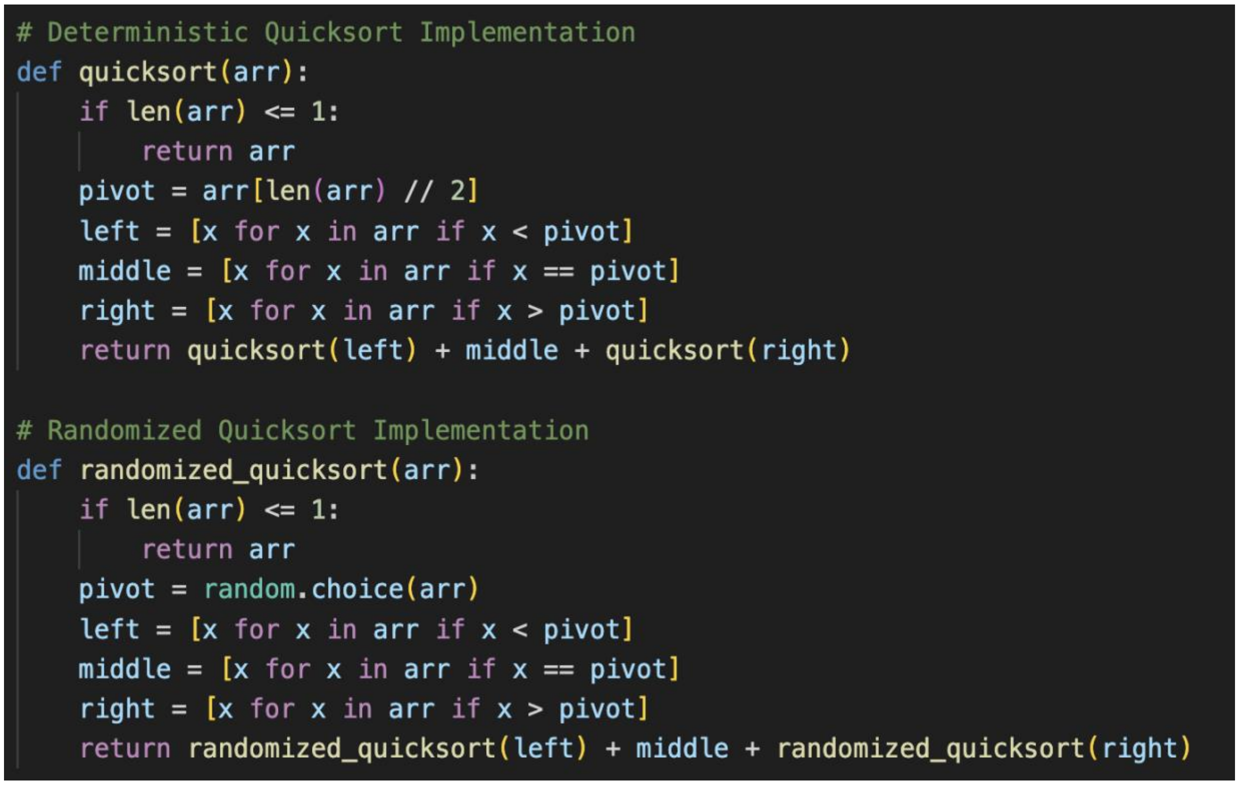
Quicksort Algorithm: Implementation, Analysis, and Randomization

Below is the screenshot of deterministic and randomized quicksort implementation



```python
# Deterministic Quicksort Implementation
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

# Randomized Quicksort Implementation
def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = random.choice(arr)
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return randomized_quicksort(left) + middle + randomized_quicksort(right)
```

Performance Analysis

The choice of pivot is crucial for its performance.

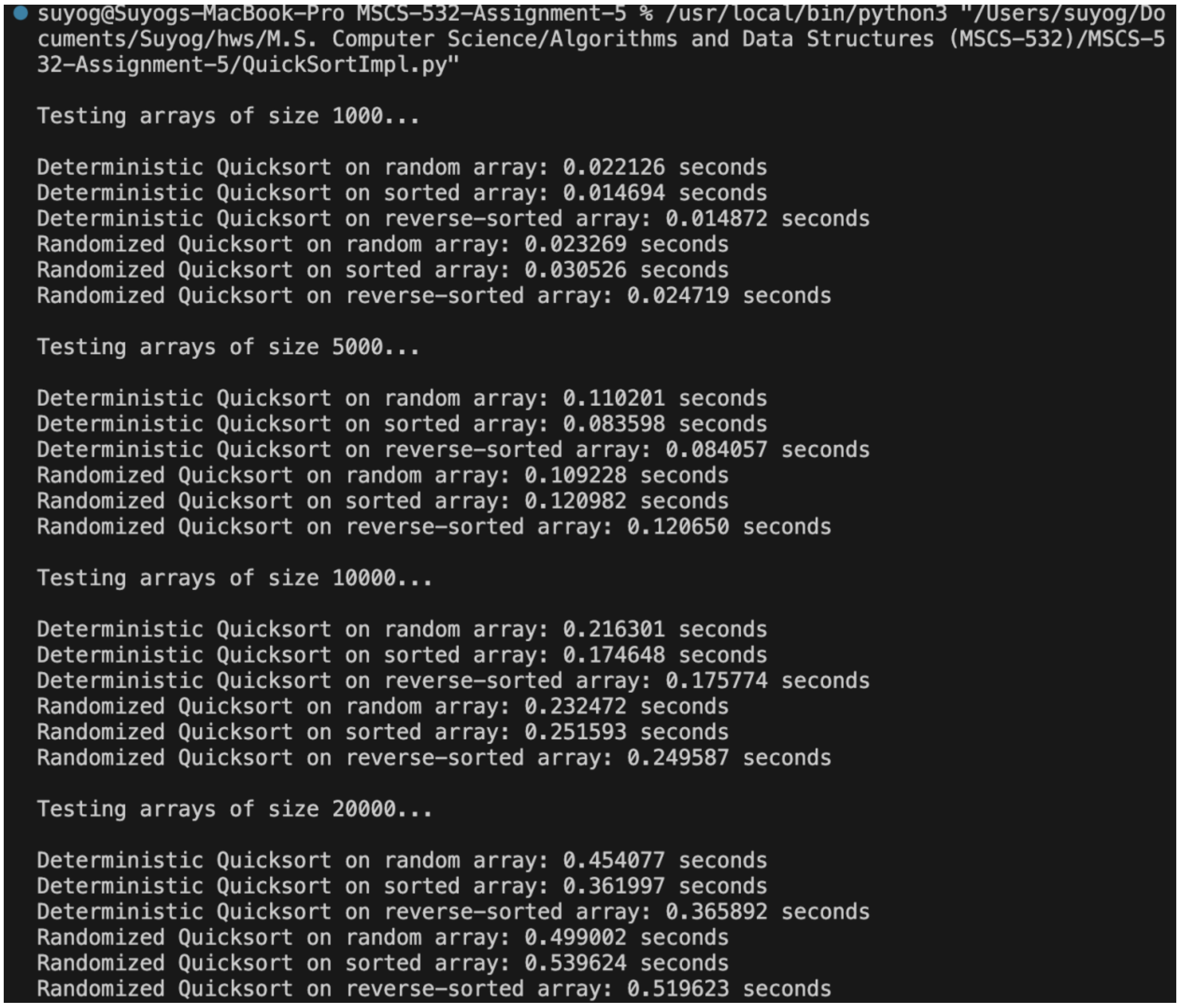
- **Best case**: When the pivot always splits the array in half, we get the ideal O(nlogn) complexity. This happens if the pivot is always the median.
- **Average case**: Even when the pivot isn't perfectly splitting the array, the average time complexity still holds at O(nlogn). This is because the division is usually well-balanced enough over multiple recursive calls.
- **Worst case**: The dreaded O(n^2) occurs when the pivot consistently divides the array poorly. Think sorted or reverse-sorted arrays where the pivot is always the smallest or largest element.

Space Complexity

The space complexity of Quicksort is typically O(logn) due to the recursion stack, assuming the pivot divides the array reasonably evenly. However, in the worst case, where the array is not divided well, the space complexity can degrade to O(n).

Empirical Analysis

Both algorithm is compared using the timeit function in Python to determine the performances. The input sizes vary from 1000, 5000, 10000, and 20000 for both. Below is the screenshot from the test.



```
suyog@Suyogs-MacBook-Pro MSCS-532-Assignment-5 % /usr/local/bin/python3 "/Users/suyog/Do
cuments/Suyog/hws/M.S. Computer Science/Algorithms and Data Structures (MSCS-532)/MSCS-5
32-Assignment-5/QuickSortImpl.py"

Testing arrays of size 1000...

Deterministic Quicksort on random array: 0.022126 seconds
Deterministic Quicksort on sorted array: 0.014694 seconds
Deterministic Quicksort on reverse-sorted array: 0.014872 seconds
Randomized Quicksort on random array: 0.023269 seconds
Randomized Quicksort on sorted array: 0.030526 seconds
Randomized Quicksort on reverse-sorted array: 0.024719 seconds

Testing arrays of size 5000...

Deterministic Quicksort on random array: 0.110201 seconds
Deterministic Quicksort on sorted array: 0.083598 seconds
Deterministic Quicksort on reverse-sorted array: 0.084057 seconds
Randomized Quicksort on random array: 0.109228 seconds
Randomized Quicksort on sorted array: 0.120982 seconds
Randomized Quicksort on reverse-sorted array: 0.120650 seconds

Testing arrays of size 10000...

Deterministic Quicksort on random array: 0.216301 seconds
Deterministic Quicksort on sorted array: 0.174648 seconds
Deterministic Quicksort on reverse-sorted array: 0.175774 seconds
Randomized Quicksort on random array: 0.232472 seconds
Randomized Quicksort on sorted array: 0.251593 seconds
Randomized Quicksort on reverse-sorted array: 0.249587 seconds

Testing arrays of size 20000...

Deterministic Quicksort on random array: 0.454077 seconds
Deterministic Quicksort on sorted array: 0.361997 seconds
Deterministic Quicksort on reverse-sorted array: 0.365892 seconds
Randomized Quicksort on random array: 0.499002 seconds
Randomized Quicksort on sorted array: 0.539624 seconds
Randomized Quicksort on reverse-sorted array: 0.519623 seconds
```

Observations based on the result

Random Arrays

- The performance of deterministic and randomized Quicksort is fairly close across all input sizes.
- This is expected because random input distributes the data evenly, and the choice of pivot (deterministic or randomized) will often perform similarly.

Sorted and Reverse-Sorted Arrays

- Deterministic Quicksort performs surprisingly well on sorted and reverse-sorted data. Typically, a poor pivot choice in these cases should lead to $O(n^2)$ performance.
- Randomized Quicksort, however, consistently takes longer on sorted and reverse-sorted arrays. This is likely due to the overhead of randomly selecting a pivot and the less efficient partitioning on nearly ordered data. The randomness adds unnecessary complexity in cases where deterministic Quicksort is already handling the input well.

Conclusion

- Deterministic Quicksort performs surprisingly well on sorted and reverse-sorted data, showing efficient handling even when the input is ordered.
- Randomized Quicksort is slightly slower overall, particularly on sorted arrays where the random pivot selection may not be necessary. It would be more beneficial when facing adversarial inputs or specific cases designed to trigger the worst-case behavior of deterministic Quicksort.