**Performance Analysis of Quicksort Algorithm: Deterministic vs Randomized**

Anjal Lawaju

University of Cumberland

Algorithms and Data Structures

Vanessa Cooper

27 July, 2025

**Performance Analysis of Quicksort Algorithm: Deterministic vs Randomized**

**1. Introduction**

Quicksort is one of the most well-known sorting algorithms in computer science. It is known for being able to sort large datasets very well. This method of divide-and-conquer, which is based on comparisons, splits an array around a pivot element and then sorts the two subarrays again. The major goals of this study are to implement the deterministic and randomized Quicksort algorithms, compare their speeds, and look into how randomization changes how Quicksort works. By testing these versions with different amounts and types of input, we look at how they work in real life and what they mean in theory.

**2. Quicksort Implementation**

**Deterministic Quicksort**:

The deterministic version of quicksort picks a fixed pivot, which is usually the final element of the array. Then, the array is split into two smaller arrays: one with items that are bigger than the pivot and the other with things that are smaller than the pivot. After that, these subarrays go through a recursive sorting procedure (Kneusel, 2024). During the partitioning step, each subarray is sorted on its own, and the elements are moved around to line up with the pivot.

**Algorithm Steps**:

* 1. Select a pivot (last element in this case).
  2. Partition the array so that elements less than the pivot are on its left, and elements greater than the pivot are on its right.
  3. Recursively sort the left and right subarrays.

**Randomized Quicksort**:

Before partitioning, the pivot in the randomized variation is chosen at random from the array. This (Vaidya, 2021) means that the worst-case time complexity (O(n^2)), which arises when the pivot selection always gives you unbalanced partitions (for example, if the pivot is always the smallest or largest element), is less likely to happen. When the pivot is random, the algorithm often works better since it is less predictable.

**Algorithm Steps**:

* 1. Randomly choose a pivot element.
  2. Swap it with the last element in the array.
  3. Partition the array around this new pivot.
  4. Recursively sort the left and right subarrays.

**3. Performance Analysis**

We look at how long it takes Quicksort's deterministic and randomized versions to sort things in different situations to see how well they work:

* **Best Case**: When the pivot divides the array evenly, O(n log n) comparisons are conducted, which is the best-case time complexity.
* **Average Case**: Quicksort typically acts like the best case, with a time complexity of O(n log n). When the pivot splits the array into two subarrays that are about the same size, this happens.
* **Worst Case**: The worst-case time complexity happens when the pivot is almost always the smallest or largest member, which means the array is split unevenly. The result is an O(n^2) time complexity. If the input array is sorted or nearly sorted, this might happen in the deterministic version. The chances of this worst-case scenario happening are far smaller in the randomized version.
* **Space Complexity**:

Both versions have a space complexity of O(log n) (Kneusel, 2024) since Quicksort is a recursive algorithm that needs space for the call stack. However, in the worst situation, the space complexity might become worse and approach O(n) when the recursion depth reaches O(n) (for example, when the pivot is constantly out of balance).

**4. Empirical Analysis** The following are the real-world results of running the deterministic and randomized Quicksort algorithms:

* **Deterministic Quicksort**: It took 0.0039985 seconds to sort a random array with 1000 elements. This is usual for Quicksort when the pivot selections frequently lead to balanced partitions.
* **Randomized Quicksort**It took 0.0020027 seconds to sort the same array. In this situation, the randomized version was faster because it made the worst-case scenario less likely to happen. This would happen more frequently in the deterministic version with highly ordered inputs.

**Impact of Randomization**:

When the input array is already sorted or nearly sorted, deterministic Quicksort is more likely to have the worst-case performance (O(n^2)). This is not the case with randomization in Quicksort. Randomized Quicksort frequently works better with input distributions like sorted, reverse-sorted, and random arrays. Choosing the pivot at random makes sure that the partitioning process is well balanced, which is important for keeping the best performance (O(n log n)).

**5. Conclusion**

Using real-world data and theoretical time complexity, we compared the performance of deterministic and randomized versions of the Quicksort algorithm. Though it often excels in typical scenarios, deterministic Quicksort may fall to quadratic time complexity in the worst case. When the input data is sorted or nearly sorted, however, the randomized version improves reliability and, in many cases, yields faster answers while decreasing the probability of encountering the worst-case scenario. By significantly enhancing its performance in real-world scenarios, our empirical tests show that randomization makes Quicksort a more robust algorithm for broad adoption.

**References**

Kneusel, R. T. (2024). *The Art of Randomness: Randomized Algorithms in the Real World*. No Starch Press.

Vaidya, K. E. (2021). *The case for a learned sorting algorithm* (Doctoral dissertation, Massachusetts Institute of Technology).