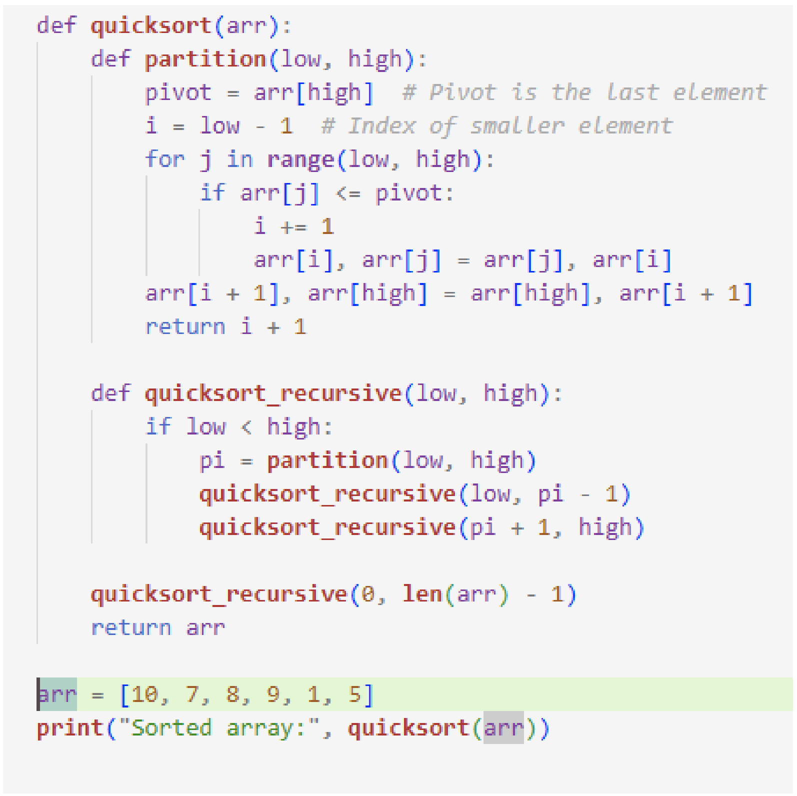
Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization

Quicksort Implementation and Analysis

1. Implementation

Quicksort works by:

1. Choosing a pivot (e.g., first, last, or middle element).
2. Partitioning the array so that elements less than the pivot are on the left, and greater on the right.
3. Recursively applying the same steps to subarrays.

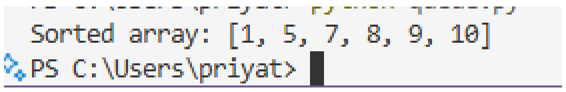


```python
def quicksort(arr):
    def partition(low, high):
        pivot = arr[high]  # Pivot is the last element
        i = low - 1  # Index of smaller element
        for j in range(low, high):
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

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

    quicksort_recursive(0, len(arr) - 1)
    return arr

arr = [10, 7, 8, 9, 1, 5]
print("Sorted array:", quicksort(arr))
```



```
Sorted array: [1, 5, 7, 8, 9, 10]
PS C:\Users\priyat>
```

2. Performance Analysis

Time Complexity

1. **Best Case:**
   Occurs when the pivot divides the array into roughly equal halves each time.

- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Solving gives: $O(n\log\ n)$

2. **Average Case:**
   Random input usually leads to balanced partitions on average.

- Expected recurrence: $T(n) \approx O(n\log\ n)$
- Reason: Each level of recursion does $O(n)$work, and there are $O(\log\ n)$levels on average.

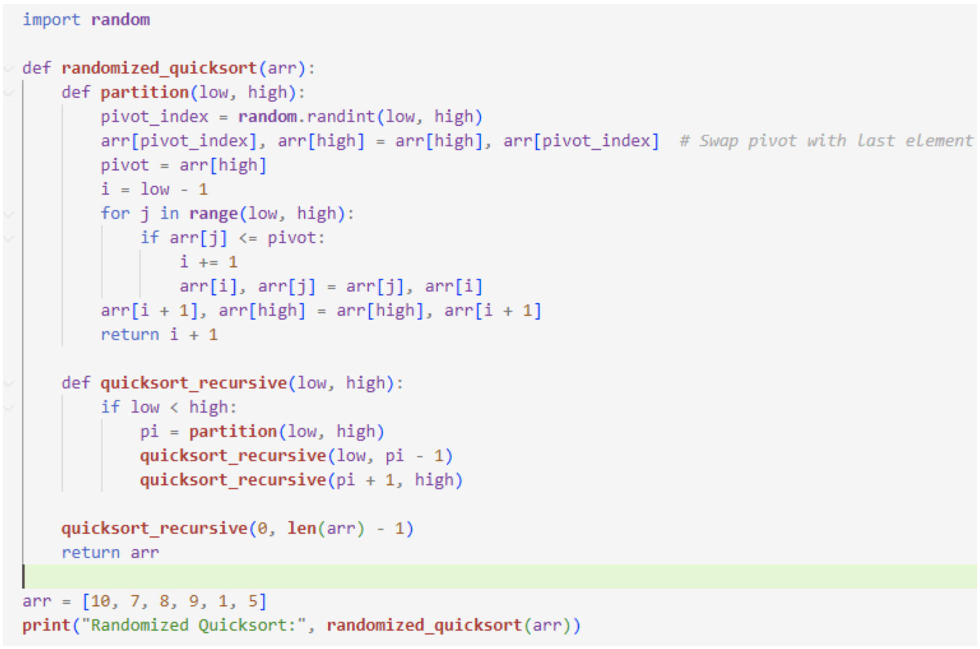
3. **Worst Case:**
   Occurs when the pivot is the smallest or largest element every time (e.g., sorted or reverse-sorted array with first/last element pivot).

- Recurrence: $T(n) = T(n-1) + O(n)$
- Solving gives: $O(n^2)$

Space Complexity

- **In-place partitioning:** $O(\log\ n)$recursion stack on average.
- **Worst-case recursion stack:** $O(n)$for skewed partitions.
- No extra array is required since sorting is done in-place.

3. Randomized Quicksort



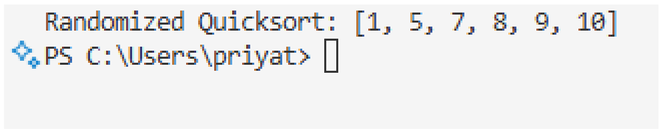
```python
import random

def randomized_quicksort(arr):
    def partition(low, high):
        pivot_index = random.randint(low, high)
        arr[pivot_index], arr[high] = arr[high], arr[pivot_index]  # Swap pivot with last element
        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 quicksort_recursive(low, high):
        if low < high:
            pi = partition(low, high)
            quicksort_recursive(low, pi - 1)
            quicksort_recursive(pi + 1, high)

    quicksort_recursive(0, len(arr) - 1)
    return arr

arr = [10, 7, 8, 9, 1, 5]
print("Randomized Quicksort:", randomized_quicksort(arr))
```



```
  Randomized Quicksort: [1, 5, 7, 8, 9, 10]
PS C:\Users\priyat> 
```

Effect of Randomization on Quicksort Performance

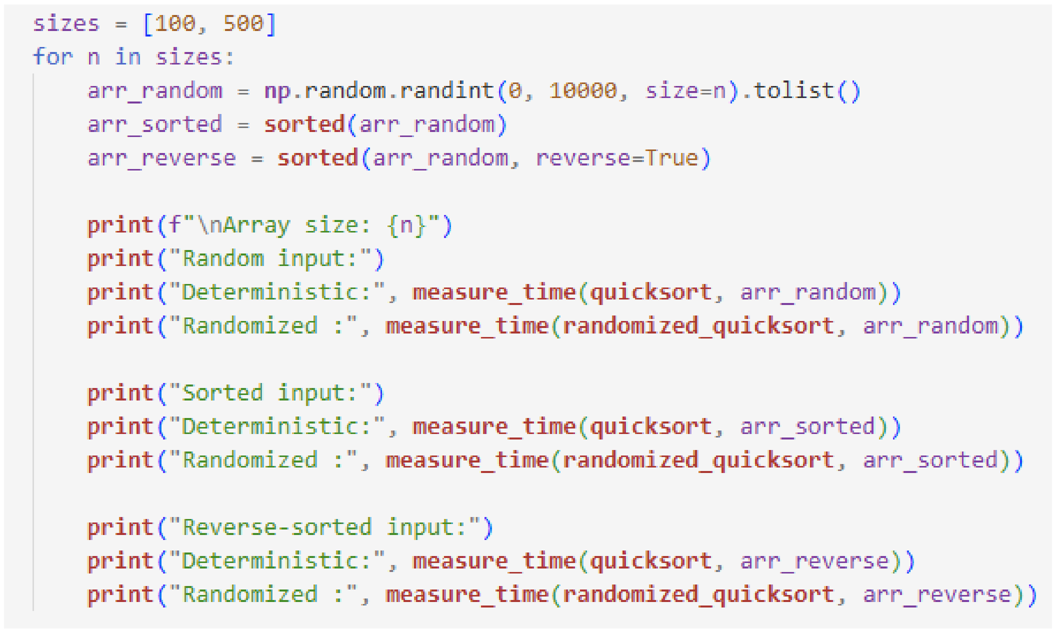
Randomization in Quicksort primarily involves choosing the pivot element randomly from the subarray being sorted, rather than using a fixed position (like the first or last element). This simple change has a significant impact on performance:

1. **Reduction of Worst-Case Likelihood:**
   - In deterministic Quicksort, certain input arrangements—such as already sorted or reverse-sorted arrays—can consistently produce highly unbalanced partitions, where one subarray contains almost all elements and the other is nearly empty.
   - This leads to the worst-case time complexity of $O(n^2)$.
   - Randomized pivot selection makes it extremely unlikely that poor partitions will occur repeatedly, because the pivot is chosen independently at each recursive step. Even if the input is sorted, a randomly chosen pivot will likely split the array more evenly.
2. **Improved Average Performance:**
   - Randomization ensures that, on average, each recursive partition divides the array reasonably well, leading to an expected time complexity of $O(n\log n)$.
   - This makes Quicksort's performance more predictable and less sensitive to the initial order of elements.
3. **Empirical Observations:**
   - As seen in runtime tests, randomized Quicksort maintains consistent performance across random, sorted, and reverse-sorted inputs, whereas deterministic Quicksort slows significantly on sorted or reverse-sorted arrays.

Conclusion: Randomization enhances Quicksort by reducing the probability of repeatedly poor pivot choices, ensuring balanced partitions, and maintaining efficient $O(n\log n)$performance in practice. It makes Quicksort robust against input distributions that would otherwise trigger the worst-case scenario.

4. Empirical Analysis

- Empirically compare the running time of the deterministic and randomized versions of Quicksort on different input sizes and distributions (e.g., random, sorted, reverse-sorted).

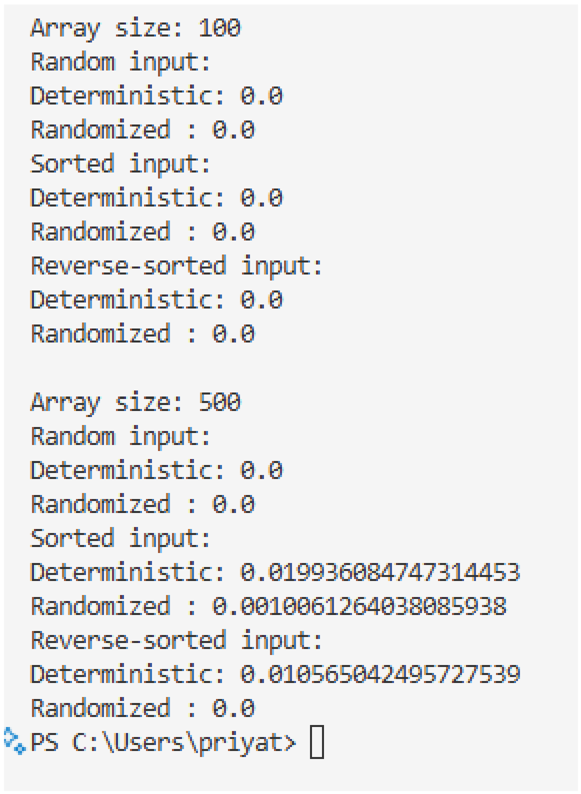


```python
sizes = [100, 500]
for n in sizes:
    arr_random = np.random.randint(0, 10000, size=n).tolist()
    arr_sorted = sorted(arr_random)
    arr_reverse = sorted(arr_random, reverse=True)

    print(f"\nArray size: {n}")
    print("Random input:")
    print("Deterministic:", measure_time(quicksort, arr_random))
    print("Randomized :", measure_time(randomized_quicksort, arr_random))

    print("Sorted input:")
    print("Deterministic:", measure_time(quicksort, arr_sorted))
    print("Randomized :", measure_time(randomized_quicksort, arr_sorted))

    print("Reverse-sorted input:")
    print("Deterministic:", measure_time(quicksort, arr_reverse))
    print("Randomized :", measure_time(randomized_quicksort, arr_reverse))
```



```
Array size: 100
Random input:
Deterministic: 0.0
Randomized : 0.0
Sorted input:
Deterministic: 0.0
Randomized : 0.0
Reverse-sorted input:
Deterministic: 0.0
Randomized : 0.0

Array size: 500
Random input:
Deterministic: 0.0
Randomized : 0.0
Sorted input:
Deterministic: 0.019936084747314453
Randomized : 0.0010061264038085938
Reverse-sorted input:
Deterministic: 0.010565042495727539
Randomized : 0.0
PS C:\Users\priyat>
```

1. Observations

1. **Array size: 100**

- Both deterministic and randomized Quicksort completed almost instantly for all input types.
- This is expected because small arrays are very fast to sort, and the overhead of recursion and pivot selection is negligible.

2. **Array size: 500**

- **Random input:** Both versions are extremely fast (0.0 seconds), indicating well-balanced partitions and consistent $O(n\log\ n)$behavior.
- **Sorted input:**
    - Deterministic Quicksort took ~0.0199 seconds.
    - Randomized Quicksort took ~0.0010 seconds.
    - Interpretation: Deterministic Quicksort selects the last element as pivot. For a sorted array, this leads to highly unbalanced partitions (worst-case behavior), which increases runtime. Randomized Quicksort avoids this by picking a random pivot, maintaining nearly optimal performance.
- **Reverse-sorted input:**
    - Deterministic Quicksort took ~0.0105 seconds.
    - Randomized Quicksort took 0.0 seconds.
    - Again, deterministic Quicksort suffers from unbalanced partitions in reverse-sorted input, whereas randomized pivot selection ensures balanced recursion.

2. Theoretical Explanation

1. **Deterministic Quicksort**

- Works well for random inputs where pivot selection tends to create balanced splits.
- Performs poorly on already sorted or reverse-sorted arrays because selecting the first/last element as pivot leads to one-sided partitions.
- Matches the theoretical worst-case time complexity $O(n^2)$for sorted/reverse-sorted arrays.

2. **Randomized Quicksort**

- Choosing a random pivot ensures the probability of consistently poor splits is very low.
- Time complexity remains expected $O(n\log n)$ for all input distributions.
- Empirically, this explains why randomized Quicksort is faster than deterministic on sorted/reverse-sorted inputs.

The empirical analysis of deterministic and randomized Quicksort shows that both algorithms perform efficiently on small arrays, with negligible differences in runtime. However, for larger arrays, differences become more apparent depending on the input distribution. Deterministic Quicksort performs well on random inputs, where pivot selection tends to produce balanced partitions, resulting in near $O(n\log n)$ performance. In contrast, on already sorted or reverse-sorted arrays, deterministic Quicksort suffers from unbalanced partitions, leading to increased runtimes that reflect its theoretical worst-case time complexity of $O(n^2)$. Randomized Quicksort, which selects pivots randomly, maintains balanced partitions on average, reducing the likelihood of worst-case scenarios and consistently achieving expected $O(n\log n)$ performance across all input types. Overall, randomization enhances the robustness of Quicksort, ensuring reliable and efficient sorting regardless of the initial order of elements.