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

Отчет по лабораторной работе №3
по курсу «Алгоритмы и структуры данных»
Тема: Быстрая сортировка, сортировки за линейное время
Вариант 5

Выполнил:
Артемов И. В.
К3141

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №1. Улучшение Quick sort

Текст задачи.

1 задача. Улучшение Quick sort

1. Используя *псевдокод* процедуры `Randomized - QuickSort`, а так же `Partition` из презентации к Лекции 3 (страницы 8 и 12), напишите программу быстрой сортировки на Python и проверьте ее, создав несколько рандомных массивов, подходящих под параметры:

   - **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^4$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, ***по модулю*** не превосходящих 10^9.
   - **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным массивом. Между любыми двумя числами должен стоять ровно один пробел.
   - Ограничение по времени. 2 сек.
   - Ограничение по памяти. 256 мб.
   - Для проверки можно выбрать наихудший случай, когда сортируется массив рамера $10^3, 10^4, 10^5$ чисел порядка 10^9, отсортированных в обратном порядке; наилучший, когда массив уже 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 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` +с `Partition` и ее с `Partition3` на сетах случайных данных, в которых содержатся всего несколько уникальных элементов при $n = 10^3, 10^4, 10^5$. Что быстрее, `Randomized-QuickSort` +с `Partition3` или `Merge-Sort`?
   - Пример:

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

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import tracemalloc
import time
import random
from lab2.utils import open_file, write_file

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()


def randomized_partition(arr, low, high):
    # Случайный выбор опорного элемента для разделения массива
    pivot_index = random.randint(low, high)
    arr[high], arr[pivot_index] = arr[pivot_index], arr[high]

    pivot = arr[high]
    i = low - 1
    # Разбиение массива на элементы, меньшие и большие опорного
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    # Обмен опорного элемента с элементом на позиции i + 1
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def partition3(arr, low, high):
    # Разбиение массива на три части относительно опорного элемента
    pivot = arr[high]
    lt = low  # Определяет конец области < pivot
    gt = high  # Начало области > pivot
    i = low

    while i <= gt:
        # Если текущий элемент меньше опорного
        if arr[i] < pivot:
            arr[lt], arr[i] = arr[i], arr[lt]
            lt += 1
            i += 1
        # Если текущий элемент больше опорного
        elif arr[i] > pivot:
            arr[gt], arr[i] = arr[i], arr[gt]
            gt -= 1
        # Если текущий элемент равен опорному
        else:
            i += 1
    return lt, gt


def randomized_quick_sort_3way(arr, low, high):
    # Рекурсивная сортировка массива с трёхсторонним разбиением
    if low < high:
        # Случайный выбор опорного элемента для разбиения на три части
        pivot_index = random.randint(low, high)
        arr[high], arr[pivot_index] = arr[pivot_index], arr[high]

        # Трёхстороннее разбиение массива
        lt, gt = partition3(arr, low, high)
```

```
        # Рекурсивная сортировка частей массива
        randomized_quick_sort_3way(arr, low, lt - 1)
        randomized_quick_sort_3way(arr, gt + 1, high)


# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    n_str, m = open_file("../txtf/input.txt")
    n = int(n_str[0])  # Преобразуем первую строку в число n

    # Проверка корректности входных данных: размер массива и элементы
    if (1 <= n <= 10 ** 4) and (all(abs(i) <= 10 ** 9 for i in m)):
        # Сортируем массив m с помощью функции randomized_quick_sort_3way
        randomized_quick_sort_3way(m, 0, n - 1)
        # Записываем отсортированный массив в файл output.txt
        write_file(" ".join(str(a) for a in m), "../txtf/output.txt")
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print('Введите корректные данные')

    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()
```

Текстовое объяснение решения.

1) Считываем данные из input.txt. Если не подходят по условиям, то проси ввести корректные данные
2) Функция randomized_partition:

- Осуществляет разбиение массива вокруг случайного опорного элемента.
- Случайный опорный элемент выбирается и перемещается в конец массива.
- Затем массив делится так, чтобы элементы, меньшие опорного, располагались слева, а большие — справа.
- Возвращает индекс опорного элемента после разбиения.

3) Функция partition3:

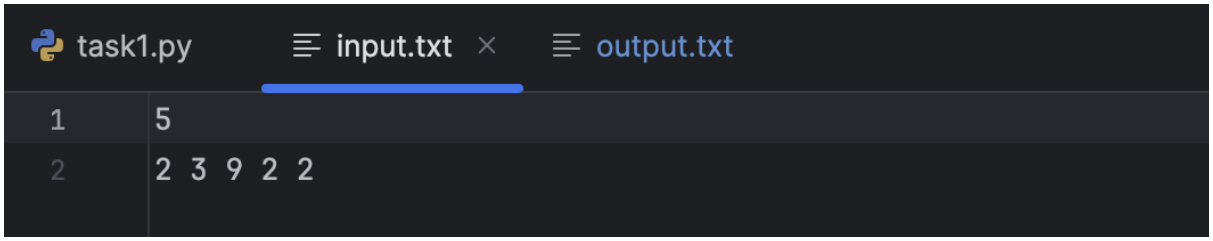
- Выполняет трёхстороннее разбиение массива на части < pivot, = pivot, > pivot.
- Элементы, меньшие опорного, перемещаются влево, большие — вправо, а равные опорному остаются в центре.
- Возвращает индексы начала и конца области, содержащей элементы, равные опорному.

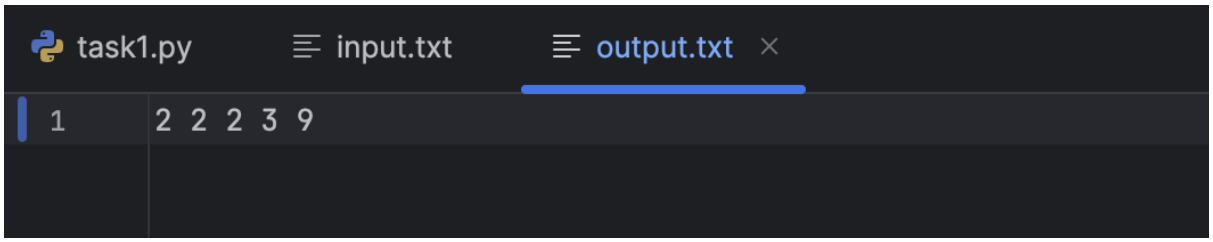
4) Функция randomized_quick_sort_3way:

- Рекурсивно сортирует массив с использованием трёхстороннего разбиения.
- Сначала выбирает случайный опорный элемент, затем делит массив с помощью partition3.
- Рекурсивно сортирует части массива, расположенные слева и справа от центральной области, равной опорному элементу.

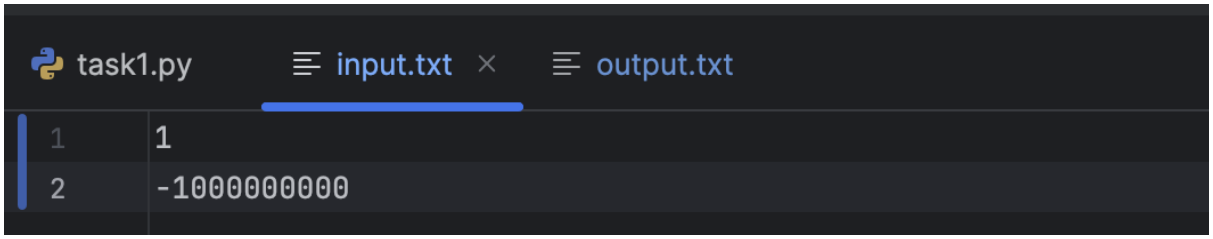
5) Если входные данные корректны, массив m сортируется с использованием функции randomized_quick_sort_3way.
6) Результат записывается в output.txt

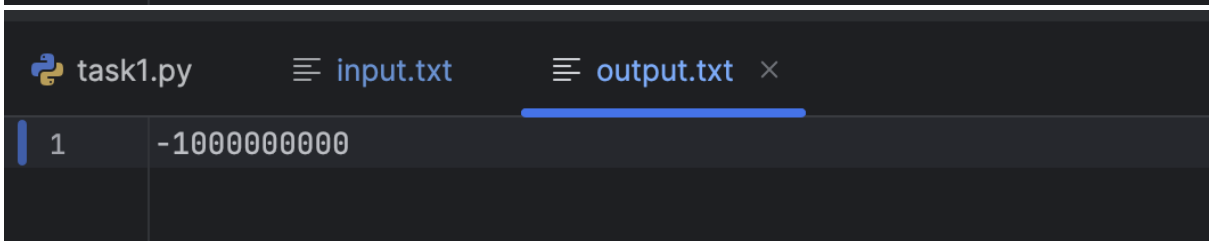
Результат работы кода на примере из задачи:

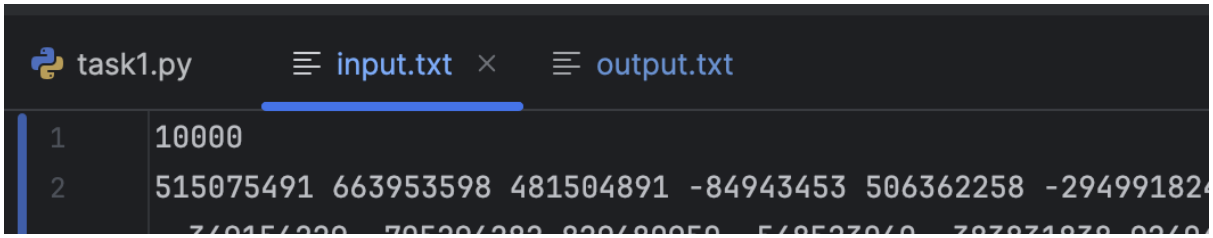


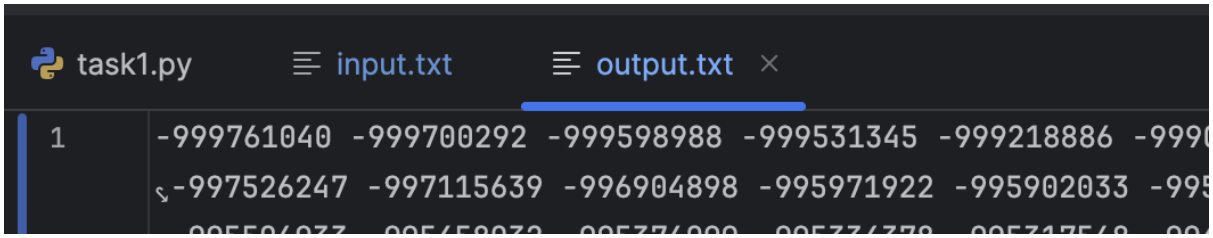


Результат работы кода на максимальных и минимальных значениях:









	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0002167500001633016 секунд	14508 байт
Пример из задачи	0.00024616699988655455 секунд	14506 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.22705929099993227 секунд	1064396 байт

Вывод по задаче: мною был изучен алгоритм быстрой сортировки массива, а также выяснилось, что чем больше массив, тем дольше будет время выполнения и больше затраты памяти.

Задача №5. Индекс Хирша

Текст задачи.

5 задача. 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?

Листинг кода.

```
# Импортируем библиотеки для отслеживания времени работы программы и
использования памяти
import tracemalloc
import time
from lab3.utils import open_file, write_file

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()
```

```
# Включаем отслеживание памяти, чтобы отслеживать, сколько памяти
использует программа
tracemalloc.start()


# Функция для вычисления индекса H (h-index) на основе списка цитирований
def h_index(citations):
    # Сортируем список цитирований по убыванию, чтобы проверить индекс H
быстрее
    citations.sort(reverse=True)

    # Проходим по отсортированному списку и находим максимальное значение,
    # которое выполняет условие citations[i] <= цитированию на позиции i
    for i in range(len(citations)):
        if citations[i] < i + 1:
            return i  # Возвращаем индекс citations[i], если условие
нарушено

    # Если все элементы соответствуют условиям, возвращаем длину списка как
индекс H
    return len(citations)


# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    citations_str = open_file("../txtf/input.txt")
    citations = citations_str[0]  # Используем первую строку, предполагая,
что это массив цитирований

    # Проверка корректности входных данных:
    # - размер массива должен быть в пределах от 1 до 5000
    # - все элементы массива должны быть в пределах от 0 до 1000
    if (1 <= len(citations) <= 5000) and (all(0 <= i <= 1000 for i in
citations)):
        # Вычисляем индекс H для списка цитирований
        h_index_value = h_index(citations)

        # Записываем полученное значение индекса H в файл output.txt
        write_file(h_index_value, "../txtf/output.txt")
    else:
        # Если данные некорректны, выводим сообщение об ошибке
        print('Введите корректные данные')

    # Выводим время выполнения программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))

    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()
```

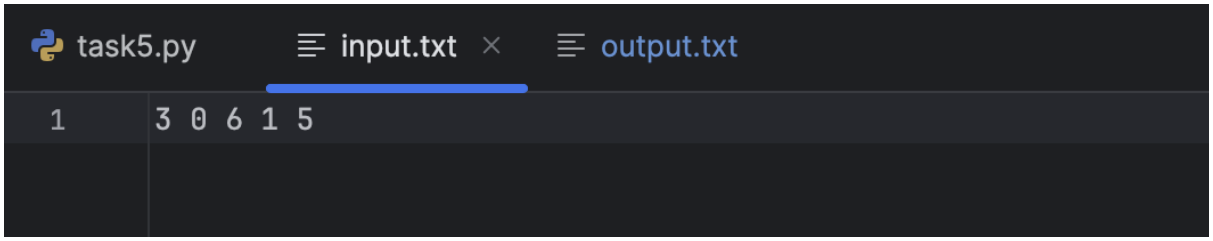
Текстовое объяснение решения.

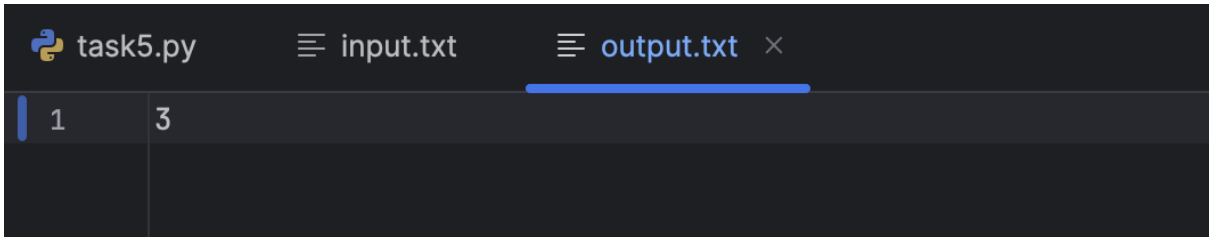
1) Считываем данные из input.txt. Если не подходят по условиям, то проси ввести корректные данные
2) Объявляем функцию h_index, которая вычисляет индекс Хирша на основе списка цитирований

- Сортируем список цитирований по убыванию, чтобы проверить индекс Хирша
- Проходим по отсортированному списку и находим максимальное значение h, которое выполняет условие h <= цитированию на позиции 1
- Возвращаем индекс citations[i], если условие нарушено
- Если все элементы соответствуют условиям, возвращаем длину списка как индекс Хирша

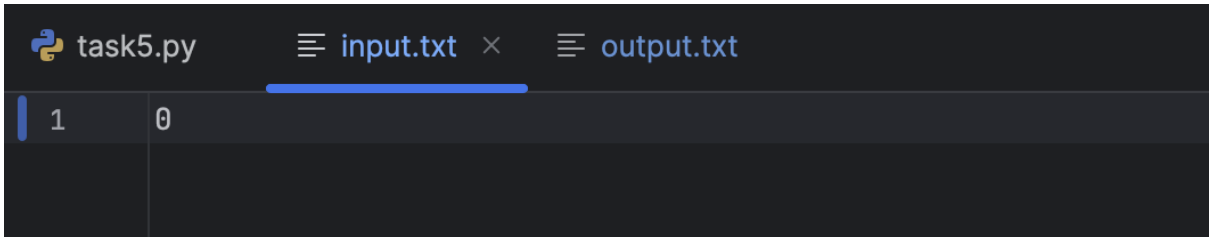
3) Вычисляем индекс Хирша для списка цитирований и записываем в output.txt

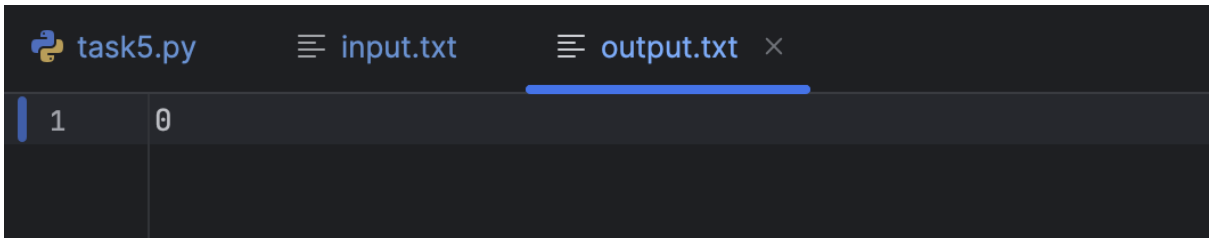
Результат работы кода на примере из текста задачи:

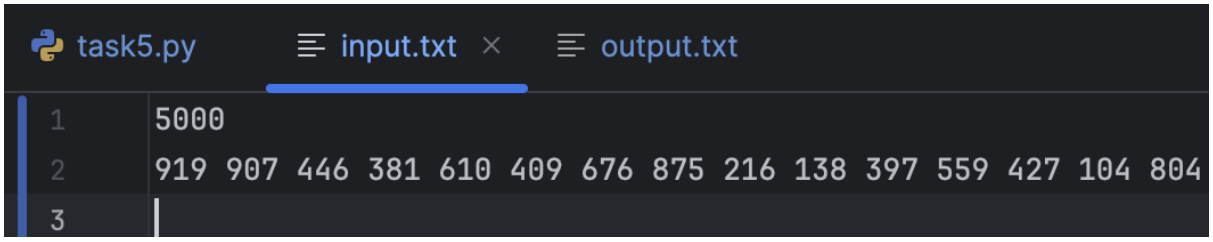




Результат работы кода на максимальных и минимальных значениях:







	Время выполнения	Затраты памяти
Нижняя граница	0.0002948339997601579	14061 байт

диапазона значений входных данных из текста задачи	секунд	
Пример из задачи	0.000959457946009934 секунд	14400 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.22705929099993227 секунд	430401 байт

Вывод по задаче: мною был изучен алгоритм подсчёта индекса Хирша.

Задача №7. Цифровая сортировка

Текст задачи.

7 задача. 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 ($1 \le n \le 10^6$, $1 \le k \le m \le 10^6$, $n \cdot m \le 5 \cdot 10^7$). 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 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».

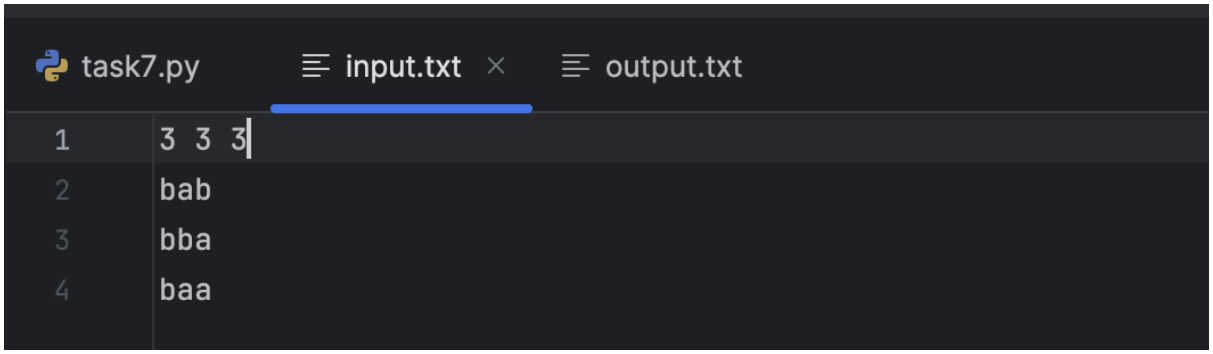
Листинг кода.

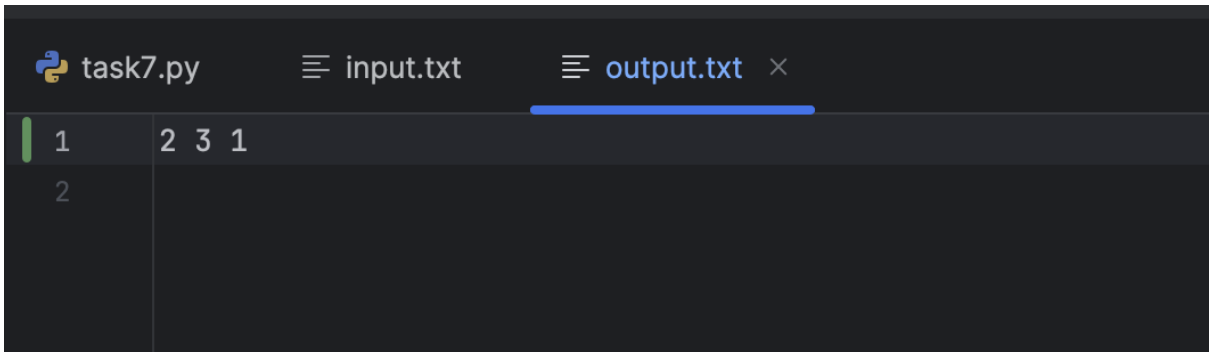
```
import tracemalloc
import time
from lab3.utils import open_file, write_file

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

def radix_sort_phase(strings, phase):
    """Сортировка по заданной фазе (символу)."""
    return sorted(strings, key=lambda x: x[1][phase])

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    (n, m, k), columns = open_file("../txtf/input.txt")

    # Проверка корректности входных данных
    if (1 <= n <= 10**6) and (1 <= k <= m <= 10**6) and (n * m <= 5 *
10**7):
        # Формируем список строк из колонок
        strings = [(i + 1, ''.join(columns[j][i] for j in range(m))) for i
in range(n)]

        # Применяем сортировку по фазам
        for phase in range(min(m, k) - 1, -1, -1):
            strings = radix_sort_phase(strings, phase)

        # Подготовка результатов (выводим индексы строк в новом порядке)
        result = [str(item[0]) for item in strings]
        write_file(result, "../txtf/output.txt")
    else:
        print('Введите корректные данные')

    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")
    tracemalloc.stop()
```

Текстовое объяснение решения.

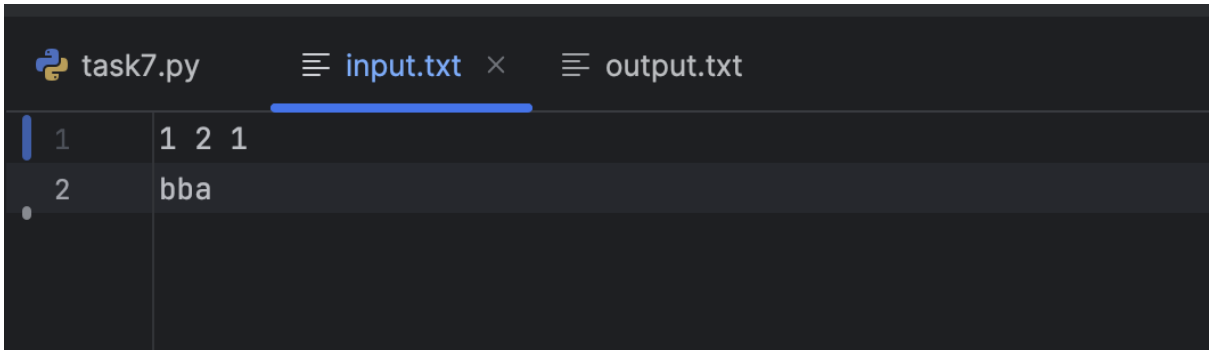
1) Проверяем удовлетворяют ли входные данные условию задачи. Если нет, то просим пользователя ввести корректные данные
2) Объявляем функцию radix_sort_phase, которая сортирует по заданной фазе (символу)
3) Формируем список строк из колонок
4) Применяем сортировку по фазам
5) Выводим индексы строк в новом порядке в output.txt

Результат работы кода на примере из задачи:





Результат работы кода на максимальных и минимальных значениях:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00027195800794288516 секунд	14188 байт
Пример из задачи	0.0004289170001356979 секунд	14252 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.09492137498455122 секунд	2127532 байт

Вывод по задаче: мною был изучен алгоритм вывода порядка n строк после k фаз цифровой сортировки

Дополнительные задачи

Задача №3. Сортировка пугалом

Текст задачи.

«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 ($1 \leq n, k \leq 10^5$) – 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

Листинг кода.

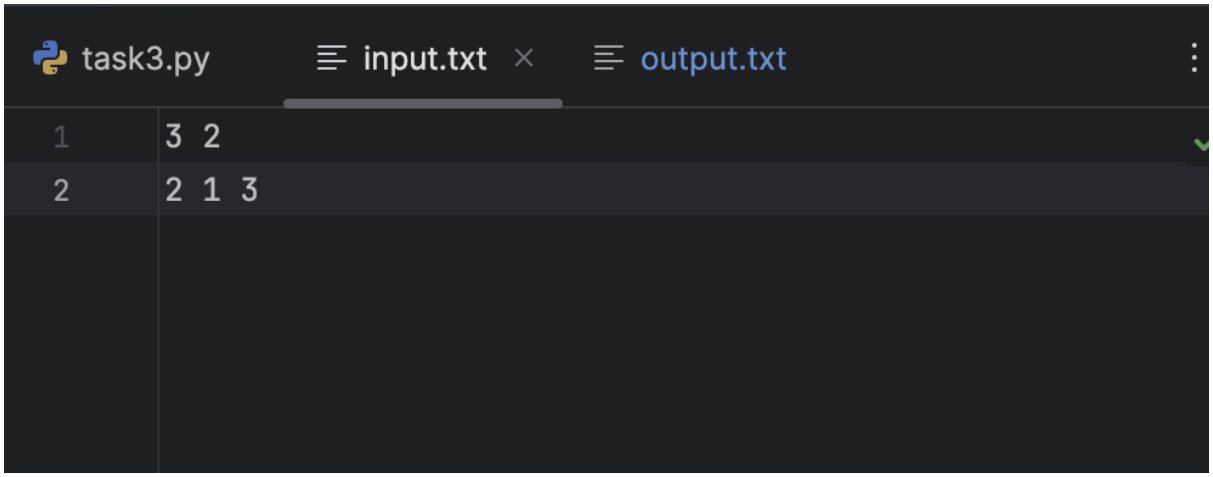
```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import tracemalloc  # Для отслеживания потребления памяти
import time  # Для измерения времени выполнения
import random  # Для случайных операций (не используется в данном коде, но
импортирован)
from lab2.utils import open_file, write_file  # Импортируем функции для
чтения и записи файлов

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()  # Засекаем время начала выполнения программы

# Включаем отслеживание памяти
tracemalloc.start()  # Стартуем отслеживание памяти, чтобы в конце узнать,
сколько памяти было использовано


# Основная функция, которая проверяет, возможно ли отсортировать массив по
описанным правилам
def can_sort(n, k, arr):
    # Разделяем элементы массива на k групп по остатку от деления на k
    groups = [[] for _ in range(k)]  # Создаем список групп, в каждой будет
храниться часть элементов массива
```

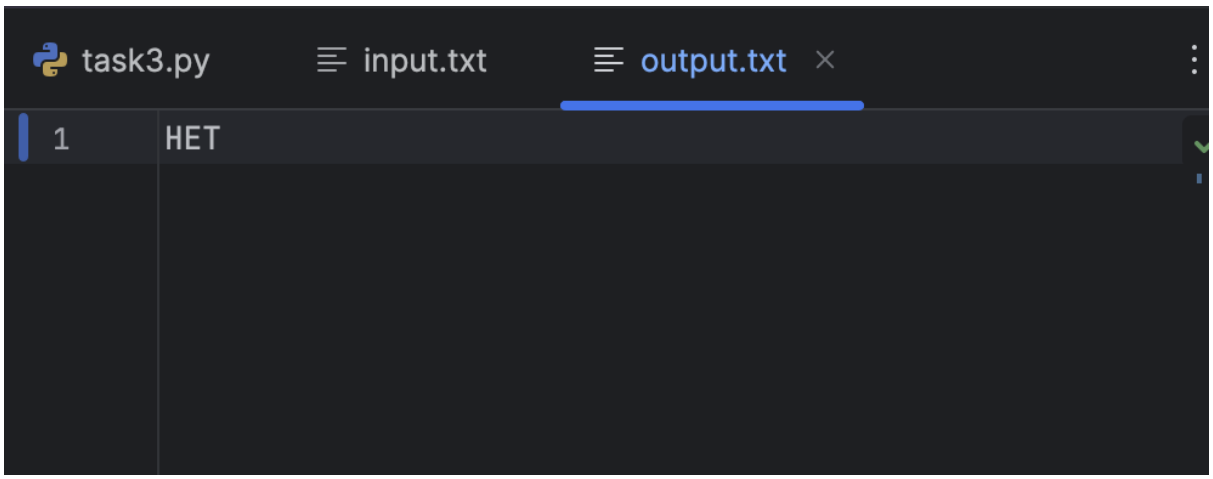
```
    for i in range(n):  # Проходим по всем элементам массива
        groups[i % k].append(arr[i])  # Распределяем элементы по группам с
учетом индекса и остатка от деления на k

    # Сортируем каждую группу по отдельности
    for i in range(k):  # Для каждой группы
        groups[i].sort()  # Сортируем элементы внутри группы

    # Воссоздаем отсортированный массив из отсортированных групп
    sorted_arr = []  # Список для хранения результирующего отсортированного
массива
    for i in range(n):  # Проходим по всем индексам массива
        sorted_arr.append(groups[i % k].pop(0))  # Берем элементы из
отсортированных групп в соответствующем порядке

    # Проверяем, совпадает ли получившийся массив с полностью
отсортированным массивом
    if sorted_arr == sorted(arr):  # Если отсортированный массив совпадает
с результатом
        return "ДА"  # Возвращаем "ДА", если сортировка возможна
    else:
        return "НЕТ"  # Возвращаем "НЕТ", если сортировка невозможна


# Основной блок программы, который выполняется при запуске скрипта
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    (n_and_k_str, m) = open_file("../txtf/input.txt")  # Считываем данные
из файла: первое число n и k, далее массив матрешек
    n, k  = n_and_k_str  # Преобразуем второе число в число k (размах рук)

    # Проверка корректности входных данных: размер массива и элементы
    if (1 <= n <= 10 ** 5) and (all(abs(i) <= 10 ** 9 for i in m)) and (1
<= k <= 10 ** 5):
        # Если данные корректны (n в пределах от 1 до 10^5, элементы
массива в пределах от -10^9 до 10^9, k в пределах от 1 до 10^5)

        # Сортируем массив m с помощью функции can_sort
        result = can_sort(n, k, m)  # Проверяем, можем ли отсортировать
массив по правилам

        # Записываем результат в файл output.txt
        write_file(result, "../txtf/output.txt")  # Записываем ответ ("ДА"
или "НЕТ") в файл output.txt
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print('Введите корректные данные')  # Если данные не соответствуют
ограничениям, выводим ошибку

    # Выводим время работы программы
    print("Время работы: %s секунд" % (
                time.perf_counter() - t_start))  # Выводим время работы
программы, рассчитанное с начала выполнения

    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1],
          "байт")  # Выводим максимальное количество памяти, затраченное на
выполнение программы

    # Останавливаем отслеживание памяти
    tracemalloc.stop()  # Останавливаем отслеживание памяти после
завершения программы
```

Текстовое объяснение решения.

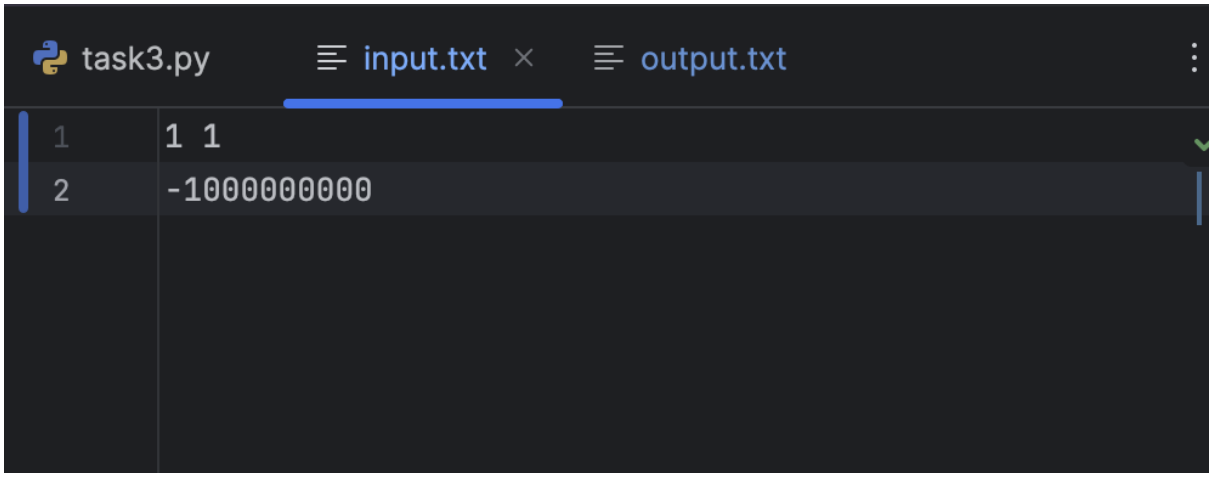
1) Проверяем удовлетворяют ли входные данные условию задачи. Если нет, то просим пользователя ввести корректные данные
2) Объявляем функцию, которая проверяет, возможно ли отсортировать массив по описанным правилам
    - Создаём список групп, в каждой будет храниться часть элементов массива
    - Распределяем элементы по группам с учётом индекса и остатка от деления на k
    - Сортируем элементы внутри группы
    - Воссоздаём отсортированный массив из отсортированных групп
    - Проверяем, совпадает ли получившийся массив с полностью отсортированным массивом
3) Сортируем массив m с помощью функции can_sort и записываем результат в output.txt

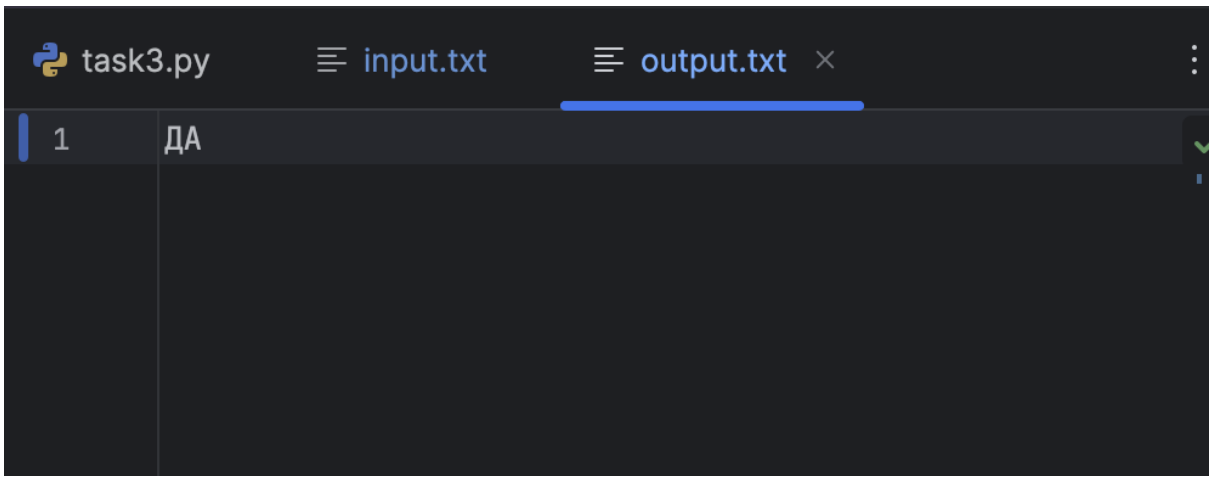
Результат работы кода на примере из текста задачи:

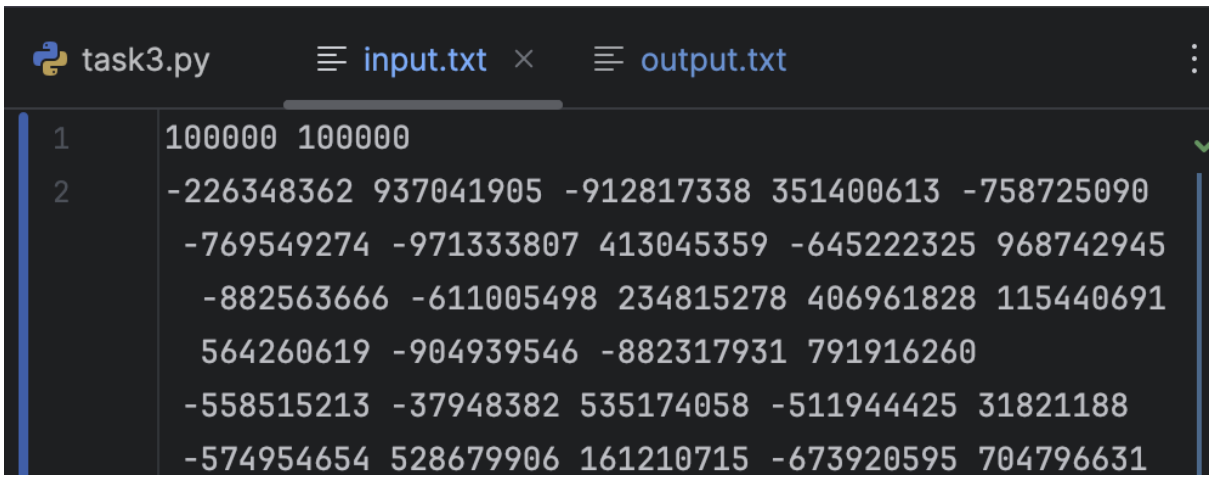




Результат работы кода на максимальных и минимальных значениях:







	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0002184580002904113 секунд	14234 байт
Пример из задачи	0.00037754100003439817 секунд	14228 байт

Верхняя граница диапазона значений входных данных из текста задачи	0.22705929099993227 секунд	1064396 байт

Вывод по задаче: мною был изучен алгоритм сортировки пугалом.

Задача №6. Сортировка целых чисел

Текст задачи.

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, $C_1 + C_{11} + C_{21} + ...$).

- **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 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 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.

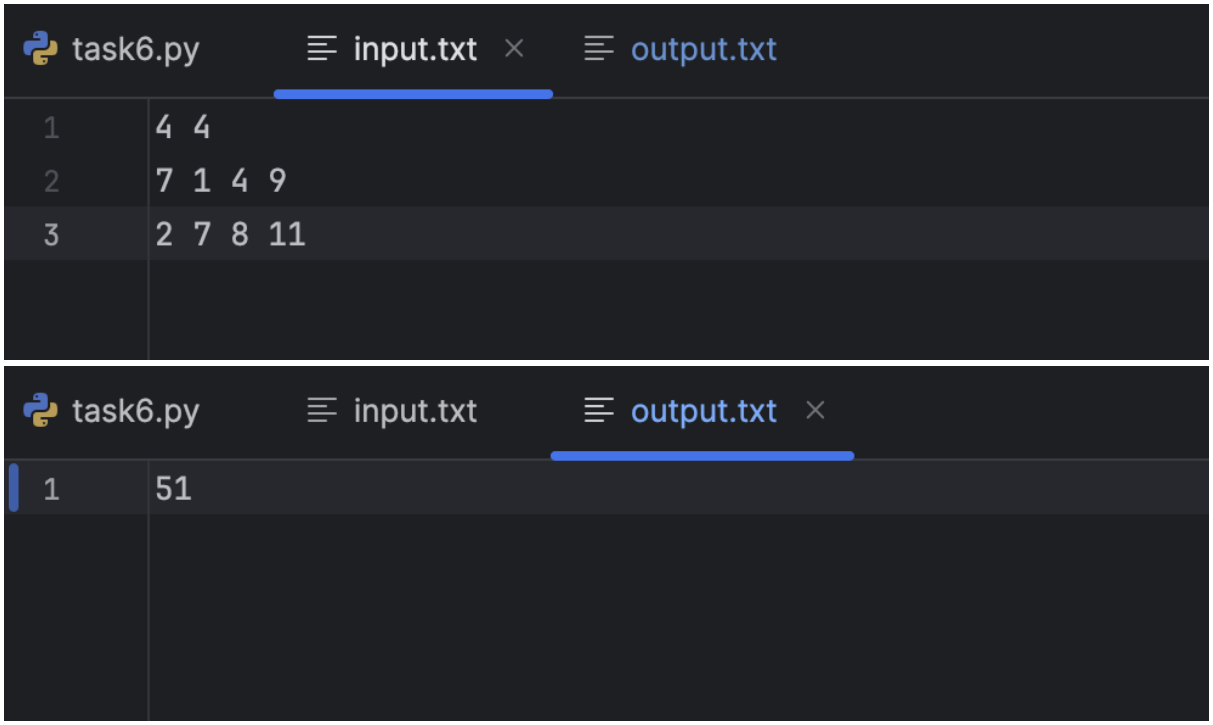
Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import tracemalloc
import time
from lab3.utils import open_file, write_file

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()


# Генерация массива попарных произведений
def generate_pairwise_array(A, B):
    C = []
    for a in A:
        for b in B:
            C.append(a * b)
    return C


# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    n_and_m_str, A, B = open_file("../txtf/input.txt")
    n = int(n_and_m_str[0])
    m = int(n_and_m_str[1])

    # Проверка корректности входных данных
    if (1 <= n, m <= 6000) and (all(0 <= i <= 40000 for i in A)) and all(0
<= i <= 40000 for i in B):
        C = generate_pairwise_array(A, B)
        C.sort()
        # Подсчёт суммы каждого десятого элемента
        result = sum(C[i] for i in range(0, len(C), 10))
        # Записываем отсортированный массив в файл output.txt
        write_file(result, "../txtf/output.txt")
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print('Введите корректные данные')

    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")
    tracemalloc.stop()  # Останавливаем отслеживание памяти
```

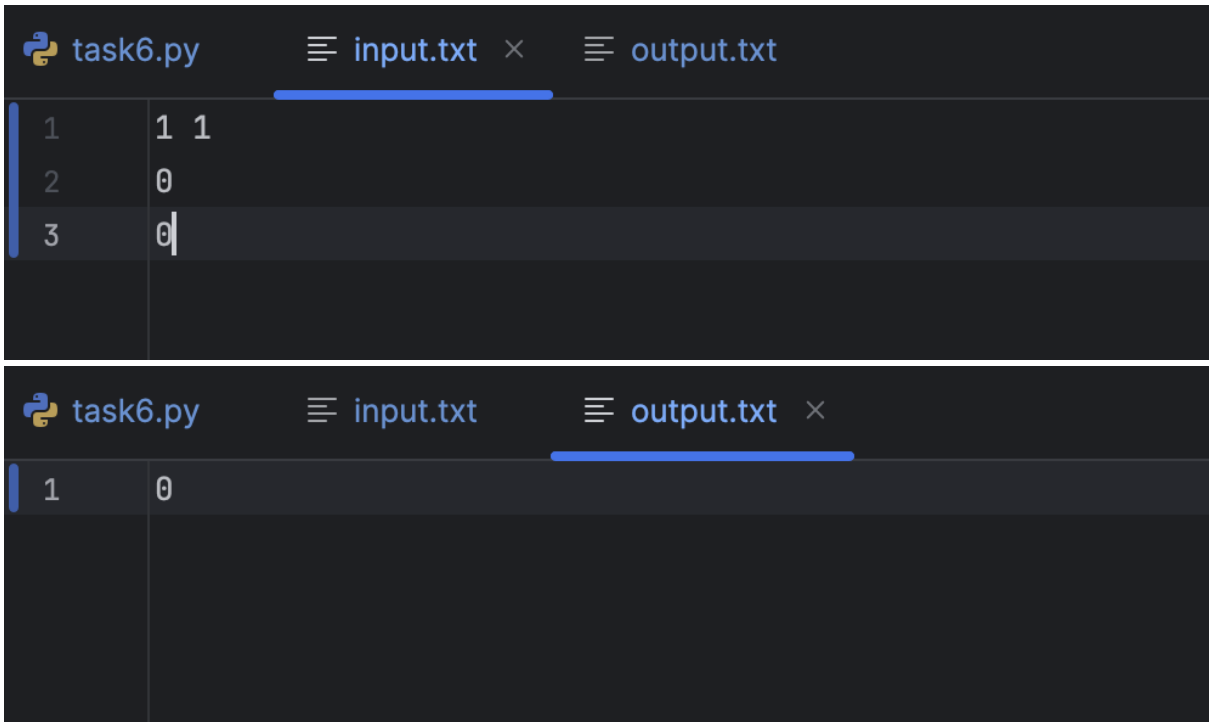
Текстовое объяснение решения.

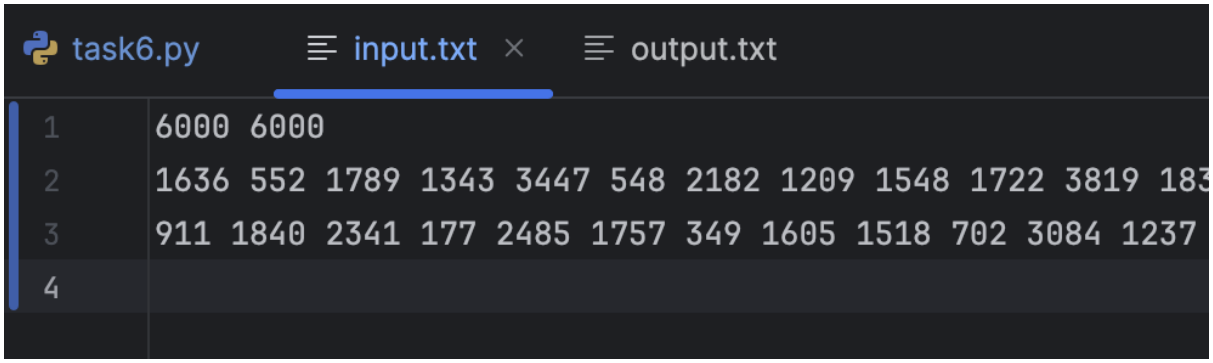
1) Проверяем удовлетворяют ли входные данные условию задачи. Если нет, то просим пользователя ввести корректные данные
2) Объявляем функцию generate_pairwise_array, которая генерирует массив попарных произведений
3) Сортируем полученный массив
4) Подсчёт суммы каждого десятого элемента и запись в output.txt

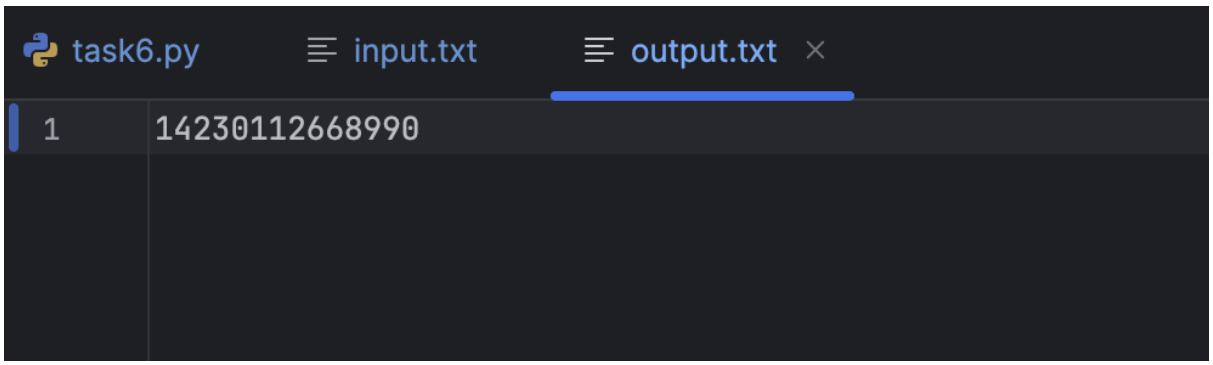
Результат работы кода на примере из текста задачи:



Результат работы кода на максимальных и минимальных значениях:







	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000254583000241837 секунд	14180 байт
Пример из задачи	0.00047691597137600183 секунд	14256 байт
Верхняя граница диапазона значений входных данных из текста задачи	30.278248833999896 секунд	15810163 байт

Вывод по задаче: мною был изучен алгоритм сортировки целых чисел и вывода суммы каждого десятого элемента этой последовательности.

Вывод

В ходе лабораторной работы были изучены алгоритмы быстрой сортировки, сортировки за линейное время.