Randomized vs Deterministic Quicksort

Quicksort is an efficient sorting algorithm with a time complexity that differs based on the sorting of input data. In the best case, the time complexity will be O(n.logn). This happens when the pivot chosen divides the array into two nearly equal halves in almost all recursive calls. For example, if the pivot is always the median of the array, the problem size is halved at each step, leading to a balanced recursion tree. The depth of this tree is logn, and at each level, O(n) work is done to partition the array, resulting in the O(n.logn) time complexity.

For the average case scenario, Quicksort again achieves $O(n.\log n)$ time complexity. This is because, on average, the pivot divides the array into two reasonably balanced subarrays. Even if the splits are not perfectly equal, the algorithm still performs efficiently due to the logarithmic depth of the recursion tree. The partitioning step takes O(n) time and is repeated for each tree level, leading to the overall O(n.logn) complexity. This makes Quicksort one of the fastest sorting algorithms in practice.

However, for the worst-case scenario, Quicksort's time complexity degrades to O(n^2). This happens when the pivot is consistently the smallest or largest element in the array, causing highly unbalanced partitions. For instance, if the array is already sorted or reverse-sorted, and the pivot is always chosen as the last element, one partition will contain all elements except the pivot, while the other will be empty. This results in a recursion tree of depth n, with each level performing O(n) work, leading to the O(n^2) time complexity.

The space complexity of the Quicksort algorithm is O(logn) in the best and average cases due to the recursion stack. Each recursive call consumes space, and the depth of the recursion tree is logn when the partitions are balanced. However, in the worst case, the recursion depth can be n, leading to O(n) space complexity. Quicksort works directly on the input array without requiring significant additional memory. It needs extra memory for the recursion stack during the sorting process. The main overhead in Quicksort arises from the

recursive function calls and the partitioning step, where elements are compared and swapped to rearrange the array around the pivot.

Randomization significantly improves Quicksort's performance by reducing the likelihood of encountering the worst-case scenario. This is because choosing a fixed pivot in a deterministic Quicksort can lead to O(n^2) time complexity for sorted or reverse-sorted inputs. Randomized Quicksort prevents creating unbalanced partitions when the data is in such sorted or reverse sorted orders. By picking the pivot randomly, the algorithm makes it really unlikely that it will always choose the smallest or largest element. This helps create more even splits in the array most of the time, which keeps the average time complexity at O(n.logn). Because of this, randomized Quicksort is fast, reliable, and works well in different situations, which is why it’s such a popular choice for sorting stuff in real-world applications.

In order to measure the performance of randomized and deterministic quicksort, I used two datasets of size 10000 and 20000 with sorted and randomized data. Using these data types, I measured both time taken as well as peak memory usage.

The time taken for each quicksort in **seconds** is shown below:

Input Type	Deterministic (10,000)	Randomized (10,000)	Deterministic (20,000)	Randomized (20,000)
Sorted Data	1.745727	0.012268	7.522507	0.020355
Reverse Sorted	1.633290	0.010356	7.212292	0.020389
Random Data	0.009230	0.011227	0.019953	0.023500

From the observed results, randomized quick sort performs consistently on all types of data. Deterministic quick sort's performance differs based on data type, as we can see a big time difference between sorted and reverse-sorted data. Deterministic quicksort takes

significantly longer for these cases due to its worst-case behavior when the pivot selection results in highly unbalanced partitions. Randomized quicksort fixes this issue by choosing pivots randomly, which ensures better partitioning on average. The results observed align with theoretical understanding, where deterministic quicksort exhibits O(n^2) worst-case complexity for already sorted or reverse-sorted data. In contrast, randomized quick sort is more likely to achieve O(n.log n) average complexity across all cases.

Here is the peak memory usage of both type of quicksorts in **KB (Kilo Bytes)**:

Input Type	Deterministic (10,000)	Randomized (10,000)	Deterministic (20,000)	Randomized (20,000)
Sorted Data	416241.53	378.48	1662105.12	823.37
Reverse Sorted	416866.40	340.14	1663354.99	689.05
Random Data	307.86	369.05	846.75	600.34

Deterministic quicksort uses a lot more memory than randomized quicksort, especially when sorting already sorted or reverse-sorted data. This happens because deterministic quicksort picks bad pivots in these cases, causing deep recursion and high stack memory usage. On the other hand, randomized quicksort chooses pivots randomly, which helps keep the recursion shallower and reduces memory use. The massive gap in memory usage for larger inputs (e.g., over 1.6GB for deterministic quicksort with 20,000 numbers compared to less than 1MB for randomized quick sort) matches the theory that randomized quicksort splits the data more evenly, leading to less recursion and lower memory overhead.