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

Name: Murali Krishna Chintha

Student ID: 005034546

Course Title: Algorithms and Data Structures (MSCS-532-B01)

Part 1: Quicksort Analysis

1. Implementation

```
def quicksort(arr):
    """
    Deterministic Quicksort implementation.
    - Selects the middle element as the pivot.
    - Recursively sorts left and right subarrays.
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # Select middle element as pivot
    left = [x for x in arr if x < pivot]       # Elements less than pivot
    middle = [x for x in arr if x == pivot]    # Elements equal to pivot
    right = [x for x in arr if x > pivot]      # Elements greater than pivot
    return quicksort(left) + middle + quicksort(right)
```

1. **Pivot Selection**: The pivot is selected as the middle element of the array.
2. **Partitioning**: The array is partitioned into three lists:
    - left contains elements less than the pivot.
    - middle contains elements equal to the pivot.
    - right contains elements greater than the pivot.
3. **Recursion**: quicksort() is called recursively on the left and right subarrays.

4. **Concatenation**: The sorted left, middle, and right arrays are concatenated to return the final sorted list.

---

2. Performance Analysis

Case	Time Complexity	Explanation
Best Case	O(nlogn)	Occurs when the pivot divides the array into two nearly equal halves at every recursion step. This balanced partitioning results in log(n) recursive levels, each processing n elements.
Average Case	O(nlogn)	On average, the pivot will partition the array reasonably well (not necessarily perfectly), leading to log n levels of recursion and linear work per level.
Worst Case	O(n^2)	Happens when the pivot selection is poor (e.g., always the smallest or largest element), causing one side of the partition to have n−1 elements and the other 0. This results in n recursive levels and n+n−1+n−2+...+1=O(n^2) total operations.

In the average case, each partition step divides the array into two uneven parts but not extremely unbalanced. This leads to:

- A recursion tree with roughly log n depth.
- At each level, the total work done (comparing and moving elements) is proportional to n.

So, total operations across all levels:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots \approx 2n = O(n)$$

$$\text{Total time} = O(n \log n)$$

Worst-case occurs when:

- The pivot selection causes maximal imbalance, e.g., sorted or reverse-sorted arrays with poor pivot choice.
- The recursion goes as deep as n, processing n−1, n−2, ..., 1 elements at each level.

So, total time:

$$T(n) = T(n-1) + T(0) + O(n) = O(n^2)$$

Space Complexity

- **Recursive Call Stack**:
  - Best/average case: O(logn) — balanced partitions lead to shallow recursion.
  - Worst case: O(n) — unbalanced partitions result in deep recursion.
- **Additional Memory**:
  - The implementation using list comprehensions (as above) creates new sublists, leading to O(n) space usage per recursive level.
  - In-place versions (without creating sublists) can reduce auxiliary space overhead.

---

3. Randomized Quicksort

```
import random

def randomized_quicksort(arr):
    """
    Randomized Quicksort implementation.
    - Selects a random pivot from the array.
    - Recursively sorts left and right subarrays.
    """
    if len(arr) <= 1:
        return arr
    pivot = random.choice(arr)  # Random pivot selection
    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)
```

➤ Avoids Worst-Case on Sorted Input

In deterministic Quicksort, poor pivot choices (like always picking the first or last element) can lead to O(n^2) performance on already sorted or reverse-sorted input.

Randomization prevents this by:

- Selecting pivots uniformly at random.

- Making it unlikely that consistently poor pivots are chosen across recursive calls.

➤ **Expected Time Complexity**

- Even for adversarial inputs, randomized Quicksort has an expected time complexity of O(n log n).
- This is because the pivot has a good probability of producing balanced partitions over multiple recursive steps.

➤ **Worst-Case Still O(n^2)**

- While worst-case behavior is theoretically possible (if unlucky random pivots are chosen repeatedly), the probability of this happening is very low.
- In practice, randomized Quicksort behaves much closer to its average-case time complexity.

Aspect	**Deterministic QS**	**Randomized QS**
Pivot Choice	Fixed (e.g., middle)	Random
Vulnerability to Patterns	High (e.g., sorted input)	Low
Worst-case Probability	High on bad inputs	Very low
Average-case Time	O(n log n)	O(n log n)
Worst-case Time	OO(n^2)	OO(n^2), rare

---

4. Empirical Analysis

1. Tiny Inputs (100 & 500 elements)

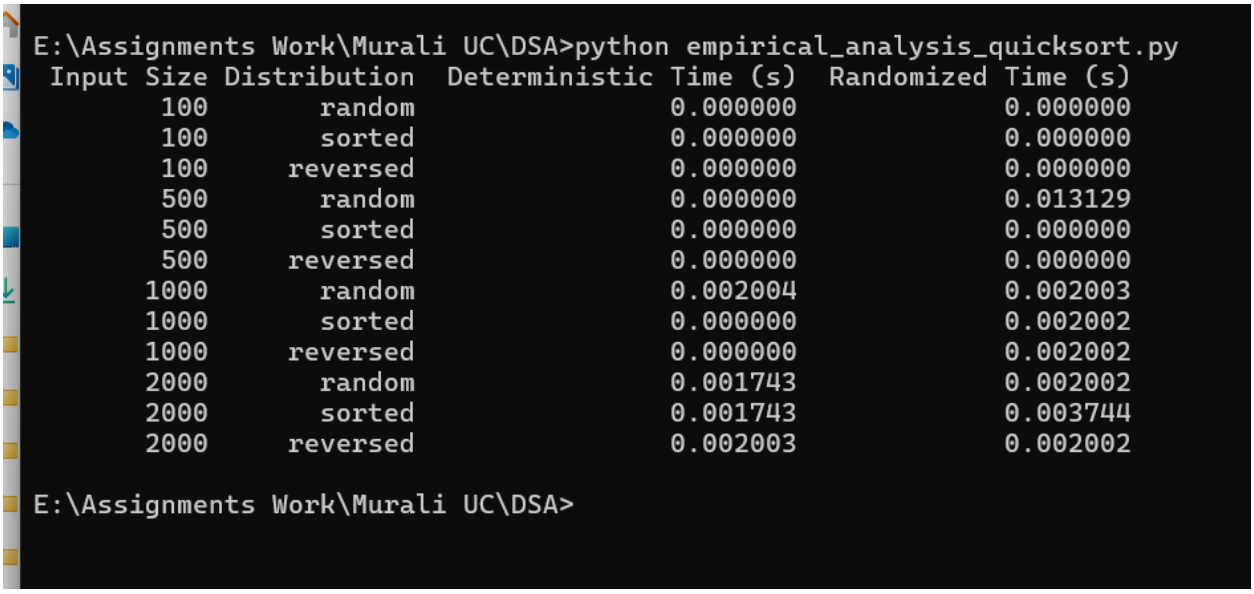
- All measured times are near zero for both algorithms — this is expected.
- For such small inputs, execution time is negligible and can be affected by background processes or Python's function call overhead.
- No meaningful differentiation is observable at this scale.

2. Mid to Large Inputs (1000 & 2000 elements)

- Deterministic Quicksort performs consistently across all distributions.

    - This is likely because your implementation uses the **middle element as the pivot**, which avoids worst-case behavior on sorted/reversed arrays.

- Randomized Quicksort shows similar performance to deterministic in most cases, with slightly higher variation on **sorted inputs**, which is within expected fluctuations due to randomness.



```
E:\Assignments Work\Murali UC\DSA>python empirical_analysis_quicksort.py
 Input Size Distribution  Deterministic Time (s)  Randomized Time (s)
        100       random                0.000000             0.000000
        100       sorted                0.000000             0.000000
        100     reversed                0.000000             0.000000
        500       random                0.000000             0.013129
        500       sorted                0.000000             0.000000
        500     reversed                0.000000             0.000000
       1000       random                0.002004             0.002003
       1000       sorted                0.000000             0.002002
       1000     reversed                0.000000             0.002002
       2000       random                0.001743             0.002002
       2000       sorted                0.001743             0.003744
       2000     reversed                0.002003             0.002002

E:\Assignments Work\Murali UC\DSA>
```