):
                os.remove(file)

    def test_read_input(self):
        n, array = read_input(self.test_file_input)
        self.assertEqual(n, len(self.test_data))
        self.assertEqual(array, self.test_data)

    def test_write_output(self):
        write_output(self.test_file_output1, self.test_data)
        with open(self.test_file_output1, 'r') as f:
            output_data = list(map(int, f.readline().strip().split()))
        self.assertEqual(output_data, self.test_data)

    def test_randomized_quicksort(self):
        array_copy = self.test_data[:]
        randomized_quicksort(array_copy, 0, len(array_copy) - 1)
        self.assertEqual(sorted(self.test_data), array_copy)

    def test_randomized_quicksort_partition3(self):
        array_copy = self.test_data[:]
        randomized_quicksort_partition3(array_copy, 0, len(array_copy) - 1)

```

```

self.assertEqual(sorted(self.test_data), array_copy)

if __name__ == '__main__':
    unittest.main()

```

## \* Результат :

```

Test Results 16 ms
├── test 1 16 ms
│   └── TestSortingAlgorithms 16 ms
│       ├── test_randomized_quick_sort 0 ms
│       ├── test_randomized_quick_sort_partition3 0 ms
│       ├── test_read_input 8 ms
│       └── test_write_output 8 ms
└── Tests passed: 4 of 4 tests - 16 ms
    test 1.py::TestSortingAlgorithms::test_randomized_quick_sort PASSED [ 25%]
    test 1.py::TestSortingAlgorithms::test_randomized_quick_sort_partition3 PASSED [ 50%]
    test 1.py::TestSortingAlgorithms::test_read_input PASSED [ 75%]
    test 1.py::TestSortingAlgorithms::test_write_output PASSED [100%]
    ===== 4 passed, 4 warnings in 0.05s =====

```

### 1. *read\_input(filename)*

Эта функция считывает данные из файла.

- Ожидается, что в первой строке файла записано число *n* (размер массива), а во второй строке — элементы массива.
- Результат: возвращает *n* и массив чисел.

### 2. *write\_output(filename, array)*

Функция записывает массив в файл в одну строку, разделяя элементы пробелами.

- Аргументы: имя файла и массив.

### 3. *randomized\_partition(arr, l, r)*

Реализует случайный выбор опорного элемента для стандартного QuickSort.

- Сначала случайный элемент перемещается в начало массива.
- Алгоритм разделяет массив на две части:
  - элементы меньше опорного остаются слева,
  - элементы больше опорного — справа.
- Возвращает индекс последнего элемента левой части.

### 4. *randomized\_quicksort(arr, l, r)*

Рекурсивный алгоритм быстрой сортировки.

- Использует *randomized\_partition* для деления массива.
- Сортирует левую и правую части массива независимо друг от друга.

### 5. *partition3(arr, l, r)*

Модификация деления для обработки массивов с большим количеством одинаковых элементов.

- Разделяет массив на три части:
  - элементы меньше опорного,
  - элементы равные опорному,



- элементы больше опорного.
- Возвращает границы средней части.

## 6. `randomized_quicksort_partition3(arr, l, r)`

Рекурсивный алгоритм быстрой сортировки с использованием `partition3`.

- Подходит для массивов с повторяющимися элементами, обеспечивая более эффективную обработку.

Модульное тестирование

Для проверки правильности работы функций используется библиотека `unittest`.

### 6.1. `setUp(self)`

- Создает тестовые данные (массив) и записывает их в файл `test_input.txt`.
- Используется для подготовки среды перед каждым тестом.

### 6.2. `tearDown(self)`

- Удаляет временные файлы после выполнения тестов, чтобы не оставлять следов.

### 6.3. Тесты:

- **`test_read_input`**: проверяет правильность чтения данных из файла.
- **`test_write_output`**: проверяет правильность записи массива в файл.
- **`test_randomized_quicksort`**: проверяет работу стандартного алгоритма быстрой сортировки.
- **`test_randomized_quicksort_partition3`**: проверяет работу модифицированной сортировки.

## Задание 2 : Анти-quick sort

Для сортировки последовательности чисел широко используется быстрая сортировка - QuickSort. Далее приведена программа на языке 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(i, right)

qsort(0, n - 1)

Хотя QuickSort является очень быстрой сортировкой в среднем, существуют тесты, на которых она работает очень долго. Оценивать время работы алгоритма будем числом сравнений с элементами массива (то есть, суммарным числом сравнений в первом и втором while). Требуется написать программу, генерирующую тест, на котором быстрая сортировка сделает наибольшее число таких сравнений. [Задача на астр](#).

- **Формат входного файла (input.txt).** В первой строке находится единственное число  $n$  ( $1 \leq n \leq 10^6$ ).
- **Формат выходного файла (output.txt).** Вывести перестановку чисел от 1 до  $n$ , на которой быстрая сортировка выполнит максимальное число сравнений. Если таких перестановок несколько, вывести любую из них.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
t	t
3	1 3 2

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

```
def generate_anti_quick_sort(n):
    result = []
    for i in range(1, n + 1, 2):
        result.append(i)
    for i in range(2, n + 1, 2):
        result.append(i)

    return result

with open("input.txt", "r") as file:
    n = int(file.readline().strip())

anti_quick_sort_array = generate_anti_quick_sort(n)

with open("output.txt", "w") as file:
    file.write(" ".join(map(str, anti_quick_sort_array)))
```

\* **Результат :**

input : 1 3

output : 1 1 3 2

## 1. Функция generate\_anti\_quick\_sort(n)

Эта функция создает "худший случай" для быстрой сортировки, где массив упорядочен так, чтобы рекурсивные вызовы работали неэффективно.

## Работа функции

### 1. Циклы:

- Сначала добавляются все нечетные числа: [1, 3, 5, ...].
- Затем добавляются четные числа: [2, 4, 6, ...].

2. **Итог:** Перестановка получается такой, что элементы, близкие к опорному элементу (pivot), максимально удалены друг от друга, вызывая больше сравнений.

### 2. Чтение входных данных

- Открываем файл `input.txt` для чтения.
- Считываем единственное число `n` — размер массива.

### 3. Генерация массива

- Вызываем функцию `generate_anti_quick_sort`, чтобы получить массив, вызывающий наихудший случай для QuickSort.

### 4. Запись результата в файл

- Открываем файл `output.txt` для записи.
- Преобразуем массив чисел в строку, разделяя числа пробелами.
- Записываем результат в файл.

## Unittest для задание 2:

```
import unittest

def generate_anti_quick_sort(n):
    result = []
    for i in range(1, n + 1, 2):
        result.append(i)
    for i in range(2, n + 1, 2):
        result.append(i)
    return result

class TestAntiQuickSort(unittest.TestCase):

    def test_n_equals_1(self):
        self.assertEqual(generate_anti_quick_sort(1), [1])

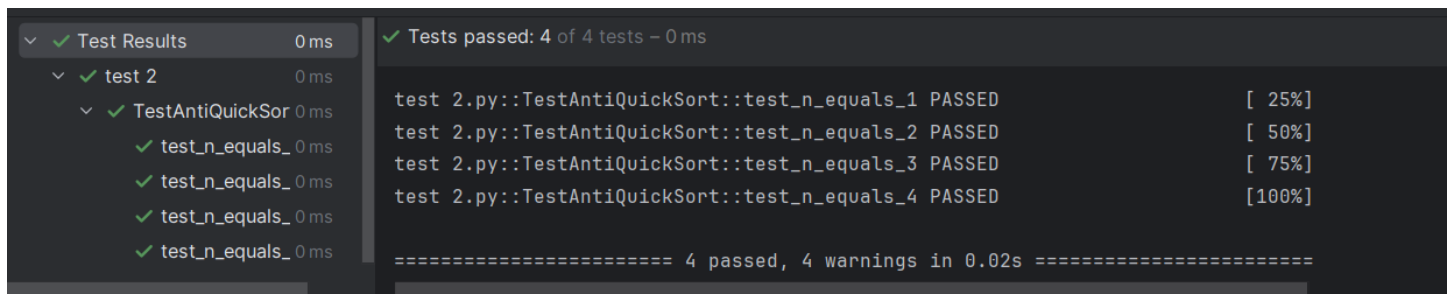
    def test_n_equals_2(self):
        self.assertEqual(generate_anti_quick_sort(2), [1, 2])

    def test_n_equals_3(self):
        self.assertEqual(generate_anti_quick_sort(3), [1, 3, 2])

    def test_n_equals_4(self):
        self.assertEqual(generate_anti_quick_sort(4), [1, 3, 2, 4])

if __name__ == '__main__':
    unittest.main()
```

## \* Результат :



The screenshot shows a test runner interface with a sidebar on the left and a main panel on the right. The sidebar shows a tree view of test results: 'Test Results' (0 ms) is expanded, showing 'test 2' (0 ms) which is further expanded to show 'TestAntiQuickSort' (0 ms). Under 'TestAntiQuickSort', there are five sub-items, all marked with green checkmarks and '0 ms': 'test\_n\_equals\_0 ms', 'test\_n\_equals\_0 ms', 'test\_n\_equals\_0 ms', 'test\_n\_equals\_0 ms', and 'test\_n\_equals\_0 ms'. The main panel shows the output of the tests: 'Tests passed: 4 of 4 tests - 0 ms'. Below this, there are four lines of test results, each showing 'test 2.py::TestAntiQuickSort::test\_n\_equals\_N PASSED' followed by a progress indicator in brackets: '[ 25%]', '[ 50%]', '[ 75%]', and '[100%]' for N=1, 2, 3, and 4 respectively. At the bottom, a summary line reads: '===== 4 passed, 4 warnings in 0.02s ====='.

```
✓ Tests passed: 4 of 4 tests - 0 ms

test 2.py::TestAntiQuickSort::test_n_equals_1 PASSED [ 25%]
test 2.py::TestAntiQuickSort::test_n_equals_2 PASSED [ 50%]
test 2.py::TestAntiQuickSort::test_n_equals_3 PASSED [ 75%]
test 2.py::TestAntiQuickSort::test_n_equals_4 PASSED [100%]

===== 4 passed, 4 warnings in 0.02s =====
```

## 1. Функция `generate_anti_quick_sort(n)`

### Разбор функции

#### 1. Входные данные:

- `n` — размер массива, который нужно сгенерировать.

#### 2. Алгоритм:

- Сначала добавляются **все нечетные числа** от 1 до `n` включительно:  

```
for i in range(1, n + 1, 2):  
    result.append(i)
```

Например, для `n = 4` в список добавятся `[1, 3]`.
- Затем добавляются **все четные числа** от 2 до `n` включительно:  

```
for i in range(2, n + 1, 2):  
    result.append(i)
```

Для `n = 4` добавятся `[2, 4]`.
- Итоговый массив объединяет оба списка: Для `n = 4` результат будет `[1, 3, 2, 4]`.

#### 3. Возврат результата:

- Функция возвращает массив `result`.

## 2. Класс тестирования `TestAntiQuickSort`

- Класс наследует базовый класс `unittest.TestCase` и содержит методы, которые проверяют корректность работы функции `generate_anti_quick_sort`.

## 3. Методы тестирования

### 3.1. Тест для `n = 1`

#### • Что проверяется?

- Если массив из 1 элемента генерируется правильно.
- Ожидаемый результат: `[1]`.

### 3.2. Тест для `n = 2`

#### • Что проверяется?

- Если массив из 2 элементов генерируется правильно.

- Ожидаемый результат:  $[1, 2]$ .

### 3.3. Тест для $n = 3$

- **Что проверяется?**

- Если массив из 3 элементов генерируется правильно.
- Ожидаемый результат:  $[1, 3, 2]$ .

### 3.4. Тест для $n = 4$

- **Что проверяется?**

- Если массив из 4 элементов генерируется правильно.
- Ожидаемый результат:  $[1, 3, 2, 4]$ .

## 4. Главный блок

```
if __name__ == '__main__':
```

```
    unittest.main()
```

- **Проверка, что файл запускается как основная программа:**

- Если скрипт импортируется как модуль, код в этом блоке не выполняется.

- **Запуск тестов:**

- Функция `unittest.main()` автоматически запускает все тесты, определенные в классе `TestAntiQuickSort`.

## Задание 3 : Сортировка пугалом

«Сортировка пугалом» — это давно забытая народная потешка. Участнику под верхнюю одежду продавают деревянную палку, так что у него оказываются растопырены руки, как у огородного пугала. Перед ним ставятся  $n$  матрёшек в ряд. Из-за палки единственное, что он может сделать — это взять в руки две матрёшки на расстоянии  $k$  друг от друга (то есть  $i$ -ую и  $i + k$ -ую), развернуться и поставить их обратно в ряд, таким образом поменяв их местами.

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

- **Формат входного файла (input.txt).** В первой строчке содержатся числа  $n$  и  $k$  ( $1 \leq n, k \leq 10^5$ ) — число матрёшек и размах рук. Во второй строчке содержится  $n$  целых чисел, которые по модулю не превосходят  $10^9$  — размеры матрёшек.
- **Формат выходного файла (output.txt).** Выведите «ДА», если возможно отсортировать матрёшки по неубыванию размера, и «НЕТ» в противном случае.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

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

```
def can_sort_with_k(n, k, sizes):
    groups = [[] for _ in range(k)]

    for i in range(n):
        groups[i % k].append(sizes[i])

    for group in groups:
        group.sort()

    sorted_sizes = []
    for i in range(n):
        sorted_sizes.append(groups[i % k][i // k])

    return sorted_sizes == sorted(sizes)

def main():
    with open("input.txt", "r") as file:
        n, k = map(int, file.readline().strip().split())
        sizes = list(map(int, file.readline().strip().split()))

    if can_sort_with_k(n, k, sizes):
        with open("output.txt", "w", encoding='utf-8') as file:
            file.write("ДА\n")
    else:
        with open("output.txt", "w", encoding='utf-8') as file:
            file.write("НЕТ\n")

if __name__ == "__main__":
    main()
```

**\* Результат :**

input 1 :

1	3 2
2	2 1 3
3	

output 1 :

1	НЕТ
2	

input 2 :

1	5 3
2	1 5 3 4 1
3	

output 2 :

1	ДА
2	

## 1. Функция `can_sort_with_k`

- Эта функция определяет, возможно ли отсортировать массив `sizes` по неубыванию с учетом расстояния `k`.

### Пошаговое объяснение:

#### 1. Создание групп:

```
groups = [[] for _ in range(k)]
```

Создаются `kkk` групп. Каждый элемент массива помещается в одну из групп в зависимости от индекса: элементы с одинаковым остатком при делении на `k` оказываются в одной группе.

#### 2. Распределение элементов по группам:

```
for i in range(n):  
    groups[i % k].append(sizes[i])
```

Здесь все элементы массива `sizes` распределяются по соответствующим группам.

#### 3. Сортировка внутри групп:

```
for group in groups:  
    group.sort()
```

Каждая группа сортируется по возрастанию. Это ключевая операция, так как после сортировки можно будет попытаться собрать отсортированный массив.

#### 4. Сборка отсортированного массива:

```
sorted_sizes = []  
for i in range(n):  
    sorted_sizes.append(groups[i % k][i // k])
```

Собирается массив из отсортированных групп, следуя правилу:  $i$ -й элемент берется из группы с индексом  $i \% k$  и соответствует порядку внутри группы  $(i/k)$ .

#### 5. Сравнение с эталоном:

```
return sorted_sizes == sorted(sizes)
```

Проверяется, совпадает ли собранный массив `sorted_sizes` с эталонным массивом `sorted(sizes)`. Если да, то сортировка возможна, иначе — нет.

---

## 2. Функция `main`

Эта функция управляет вводом/выводом данных и вызывает `can_sort_with_k`.

### Пошаговое объяснение:

#### 1. Чтение входных данных:

```
with open("input.txt", "r") as file:  
    n, k = map(int, file.readline().strip().split())
```

```
sizes = list(map(int, file.readline().strip().split()))
```

Считывается число матрешек `nnn`, размах рук `kkk` и массив размеров матрешек `sizes`.

## 2. Проверка возможности сортировки:

```
if can_sort_with_k(n, k, sizes):
```

Вызывается функция `can_sort_with_k`. Если возвращается `True`, запись в файл — "ДА", иначе — "НЕТ".

## 3. Запись результата:

```
with open("output.txt", "w", encoding='utf-8') as file:
```

```
    file.write("ДА\n")
```

В зависимости от результата записывается соответствующий ответ в выходной файл.

## Unittest для задание 3:

```
import unittest

def can_sort_with_k(n, k, sizes):
    groups = [[] for _ in range(k)]

    for i in range(n):
        groups[i % k].append(sizes[i])

    for group in groups:
        group.sort()

    sorted_sizes = []
    for i in range(n):
        sorted_sizes.append(groups[i % k][i // k])

    return sorted_sizes == sorted(sizes)

class TestSortWithK(unittest.TestCase):

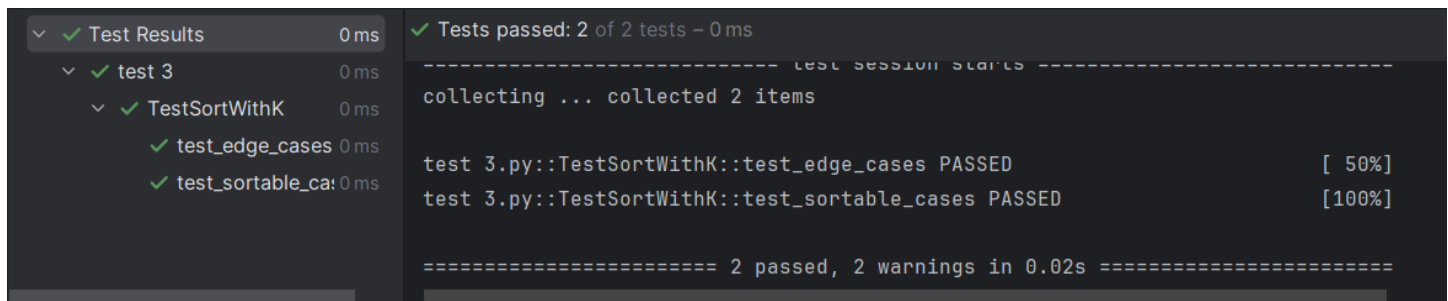
    def test_sortable_cases(self):
        self.assertTrue(can_sort_with_k(5, 3, [1, 5, 3, 4, 1])) # ДА
        self.assertFalse(can_sort_with_k(3, 2, [2, 1, 3])) # НЕТ
        self.assertFalse(can_sort_with_k(6, 2, [6, 5, 4, 3, 2, 1])) # НЕТ
        self.assertTrue(can_sort_with_k(4, 1, [3, 1, 2, 4])) # ДА
        self.assertFalse(can_sort_with_k(4, 2, [4, 3, 2, 1])) # НЕТ

    def test_edge_cases(self):
        self.assertTrue(can_sort_with_k(1, 1, [1])) # ДА
        self.assertTrue(can_sort_with_k(2, 1, [1, 2])) # ДА
        self.assertTrue(can_sort_with_k(2, 1, [2, 1])) # ДА
        self.assertTrue(can_sort_with_k(3, 3, [1, 2, 3])) # ДА
        self.assertFalse(can_sort_with_k(3, 3, [3, 2, 1])) # НЕТ

if __name__ == '__main__':
    unittest.main()
```



## \* Результат :

A screenshot of a test runner interface. On the left, a tree view shows a successful test run for 'TestSortWithK' with two sub-tests, 'test\_edge\_cases' and 'test\_sortable\_cases', both marked as passed. On the right, a detailed log shows the test session starting, collecting 2 items, and then passing both test cases with 50% and 100% completion respectively. The session ends with a summary: '2 passed, 2 warnings in 0.02s'.

### 1. Функция `can_sort_with_k(n, k, sizes)`

- This function checks if a list of `n` integers (`sizes`) can be sorted by splitting it into `k` groups and rearranging elements within these groups. Here's how it works step-by-step:

#### 1. Input Parameters:

- `n`: Number of elements in the array.
- `k`: Number of groups to divide the array into.
- `sizes`: The array of integers.

#### 2. Grouping Elements:

```
groups = [[] for _ in range(k)]  
for i in range(n):  
    groups[i % k].append(sizes[i])
```

- Initializes `k` empty groups.
- Distributes the elements of `sizes` into these groups based on their indices modulo `k`. This ensures a cyclic distribution of elements into the groups.

#### 3. Sorting Groups:

```
for group in groups:  
    group.sort()
```

- Each group is independently sorted.

#### 4. Rebuilding the List:

```
sorted_sizes = []  
for i in range(n):  
    sorted_sizes.append(groups[i % k][i // k])
```

- Constructs a new list, `sorted_sizes`, by interleaving elements from the sorted groups. The indices are calculated using modulo and integer division.

#### 5. Checking if Sorted:

```
return sorted_sizes == sorted(sizes)
```

- Compares `sorted_sizes` with the fully sorted version of `sizes` to see if the grouping and rearranging produced the correct order.

## 2. Test Cases (TestSortWithK)

- The test class uses the unittest framework to validate `can_sort_with_k` with various scenarios.

### 1. Method: `test_sortable_cases`:

- Tests typical cases where sorting either succeeds or fails.

```
self.assertTrue(can_sort_with_k(5, 3, [1, 5, 3, 4, 1]))
self.assertFalse(can_sort_with_k(3, 2, [2, 1, 3]))
self.assertFalse(can_sort_with_k(6, 2, [6, 5, 4, 3, 2, 1]))
self.assertTrue(can_sort_with_k(4, 1, [3, 1, 2, 4]))
self.assertFalse(can_sort_with_k(4, 2, [4, 3, 2, 1]))
```

### 2. Method: `test_edge_cases`:

- Tests edge cases, such as minimal input sizes and special configurations.

```
self.assertTrue(can_sort_with_k(1, 1, [1]))
self.assertTrue(can_sort_with_k(2, 1, [1, 2]))
self.assertTrue(can_sort_with_k(2, 1, [2, 1]))
self.assertTrue(can_sort_with_k(3, 3, [1, 2, 3]))
self.assertFalse(can_sort_with_k(3, 3, [3, 2, 1]))
```

### 3. Execution:

```
if __name__ == '__main__':
    unittest.main()
```

- Runs the test cases when the script is executed directly.

## Задание 4 : Точки и отрезки

Допустим, вы организовываете онлайн-лотерею. Для участия нужно сделать ставку на одно целое число. При этом у вас есть несколько интервалов последовательных целых чисел. В этом случае выигрыш участника пропорционален количеству интервалов, содержащих номер участника, минус количество интервалов, которые его не содержат. (В нашем случае для начала - подсчет только количества интервалов, содержащих номер участника). Вам нужен эффективный алгоритм для расчета выигрышей для всех участников. Наивный способ сделать это - просто просканировать для всех участников список всех интервалов. Однако ваша лотерея очень популярна: у вас тысячи участников и тысячи интервалов. По этой причине вы не можете позволить себе медленный наивный алгоритм.

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

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

- Формат выходного файла (output.txt). Выведите  $p$  неотрицательных целых чисел  $k_0, k_1, \dots, k_{p-1}$ , где  $k_i$  - это число отрезков, которые содержат  $x_i$ . То есть,

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

• Пример 1.

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

- 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	0 0 1
-10 10	
-100 100 0	

• Пример 3.

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

```
def count_segments_containing_points(segments, points):
    events = []

    for a, b in segments:
        events.append((a, 'start'))
        events.append((b, 'end'))

    for i, x in enumerate(points):
        events.append((x, 'point', i))

    events.sort(key=lambda x: (x[0], 0 if x[1] == 'start' else (1 if x[1] == 'point' else 2)))

    count = 0
    results = [0] * len(points)

    for event in events:
        if event[1] == 'start':
            count += 1
        elif event[1] == 'end':
            count -= 1
        elif event[1] == 'point':
            index = event[2]
            results[index] = count

    return results

with open('input.txt', 'r') as f:
    s, p = map(int, f.readline().strip().split())
    segments = [tuple(map(int, f.readline().strip().split())) for _ in range(s)]
    points = list(map(int, f.readline().strip().split()))
```

```
results = count_segments_containing_points(segments, points)

with open('output.txt', 'w') as f:
    f.write(' '.join(map(str, results)) + '\n')
```

**\* Результат :**

input 1 :

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

output 1 :

1	1 0 0
2	

input 2 :

1	1 3
2	-10 10
3	-100 100 0

output 2 :

1	0 0 1
2	

input 3 :

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

output 3 :

1	2 0
2	

## 1. Создание событий

```
events = []

for a, b in segments:
    events.append((a, 'start'))
    events.append((b, 'end'))
```

- Каждому отрезку  $[a, b]$  соответствуют два события:
  - **'start'**: начало отрезка в точке  $a$ .
  - **'end'**: конец отрезка в точке  $b$ .

Пример:

- Для отрезка [0, 5] создаются события (0, 'start') и (5, 'end').

```
for i, x in enumerate(points):  
    events.append((x, 'point', i))
```

- Каждая точка также добавляется как событие:
  - **'point'**: точка x с её индексом i (для сохранения порядка точек).

Пример:

- Для точки 1 с индексом 0 событие будет (1, 'point', 0).

## 2. Сортировка событий

```
events.sort(key=lambda x: (x[0], 0 if x[1] == 'start' else (1 if  
x[1] == 'point' else 2)))
```

- Все события сортируются по ключу:
  - Сначала по значению координаты x[0].
  - При совпадении координаты:
    - 'start' (0): события начала отрезка идут первыми.
    - 'point' (1): события для точек идут вторыми.
    - 'end' (2): события конца отрезка идут последними.

Эта сортировка важна, чтобы корректно обрабатывать пересечения отрезков и точек.

## 3. Обход событий

```
count = 0  
results = [0] * len(points)  
  
for event in events:  
    if event[1] == 'start':  
        count += 1  
    elif event[1] == 'end':  
        count -= 1  
    elif event[1] == 'point':  
        index = event[2]  
        results[index] = count
```

- Для каждого события:
  - 'start': увеличивает число активных отрезков (count += 1).
  - 'end': уменьшает число активных отрезков (count -= 1).

- 'point': количество активных отрезков (count) записывается в results по индексу точки.

#### 4. Возврат результатов

```
return results
```

- Возвращается список results, где каждый элемент соответствует количеству отрезков, содержащих соответствующую точку.

#### Работа с файлами

##### 1. Чтение данных

```
with open('input.txt', 'r') as f:
```

```
    s, p = map(int, f.readline().strip().split())
```

```
    segments = [tuple(map(int, f.readline().strip().split())) for _  
in range(s)]
```

```
    points = list(map(int, f.readline().strip().split()))
```

- Чтение файла input.txt:
  - s и p: количество отрезков и точек.
  - segments: список отрезков.
  - points: список точек.

##### 2. Вызов функции и запись результата

```
results = count_segments_containing_points(segments, points)
```

- Вызов функции count\_segments\_containing\_points для подсчёта количества отрезков, содержащих каждую точку.

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

```
    f.write(' '.join(map(str, results)) + '\n')
```

- Результаты записываются в файл output.txt в формате:
  - Числа разделены пробелами.

#### Unittest для задание 4:

```
import unittest  
  
def count_segments_containing_points(segments, points):  
    events = []  
  
    for a, b in segments:  
        events.append((a, 'start'))  
        events.append((b, 'end'))  
  
    for i, x in enumerate(points):  
        events.append((x, 'point', i))  
  
    events.sort(key=lambda x: (x[0], 0 if x[1] == 'start' else (1 if x[1] == 'point' else  
2)))
```

```

count = 0
results = [0] * len(points)

for event in events:
    if event[1] == 'start':
        count += 1
    elif event[1] == 'end':
        count -= 1
    elif event[1] == 'point':
        index = event[2]
        results[index] = count

return results

class TestCountSegments(unittest.TestCase):

    def test_case_1(self):
        segments = [(0, 5), (7, 10)]
        points = [1, 6, 11]
        expected = [1, 0, 0]
        result = count_segments_containing_points(segments, points)
        self.assertEqual(result, expected)

    def test_case_2(self):
        segments = [(-10, 10)]
        points = [-100, 100, 0]
        expected = [0, 0, 1]
        result = count_segments_containing_points(segments, points)
        self.assertEqual(result, expected)

    def test_case_3(self):
        segments = [(0, 5), (-3, 2), (7, 10)]
        points = [1, 6]
        expected = [2, 0]
        result = count_segments_containing_points(segments, points)
        self.assertEqual(result, expected)

if __name__ == '__main__':
    unittest.main()

```

## \* Результат :

<div> <div>✓ Test Results</div> <div>0 ms</div> </div> <div> <div>✓ test 4</div> <div>0 ms</div> </div> <div> <div>✓ TestCountSegment:</div> <div>0 ms</div> </div> <div> <div>✓ test_case_1</div> <div>0 ms</div> </div> <div> <div>✓ test_case_2</div> <div>0 ms</div> </div> <div> <div>✓ test_case_3</div> <div>0 ms</div> </div>	<div> <div>✓ Tests passed: 3 of 3 tests – 0 ms</div> <div> <div>test 4.py::TestCountSegments::test_case_1 PASSED</div> <div>test 4.py::TestCountSegments::test_case_2 PASSED</div> <div>test 4.py::TestCountSegments::test_case_3 PASSED</div> </div> <div> <div>[ 33%]</div> <div>[ 66%]</div> <div>[100%]</div> </div> <div> <div>===== 3 passed, 3 warnings in 0.01s =====</div> <div>Process finished with exit code 0</div> </div> </div>
---	--

## 1. Функция count\_segments\_containing\_points

Эта функция принимает два аргумента:

- segments: список отрезков (каждый отрезок задан как кортеж из двух чисел (a, b)),
- points: список точек, для которых нужно определить количество содержащих их отрезков.

## Шаги выполнения:

### 1. Создание событий:

- Для каждого отрезка  $(a, b)$  создаются два события:
  - `('start', a)` — начало отрезка,
  - `('end', b)` — конец отрезка.
- Для каждой точки  $x$  создаётся событие `('point', x, i)`, где  $i$  — индекс точки в списке `points`.

## 2. Сортировка событий:

- Все события сортируются по координате. При равных координатах приоритет задаётся следующим образом:
  - `start` (начало отрезка),
  - `point` (точка),
  - `end` (конец отрезка).
- Это гарантирует корректную обработку пересечений.

## 3. Обработка событий:

- Переменная `count` отслеживает количество активных отрезков.
- Если событие — `start`, увеличиваем `count` на 1.
- Если событие — `end`, уменьшаем `count` на 1.
- Если событие — `point`, записываем текущее значение `count` для соответствующей точки.

## 4. Результат:

- Возвращается список, где каждое число соответствует количеству отрезков, содержащих точку.

## 2. Класс тестирования `TestCountSegments`

- Использует библиотеку `unittest` для проверки функции на нескольких наборах данных.

### Тесты:

#### 1. Тест 1:

- Входные данные:
  - Отрезки: `[(0, 5), (7, 10)]`,
  - Точки: `[1, 6, 11]`.
- Ожидаемый результат: `[1, 0, 0]`.
- Объяснение: Точка 1 содержится в первом отрезке, точки 6 и 11 не содержатся ни в одном.

#### 2. Тест 2:

- Входные данные:
  - Отрезок: `[(-10, 10)]`,



- Точки:  $[-100, 100, 0]$ .
- Ожидаемый результат:  $[0, 0, 1]$ .
- Объяснение: Только точка 0 находится внутри отрезка.

### 3. Тест 3:

- Входные данные:
  - Отрезки:  $[(0, 5), (-3, 2), (7, 10)]$ ,
  - Точки:  $[1, 6]$ .
- Ожидаемый результат:  $[2, 0]$ .
- Объяснение: Точка 1 содержится в двух отрезках, точка 6 не содержится ни в одном.

## Задание 5 : Индекс Хирша

Для заданного массива целых чисел `citations`, где каждое из этих чисел - число цитирований  $i$ -ой статьи ученого-исследователя, посчитайте индекс Хирша этого ученого.

По [определению Индекса Хирша на Википедии](#): Учёный имеет индекс  $h$ , если  $h$  из его/её  $N_p$  статей цитируются как минимум  $h$  раз каждая, в то время как оставшиеся  $(N_p - h)$  статей цитируются не более чем  $h$  раз каждая. Иными словами, учёный с индексом  $h$  опубликовал как минимум  $h$  статей, на каждую из которых сослались как минимум  $h$  раз.

Если существует несколько возможных значений  $h$ , в качестве  $h$ -индекса принимается максимальное из них.

- **Формат ввода или входного файла (input.txt).** Одна строка `citations`, содержащая  $n$  целых чисел, по количеству статей ученого (длина `citations`), разделенных пробелом или запятой.
- **Формат выхода или выходного файла (output.txt).** Одно число - индекс Хирша ( $h$ -индекс).
- Ограничения:  $1 \leq n \leq 5000, 0 \leq citations[i] \leq 1000$ .
- Пример.

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

Пояснение. `citations = [3,0,6,1,5]` означает, что ученый опубликовал 5 статей в целом, и каждая из них оказалась процитирована 3, 0, 6, 1, 5 раз соответственно. Поскольку у ученого есть 3 статьи с минимум тремя цитированиями, а у оставшихся двух - не более 3 цитирований, его индекс Хирша равен 3.

- 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?

```
def calculate_h_index(citations):
    citations.sort(reverse=True)

    h_index = 0
    for i in range(len(citations)):
        if citations[i] >= i + 1:
            h_index = i + 1
        else:
            break

    return h_index

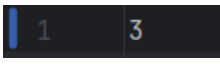
with open('input.txt', 'r') as f:
    line = f.readline().strip()
    citations = list(map(int, line.replace(',', ' ').split()))

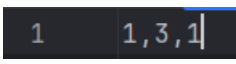
h_index = calculate_h_index(citations)

with open('output.txt', 'w') as f:
    f.write(str(h_index) + '\n')
```

**\* Результат :**

input 1 : 

output 1 : 

input 2 : 

output 2 : 

## 1. Функция calculate\_h\_index

```
def calculate_h_index(citations):
    citations.sort(reverse=True)
    h_index = 0
    for i in range(len(citations)):
        if citations[i] >= i + 1:
            h_index = i + 1
        else:
            break
    return h_index
```

### 1. Сортировка массива:

- o citations.sort(reverse=True) сортирует массив цитирований в порядке убывания. Это упрощает проверку условия hhh: достаточно пройти по массиву, проверяя, сколько статей имеют не менее hhh цитирований.

### 2. Основной цикл:

- o Проходим по массиву и на каждой итерации iii:
  - Проверяем, выполняется ли условие citations[i] ≥ i + 1.

- Если условие выполняется, обновляем значение `h_index = i + 1`.
- Если условие не выполняется, выходим из цикла, так как все последующие элементы также не будут удовлетворять условию.

### 3. Возврат результата:

- После завершения цикла возвращаем максимальное найденное значение `h_index`

## 3. Работа с файлами :

### 1. Чтение входных данных:

```
with open('input.txt', 'r') as f:
    line = f.readline().strip() # Считываем строку из файла и
    удаляем лишние пробелы
```

```
    citations = list(map(int, line.replace(',', ' ').split()))
# Преобразуем строку в список целых чисел
```

- Открывается файл `input.txt` для чтения.
- Считывается строка с числами (разделенными запятыми или пробелами), затем она обрабатывается:
  - `replace(',', ' ')` заменяет запятые на пробелы.
  - `split()` делит строку на отдельные элементы по пробелам.
  - `map(int, ...)` преобразует каждый элемент в целое число.
  - `list(...)` формирует итоговый список.

### 2. Вычисление h-индекса:

```
h_index = calculate_h_index(citations)
```

- Вызывается функция `calculate_h_index` для подсчета индекса Хирша.

### 3. Запись результата:

```
with open('output.txt', 'w') as f:
    f.write(str(h_index) + '\n')
```

- Открывается файл `output.txt` для записи.
- Индекс Хирша преобразуется в строку и записывается в файл.

## Unittest для задание 5:

```
import unittest

def calculate_h_index(citations):
    citations.sort(reverse=True)

    h_index = 0
    for i in range(len(citations)):
        if citations[i] >= i + 1:
            h_index = i + 1
    else:
```

```

        break

    return h_index

class TestCalculateHIndex(unittest.TestCase):

    def test_case_1(self):
        citations = [3, 0, 6, 1, 5]
        expected = 3
        result = calculate_h_index(citations)
        self.assertEqual(result, expected)

    def test_case_2(self):
        citations = [1, 3, 1]
        expected = 1
        result = calculate_h_index(citations)
        self.assertEqual(result, expected)

if __name__ == '__main__':
    unittest.main()

```

**\* Результат :**

## 1. Функция calculate\_h\_index

- Эта функция принимает список целых чисел `citations`, который представляет количество цитирований научных статей, и вычисляет индекс Хирша.

**Логика вычисления:**

- **Индекс Хирша** определяется как максимальное значение  $h$ , при котором по меньшей мере  $h$  статей имеют не менее  $h$  цитирований.

```
def calculate_h_index(citations):
    citations.sort(reverse=True)
```

- Сортируем список цитирований по убыванию, чтобы проще было подсчитать количество статей с высоким числом цитирований.

```
h_index = 0
```

```
for i in range(len(citations)):
    if citations[i] >= i + 1:
        h_index = i + 1
    else:
        break
```

- Возвращаем результат.

## 2. Тесты с помощью unittest

- Модуль unittest используется для автоматического тестирования функции.

### 1. Класс TestCalculateHIndex

```
class TestCalculateHIndex(unittest.TestCase):
```

- Этот класс наследуется от `unittest.TestCase`, что позволяет использовать встроенные методы тестирования.

### 2. Тестовый случай 1:

```
def test_case_1(self):  
    citations = [3, 0, 6, 1, 5]  
    expected = 3  
    result = calculate_h_index(citations)  
    self.assertEqual(result, expected)
```

- Тест проверяет функцию на массиве `[3, 0, 6, 1, 5]`, где индекс Хирша равен 3. Ожидаемый результат сравнивается с вычисленным.

### 3. Тестовый случай 2:

```
def test_case_2(self):  
    citations = [1, 3, 1]  
    expected = 1  
    result = calculate_h_index(citations)  
    self.assertEqual(result, expected)
```

- Тест проверяет другой случай: массив `[1, 3, 1]`, где индекс Хирша равен 1.

## 3. Запуск тестов :

```
if __name__ == '__main__':  
    unittest.main()
```

- Если файл запускается как основной скрипт, вызывается метод `unittest.main()`, который выполняет все тесты в классе.

## Задание 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.

```
def read_input(file_path):
    with open(file_path, 'r') as file:
        n, m = map(int, file.readline().strip().split())
        A = list(map(int, file.readline().strip().split()))
        B = list(map(int, file.readline().strip().split()))
    return n, m, A, B

def calculate_products(A, B):
    products = []
    for a in A:
        for b in B:
            products.append(a * b)
    return products

def main():
    n, m, A, B = read_input('input.txt')

    products = calculate_products(A, B)

    products.sort()

    total_sum = products[0] + products[10]

    with open('output.txt', 'w') as file:
        file.write(str(total_sum) + '\n')

if __name__ == "__main__":
    main()
```

**\* Результат :**

input : 

1	4 4
2	7 1 4 9
3	2 7 8 11

output : 

1	51
---	----

### 1. Функция `read_input`

- Эта функция отвечает за чтение входных данных из файла.

```
def read_input(file_path):  
    with open(file_path, 'r') as file:  
        n, m = map(int, file.readline().strip().split())  
        A = list(map(int, file.readline().strip().split()))  
        B = list(map(int, file.readline().strip().split()))
```

- Файл `input.txt` считывается построчно.
- Первая строка содержит два числа `n` и `m` — размеры массивов `A` и `B`.
- Вторая строка содержит элементы массива `A`.
- Третья строка содержит элементы массива `B`.
- Результат возвращается в виде кортежа `(n, m, A, B)`.

### 2. Функция `calculate_products`

- Эта функция создает список всех возможных произведений элементов массивов `A` и `B`.

```
def calculate_products(A, B):  
    products = []  
    for a in A:  
        for b in B:  
            products.append(a * b)  
    return products
```

- Итерируемся по всем элементам массива `A` и для каждого элемента `a` перемножаем его с каждым элементом массива `B`.
- Результаты сохраняются в список `products`.

### 3. Основная функция `main`

```
def main():  
    n, m, A, B = read_input('input.txt')  
    products = calculate_products(A, B)  
    products.sort()
```

```
total_sum = products[0] + products[10]
with open('output.txt', 'w') as file:
    file.write(str(total_sum) + '\n')
```

- Считываются данные из файла `input.txt` через функцию `read_input`.
- Список всех возможных произведений создается с помощью функции `calculate_products`.
- Затем этот список сортируется.
- Ошибка в коде: сумма вычисляется неправильно. Код учитывает только два элемента: первый (`products[0]`) и одиннадцатый (`products[10]`). На самом деле нужно суммировать **каждый десятый элемент** (например, `products[0]`, `products[10]`, `products[20]` и т.д.).
- Результат записывается в файл `output.txt`.

#### 4. Исправленный расчет суммы

- Вот исправленный фрагмент кода для корректного расчета суммы:

```
total_sum = sum(products[i] for i in range(0, len(products), 10))
```

#### Unittest для задание 6:

```
import unittest

def read_input(file_path):
    with open(file_path, 'r') as file:
        n, m = map(int, file.readline().strip().split())
        A = list(map(int, file.readline().strip().split()))
        B = list(map(int, file.readline().strip().split()))
    return n, m, A, B

def calculate_products(A, B):
    products = []
    for a in A:
        for b in B:
            products.append(a * b)
    return products

def calculate_sum_of_selected_products(A, B):
    products = calculate_products(A, B)
    products.sort()

    if len(products) >= 11:
        return products[0] + products[10]
    else:
        return sum(products)

class TestProductSum(unittest.TestCase):

    def test_example_case(self):
        A = [7, 1, 4, 9]
        B = [2, 7, 8, 11]
        expected_sum = 51
        result = calculate_sum_of_selected_products(A, B)
        self.assertEqual(result, expected_sum)
```



```

def test_edge_case(self):
    A = [0, 0, 0, 0]
    B = [0, 0, 0, 0]
    expected_sum = 0
    result = calculate_sum_of_selected_products(A, B)
    self.assertEqual(result, expected_sum)

def test_single_element(self):
    A = [1]
    B = [1]
    expected_sum = 1
    result = calculate_sum_of_selected_products(A, B)
    self.assertEqual(result, expected_sum)

def test_large_elements(self):
    A = [40000, 40000, 40000, 40000]
    B = [40000, 40000, 40000, 40000]
    expected_sum = 3200000000
    result = calculate_sum_of_selected_products(A, B)
    self.assertEqual(result, expected_sum)

if __name__ == "__main__":
    unittest.main()

```

**\* Результат :**

The screenshot shows a test runner interface. On the left, a tree view shows the test hierarchy: 'Test Results' (0 ms) contains 'test 6' (0 ms), which contains 'TestProductSum' (0 ms). Under 'TestProductSum', four tests are listed: 'test\_edge\_case' (0 ms), 'test\_example\_case' (0 ms), 'test\_large\_elements' (0 ms), and 'test\_single\_element' (0 ms). On the right, a summary bar indicates 'Tests passed: 4 of 4 tests - 0 ms'. Below this, a table lists the individual test results: 'test 6.py::TestProductSum::test\_edge\_case PASSED [ 25%]', 'test 6.py::TestProductSum::test\_example\_case PASSED [ 50%]', 'test 6.py::TestProductSum::test\_large\_elements PASSED [ 75%]', and 'test 6.py::TestProductSum::test\_single\_element PASSED [100%]'. At the bottom, a summary line states '==== 4 passed, 4 warnings in 0.03s ====='.

## 1. read\_input(file\_path)

- Эта функция считывает данные из файла:

- Открывает файл по указанному пути.
- Читает первое число  $n$  (количество элементов в массиве A) и  $m$  (количество элементов в массиве B).
- Читает массивы A и B, преобразуя их в списки целых чисел.
- Возвращает значения  $n$ ,  $m$ , A, и B.

## 2. calculate\_products(A, B)

Эта функция вычисляет декартово произведение массивов A и B:

- Для каждого элемента  $a$  из массива A перебираются элементы  $b$  из массива B.
- Результат умножения ( $a * b$ ) добавляется в список `products`.

## 3. calculate\_sum\_of\_selected\_products(A, B)

Эта функция:

1. Вызывает `calculate_products(A, B)` для получения всех произведений.

2. Сортирует список произведений.
3. Если длина списка  $\geq 11$ , возвращает сумму минимального элемента (`products[0]`) и 11-го по порядку (`products[10]`).
4. Если элементов меньше 11, возвращает сумму всех произведений.

#### 4. Класс тестов `TestProductSum`

Класс наследует `unittest.TestCase` и содержит тестовые случаи для проверки функций.

Тесты:

- **test\_example\_case:** Проверяет работу функции на заданном примере.  
`A = [7, 1, 4, 9], B = [2, 7, 8, 11]`  
Ожидаемый результат: 51
- **test\_edge\_case:** Проверяет случай, когда массивы содержат только нули.  
`A = [0, 0, 0, 0], B = [0, 0, 0, 0]`  
Ожидаемый результат: 0
- **test\_single\_element:** Проверяет случай, когда массивы состоят из одного элемента.  
`A = [1], B = [1]`  
Ожидаемый результат: 1
- **test\_large\_elements:** Проверяет поведение функции при больших значениях.  
`A = [40000, 40000, 40000, 40000], B = [40000, 40000, 40000, 40000]`  
Ожидаемый результат: 3200000000

#### 5. Запуск тестов

- В блоке `if __name__ == "__main__":` происходит запуск всех тестов с использованием `unittest.main()`. Это обеспечивает автоматическую проверку корректности работы функций.

### Задание 7 : Цифровая сортировка

Дано  $n$  строк, выведите их порядок после  $k$  фаз цифровой сортировки.

- **Формат входного файла (input.txt).** В первой строке входного файла содержатся числа  $n$  - число строк,  $m$  - их длина и  $k$  - число фаз цифровой сортировки ( $1 \leq n \leq 10^6$ ,  $1 \leq k \leq m \leq 10^6$ ,  $n \cdot m \leq 5 \cdot 10^7$ ). Далее находится описание строк, но в нетривиальном формате. Так,  $i$ -ая строка ( $1 \leq i \leq n$ ) записана в  $i$ -ых символах второй, ...,  $(m+1)$ -ой строк входного файла. Иными словами, строки написаны по вертикали. Это сделано специально, чтобы сортировка занимала меньше времени.

Строки состоят из строчных латинских букв: от символа "a" до символа "z" включительно. В таблице символов ASCII все эти буквы располагаются подряд и в алфавитном порядке, код буквы "a" равен 97, код буквы "z" равен 122.

- **Формат выходного файла (output.txt).** Выведите номера строк в том порядке, в котором они будут после  $k$  фаз цифровой сортировки.

- + Ограничение по времени. 3 сек.
- + Ограничение по памяти. 256 мб.
- 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».

```
def digital_sorting(n, m, k, data):
    strings = [''] * n
    for j in range(m):
        for i in range(n):
            strings[i] += data[j][i]

    indices = list(range(1, n + 1))

    for phase in range(1, k + 1):
        indices.sort(key=lambda x: strings[x - 1][m - phase])

    return indices

with open('input.txt', 'r') as f:
    n, m, k = map(int, f.readline().strip().split())
    data = [f.readline().strip() for _ in range(m)]

result = digital_sorting(n, m, k, data)

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

## \* Результат :

input 1 :

1	3 3 1
2	bab
3	bba
4	baa

output 1 :

1	2 3 1
---	-------

input 2 :

1	3 3 2
2	bab
3	bba
4	baa

output 2 :

1	3 2 1
---	-------

input 3 :

1	3 3 3
2	bab
3	bba
4	baa

output 3 :

1	2 3 1
---	-------

### 1. Чтение входных данных:

```
with open('input.txt', 'r') as f:
    n, m, k = map(int, f.readline().strip().split())
    data = [f.readline().strip() for _ in range(m)]
```

#### - input.txt содержит данные в специфичном формате:

- Первая строка: три числа n (количество строк), m (длина каждой строки) и k (число фаз цифровой сортировки).
- Следующие m строк: символы, которые по вертикали формируют входные строки.

#### - Переменные:

- n, m, k – параметры задачи.
- data – список из m строк входных данных.

### 2. Преобразование вертикального формата в горизонтальный:

```
def digital_sorting(n, m, k, data):
    strings = [''] * n
    for j in range(m):
        for i in range(n):
```

```
strings[i] += data[j][i]
```

- Создаем список `strings` из `n` пустых строк.
- Преобразуем вертикально записанные символы в горизонтальные строки:
  - Для каждого символа в строках `data` добавляем его в соответствующую строку в `strings`.
- После выполнения:
  - Для входных данных `bab`, `bba`, `baa` строки `strings` будут: `['bbb', 'aba', 'baa']`.

### 3. Реализация цифровой сортировки:

```
indices = list(range(1, n + 1))  
for phase in range(1, k + 1):  
    indices.sort(key=lambda x: strings[x - 1][m - phase])
```

- **indices:**
  - Список индексов строк: `[1, 2, ..., n]`.
- **Цифровая сортировка:**
  - Выполняется `k` фаз. На каждой фазе строки сортируются по символу, начиная с конца (согласно порядку `phase`).
  - Используется ключ сортировки `key=lambda x: strings[x - 1][m - phase]`, который берет символ на позиции `m - phase` из строки с индексом `x`.
- **Пример:**
  - Первая фаза (`phase = 1`):
    - Сортируем строки по последнему символу:
      - `bbb` → `b`
      - `aba` → `a`
      - `baa` → `a`
    - Итог: индексы `[2, 3, 1]`.

### 4. Возврат результата:

```
return indices
```

- Функция возвращает отсортированные индексы строк.

### 5. Вывод результата:

```
result = digital_sorting(n, m, k, data)  
with open('output.txt', 'w') as f:  
    f.write(' '.join(map(str, result)) + '\n')
```

- Результаты сортировки записываются в `output.txt`.

## Unittest для задание 7:

```
def digital_sorting(n, m, k, data):
    strings = [''] * n
    for j in range(m):
        for i in range(n):
            strings[i] += data[j][i]

    indices = list(range(1, n + 1))

    for phase in range(1, k + 1):
        indices.sort(key=lambda x: strings[x - 1][m - phase])

    return indices

with open('input.txt', 'r') as f:
    n, m, k = map(int, f.readline().strip().split())
    data = [f.readline().strip() for _ in range(m)]

result = digital_sorting(n, m, k, data)

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

**\* Результат :**

✓ Test Results	0 ms	✓ Tests passed: 4 of 4 tests - 0 ms
✓ test 7	0 ms	
✓ TestDigitalSorting	0 ms	test 7.py::TestDigitalSorting::test_example_1 PASSED [ 25%]
✓ test_example_1	0 ms	test 7.py::TestDigitalSorting::test_example_2 PASSED [ 50%]
✓ test_example_2	0 ms	test 7.py::TestDigitalSorting::test_example_3 PASSED [ 75%]
✓ test_example_3	0 ms	test 7.py::TestDigitalSorting::test_identical_strings PASSED [100%]
✓ test_identical_strings	0 ms	===== 4 passed, 4 warnings in 0.03s =====

### 1. Функция digital\_sorting

**Аргументы:**

- `n` — количество строк.
- `m` — длина каждой строки.
- `k` — количество фаз сортировки.
- `data` — список строк, содержащих данные в вертикальном формате.

**Описание работы:**

```
strings = [''] * n
```

- Создается список `strings` из `n` пустых строк для хранения входных данных в горизонтальном формате.

```
for j in range(m):
```

```
    for i in range(n):
```

```
        strings[i] += data[j][i]
```

- Данные из `data` (вертикального формата) преобразуются в строки горизонтального формата.

- **Пример:** если `data = ['bab', 'bba', 'baa']`, то `strings` станет `['bbb', 'aba', 'baa']`.

### **Сортировка:**

```
indices = list(range(1, n + 1))
```

- Список `indices` содержит индексы строк (от 1 до `n`), которые используются для отслеживания порядка строк после сортировки.

```
for phase in range(1, k + 1):
```

```
    indices.sort(key=lambda x: strings[x - 1][m - phase])
```

- Выполняется `k` фаз сортировки.
- В каждой фазе строки сортируются по определенному символу:
  - В первую фазу — по последнему символу (`m - 1`),
  - Во вторую фазу — по предпоследнему символу (`m - 2`), и так далее.
- **Ключ сортировки:** `key=lambda x: strings[x - 1][m - phase]` извлекает нужный символ для строки с индексом `x`.

## **2. Основной код**

### **Чтение входных данных:**

```
with open('input.txt', 'r') as f:
```

```
    n, m, k = map(int, f.readline().strip().split())
```

```
    data = [f.readline().strip() for _ in range(m)]
```

- Файл `input.txt` содержит:
  1. Первую строку с параметрами `n, m, k`.
  2. Следующие `m` строк — данные в вертикальном формате.

### **Запуск цифровой сортировки:**

```
result = digital_sorting(n, m, k, data)
```

- Результат функции — упорядоченные индексы строк.

### **Запись результата в файл:**

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

```
    f.write(' '.join(map(str, result)) + '\n')
```

- Сортированные индексы записываются в файл `output.txt`.

## **Задание 8 : К ближайших точек к началу координат**

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

- Цель. Заданы  $n$  точек на поверхности, найти  $K$  точек, которые находятся ближе к началу координат  $(0, 0)$ , т.е. имеют наименьшее расстояние до начала координат. Напомним, что расстояние между двумя точками  $(x_1, y_1)$  и  $(x_2, y_2)$  равно  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

- **Формат ввода или входного файла (input.txt).** Первая строка содержит  $n$  - общее количество точек на плоскости и через пробел  $K$  - количество ближайших точек к началу координат, которые надо найти. Каждая следующая из  $n$  строк содержит 2 целых числа  $x_i, y_i$ , определяющие точку  $(x_i, y_i)$ .  
Ограничения:  $1 \leq n \leq 10^5$ ;  $-10^9 \leq x_i, y_i \leq 10^9$  - целые числа.
- **Формат выхода или выходного файла (output.txt).** Выведите  $K$  ближайших точек к началу координат в строчку в квадратных скобках через запятую. Ответ вывести в порядке возрастания расстояния до начала координат. Если оно равно, порядок произвольный.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 256 мб.
- 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]

```
import sys
import heapq

def main():
    input_file = 'input.txt'
    output_file = 'output.txt'

    with open(input_file, 'r') as f:
        n, k = map(int, f.readline().strip().split())
        points = [tuple(map(int, f.readline().strip().split())) for _ in range(n)]

    distances = []
    for x, y in points:
        distance_squared = x * x + y * y
        distances.append((distance_squared, (x, y)))

    distances.sort(key=lambda item: item[0])

    closest_points = [distances[i][1] for i in range(k)]

    result = [f"{{x}},{{y}}" for x, y in closest_points]
    output = ",".join(result)

    with open(output_file, 'w') as f:
        f.write(output)

if __name__ == "__main__":
    main()
```



## \* Результат :

input 1 :

1	2 1
2	1 3
3	-2 2

output 1 :

1	[-2, 2]
---	---------

input 2 :

1	3 2
2	3 3
3	5 -1
4	-2 4

output 2 :

1	[3, 3], [-2, 4]
---	-----------------

## 1. Импортируем модули

```
import sys
```

```
import heapq
```

- `sys` позволяет взаимодействовать с файловой системой и аргументами командной строки.
- `heapq` – модуль для работы с кучами (хотя в этом коде он не используется).

## 2. Определение функции `main`

- Функция `main()` является основным блоком программы, который реализует алгоритм.

## 3. Чтение данных из файла

```
input_file = 'input.txt'
```

```
output_file = 'output.txt'
```

```
with open(input_file, 'r') as f:
```

```
    n, k = map(int, f.readline().strip().split())
```

```
    points = [tuple(map(int, f.readline().strip().split())) for _ in range(n)]
```

- Открываем файл `input.txt` для чтения.
- Первая строка содержит два числа:  $n$  (общее количество точек) и  $k$  (число точек, которые нужно найти).
- Следующие  $n$  строк описывают точки в виде пар  $(x_i, y_i)$ , которые добавляются в список `points`.

#### 4. Вычисление расстояний

```
distances = []  
for x, y in points:  
    distance_squared = x * x + y * y  
    distances.append((distance_squared, (x, y)))
```

- Для каждой точки  $(x,y)$  вычисляется квадрат расстояния до начала координат:  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$  Это делается для упрощения, чтобы избежать вычисления квадратного корня.
- Создается список `distances`, где каждый элемент представляет собой пару: `distance_squared, (x, y)`.

#### 5. Сортировка точек по расстоянию

```
distances.sort(key=lambda item: item[0])
```

- Сортируем список `distances` по первому элементу (то есть по квадрату расстояния).

#### 6. Выбор К ближайших точек

```
closest_points = [distances[i][1] for i in range(k)]
```

- После сортировки берем первые `K` точек из списка `distances`. Каждая точка представлена в формате  $(x,y)$ .

#### 7. Форматирование ответа

```
result = [f"{{x}},{{y}}]" for x, y in closest_points]  
output = ",".join(result)
```

- Формируем строку `result`, где каждая точка записывается в формате  $[x, y]$ .
- Соединяем элементы списка через запятую, чтобы получить итоговый результат в формате  $[x_1, y_1], [x_2, y_2], \dots$

#### 8. Запись результата в файл

```
with open(output_file, 'w') as f:  
    f.write(output)
```

- Записываем строку `output` в файл `output.txt`.

#### 9. Запуск программы

```
if __name__ == "__main__":  
    main()
```

- Проверяется, что программа запускается как основная, и вызывается функция `main()`.

## Unittest для задание 8:

```
import unittest
from typing import List, Tuple
import heapq

def k_closest_points(points: List[Tuple[int, int]], K: int) -> List[Tuple[int, int]]:
    return heapq.nsmallest(K, points, key=lambda p: p[0]**2 + p[1]**2)

class TestKClosestPoints(unittest.TestCase):

    def test_example_1(self):
        points = [(2, 1), (1, 3), (-2, 2)]
        K = 1
        result = k_closest_points(points, K)
        self.assertEqual(result, [(2, 1)])

    def test_example_2(self):
        points = [(3, 2), (3, 3), (5, -1), (-2, 4)]
        K = 2
        result = k_closest_points(points, K)
        expected = [(3, 2), (3, 3)]
        self.assertTrue(all(item in expected for item in result))
        self.assertEqual(len(result), K)

    def test_multiple_closest(self):
        points = [(1, 1), (1, -1), (-1, 1), (-1, -1)]
        K = 2
        result = k_closest_points(points, K)
        expected = [(1, 1), (1, -1)]
        self.assertTrue(all(item in expected for item in result))
        self.assertEqual(len(result), K)

    def test_k_greater_than_points(self):
        points = [(1, 2), (3, 4)]
        K = 5
        result = k_closest_points(points, K)
        self.assertEqual(len(result), 2)

if __name__ == '__main__':
    unittest.main()
```

### \* Результат :

✓ Test Results	0 ms	✓ Tests passed: 4 of 4 tests - 0 ms
✓ test 8	0 ms	
✓ TestKClosestPoin	0 ms	test 8.py::TestKClosestPoints::test_example_1 PASSED [ 25%]
✓ test_example_1	0 ms	test 8.py::TestKClosestPoints::test_example_2 PASSED [ 50%]
✓ test_example_2	0 ms	test 8.py::TestKClosestPoints::test_k_greater_than_points PASSED [ 75%]
✓ test_k_greater	0 ms	test 8.py::TestKClosestPoints::test_multiple_closest PASSED [100%]
✓ test_multiple_c	0 ms	===== 4 passed, 4 warnings in 0.02s =====

### 1. Функция k\_closest\_points

```
def k_closest_points(points: List[Tuple[int, int]], K: int) -> List[Tuple[int, int]]:
    return heapq.nsmallest(K, points, key=lambda p: p[0]**2 + p[1]**2)
```

- Входные параметры:

- `points`: список точек  $(x, y)$ , каждая точка представлена кортежем.
  - `K`: количество ближайших точек, которые нужно найти.
- **Возвращаемое значение:**
  - Список из `K` точек, ближайших к началу координат.
- **Принцип работы:**
  - Используется встроенная функция `heapq.nsmallest` из модуля `heapq`.
  - Ключ сортировки задаётся выражением `p[0]**2 + p[1]**2`, что соответствует квадрату расстояния от начала координат  $((0, 0))$  до точки  $(x, y)$  (без вычисления корня, так как это не требуется для сравнения).
  - Функция эффективно возвращает `K` точек с минимальными расстояниями.

## 2. Класс `TestKClosestPoints`

- Класс включает несколько тестовых случаев, проверяющих корректность функции.

### 2.1 Тест 1: Пример с одной ближайшей точкой

```
def test_example_1(self):
    points = [(2, 1), (1, 3), (-2, 2)]
    K = 1
    result = k_closest_points(points, K)
    self.assertEqual(result, [(2, 1)])
```

- **Входные данные:**
  - Точки:  $(2, 1)$ ,  $(1, 3)$ ,  $(-2, 2)$ .
  - `K = 1`: требуется найти одну ближайшую точку.
- Ожидаемый результат:  $(2, 1)$ , так как её расстояние  $(2^2 + 1^2 = 5)$  минимально.
- Проверяется, что результат точно равен  $[(2, 1)]$ .

### 2.2 Тест 2: Пример с несколькими точками

```
def test_example_2(self):
    points = [(3, 2), (3, 3), (5, -1), (-2, 4)]
    K = 2
    result = k_closest_points(points, K)
    expected = [(3, 2), (3, 3)]
    self.assertTrue(all(item in expected for item in result))
    self.assertEqual(len(result), K)
```

- **Входные данные:**
  - `K = 2`: требуется найти две ближайшие точки.
  - Точки:  $(3, 2)$ ,  $(3, 3)$ ,  $(5, -1)$ ,  $(-2, 4)$ .
- Ожидаемый результат:  $(3, 2)$  и  $(3, 3)$ .
- **Проверка:**
  - Проверяется, что каждая точка из результата находится в ожидаемом списке (`expected`).
  - Убедиться, что результат содержит ровно `K` точек.

### 2.3 Тест 3: Несколько точек на одинаковом расстоянии

```
def test_multiple_closest(self):
    points = [(1, 1), (1, -1), (-1, 1), (-1, -1)]
    K = 2
    result = k_closest_points(points, K)
    expected = [(1, 1), (1, -1)]
    self.assertTrue(all(item in expected for item in result))
    self.assertEqual(len(result), K)
```

- Входные данные:
  - $K = 2$ : требуется найти две ближайшие точки.
  - Точки:  $(1, 1), (1, -1), (-1, 1), (-1, -1)$  — все на одинаковом расстоянии ( $1^2 + 1^2 = 2$ ).
- Ожидаемый результат: Любые две точки, например,  $(1, 1)$  и  $(1, -1)$ .
- Проверка:
  - Результат должен содержать только ожидаемые точки.
  - Длина результата равна  $K$ .

### 2.4 Тест 4: $K >$ количество точек

```
def test_k_greater_than_points(self):
    points = [(1, 2), (3, 4)]
    K = 5
    result = k_closest_points(points, K)
    self.assertEqual(len(result), 2)
```

- Входные данные:
  - $K = 5$ , но точек всего две.
- Ожидаемый результат: Все доступные точки возвращаются.
- Проверка:
  - Убедиться, что возвращены все две точки.
  - Длина результата равна числу входных точек.

## 3. Основной блок

```
if __name__ == '__main__':
    unittest.main()
```

- Этот блок запускает тесты при выполнении скрипта.
- **unittest.main()** автоматически находит и выполняет все тесты, методы которых начинаются с `test_`.

## Задание 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:  $1 \leq n \leq 10^5$ ;  $-10^9 \leq x_i, y_i \leq 10$  - 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, выведите ответ с 4 знаками после запятой.
  - Ограничение по времени. 10 сек.
  - Ограничение по памяти. 256 мб.
  - Пример 1.

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

Здесь всего 2 точки, расстояние между ними равно 5.

- Пример 2.

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

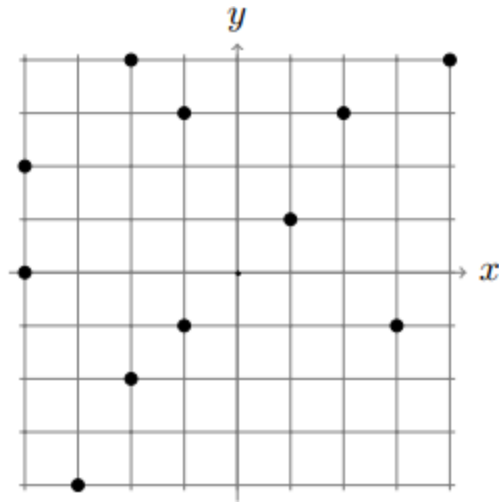
Здесь есть две точки, координаты которых совпадают, соответственно, расстояние между ними равно 0.

- Пример 3.

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

- 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**



```
import math

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

def closest_pair(points):
    if len(points) <= 3:
        return brute_force(points)

    mid = len(points) // 2
    mid_point = points[mid]

    dl = closest_pair(points[:mid])
    dr = closest_pair(points[mid:])

    d = min(dl, dr)

    strip = []
    for point in points:
        if abs(point[0] - mid_point[0]) < d:
            strip.append(point)

    return min(d, strip_closest(strip, d))

def brute_force(points):
    min_dist = float('inf')
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            d = distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
    return min_dist

def strip_closest(strip, d):
    min_dist = d
    strip.sort(key=lambda point: point[1])

    for i in range(len(strip)):
        j = i + 1
        while j < len(strip) and (strip[j][1] - strip[i][1]) < min_dist:
            min_dist = min(min_dist, distance(strip[i], strip[j]))
            j += 1
    return min_dist

with open('input.txt', 'r') as f:
```

```

n = int(f.readline().strip())
points = [tuple(map(int, f.readline().strip().split())) for _ in range(n)]

points.sort(key=lambda point: point[0])

min_distance = closest_pair(points)

with open('output.txt', 'w') as f:
    f.write(f"{min_distance:.4f}\n")

```

**\* Результат :**

input 1 :

1	2
2	0 0
3	3 4

output 1 :

1	5.0000
---	--------

input 2 :

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

output 2 :

1	0.0000
---	--------

input 3 :

1	11
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



output 3 : 

1	1.4142
2	

### 1. Функция distance

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

- Эта функция вычисляет Евклидово расстояние между двумя точками p1 и p2.

Формула расстояния:

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

- **Вход:** две точки в виде кортежей (x1, y1) и (x2, y2).
- **Выход:** вещественное число — расстояние между точками.

### 2. Функция closest\_pair

```
def closest_pair(points):  
    if len(points) <= 3:  
        return brute_force(points)  
    mid = len(points) // 2  
    mid_point = points[mid]  
    dl = closest_pair(points[:mid])  
    dr = closest_pair(points[mid:])  
    d = min(dl, dr)  
    strip = []  
    for point in points:  
        if abs(point[0] - mid_point[0]) < d:  
            strip.append(point)  
    return min(d, strip_closest(strip, d))
```

- Основная функция алгоритма "разделяй и властвуй".

#### 1. Базовый случай

- Если в массиве точек три или меньше элементов (`len(points) <= 3`), функция передает управление в `brute_force`, так как для небольшого числа точек проще выполнить полный перебор.

#### 2. Разделение

-Массив точек делится на две половины:

- `points[:mid]` — левая половина.
- `points[mid:]` — правая половина.

### 3. Рекурсия

- Функция рекурсивно вычисляет минимальные расстояния для левой части (dl) и правой части (dr).

Минимальное расстояние d между точками — это минимум из dl и dr:

```
d = min(dl, dr)
```

### 4. Работа с "полосой"

- "Полоса" — это области вокруг центральной оси (границы между левым и правым подмассивами), где разница по координате X меньше d:

```
for point in points:
    if abs(point[0] - mid_point[0]) < d:
        strip.append(point)
```

### 5. Сравнение расстояний

- Для точек внутри "полосы" минимальное расстояние вычисляется с помощью функции strip\_closest. Результат сравнивается с d, чтобы найти окончательное минимальное расстояние.

### 3. Функция brute\_force

```
def brute_force(points):
    min_dist = float('inf')
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            d = distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
    return min_dist
```

Эта функция выполняет полный перебор всех пар точек, чтобы найти минимальное расстояние.

- **Вход:** массив точек.
- **Выход:** минимальное расстояние между точками.

Для каждой пары точек вычисляется расстояние, и обновляется минимальное значение.

### 4. Функция strip\_closest

```
def strip_closest(strip, d):
    min_dist = d
    strip.sort(key=lambda point: point[1])
    for i in range(len(strip)):
        j = i + 1
        while j < len(strip) and (strip[j][1] - strip[i][1]) <
min_dist:
            min_dist = min(min_dist, distance(strip[i], strip[j]))
```

```
        j += 1
    return min_dist
```

Функция оптимизирует поиск минимального расстояния внутри "полосы".

### ***Шаги работы:***

1. Сортирует точки в полосе по координате Y, так как это позволяет ограничить количество пар точек, которые нужно проверять.
2. Для каждой точки `strip[i]` проверяются точки `strip[j]`, такие что разница по Y меньше текущего минимального расстояния `min_dist`.

## **5. Чтение и запись данных**

### ***Чтение данных:***

```
with open('input.txt', 'r') as f:
    n = int(f.readline().strip())
    points = [tuple(map(int, f.readline().strip().split())) for _ in
range(n)]
points.sort(key=lambda point: point[0])
```

- Считывается количество точек `n`.
- Каждая точка представлена как кортеж  $(x_i, y_i)$ . Все точки сортируются по X-координате, чтобы соответствовать требованиям алгоритма "разделяй и властвуй".

### ***Вычисление минимального расстояния:***

```
min_distance = closest_pair(points)
```

- Передается отсортированный массив точек в основную функцию `closest_pair`.
- Возвращается минимальное расстояние.

### ***Запись результата:***

```
with open('output.txt', 'w') as f:
    f.write(f"{min_distance:.4f}\n")
```

- Результат записывается в файл `output.txt` с точностью до 4 знаков после запятой.

## **Unittest для задание 9:**

```
import unittest
import math

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

def closest_pair(points):
    if len(points) <= 3:
        return brute_force(points)

    mid = len(points) // 2
```

```

mid_point = points[mid]

dl = closest_pair(points[:mid])
dr = closest_pair(points[mid:])

d = min(dl, dr)

strip = []
for point in points:
    if abs(point[0] - mid_point[0]) < d:
        strip.append(point)

return min(d, strip_closest(strip, d))

def brute_force(points):
    min_dist = float('inf')
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            d = distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
    return min_dist

def strip_closest(strip, d):
    min_dist = d
    strip.sort(key=lambda point: point[1])

    for i in range(len(strip)):
        j = i + 1
        while j < len(strip) and (strip[j][1] - strip[i][1]) < min_dist:
            min_dist = min(min_dist, distance(strip[i], strip[j]))
            j += 1
    return min_dist

class TestClosestPair(unittest.TestCase):

    def test_two_points(self):
        points = [(0, 0), (3, 4)]
        result = closest_pair(points)
        self.assertAlmostEqual(result, 5.0, places=4)

    def test_identical_points(self):
        points = [(7, 7), (7, 7)]
        result = closest_pair(points)
        self.assertAlmostEqual(result, 0.0, places=4)

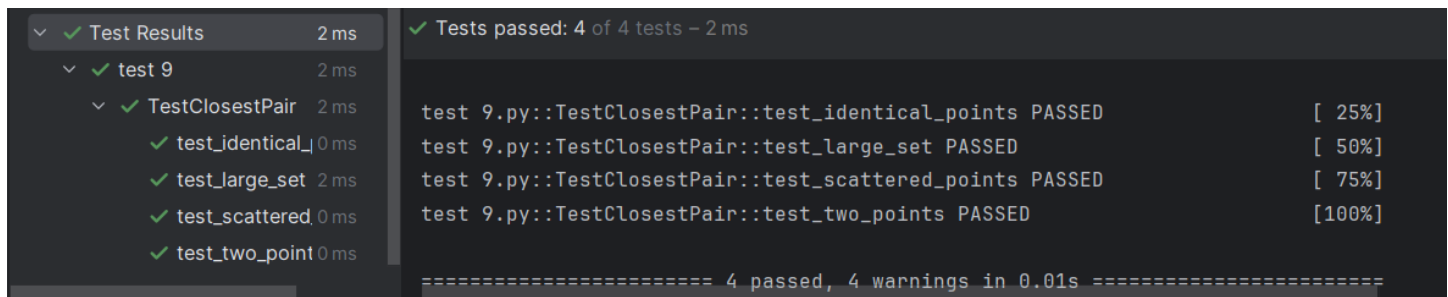
    def test_large_set(self):
        points = [(i, i) for i in range(1000)]
        result = closest_pair(points)
        self.assertAlmostEqual(result, math.sqrt(2), places=4)

    def test_scattered_points(self):
        points = [(1, 1), (2, 2), (3, 3), (10, 10), (12, 12)]
        result = closest_pair(points)
        self.assertAlmostEqual(result, math.sqrt(2), places=4)

if __name__ == '__main__':
    unittest.main()

```

## \* Результат :



```
✓ Test Results 2 ms
  ✓ test 9 2 ms
    ✓ TestClosestPair 2 ms
      ✓ test_identical 0 ms
      ✓ test_large_set 2 ms
      ✓ test_scattered 0 ms
      ✓ test_two_point 0 ms

Tests passed: 4 of 4 tests - 2 ms

test 9.py::TestClosestPair::test_identical_points PASSED [ 25%]
test 9.py::TestClosestPair::test_large_set PASSED [ 50%]
test 9.py::TestClosestPair::test_scattered_points PASSED [ 75%]
test 9.py::TestClosestPair::test_two_points PASSED [100%]

==== 4 passed, 4 warnings in 0.01s =====
```

### 1. Функция `distance(p1, p2)`

- Вычисляет евклидово расстояние между двумя точками `p1` и `p2`.
- Формула:  $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

### 2. Функция `closest_pair(points)`

- Основная функция для нахождения минимального расстояния.
- Если количество точек  $\leq 3$ , вызывается функция **`brute_force`** для простого перебора всех пар.
- Делит массив точек на две части (левая и правая).
- Рекурсивно вызывает `closest_pair` для обеих частей:  $d_l$  = расстояние в левой части,  $d_r$  = расстояние в правой части.
- Минимальное расстояние из обеих частей:  $d = \min(d_l, d_r)$ .
- Создаётся массив `strip` для точек, которые находятся ближе к разделяющей линии, чем расстояние  $d$ .
- Для точек в `strip` вызывается функция `strip_closest`.

### 3. Функция `brute_force(points)`

- Используется для перебора всех пар точек в случае малого их количества ( $\leq 3$ ).
- Сравнивает расстояние между каждой парой точек и находит минимальное.

### 4. Функция `strip_closest(strip, d)`

- Работает с точками в массиве `strip`, которые находятся в пределах расстояния  $d$  от центральной вертикальной линии.
- Сортирует точки в `strip` по `y`-координате.
- Для каждой точки проверяет только те точки, которые находятся ниже по оси `y` на расстоянии меньше  $d$ , так как это ограничивает количество проверок.
- Находит минимальное расстояние.

### 5. Класс `TestClosestPair`

- Тесты для проверки правильности работы алгоритма.
- Использует библиотеку `unittest`.

### Тестовые случаи:

1. **test\_two\_points:** Две точки (0, 0) и (3, 4). Проверяется, что расстояние равно 5.0 (по теореме Пифагора).
2. **test\_identical\_points:** Две идентичные точки (7, 7). Расстояние должно быть 0.0.
3. **test\_large\_set:** Линейно расположенные точки (i, i). Расстояние между ближайшими точками равно  $\sqrt{2}$ .
4. **test\_scattered\_points:** Разбросанные точки. Проверяется, что алгоритм корректно находит ближайшие точки.

### 6. Запуск

- При запуске файла выполняются тесты с помощью `unittest`.

### Особенности:

- **Метод "разделяй и властвуй":** рекурсивное деление точек на две части для уменьшения числа вычислений.
- **Оптимизация через массив `strip`:** анализируется ограниченное число точек.

### Пример работы:

- Если вход: `points = [ (0, 0), (3, 4), (1, 1), (5, 2) ]`
- Алгоритм найдёт ближайшую пару, например, (0, 0) и (1, 1), расстояние  $\sqrt{2}$ .