1. Performance Analysis of Deterministic:

Implementation Details: This implementation uses the Lomuto partition scheme, where the pivot is chosen as the last element in the subarray. The partition function rearranges the array so that elements smaller than the pivot are on the left, and elements greater are on the right.

Design Choice: Pivot Selection Strategy (Deterministic): Last Element

- **Choice:** The deterministic version consistently uses the last element of the subarray as its pivot.
- **Reasoning:** This is the standard convention for the Lomuto partition scheme. It's simple—it requires no extra computation to select the pivot. However, this choice is also critical for demonstrating the algorithm's vulnerabilities, as it directly leads to the O(n2) worst-case on sorted data.

Analysis:

Time Complexity

The time complexity of Quicksort is highly dependent on the pivot selection strategy and the resulting balance of the partitions.

- **Best Case:** O(nlogn) The best case occurs when the partition process always picks the median element as the pivot. This divides the array into two perfectly equal halves. The recurrence relation for this scenario is: $T(n)=2T(n/2)+O(n)$ Here, O(n) is the time taken by the partition function to scan the subarray. This recurrence solves to O(nlogn). The recursion tree has a depth of logn, and at each level, the total work done is O(n).

- **Average Case:** O(nlogn) The average case occurs when the input array is in a random order. It's not necessary to pick the exact median every time. As long as the partitions are reasonably balanced (e.g., a 10%-90% split), the resulting complexity remains O(nlogn). For a random input, the probability of consistently choosing a bad pivot is extremely low. The mathematical analysis, while complex, confirms that the expected depth of the recursion tree is still O(logn), leading to the same average-case time complexity as the best case.

- **Worst Case:** O(n2) The worst case occurs when the partition process consistently picks the smallest or largest element as the pivot. This creates highly unbalanced partitions: one with $n-1$ elements and the other with 0. This happens, for example, when applying this specific implementation to an already sorted or reverse-sorted array. The recurrence relation becomes: $T(n)=T(n-1)+O(n)$ This telescopes to $T(n)=O(n)+O(n-1)+...+O(1)$, which is an arithmetic series summing to O(n2). For large n, this can be extremely slow and may lead to a RecursionError due to excessive stack depth.

Space Complexity

The space complexity is determined by the depth of the recursion stack.

- **Best/Average Case**: O(logn). The recursion depth is logarithmic because the problem size is halved at each step.
- **Worst Case:** O(n). In the worst-case scenario, the recursion depth becomes linear, as each recursive call reduces the problem size by only one element. This can be a significant issue for memory.

Quicksort is an in-place sorting algorithm, meaning it does not require auxiliary space proportional to the input size for storing elements. The space overhead comes purely from the recursion stack.

2.Performance Analysis of Randomization

Implementation Details: To mitigate the chance of hitting the worst-case scenario, we can introduce randomization into our pivot selection. Instead of always picking the last element, we randomly select an element from the subarray and use it as the pivot.

Design Choice: Pivot Selection Strategy (Randomized): Random Element

- **Choice:** The randomized version picks a random element from the subarray and swaps it into the pivot position (the last element) before partitioning.
- **Reasoning:** This is a robust and widely-used strategy to mitigate the risk of worst-case behavior. It effectively decouples the algorithm's performance from the input order, ensuring an expected O(nlogn) time complexity for any dataset. It's a simple change that dramatically improves real-world performance and reliability.

Analysis:

Randomization makes the algorithm's performance independent of the input data's initial order. By choosing a random pivot, the probability of consistently picking a "bad" pivot (the smallest or largest element) becomes vanishingly small, even for sorted or reverse-sorted inputs.
This ensures that the expected time complexity is O(nlogn) for *any* input distribution. The worst-case complexity remains O(n2), but it would require an extraordinary run of bad luck in random choices, making it a theoretical possibility rather than a practical concern. In essence, randomization converts the "average case" into the expected case for all inputs.

3.Observed Results:

Input Size (n)	Dataset	Deterministic Time (s)	Randomized Time (s)
	Random	0.002	0.0021

1000

	Sorted	0.0528	0.0024
	Reverse-Sorted	0.0515	0.0022
	Random	0.0049	0.0051
	Sorted	0.2088	0.0054
2000	Reverse-Sorted	0.2075	0.0055
	Random	0.0155	0.0159
	Sorted	1.3421	0.0163
5000	Reverse-Sorted	1.3298	0.0165
	Random	0.0381	0.0385
	Sorted	5.4593	0.0392
9999	Reverse-Sorted	5.3997	0.0398

4.Discussion:

1. **Random Data:** As predicted by the theoretical analysis, both deterministic and randomized Quicksort perform exceptionally well on random data. Their running times are nearly identical and scale in a way that is consistent with O(nlogn) growth.
2. **Sorted & Reverse-Sorted Data:** The results here are stark and perfectly illustrate the core problem of a fixed-pivot strategy.
    - The **deterministic Quicksort** slows down dramatically. The runtime grows quadratically (O(n2)), as seen by the huge jump in time when n doubles. For n=9999, it takes over 5 seconds, whereas it took only milliseconds for the random array of the same size. This is the worst-case scenario in action.
    - The **randomized Quicksort**, in contrast, shows no performance degradation. Its runtime on sorted and reverse-sorted data is virtually identical to its runtime on random data. By randomly selecting a pivot, it avoids the pitfall of consistently picking the worst element and maintains its expected O(nlogn) performance.

This empirical analysis provides clear, practical evidence for the theoretical concepts. It demonstrates that while Quicksort is an extremely fast algorithm on average, the deterministic version is vulnerable to common data patterns. Randomization is a simple but powerful technique to fortify the algorithm against these vulnerabilities, making it a robust and reliable choice for general-purpose sorting.