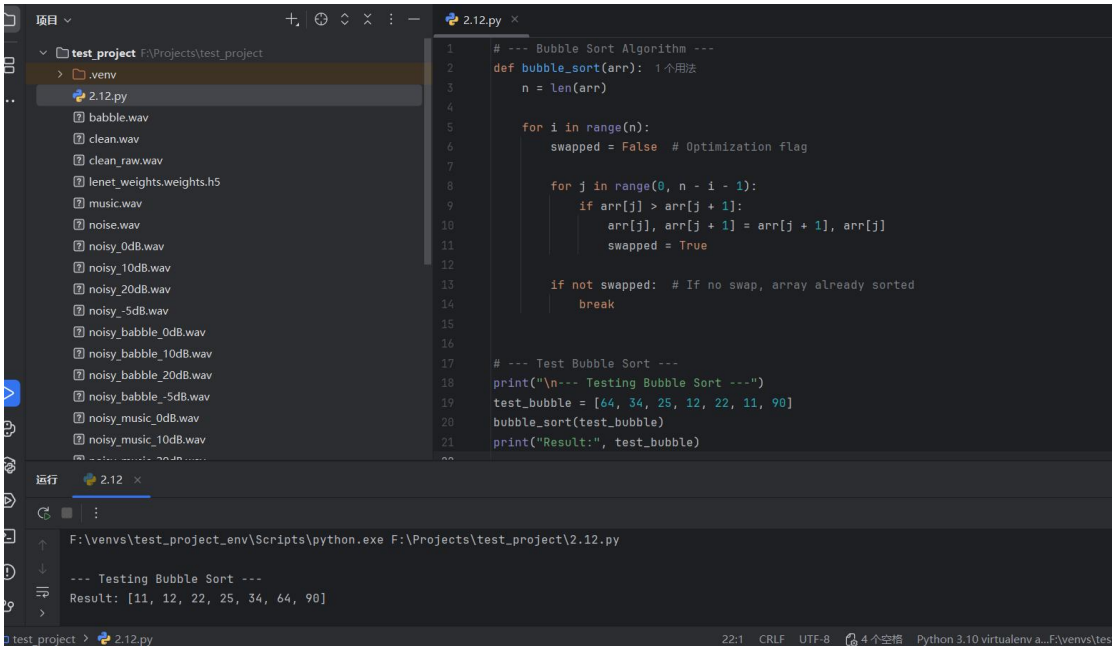
Problem 1 — Sorting Algorithms

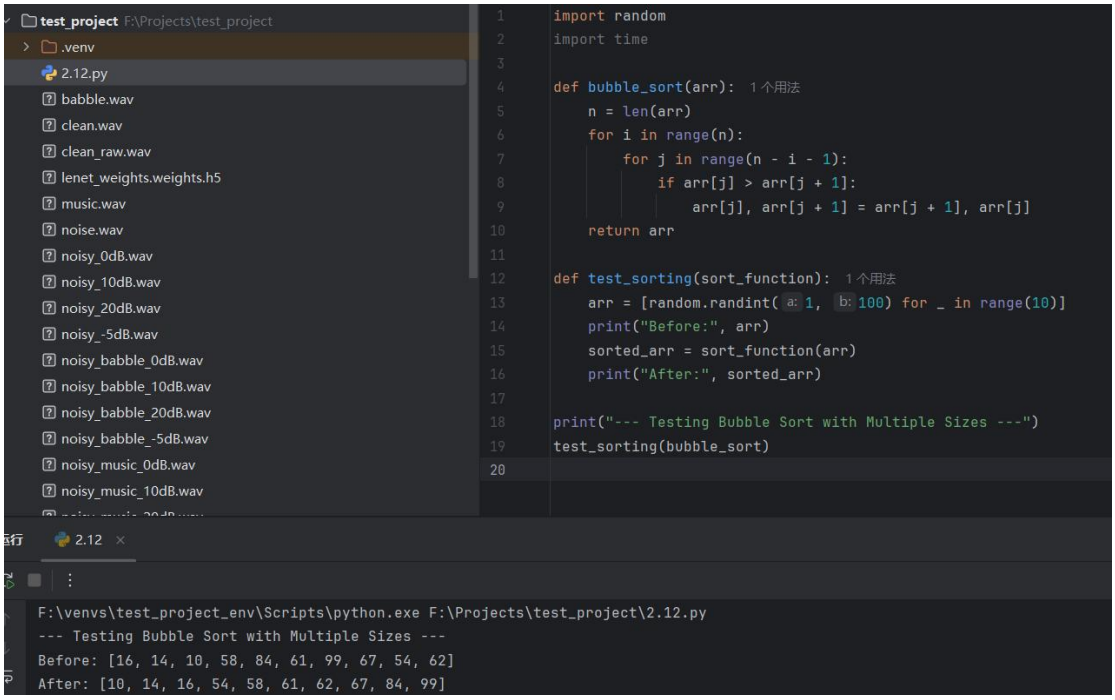
1. write own bad $O(n\ 2)$ sorting (bubble sorting)



Code

```
# --- Bubble Sort Algorithm ---
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        swapped = False    # Optimization flag

        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        if not swapped:    # If no swap, array already sorted
            break

# --- Test Bubble Sort ---
print("\n--- Testing Bubble Sort ---")
test_bubble = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(test_bubble)
print("Result:", test_bubble)
```

2. Tests of Various Sizes

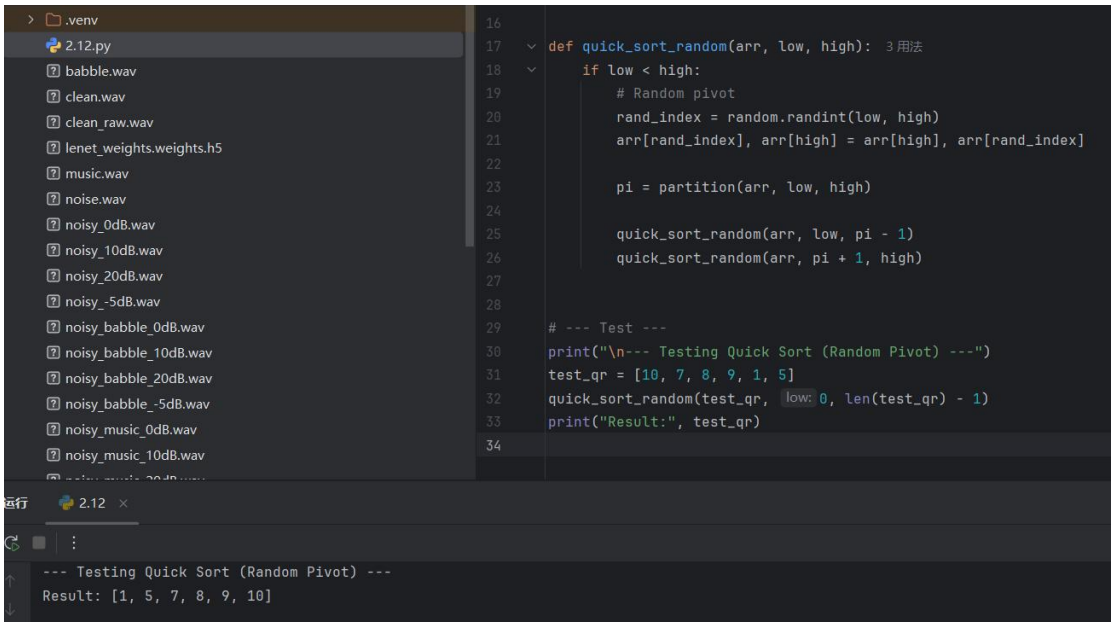


Code

```
import random
import time

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

def test_sorting(sort_function):
    arr = [random.randint(1, 100) for _ in range(10)]
    print("Before:", arr)
    sorted_arr = sort_function(arr)
    print("After:", sorted_arr)

print("--- Testing Bubble Sort with Multiple Sizes ---")
test_sorting(bubble_sort)
```

3. Quick Sort

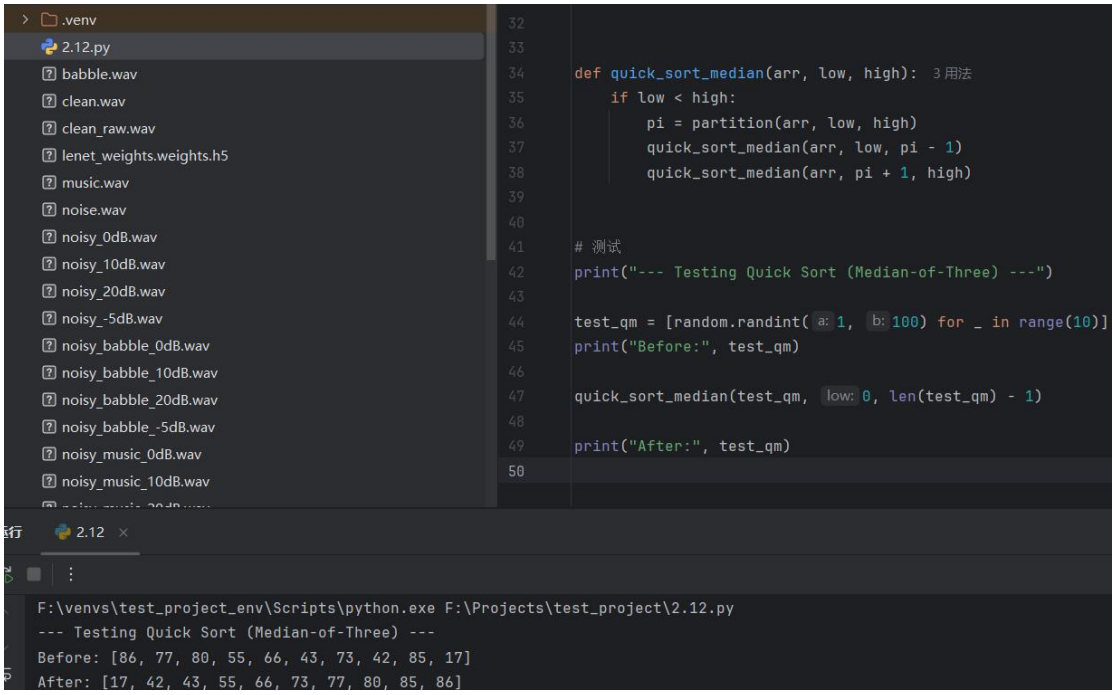
(a) Quick Sort with Random Pivot




```
import random

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quick_sort_random(arr, low, high):
    if low < high:
        # Random pivot
        rand_index = random.randint(low, high)
        arr[rand_index], arr[high] = arr[high], arr[rand_index]

        pi = partition(arr, low, high)

        quick_sort_random(arr, low, pi - 1)
        quick_sort_random(arr, pi + 1, high)
```

```
# --- Test ---
print("\n--- Testing Quick Sort (Random Pivot) ---")
test_qr = [10, 7, 8, 9, 1, 5]
quick_sort_random(test_qr, 0, len(test_qr) - 1)
print("Result:", test_qr)
```

(b) Quick Sort with Median-of-Three Pivot

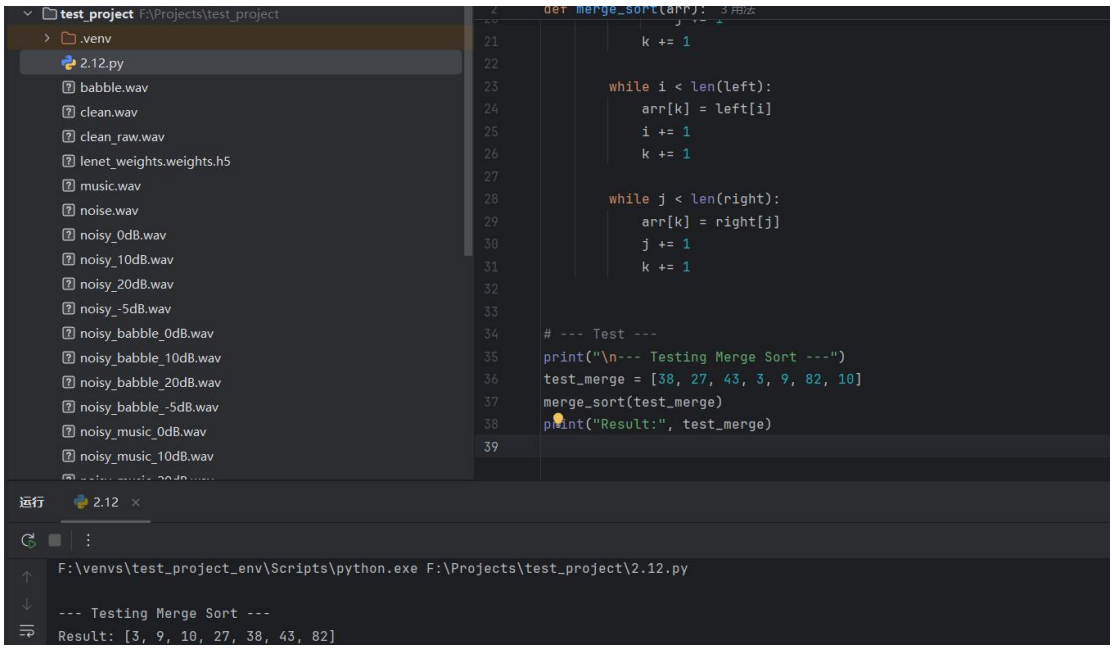


code

```
import random

# 选择三个数的中位数作为 pivot
def median_of_three(arr, low, high):
    mid = (low + high) // 2

    if arr[low] > arr[mid]:
        arr[low], arr[mid] = arr[mid], arr[low]
    if arr[low] > arr[high]:
        arr[low], arr[high] = arr[high], arr[low]
    if arr[mid] > arr[high]:
        arr[mid], arr[high] = arr[high], arr[mid]

    # 把中位数放到 high 位置
    arr[mid], arr[high] = arr[high], arr[mid]
    return arr[high]
```

```
# ⭐ 这个就是你缺少的函数
def partition(arr, low, high):
    pivot = median_of_three(arr, low, high)
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quick_sort_median(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort_median(arr, low, pi - 1)
        quick_sort_median(arr, pi + 1, high)


# test
print("--- Testing Quick Sort (Median-of-Three) ---")

test_qm = [random.randint(1, 100) for _ in range(10)]
print("Before:", test_qm)

quick_sort_median(test_qm, 0, len(test_qm) - 1)

print("After:", test_qm)
```

4. Merge Sort

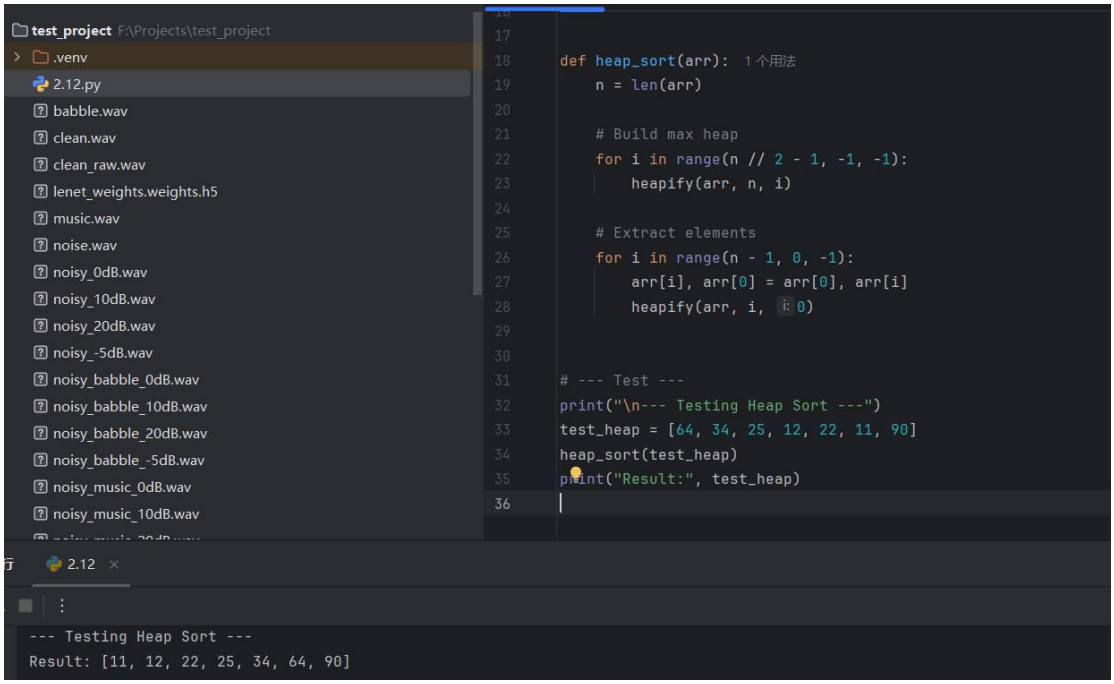


code

```
# --- Merge Sort ---
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2

        left = arr[:mid]
        right = arr[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            arr[k] = left[i]
```

```
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1


# --- Test ---
print("\n--- Testing Merge Sort ---")
test_merge = [38, 27, 43, 3, 9, 82, 10]
merge_sort(test_merge)
print("Result:", test_merge)
```

5. Heap Sort

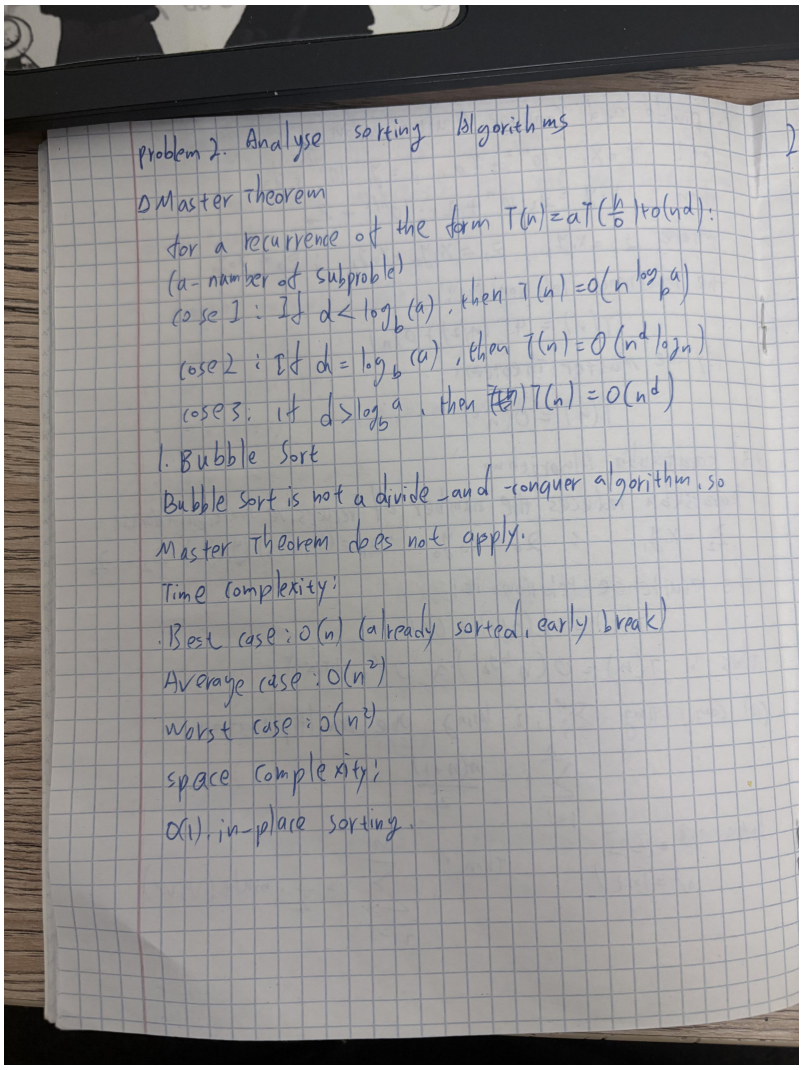


code

```
# --- Heapify ---
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
```

```
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heap_sort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)


# --- Test ---
print("\n--- Testing Heap Sort ---")
test_heap = [64, 34, 25, 12, 22, 11, 90]
heap_sort(test_heap)
print("Result:", test_heap)
```

Problem 2 (Analyse Sorting Algorithms). Analyse succinctly, for all sorting Algorithms above, time and space complexities using the master theorem where applicable



2. Quick Sort (Random Pivot)

Average case recurrence:

$T(n) = 2T(n/2) + O(n)$

using Master Theorem:

$a = 2$

$b = 2$

$f(n) = O(n)$

Since $n^{\log_b a} = n$, Case 2 applies:

$T(n) = O(n \log n)$

worst case: $T(n) = T(n-1) + O(n) = O(n^2)$

Space Complexity:

Average: $O(\log n)$

worst: $O(n)$

3. Quick Sort (Median-of-Three)

improves pivot selection, making balanced partitions more likely.

Average case: $O(n \log n)$

worst case: $O(n^2)$

space: $O(\log n)$ average

4. Merge Sort

Recurrence:

$$T(n) = 2T(n/2) + O(n)$$

By Master Theorem:

$$T(n) = O(n \log n)$$

space complexity:

$O(n)$. extra temporary arrays.

5. Heap Sort

Building heap : $O(n)$

Extracting elements : $n \log n$

Total time : $O(n \log n)$

space : $O(1)$