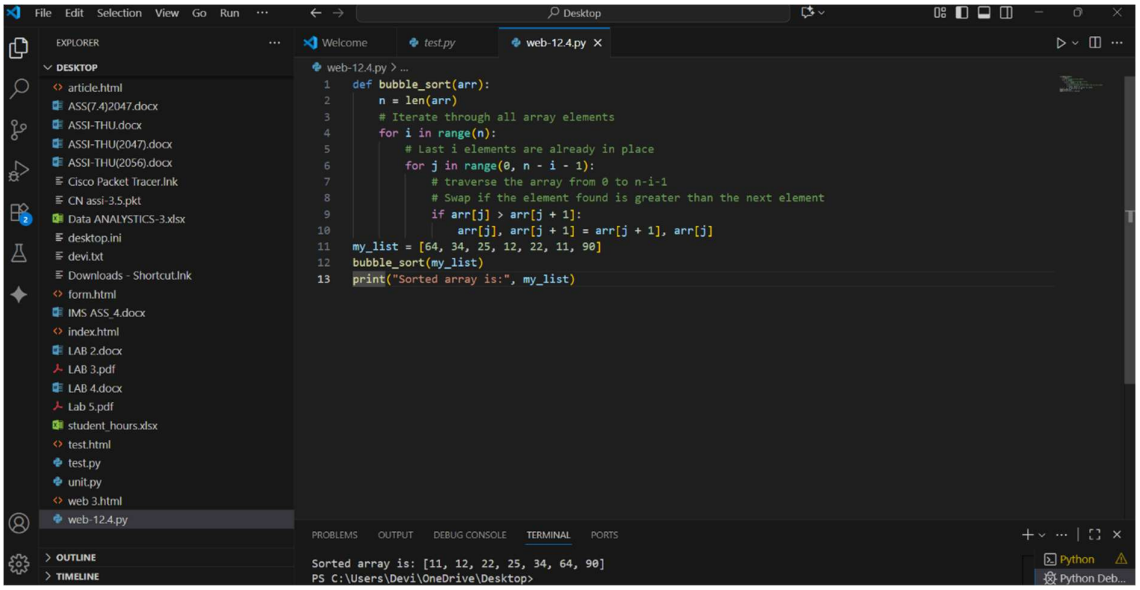
ASSIGNMENT-12.4

NAME:DEVI PRIYA.G

2403A52067

BATCH-04

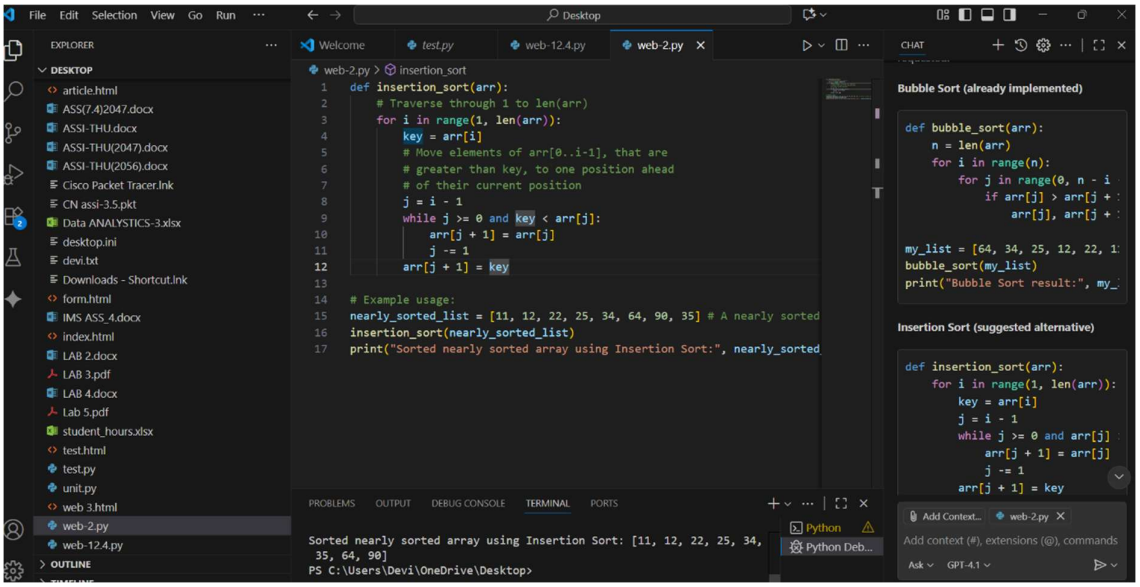
TASK-1: Write a Python implementation of Bubble Sort



EXPLANATION:

- **def bubble_sort(arr):**: This line defines a function named bubble_sort that takes one argument, arr, which is the list to be sorted.
- **n = len(arr)**: This gets the number of elements in the input list and stores it in the variable n.
- **for i in range(n):**: This is the outer loop. It iterates n times. In each iteration, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
- **for j in range(0, n - i - 1):**: This is the inner loop. It traverses the unsorted portion of the array (from the beginning up to the i-th element from the end, which is already sorted).

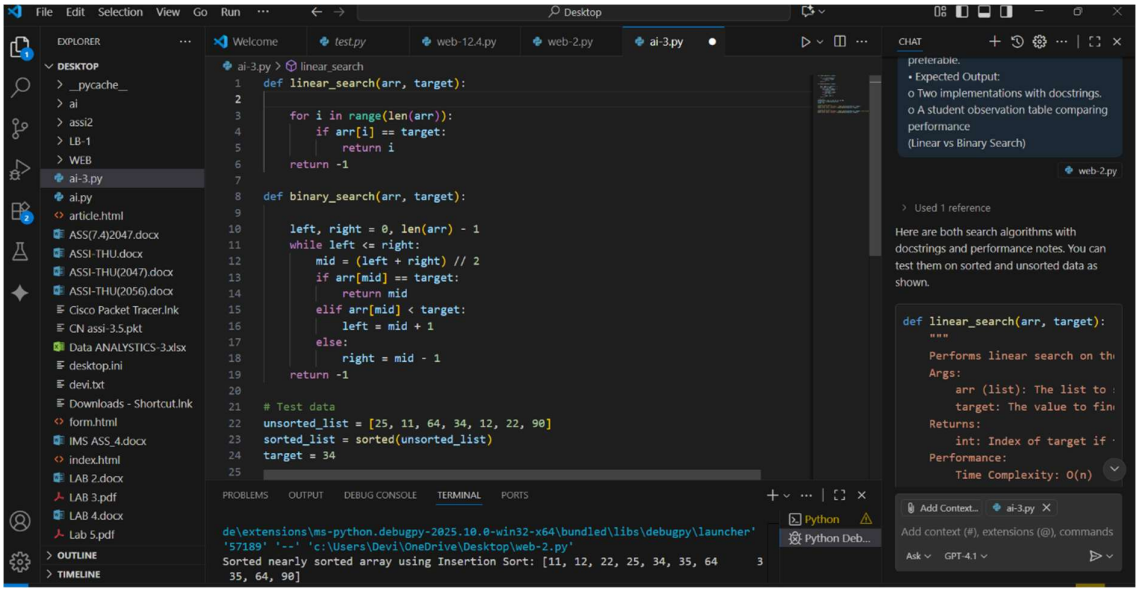
TASK-2: Provide Bubble Sort code to AI and ask it to suggest a more efficient algorithm for partially sorted arrays

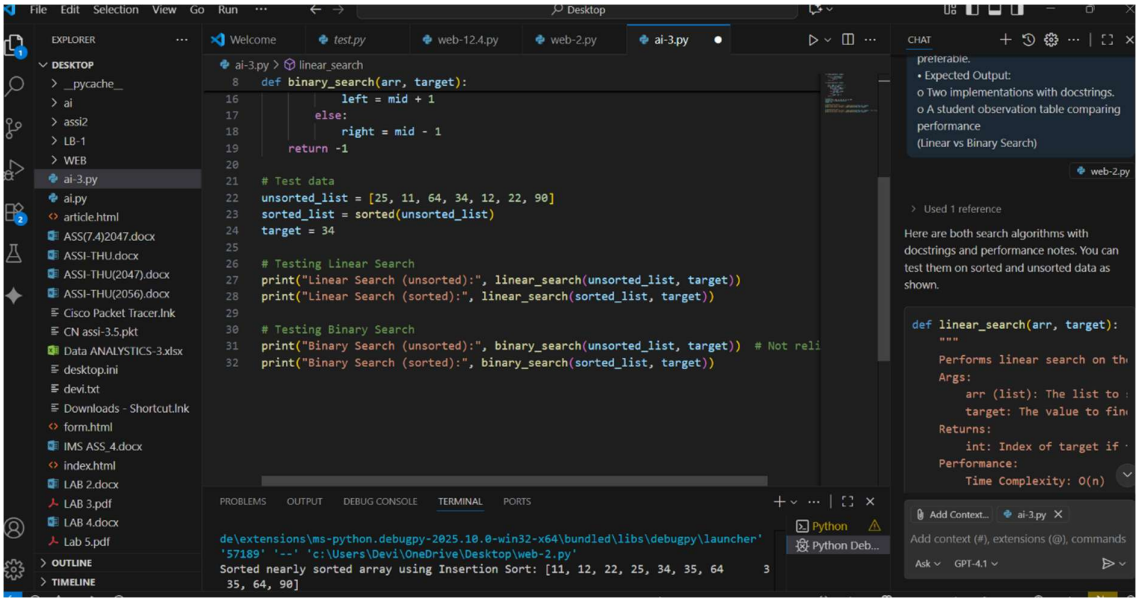


EXPLANATION:

- **def insertion_sort(arr):**: This line defines a function named insertion_sort that takes one argument, arr, which is the list to be sorted.
- **for i in range(1, len(arr)):**: This is the main loop. It iterates through the list starting from the second element (i = 1) up to the last element. It considers each element arr[i] as the "key" to be inserted into the already sorted portion of the array to its left.
- **key = arr[i]**: This line stores the current element being considered for insertion in the key variable.
- **j = i - 1**: This initializes a variable j to the index of the last element in the sorted portion of the array (to the left of the key).

TASK-3: Implement both Linear Search and Binary Search



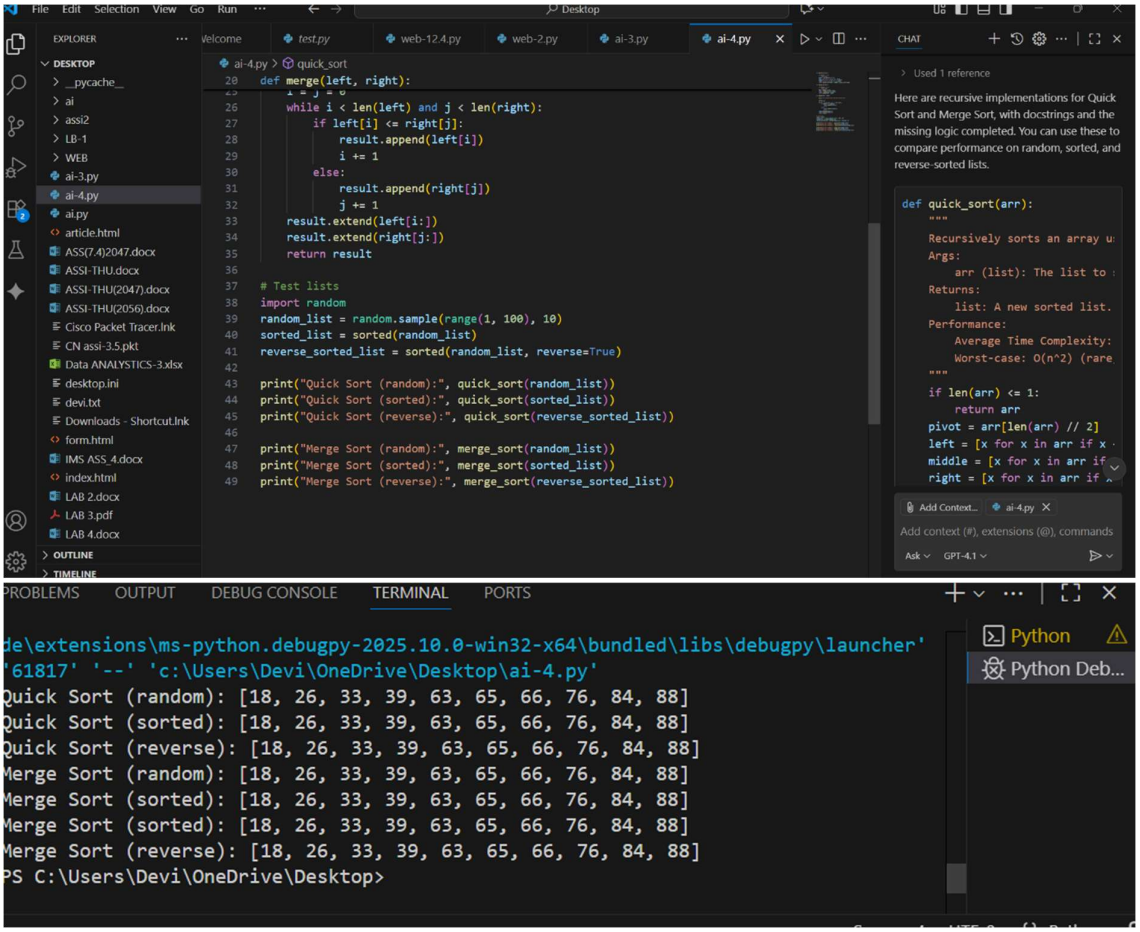


EXPLANATION:

- **def linear_search(arr, target):**: This line defines a function named linear_search that takes two arguments: arr (the list to search within) and target (the value to search for).
- **""" ... """**: This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns).
- **for index, element in enumerate(arr):**: This loop iterates through each element in the input list arr. The enumerate() function provides both the index and the value of each element.

- **if element == target:**: This condition checks if the current element is equal to the target value.

TASK-4: Quick Sort and Merge Sort Comparison



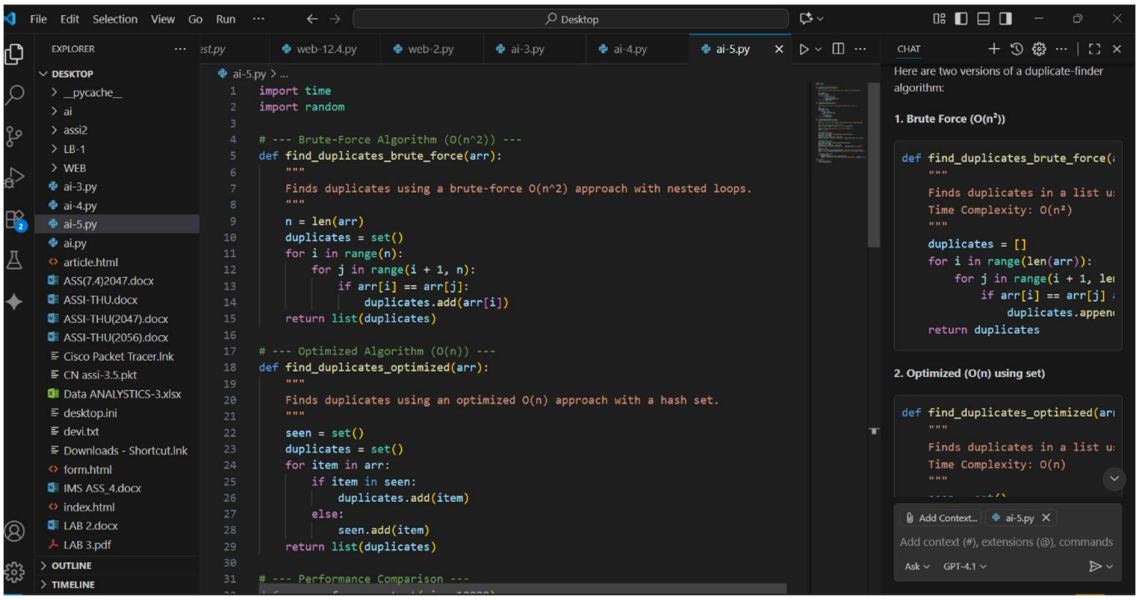
EXPLANATION: **def quick_sort(arr):**: This defines the recursive function quick_sort that takes a list arr as input.

""" ... """: This is a docstring explaining the function's purpose, arguments, and return value.

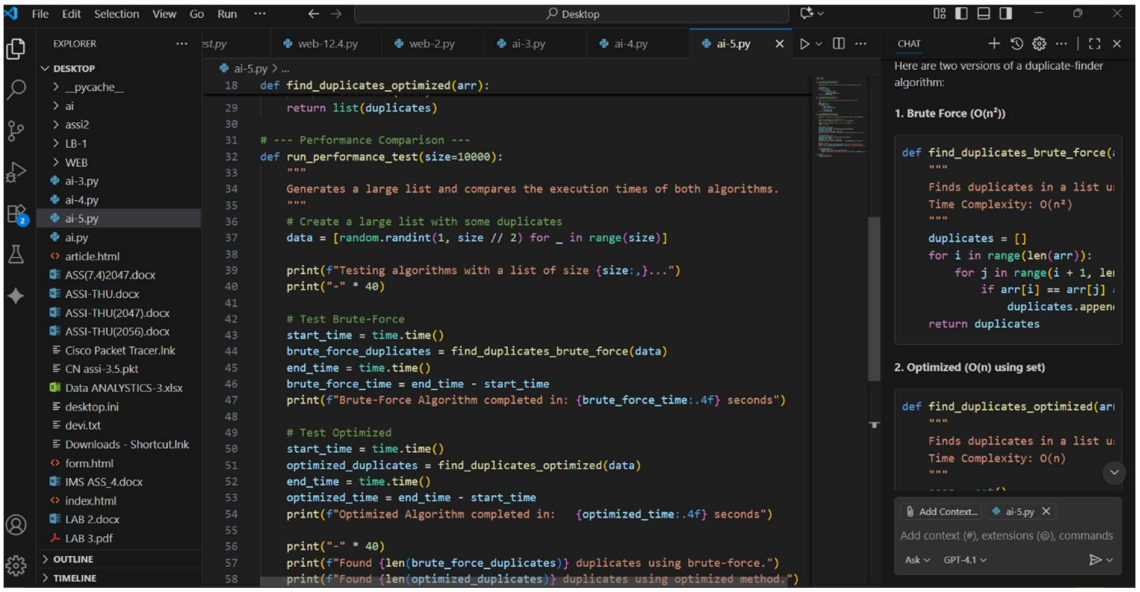
if len(arr) <= 1:: This is the base case for the recursion. If the list has 0 or 1 element, it's already sorted, so the function simply returns the list.

pivot = arr[0]: This selects the first element of the list as the pivot. *Note: Different pivot selection strategies exist, and the choice of pivot significantly impacts performance.*

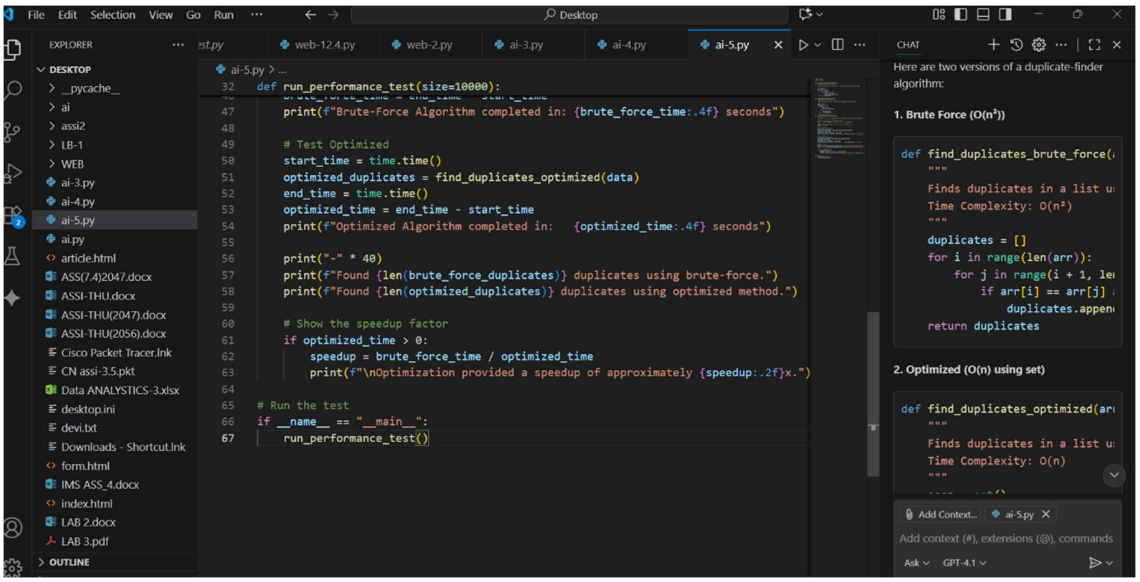
TASK-5: Give AI a naive algorithm (e.g., O(n²) duplicate search).

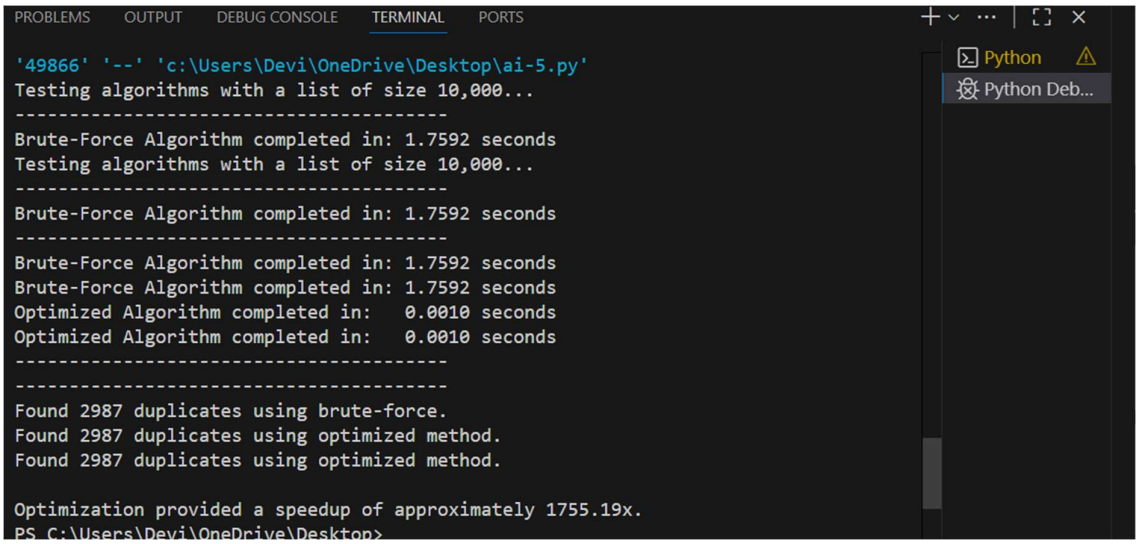



```python
import time
import random

# --- Brute-Force Algorithm (O(n^2)) ---
def find_duplicates_brute_force(arr):
    """
    Finds duplicates using a brute-force O(n^2) approach with nested loops.
    """
    n = len(arr)
    duplicates = set()
    for i in range(n):
        for j in range(i + 1, n):
            if arr[i] == arr[j]:
                duplicates.add(arr[i])
    return list(duplicates)

# --- Optimized Algorithm (O(n)) ---
def find_duplicates_optimized(arr):
    """
    Finds duplicates using an optimized O(n) approach with a hash set.
    """
    seen = set()
    duplicates = set()
    for item in arr:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)

# --- Performance Comparison ---
```




```python
def find_duplicates_optimized(arr):
    return list(duplicates)

# --- Performance Comparison ---
def run_performance_test(size=10000):
    """
    Generates a large list and compares the execution times of both algorithms.
    """
    # Create a large list with some duplicates
    data = [random.randint(1, size // 2) for _ in range(size)]

    print(f"Testing algorithms with a list of size {size:,}...")
    print("-" * 40)

    # Test Brute-Force
    start_time = time.time()
    brute_force_duplicates = find_duplicates_brute_force(data)
    end_time = time.time()
    brute_force_time = end_time - start_time
    print(f"Brute-Force Algorithm completed in: {brute_force_time:.4f} seconds")

    # Test Optimized
    start_time = time.time()
    optimized_duplicates = find_duplicates_optimized(data)
    end_time = time.time()
    optimized_time = end_time - start_time
    print(f"Optimized Algorithm completed in:   {optimized_time:.4f} seconds")

    print("-" * 40)
    print(f"Found {len(brute_force_duplicates)} duplicates using brute-force.")
    print(f"Found {len(optimized_duplicates)} duplicates using optimized method.")
```





EXPLANATION:

- **performance_data = []**: This initializes an empty list called performance_data. This list will be used to store dictionaries, where each dictionary will represent a row in the final performance table.
- **for size, data_types in execution_times.items():**: This is the outer loop that iterates through the execution_times dictionary. execution_times likely contains the measured execution times for different list sizes.
- **for data_type, algorithms in data_types.items():**: This inner loop iterates through the data types within each list size (e.g., 'random', 'sorted', 'reverse-sorted').