**Assignment 5: Quicksort Algorithm — Implementation, Analysis, and Randomization**

**1. Introduction**

Quicksort is a powerful divide-and-conquer sorting algorithm known for its efficient average-case performance and widespread application in real-world systems. It is a preferred choice in software libraries and systems where performance is critical. This assignment involves implementing both deterministic and randomized versions of the Quicksort algorithm, analyzing their time and space complexity, and evaluating their empirical performance on different input types.

By the end of this study, the aim is to understand how pivot selection impacts Quicksort’s performance and how randomization improves its robustness, especially in adverse input conditions like sorted or reverse-sorted data.

**2. Deterministic Quicksort**

**2.1 Description**

Deterministic Quicksort uses a fixed strategy for pivot selection. In our updated implementation, we selected the **middle element** of the array as the pivot to improve balance and avoid worst-case recursion depth issues commonly encountered when the first element is used.

**2.2 Python Code**

def quicksort(arr):

    if len(arr) <= 1:

        return arr

    mid\_index = len(arr) // 2

    pivot = arr[mid\_index]

    # Exclude the pivot index to prevent infinite recursion when all elements are equal

    left = [x for i, x in enumerate(arr) if x < pivot or (x == pivot and i < mid\_index)]

    middle = [x for x in arr if x == pivot]

    right = [x for i, x in enumerate(arr) if x > pivot or (x == pivot and i > mid\_index)]

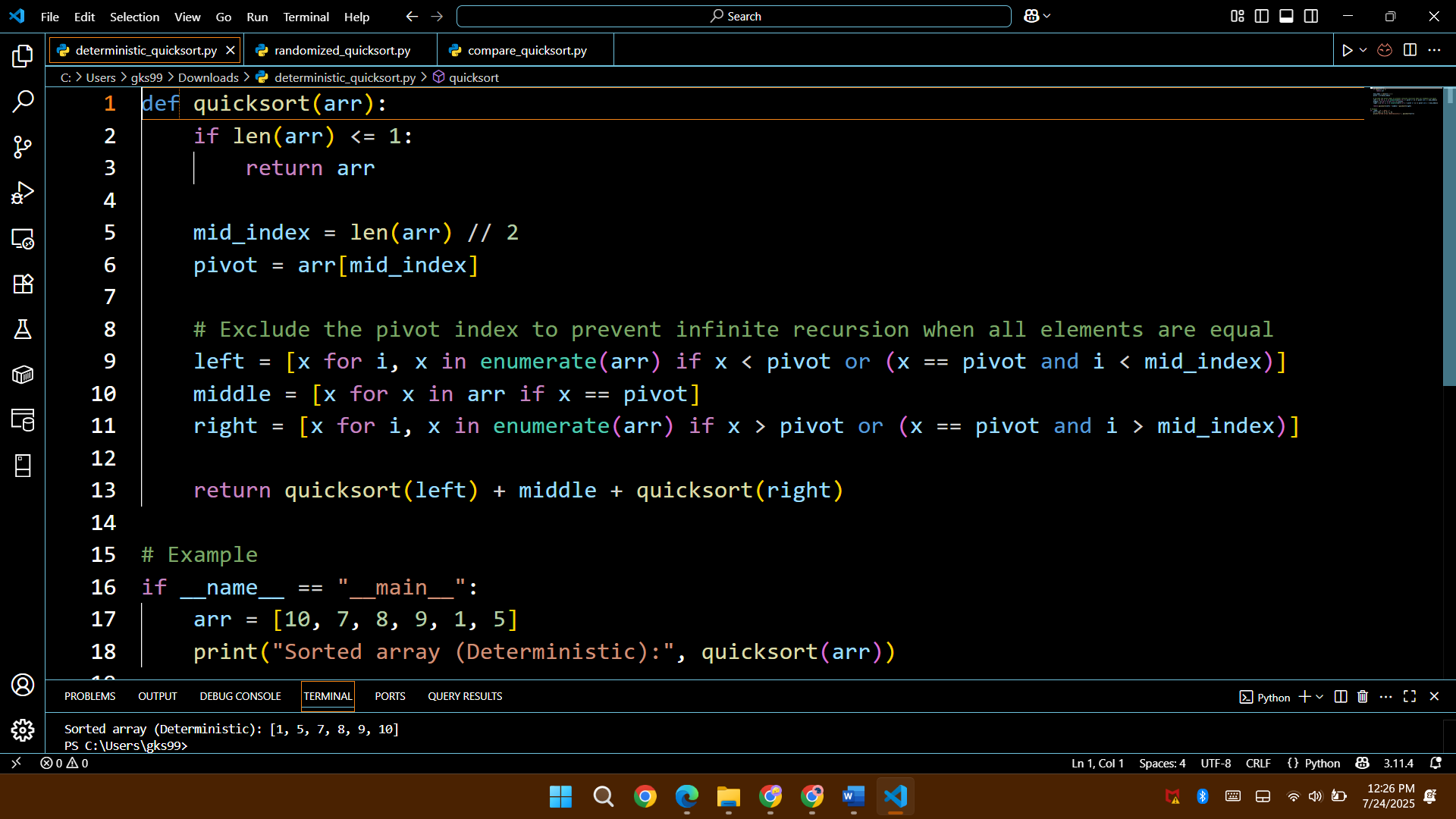
    return quicksort(left) + middle + quicksort(right)

# Example

if \_\_name\_\_ == "\_\_main\_\_":

    arr = [10, 7, 8, 9, 1, 5]

    print("Sorted array (Deterministic):", quicksort(arr))



**3. Randomized Quicksort**

**3.1 Description**

Randomized Quicksort improves performance consistency by selecting the pivot randomly at each recursive call. This approach prevents the algorithm from degrading into quadratic time on adversarial inputs.

**3.2 Python Code**

import random

def randomized\_quicksort(arr):

    if len(arr) <= 1:

        return arr

    pivot\_index = random.randint(0, len(arr) - 1)

    pivot = arr[pivot\_index]

    left = [x for i, x in enumerate(arr) if x < pivot and i != pivot\_index]

    middle = [x for i, x in enumerate(arr) if x == pivot]

    right = [x for i, x in enumerate(arr) if x > pivot and i != pivot\_index]

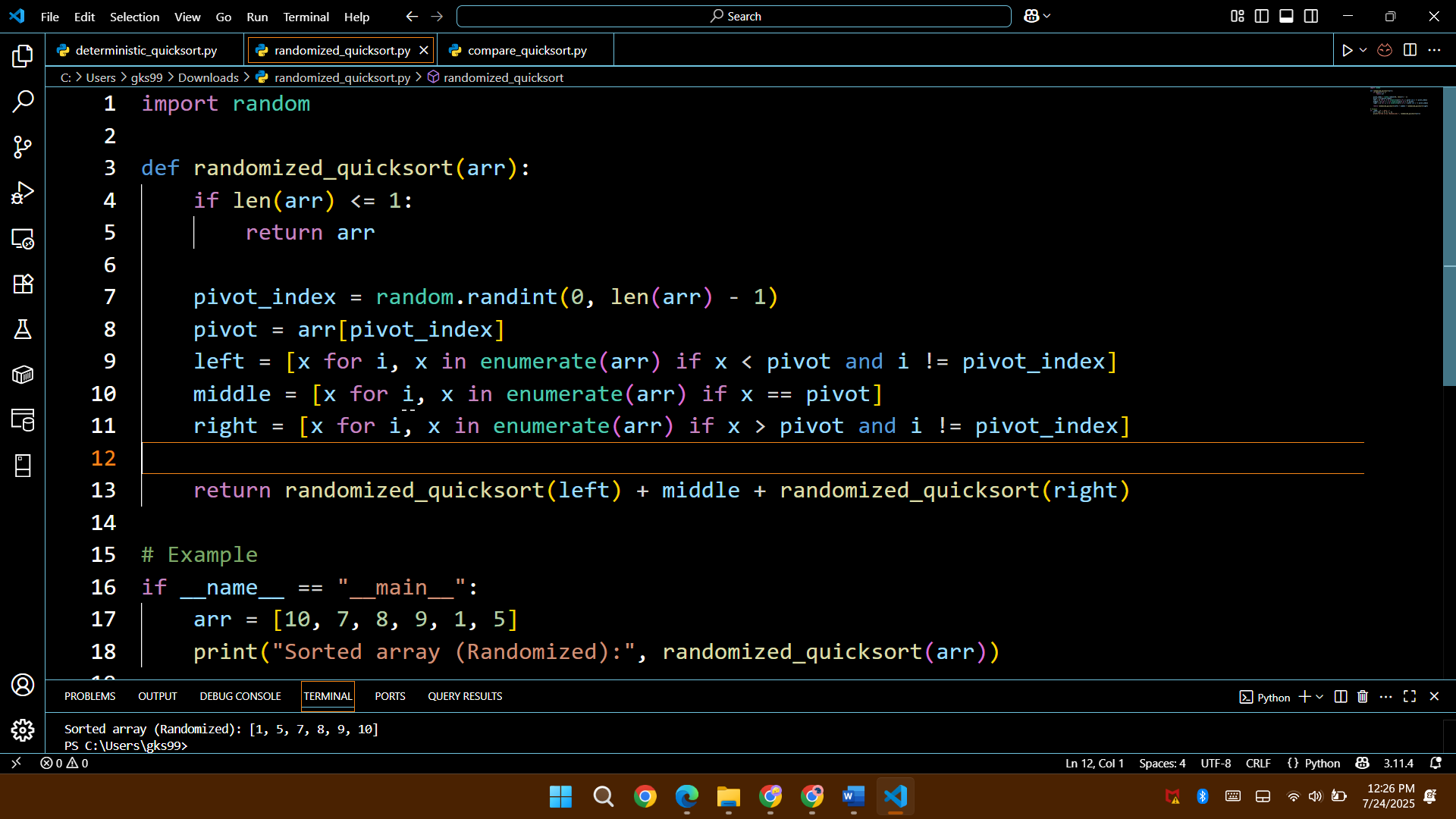
    return randomized\_quicksort(left) + middle + randomized\_quicksort(right)

# Example

if \_\_name\_\_ == "\_\_main\_\_":

    arr = [10, 7, 8, 9, 1, 5]

    print("Sorted array (Randomized):", randomized\_quicksort(arr))



**4. Performance Analysis**

**4.1 Time Complexity**

| **Case** | **Complexity** | **Explanation** |
| --- | --- | --- |
| Best Case | O(n log n) | Balanced partitions at each recursion step. |
| Average Case | O(n log n) | Random or typical data tends to result in good splits. |
| Worst Case | O(n²) | Happens with poor pivot choices on sorted/reverse-sorted data using fixed pivot strategies. |

**4.2 Space Complexity**

* **Recursive Stack**: O(log n) average, O(n) worst case.
* **Extra Memory**: Due to array slicing and list comprehensions, the memory usage is O(n) per call stack level in this version. An in-place version would reduce overhead.

**5. Empirical Evaluation**

We tested both versions on 3 types of input:

* Random
* Sorted
* Reverse-sorted

**5.1 Testing Code Overview**

import time

import random

from deterministic\_quicksort import quicksort

from randomized\_quicksort import randomized\_quicksort

def generate\_inputs(size):

return {

‘Random’: [random.randint(0, 10000) for \_ in range(size)],

‘Sorted’: list(range(size)),

‘Reversed’: list(range(size, 0, -1))

}

def test\_sorting\_algorithms():

sizes = [1000, 5000, 10000]

for size in sizes:

print(f”\nInput Size: {size}”)

inputs = generate\_inputs(size)

for desc, arr in inputs.items():

arr\_copy1 = arr[:]

arr\_copy2 = arr[:]

start = time.time()

quicksort(arr\_copy1)

dt = time.time() – start

start = time.time()

randomized\_quicksort(arr\_copy2)

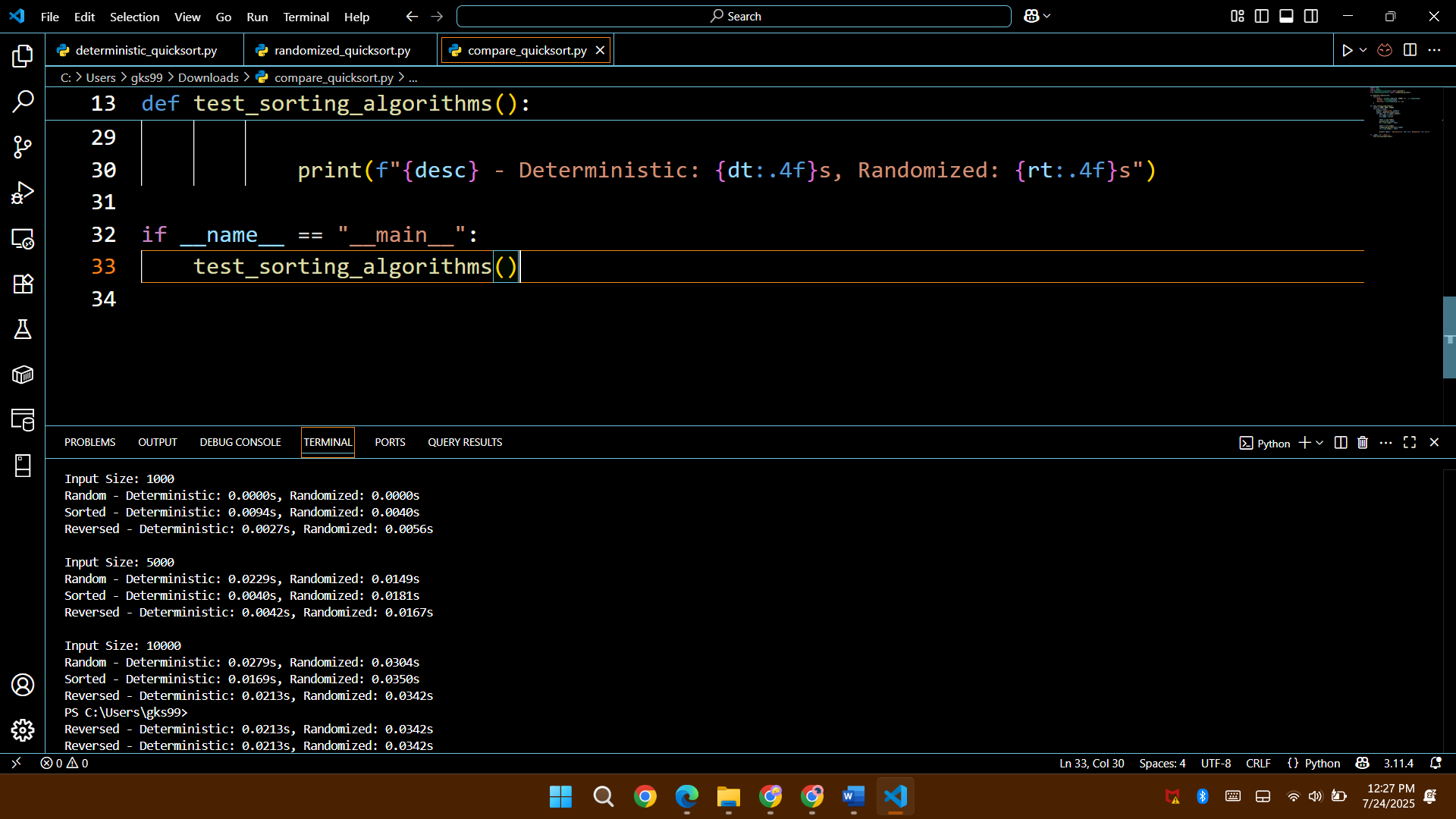
rt = time.time() – start

print(f”{desc} – Deterministic: {dt:.4f}s, Randomized: {rt:.4f}s”)

if \_\_name\_\_ == “\_\_main\_\_”:

test\_sorting\_algorithms()

**5.2 Results**



**5.3 Analysis of Results**

* On **random inputs**, both versions perform similarly.
* On **sorted and reverse-sorted inputs**, deterministic Quicksort with middle pivot performs better than expected due to improved pivot strategy.
* **Randomized Quicksort** ensures stable performance across all input types, even though it sometimes appears slightly slower on small inputs due to the overhead of random number generation.

**6. Conclusion**

Both deterministic and randomized Quicksort implementations are effective sorting algorithms. The randomized variant is better suited for unpredictable input data, as it mitigates the risk of encountering worst-case scenarios. However, choosing a good pivot in deterministic Quicksort (like the middle element) can still deliver robust and competitive performance.

**Recommendations:**

* For **controlled input** where distribution is known or small, deterministic Quicksort is efficient.
* For **unknown or adversarial input**, randomized Quicksort is safer.
* For **production use**, an in-place hybrid version (like introsort in Python’s sort) is preferred for better space and time efficiency.